NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM

by

DAVID PATRICK REED

B.S., Massachusetts Institute of Technology
(1973)

S.M., Massachusetts Institute of Technology
(1976)

E.E., Massachusetts Institute of Technology
(1977)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1978
(i.e. February 1979)

Signature of Author.....
Department of Electrical Engineering and Computer Science, September, 1978

Certified by .....
Thesis Supervisor

Accepted by .....
Chairman, Department Committee

ACKNOWLEDGMENTS

This dissertation would not have existed were it not for aid and comfort from many sources. I can hope to mention only a few important people. To all of the others whose belief in me made life worthwhile, please accept my thanks.

Professor Saltzer is one of the most helpful critics I have had, and his diligence in reading the many drafts always exceeded my expectations.

Dr. David Clark and Professor Steve Ward, my readers, helped to clarify my explanations and by urging more work on the mechanisms of NAMOS, contributed to what I think are significant improvements in my original ideas.

Jim Gray of IBM San Jose Research deserves special mention, for although I had relatively few discussions with him, every one led to new clarifications of my ideas. His insight into the system aspects of concurrency was always clear and fresh.

All of the members of the Computer Systems Research Group, who provided a home away from home, deserve special thanks for just being who they are.

Lynn, my wife, and Colin, my son, deserve all the thanks I can give for helping me through the grueling phases of this research, particularly the last month or so. Without their help, I probably would never have made it.

The work reported here was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science, an interdepartmental laboratory. This research was sponsored in part by the Advanced Research Projects Agency (ARPA) of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661.

CONTENTS

FIGURES

Chapter One

Introduction

With the advent of minicomputers and low-cost computer networking technology, a new sort of computing technology is becoming quite important. The basic characteristic of this technology is the development of a decentralized set of computing resources (computers and terminals) organized to provide computers, terminals, and storage devices that are located near their ultimate users. Computer networks, either of high-bandwidth typical to local network technologies such as the ETHERNET[Metcalfe76] or the long-distance networks such as the ARPANET[Metcalfe73], provide the necessary sharing of data and computational resources among geographically decentralized but closely related computer applications.

The term *distributed computing* has been used to describe the loosely coupled systems built using this technology. But like many other fashionable terms, distributed computing means different things to different users of the term. It has been applied to parallel computation (in this use, distribution is parallelism), offloading of computation from a mainframe computer to a front-end mini or intelligent terminal, construction of computational engines via elaborate interconnections of microprocessors, and a host of other variations on the theme of several computers tied together by some communications medium. Our use of the term as defined above specifically emphasizes decentralization as a key attribute obtainable by introducing communications into a system design. Such decentralization involves separation of the computers in the system by physical distance, by boundaries of administrative responsibility for individual computers and their applications, and by firewalls intended to increase the overall availability of the system as seen by its users in the face of component failures.

One of the major forces in the move to decentralized, distributed computing is the opportunity for autonomy gained by having direct, physical control over the source of one's computing resources. Traditionally, the computing resources of a large company or organization have been provided by a large central computer facility managed by a separate division of the company. The main reasons for this traditional structure have been the large cost benefits of sharing large computer systems that provide very high price performance, and the control over computing usage afforded by the centralization. As pointed out by d'Oliveira[DOliveira77], many organizations have very strong forces that lead to decentralization, including psychological and economic ones. Given the decrease in hardware costs for small computing facilities, this has led to, and probably will continue to lead to, more autonomously operated computer facilities in the context of these organizations. The need for autonomous control of computing resources seems to be often more important than cost.

Although the hardware technology for distributed computing is well developed, the protocols and conventions for the design of systems that support distribution of data and application programs are still in their infancy. Perhaps the major effort in this area has been the resource sharing research carried out in conjunction with the development of the ARPANET. The most sophisticated product so far of this research has been the National Software Works[Crocker75], a distributed set of application development tools that can transparently share files across a network. Although this sort of research has led to a great deal of insight on how to distribute an application, it has not yet reached the point where the design of such systems is simply the application of well understood methodologies.

Perhaps the greatest problem in the development of distributed systems is the development of methods that allow local applications and data bases to be created autonomously, then integrated with other applications and data accessible in the distributed system in a *post hoc*

fashion. Evolution of distributed system applications by integration of existing applications seems to be a natural result of the reasons for decentralization.

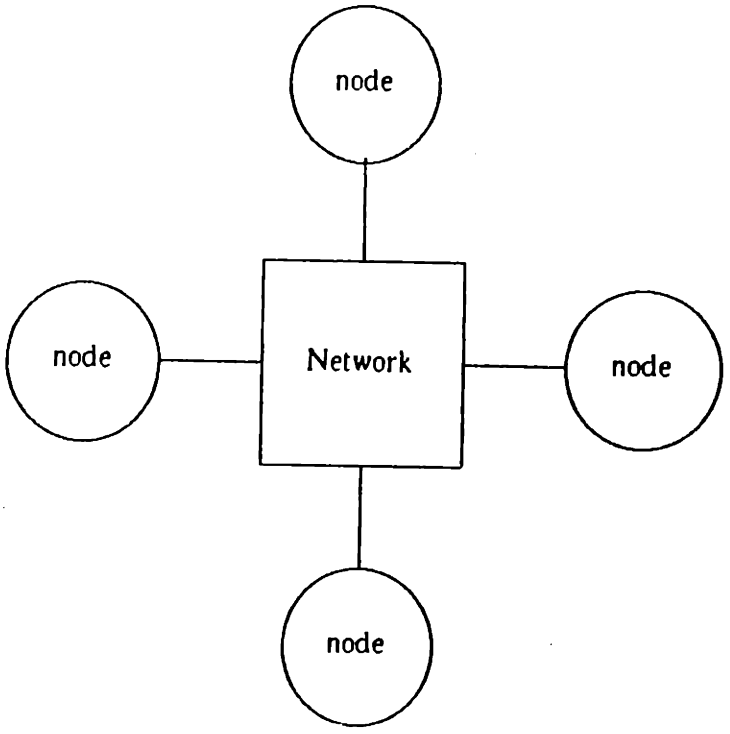
In order for this problem of *post hoc* integration to be solved, a coherent set of protocols and standard interfacc tools must be developed. Then the task of constructing new systems by integrating existing systems on multiple nodes can be simplified to matching interfaces specified in the same way.[1] Important factors in these interfaces are data types, naming of individual objects, synchronization of accesses to sharable objects and reliability of computation in the face of failures of individual communications lines and computers. The latter two problems, synchronization and reliability, form the focus of my thesis. My goal has been to provide a standard set of mechanisms for the implementation of interfaces that provide control of computations in an environment that is prone to failure and has concurrent computations. The solution proposed is to extend the object naming mechanism so that it can be used for proper synchronization and failure recovery.

1.1 The problem of a useful internode interface

A typical distributed computer system of the kind we want to consider will look like a collection of autonomous *nodes* connected by a communications network, as in figure 1. These nodes may be individual processors with their own memory, or they may be multi-processor systems of any variety. For our purposes, they are distinguished because each node behaves like a single system -- each node is either completely available to accesses through the network, or is

1. It is interesting to note that Backus, in his Turing award lecture,[Backus78] has pointed out that he also believes that construction of systems from pre-existing modules is a problem not yet properly solved. His proposal is somewhat different from ours, in that he uses composable modules athat are pure functions from inputs to output that are combined into top-level transactions that work on a shared system state. Our proposal allows the composition of transactions out of existing transactions that are designed themseves in terms of modifying the system state.

Fig. 1. Distributed System



completely unavailable. Further, the resources owned by each node are wholly controlled by the owner of the node; there is no higher authority that controls the resources on all nodes of the network.

Nonetheless, there is a great need for the ability to share the use of data and computational resources among nodes with different owners. For example, consider a relatively decentralized corporation that has several independently developed inventory control data bases residing in different computational nodes. Eventually there will probably be pressure to have an information system for overall management of the divisions owning each node that can look at the state of all data bases. Because the cost of replacing existing systems with an overall system would be prohibitive (as well as infringing on autonomy - see below), that system must be built out of the old system by making use of preexisting interfaces, if possible. It is thus a great

advantage if interfaces can be designed so that they can be later used in unplanned-for ways. In addition, by providing interfaces that other nodes can use, a node can offer and obtain information about other divisions use of the parts it makes and consumes, thereby obtaining a greater degree of optimization of its own operations.

Well designed, semantically clean interfaces that allow for unplanned later uses seem to be the key to successful sharing of programs and data in these new distributed systems. Such interfaces must not interfere with, and we hope will contribute positively to, solutions to problems that become important in decentralized systems, such as autonomy, failure management, synchronization, and conversational interaction. These issues will be discussed in detail presently.

It is not the goal of the thesis to solve the problem of constructing distributed applications out of existing (as of 1978) programs and data bases, under existing operating systems connected by some kind of network (a system that does attempt part of this goal is the National Software Works (NSW)[Crocker75]). Many such existing programs and data bases simply don't support compatible approaches to synchronization and reliability that are needed to achieve reasonable results. The primary goal is to provide tools to aid in the construction of new software for the distributed environment. If these tools are used, then the task of *post hoc* integration will be much simpler.

It may turn out to be fairly easy to adapt some existing programs or data bases to fit within the scheme developed here in the thesis. If so, then the tools developed here can certainly have a more immediate impact. The primary impact that these ideas should have, though, is in the design of future applications either for distributed systems or for computers that may eventually become nodes of distributed systems.

1.1.1 Autonomy

Freedom means
you're free to do
just whatever
pleases you;
- if, of course
that is to say,
what you please
is what you may.

-- Piet Hein

The nature of a node is captured in the notion of autonomy. That is, a node is basically free to manage its own use of its own resources in any way it sees fit. That is, a node may be available only occasionally for communications with other nodes (e.g., because the power is off from 5 P.M. to the following A.M.), much of the data stored on the node may be completely private and never accessible through the network, other data may be usable only in a laundered form (for example, only statistics of a general sort about a corporate division's production may be available outside the division), and the sorts of actions that can be carried out at a node on behalf of a remote node may be severely constrained to limit computer time resources and/or interference with local computing tasks.

Moderating the complete freedom given a node is the need for providing some kind of reliable sharing of useful data and resources. If a node is to usefully offer an interface whereby other nodes can access some of its data or resources, it needs to provide some reasonable guarantees of availability and proper behavior of the interface. The node does not want to give up more autonomy than necessary, though. Consequently, the impact of autonomy on the design of internode interfaces is the need for interfaces that are the minimal necessary infringements on node autonomy.

It is interesting to note how the picture of a distributed system with autonomous nodes differs from the notion of a computer utility[Corbato65,Frankston74], a centralized marketplace in which data, programs, and services can be shared. A computer utility might be best defined as a vast repository of data and programs that can be simultaneously manipulated by the users of the system. The availability and integrity of the underlying hardware and software mechanisms that support the shared data must be as high as that needed by the most demanding application using the system. Protection mechanisms must also exist to ensure that unauthorized sharing of or tampering with data does not occur. In the distributed system, the network is a marketplace for services and data, where the nodes may or may not offer services. The major result of decentralization is that the entire system need not be designed to meet the most stringent requirements of availability and mutual protection. Only those nodes implementing and using services with stringent requirements must be specially designed and built to meet such requirements.

1.1.2 Object Interfaces

In order to share data or programs with users outside his node, the owner of a node must provide some way for the users to refer to the data and request execution of the programs. Simply giving out disk addresses and enabling the ability of remote computers to load programs into his computer's memory would certainly allow remote use of the node's resources. However, giving such low-level information may make it difficult for the node's owner to retain much control over the ways in which the resources are used.

Object-oriented programming languages systems such as CLU[Liskov77a] and ALPHARD[Wulf74] provide an object abstraction that limits the view of the internals of data objects. A data object is just a named entity wholly characterized by its behavior in response to operations applied to it. The major benefit ascribed to object-oriented systems is that the behavior of an object can be understood, specified, and used without reference to the actual implementation of the object and operations in terms of primitive objects and operations. In supporting autonomy, a dual benefit of great importance is also obtained by using object-oriented systems. The behavior of objects can be understood, specified, and implemented without reference to their eventual use. Since the implementation of objects is hidden from the users of the objects, restructuring the implementation is easily done. Protection constraints often are more easily expressed in terms of allowable uses of abstract operations on particular abstract objects.

In this thesis, a goal is to support modularity as provided by object-oriented systems. In particular, the construction of abstract operations out of simpler abstract operations and primitive operations and the construction of abstract objects out of simpler objects and operations are supported.

There are some problems in translating the object notion to a distributed system, however. The primary ones have to do with the opportunity for concurrent accesses to shared objects, the possibility of failure in the middle of executing an operation on some resource, and the need to accomplish reliable coordinated operations involving more than one node.

1.1.3 Concurrency

The main focus of the thesis is providing reliable interfaces to objects in the presence of concurrency. Now, concurrency has been a heavily studied area of computer science, so the immediate question is, why are not the existing techniques for managing concurrent computation adequate for the present situation? Perhaps the major difference between my assumptions and those common to much of the research into concurrent programming is that I am not willing to assume that all of the users of a shared interface are designed at the time a shared interface is designed.

Consider an object that can be manipulated by some using processes. The usual approach (using monitors[Hoare74], for example) to defining an interface to that object is to define operations that reserve the exclusive use of the object to the calling process, and release exclusive use. Two problems arise in defining this interface. First, one must be aware that the object might be concurrently accessed, so that the appropriate synchronization operations can be defined. Second, one must have a way of enforcing the use of the synchronization operations in order to prevent unexpected concurrency. If all the users of the object are designed at or before the time the object's interface is designed, neither problem is particularly hard, but in the case of interest, where unplanned sharing and concurrency are likely, both problems become extremely difficult.

Another synchronization problem in the use of objects is the need for unplanned composite operations on multiple, independently designed objects. A simple example in the distributed system environment might be where two inventory control systems are brought together after being independently designed. Suppose that a new function of the combined system

is to be the ability to place an order, involving "atomically" checking the supplier system for sufficient supply, marking the ordered items as destined for the supplied system, and adding to the list of expected shipments in the supplied system the ordered items and estimated arrival. Getting this action to be atomic with respect to other concurrent actions of each inventory system that may involve the same parts is quite difficult to do without risking deadlock. A redesign of the individual systems may even be required.

The construction of new operations out of existing atomic operations is particularly difficult in a system that uses locking for synchronization. Suppose that two modules dealing with different sets of data are to be used together to create a new composite operation. It is not sufficient to let each module set and release its own locks, because then the composite operation would not be atomic -- it would be possible for another concurrent program to observe the state of the system after executing one module but before starting the second if the second was somehow delayed for a while. Consequently, in combining atomic operations that use locking, the composite module must be aware of the locking conventions of the combined operations. Since the program invoking the operations being combined is responsible for properly setting the locks, suddenly the modules are not so modular any more. They are dependent on their caller to properly set locks and avoid deadlock. An even more serious problem arises if the locks set by a module are dependent on the parameters with which it is invoked, because using such a module in the construction of larger atomic operations would require that the using program be aware of the internal construction of the module to an even greater degree. In the extreme case, the using program would have to execute the same steps the module would execute to determine what locks needed to be set before actually calling the module.

A goal of the thesis is to define a method for handling concurrency that can be easily and naturally used in the construction of abstract objects and operations. The concurrency control method is to be built into an abstraction, so that concurrent use is never "unexpected". The construction of new abstractions out of existing ones containing their own built-in concurrency control is to be supported, so that it is rare that a system must be entirely redesigned just because of a new use in conjunction with some other system that has its own concurrency control.

Very few synchronization schemes can handle this requirement for an unplanned atomic action composed of predesigned operations. A notable exception is the concept of a *supervisory computer program* in the IDA operating system described by Gaines[Gaines72], in which relatively arbitrary programs could be specified to act atomically. The IDA system idea could not be easily implemented on a distributed system because it depends on the centralized operating system notion of a "locked" supervisory state.

1.1.4 Need for a Robust Interface

An important class of failures in a decentralized system result in either temporary or permanent loss of availability of some set of resources. Examples include communications failure that might cut off access to some set of nodes and data objects from some using program, a crash of a computer that might have similar effects in addition to destroying the state of any computations in the middle of execution, and detected software errors indicating that an object is in an impossible state.

These failures can occur at any time an attempt is made to use some resource. A program that executes in the distributed environment must always be prepared to discover that some resource it is using is suddenly unavailable for some reason.

Autonomy can also lead to reasons for resources to become suddenly unavailable. The owner of a node may suddenly turn off his machine, revoke access rights for a particular set of objects granted to some program while that program is using them, etc. Such examples are not limited to distributed systems. In Multics, for example, a computation may at any microsecond of its execution discover that its right to use a segment it has been reading for the last ten minutes have been revoked.

Since abstract operations and objects are constructed out of simpler ones, such that the execution of an abstract operation may involve many steps dealing with many different resources, there are many different points at which loss of availability can strike the implementation. Nonetheless, a desirable feature of abstractions is that their behavior should not be strongly dependent on the implementation. Thus, an attempt to perform an operation on some abstract object ought to have a well defined effect if it cannot complete due to some loss of availability encountered during its execution. This effect should be specified in terms of the abstract view of the operation, not in terms of the program and data it uses in its implementation.

Abstract operations that modify the (abstract) state of some abstract object or objects become quite difficult to support in an environment where sudden loss of availability must be expected. Since the implementation of the abstract operation makes the modification by a number of steps, there may be points during the execution of these steps where loss of availability of some resource leaves the implementation at a point that has no meaning in terms of the abstractions

being implemented. Some recovery from this is necessary. The simplest approach to recovery is to undo the steps already taken in the operation, so that the operation can be thought of as having no effect if any resource it uses is unavailable during the operation.

Unfortunately, in the case of failures undoing what has already been done is not straightforward. Further loss of availability may make it impossible to undo what has been done by simply reversing the changes made. If the resource that becomes unavailable is the processor that is executing the operation, we have an extreme case of being unable to undo what has been done. We may not even know what has been done so far.

Correction of failures by undoing results also interacts strongly with synchronization. In order to properly undo a computation, one must also ensure that independent computations do not observe the transient state during which the abstract operation was attempted but not yet undone.

Although the user of an operation may not be in a position to know how to recover from a failure, the system as a whole (all nodes involved in the operation) can maintain this knowledge. In order to do this, the system must be aware of interfaces, and must be able to decide that a computation has failed and effect the undoing of operations when a failure occurs. The system, in order to decide that a computation has failed, cannot depend on the program or the user of the program, since one or both of these may have also failed. Nor can the system depend on being able to access all of the nodes containing objects that have to be corrected at the time failure is detected. Consequently, the algorithms used by the system for recovery must be very carefully designed to work correctly in the face of the same loss of availability that caused the original failure.

An alternative approach that might be taken to handle failures that result in loss of availability is to build the system so that such failures never show through to the programs executing on the system. Essentially this approach amounts to guaranteeing availability. It is usually possible to guarantee that resources are available to computations that use them given either that the computation can afford to wait, or that enough money is allocated to buy sufficient redundancy within the system to reduce the probability of failure. The tradeoff is not always possible, however. Money is often in short supply. Computations may often be executing in behalf of an interactive user at a terminal who cannot afford to wait until some remote node his program had started to use is repaired. Consequently, the approach of having the system provide a mechanism that allows undoing of computations that fail in the middle is often the best.

The case where the owner of a node decides to make it unavailable differs slightly from the failure case in that by reducing the owner's autonomy it is possible to reduce the expectation that loss of availability of this sort interferes in a bad way with users of shared objects. Nonetheless, the name of the game is to permit as much autonomy as possible. Were the owner to discover that a bug was allowing remote users to access too much of his data, it would be nice if he could shut off access to his shared objects immediately, even in the middle of "atomic" actions, should that action not cause his own data to come to harm. Analogously, even in a central system, protection mechanisms that allow for immediate access revocation can introduce severe malfunctions into operations on shared data if invoked at the wrong time. Consequently, the strong degrees of autonomy allow actions by owners that look a lot like unpredictable failures from the user's point of view.

As a result of these arguments, object interfaces that can handle sudden loss of availability are absolutely essential in the autonomous distributed system environment.

The model of failure recovery we have specified is closely allied with the *termination model* of exception handling espoused in the exception handling mechanisms of the CLU language[Liskov77b]. Upon encountering a failure that prevents execution of the module, the execution of the module is terminated. In the CLU termination model, the effect of the module for each type of failure is specified as part of the interface; in contrast, we have taken the "stronger" view that a failure is to be made equivalent to never executing the module, unless that module explicitly chooses otherwise. Because the programmer of a module cannot be expected to know about all possible failures that may result during the steps of execution of the module, the "stronger" view is safer, handling unexpected failures more effectively.

An alternative exception handling mechanism is the *resumption model* described by Goodenough[Goodenough75] and Levin[Levin77], in which the module encountering an error is suspended and possibly resumed after recovering from the error. Such a model did not seem appropriate for handling failures resulting in loss of availability because a) the executing module may lose its state, b) after an availability loss, it is difficult to tell a handler what to do to recover from the the error, and c) it is hard to design the handler of such errors without it having to include detailed knowledge of the internal workings of a module, so that the level of abstraction of the interface is compromised.

1.1.5 Conversational Interactions with Multiple Machines

Another aspect of the distributed system organization is that it ought to allow on-line construction of unplanned-for actions determining the state of shared objects at several nodes, and possibly even modifying other nodes as the result of some decision made by a person sitting at a terminal. An important special case occurs when the several machines are independently designed databases. Here the problem is that the program and its actions are being created as the program is executed, and outside the control of the system itself. Synchronization and failure management techniques that work when the program is executing completely under the control of the machine may not work. In addition, even if programs can be prevented from making mistakes by some kind of verification method built into the system, the user will make mistakes, and should have at hand means to recover from his mistakes. If the failure management techniques built into the system can generalize to the case of recovering from user mistakes, this would tremendously aid in conversational use of systems.

1.2 Naming Mechanism as a Solution

This dissertation describes a system called NAMOS (Naming Applied to Modular Object Synchronization). NAMOS consists of a unified approach to the problems of synchronization and reliability just described. Of particular concern are the problems of constructing modular systems and the problems of unplanned concurrency and unexpected failure that we expect to arise in the construction of distributed systems.

The central idea of the thesis is an unusual view of synchronization of accesses to shared data. Traditionally synchronization has been achieved by mutual exclusion. The approach to synchronization used in NAMOS is based on a mechanism for naming states of the system and objects (hence the title of the dissertation). To understand the difference between the two approaches it is helpful to use a non-computer analogy in which both techniques are well developed.

Consider a set of files (say personnel files) kept in a file storage room of some organization. We may think of each file folder, labeled by a person's name, as an object of the database. Occasionally, files must be inspected or updated. For example, one might wish to compute the average salary of women in the company. Or as a sample update, one might wish to modify the salaries of the women in the company by an appropriate percentage so that the average woman's salary is equal to the average man's. We impose a rather strong constraint on these operations that is not usually required in such a set of files. The constraint is that these "transactions" on the files must be atomic operations. That is, during the computation of the average woman's salary, no woman's salary is changed by some other clerk. Similarly, in updating the women's salaries, no other clerk is to change any of the men's or the women's salaries, to ensure that equality is achieved. The strong constraint is not normally required in human-managed systems because people are good at dealing with inconsistency. Computer programs using a database, on the other hand, have few checks built in to deal with inconsistency, so avoiding inconsistency is much more important.

One simple way to solve the problem of synchronizing the accesses made by clerks to the database is to allow only one clerk into the room containing the files at a time, and requiring that he remain there until completing the transaction. This is the basic idea of mutual exclusion. The clerk gains exclusive access to the entire state of the database, and can then make the modifications needed to construct the next state of the database. A refinement can be made to this approach, because normally clerks will need to access only a subset of the database. The refinement consists of having the clerk go into the room and collect all of the files he needs to read and update, replacing each file folder with a note that indicates that the clerk has taken the file folder to his desk. The clerk can then work with the set of file folders at his desk in private, and other clerks can work on different sets of files independently. A clerk needing to access a file folder that has been removed must wait for the folder to be returned. This approach to synchronization is analogous to locking in the use of a computerized database. Each clerk performs transactions by gaining exclusive access to the group of files he needs to access for some period of time.

The approach in NAMOS is quite different. Assume that each file folder contains only one item, say the salary of the person whose file it is. Instead of erasing the current salary and replacing it with a new one when the salary is changed, the salary of an individual is changed by adding to the file a new sheet of paper with the new salary. Each sheet of paper with a salary is stamped with the date and time when the salary becomes effective. How does this help? First of all, consider a transaction that only reads the database, such as the one to compute the average salary of all of the women. Instead of having to seize all of the folders to prevent changes from happening, the clerk can simply take his time in processing the folders; he simply must choose a date and time for which the average is to be effective, then go to each woman's folder and read

out the salary corresponding to that date and time. Concurrently, other clerks may update the women's salaries. However, a consistent computation of the average salary does not interfere with the clerks that are updating salaries.

This strategy is equally applicable in a distributed database. If the personnel files of a company are distributed to the company's many locations, it may be unacceptable to gather up all of the files of women employees in order to send them to company headquarters to compute the average. Instead, company headquarters can send a memo to clerks at each site with instructions to sum up the women's salaries effective as of a common date and time. The key idea here is that the central headquarters can construct a name (consisting of the effective date and time itself) for the state of the database at a particular time, and then use that name to gain access to that state of the database even though the database is under constant change. NAMOS provides a similar mechanism to computer programs -- the ability to name particular states of the data stored in the system along with the ability to use those names to gain access to the values of data objects in the state.

Synchronizing updates in the salary database is somewhat tricky, however. The salary adjustment for the women is performed by having the clerk decide upon a time when the adjustment is to be effective. He then must compute the average salary of the men and the average salary of the women as of just before the effective time of the adjustment, to preclude any other changes to individual salaries happening between the computation of the adjustment percentage and the actual application of the adjustment to the women's salaries. After computing the adjustment percentage, the clerk then can go to each file and add a new salary sheet with the adjusted salary.

The tricky part is in the interaction between the adjustment transaction and any other clerk's attempt to read salaries. Let T be the time the adjustment is to be effective, and T-1 be the time at which the averages are computed in order to compute the adjustment percentage. Although the adjustment is effective at T, it may be that the performance of the adjustment is not completed until sometime after T because the job is so difficult. Then it is possible that another clerk will want to know the salary of Jane Jones at time T even though it has not been computed yet. He may not even be aware that an update is in progress. The most recent salary of Jane Jones recorded in her folder is that of an earlier time. If he takes that salary as the value effective at T, and then the adjustment is completed, then he will be wrong. There are two solutions to this problem incorporated in NAMOS. First of all, reading a value out of the folder always includes making a notation on the sheet containing the salary read that indicates the effective time of the read. Thus, if Jane Jones's salary is read as of time T, the sheet containing the salary believed to be effective then is noted to have been read at time T. When the adjustment is applied, it will be discovered that someone has already read a different salary than the one that has been computed to be effective at time T. The clerk doing the adjustment would then have no choice but to undo all of the adjustments to salaries he has made thus far, choosing a new time for his adjustment to be effective. This is NAMOS's primary solution, guaranteeing that each transaction is atomic.

In the case of the salary file, however, aborting the adjustment of all women's salaries because someone at random read Jane Jones's salary is a bit impractical. The solution does, however, work well in many applications. For cases like the salary adjustment, though, NAMOS provides a second mechanism as a refinement (the refinement is not described until chapter six). Essentially the refinement amounts to allowing the clerk to mark all of the women's folders in

advance that a change may be made to the salaries as of time T. Thus a clerk that queries Jane Jones's file will observe that the most recent salary on file for Jane Jones is likely to change as of time T. He can then wait until the change is made, or he can ignore the notation and read the most recent salary (deciding that it is effective as of time T, and eventually aborting the adjustment as before).

A rather surprising property of the naming approach is that it is not necessary to predict in advance what records might be accessed. Instead the naming mechanism ensures that whenever a particular file is accessed, the proper version is obtained. In the locking or mutual exclusion approach, a consistent state is observed by assembling all of the relevant folders on one's desk at the same time. In the naming approach, one can get a folder at a time, read the correct version, and return it to the files, and still obtain a consistent state of the system. This property is the essence of NAMOS's solution for the problem of constructing new systems from existing ones. In a locking system, composition of a new function from several pre-existing ones usually requires doing all seizing for the composite operation before any component operation is executed.

NAMOS includes, as well as the naming approach to synchronization, an approach to recovering from failures. Essentially the problem can be modeled in the personnel file case as what happens when a clerk has a heart attack while in the middle of carrying out a transaction (or in a distributed system, if one of the planes carrying a message to a clerk at a remote site crashes). Part of the update may have already been made, and no other clerk may know how much the clerk had actually done. In the mutual exclusion case, to prepare for such an eventuality, each clerk will have to diligently record the old value of any salaries he updates in

some sort of update log. He cannot return any files to the file room until he has completed the update, because it might happen that that file would be picked up by another clerk, used, returned, and then the original clerk may have died. The problem is that the original clerk then must have his work erased, but the other clerk has read the output of that work and there is no record that he has read it.

NAMOS takes a different approach, that has a similar effect. Since an update generates a new version, it is only that version that is affected if the clerk dies after performing part of an update. The solution is to add to the sheet of paper containing the new version a note to the effect that "this sheet is part of a coordinated update being performed by clerk John Jones. To find out if the update is completed, call John and ask if update 0987654327 has been completed." This has the effect that if John dies, someone will answer the phone and say that John has died. John then merely has to record somewhere what the numbers of the updates he has performed are, so that when he dies, the person taking over his job can answer the question. The update in progress when he dies will not be performed, which is what is desired.

In some databases, recovery from failure is achieved by recording in a central place a log, called an update log, of all changes made by transactions. An entry in the log consists of a "transaction identifier" to identify the transaction that changed the value, the name of the object changed, and the old value before the change. A partly completed transaction can be undone by searching backwards through the update log and undoing all of the changes made. In a sense, the multiple versions of objects kept by NAMOS encode the same information as the update log, but the old versions of objects are also directly accessible for transactions to read after changing the objects, simplifying the synchronization of concurrent transactions.

With this example the basic elements of NAMOS have been characterized. In the remainder of the thesis, we explore the actual mechanisms needed to make the analogy work in real computer systems. This involves some careful definition of the exact behavior of the synchronization and reliability mechanisms, and some engineering tradeoffs in making the system perform well.

What has not been captured in the analogy is the notion of constructing transactions as modules that can be used in the building of other transactions. It was noted above that because synchronization is achieved by naming states of objects rather than seizing control of the objects, the NAMOS approach does not require all resources used by a module to be known to its caller. Exploiting this property requires designing in additional structure to the names used for states of the system that is not present in the "date and time effective" name used for states of the personnel files. The structure needed will be described in chapter three.

1.3 Related Work

The fields of synchronization and reliability in computer systems are old, so any attempt to list exhaustively the related literature would be unfortunately long. However, a number of relatively recent developments in these fields have particular bearing on the problems and approaches described in the thesis.

Distributed systems are a more recent phenomenon, and the related literature on the kinds of approaches referred to here is very small. The idea of distributed systems that are composed of autonomous nodes integrated only loosely through agreements to use a common mechanism for sharing information and services through the network is best described by

d'Oliveira[DOliveira77] and Saltzer[Saltzer78]. Related work to develop a system for integration of existing programs on a set of relatively autonomous nodes is to be found in the National Software Works Project described by Crocker[Crocker75]. Our work differs from the National Software Works project in that it does not take as a requirement the fact that existing programs and data need to be integrated, and can thus define a much more coherent interface to be used by programs to facilitate easy sharing.

The work in developing languages to support design of objects and operations in which information about the details of the implementation is largely hidden from the user is basic to our approach to defining a system to support decentralized development of software that is later shared. The languages CLU[Liskov76,Liskov77a] and ALPHARD[Wulf74], along with the operating system kernel Hydra[Wulf75], are essential precursors of the present thesis.

Our approach to synchronization is derived from two distinct but closely related ideas. First, the notion of version numbering to achieve synchronization is closely related to the synchronization mechanism developed by the author and Kanodia[Reed78]. Maintaining a sequence of versions of an object was inspired by an idea present in both the TENEX file system[Bobrow72] and the ITS file system[Eastlake69] where multiple versions of a file can be catalogued with successive version numbers while accessing a file gets the most recently created version by default. This provides a primitive synchronization mechanism among the accesses to a file, allowing a new version of the file to be written while the older version is still being read, giving a sort of "read-locking" for free.

The second related group of ideas involves the use of timestamps for synchronization. Johnson and Thomas[Johnson75] suggested the first such mechanism, which used timestamps plus an underlying property of the network that messages are delivered in order to assure synchronization of a simple distributed data base. Thomas[Thomas76] elaborated this approach to allow somewhat more general operations, while still requiring that the database be completely replicated at each node of the system. Lamport[Lamport78] describes the use of timestamps to define an ordering among requests that can be used for synchronization, and a simple algorithm to achieve mutually exclusive execution of sequences of program in time in a distributed system based on timestamps. The SDD-1 distributed database system developed by Computer Corporation of America uses timestamps internally to enforce a locking strategy[Bernstein77,Rothnie77].

The major difference between the use of timestamps to achieve synchronization by these projects and our use of pseudo-time in NAMOS is that pseudo-time is a part of the "programmer's box of tools" in NAMOS, whereas timestamps are hidden mechanisms in the above approaches.

The notion of designing a program that accesses shared objects by building it out of pieces that execute as if they are the sole agents of change to shared objects has its roots in the concept of a database transaction. The essence of the database transaction concept is described quite well by Eswaran, *et al.*[Eswaran76].

Two nice overviews of the field of designing reliable systems are Gray[Gray77] and Randell[Randell78]. They define the basic strategy of backward error recovery used to handle loss of availability within NAMOS.

The implementation mechanism used to support possibilities, the commit record, is closely related to the intentions list of the algorithm used to achieve coordinated reliable updates in Lampson and Sturgis[Lampson76]. Also related is the log mechanism described by Gray[Gray77] for handling the backwards recovery of failed transactions in a central data base system. The two-phase commit protocol described by Gray is essentially the same as the notion of making all changes conditional on the eventual state of a commit record in NAMOS. However, NAMOS supports modular construction of operations and objects, while these other systems do not necessarily do so.

Lampson and Sturgis's approach to recovery also shares our basic assumptions about the properties of memory described in chapter two, categorizing memory into two classes -- stable and volatile.

In essence, the NAMOS system developed here differs from other work in synchronization and reliability because it ties together four related concepts in one framework -- synchronization, reliability, modular construction of programs, and decentralized, distributed systems. Not only does this ensure that the solutions harmonize with one another, but in fact each problem is simpler when solved in conjunction with the others. The simplification occurs because it is hard to separate the four areas of synchronization, reliability, modular programming, and distributed systems cleanly from one another -- one has to think about certain parts of the other areas in dealing with any one area.

1.4 Thesis Plan

The remainder of the thesis is divided into two parts. The first part consists of an explanation of the object level interfaces, and the semantics of operations at this level. The second part discusses the issues of implementing the interface on a distributed system, handling low-level failures of communications and nodes, managing storage, and so forth. The description is done this way to adhere to standard top-down design.

In fact, I think it is interesting to note that the actual thinking process was not top down at all -- I was much more concerned with what was implementable than with what to implement. Especially when dealing with fault-tolerant mechanisms, one has to be careful not to ask for too much from a mechanism -- it may not be achievable. No matter how much one may try to sweep the consequences of failures under the cover of the "system blanket," it keeps burning its way through. Consequently, some notion of the kinds of implementations that are possible shows through in the interface semantics, and I will allude to various notions in the description of the interface.

Chapter two, then, provides some background in the problems of implementation. Failures of communications and nodes are described, and an argument is made that low-level error correction (such as link-by-link error correction, or reliable stream communication) is insufficient to solve the problem of failure recovery. Similarly, some of the interaction between communications technology of the packet network, reliability mechanisms, and proper synchronization will be discussed.

Chapter three begins the discussion of the object interface. Three basic notions, the system history, creation of hypothetical states, and the frozen system states in which computations can be executed (called *pseudo-temporal environments*), are described, and linguistic constructs that reflect these notions are developed. This chapter is long and involved because it presents all of the aspects of the interface, which is quite different in some respects from traditional synchronization mechanisms. Nonetheless, it is essential to the understanding of the rest of the thesis.

Chapter four discusses several ways in which the object interface can be used. A very important case, the creation of multi-node *transactions* (as defined by Eswaran *et al.*[Eswaran76]), will be shown to be easily handled with the object interface. More importantly, the definition of unplanned transactions, and the creation of conversational transactions will be shown to be easily accomplished. Other uses, such as consistent recovery from permanent mistakes or errors (backup), will be described. Finally, some "unstructured" ways to use the mechanisms will be shown, for two reasons. First, I want to show that even with my scheme, all is not perfect, and second, if there is no way for a mechanism to be misused, one should suspect that there are probably some perfectly reasonable uses that it cannot support.

Chapter five begins the discussion of implementation, talking about a mechanism for implementing the hypothetical-ness of hypothetical states of the system. The basic idea is to build at a low-level in the implementation data representations that allow operations to be *recoverable* -- a term used by Gray[Gray77] to mean that there is a single instant during their execution when they "happen". If a failure occurs before this instant, it is as if nothing had happened, and if failure occurs afterwards, the operation is guaranteed to appear to have completed correctly. A

mechanism called a *commit record* that I have developed is described, and various implementations are discussed.

Chapter six continues the discussion of the implementation, talking about how the system state is built of states of individual objects related through the concept of pseudo-time, and the representation of object histories. Issues that are key to the implementation, such as the necessary synchronization of clocks, management of storage for objects, and the details of representation of object histories needed to ensure correct operation in the face of failure, are discussed. Optimization of performance of the mechanisms, and ways to eliminate deadlock are also discussed.

Finally, chapter seven summarizes the thesis, giving goals and directions for future work. A particular issue, that of the compatibility of the mechanisms of this thesis with specially designed hardware that makes the synchronization of objects on the same system quite inexpensive, is elaborated in some depth.

Chapter Two

The communications system and storage system

In this chapter I want to discuss the interactions of the low-level components of the distributed system with synchronization and failure management. The important components are the long-term memories of each node and the message passing network that connects the nodes, allowing them to call upon each other for services and access to data. In a sense, the characteristics of the components constitute my assumptions about where the technology is and where it is likely to go.

The major problems with each component consist of reliability problems and synchronization problems. In turn, reliability breaks down into availability -- whether the component is available to have data placed into it and taken out of it -- and integrity -- whether the data entrusted to the component is damaged or not. Synchronization problems involve the ability to use the basic properties of both the message system and the storage system to control the order of actions taken by computations in the system. Since multiple computations will be proceeding in parallel, with only loose coupling between the computations at best, the order of actions taken in the system is relatively unconstrained. As shall be discussed, the message system and the storages system each add the opportunity for more unconstrained ordering of actions.

Both the message communications system and the data storage system are quite similar in function. In each system, one places data into the system with some tagging of the intended destination, and then later the data is taken out, selecting the data by means of its destination tag. The differences between the two are basically either technological or in their intended use. Typically the data sent in a message is intended to be transient, used only once or not at all, and in any case, used fairly promptly. In contrast, the data stored in a data record is to be saved for many potential later uses, that can be separated by quite a long time from the initial transmission.

It is, in fact, the case that a communications system can be built on what seems to be a "memory" technology; for example, networks have been built by connecting multiple processors to a shared bulk memory, such that messages to the other processors are stored in queues polled by the other processors. The distinction between such a system and a shared memory multiprocessor system is slight. The construction of a memory system using communications hardware is conceivable, but seems not to be a viable way to go (although once upon a time, the cost per bit of delay lines was relatively quite cheap). Thus we cannot make the simple argument that communications and memory are basically different, and that we must therefore distinguish the two.

I do, however, assume that there are two components, the communications system and the storage system. The communications system is an abstraction designed to capture the notion of data transport. The storage system is an abstraction designed to capture the notion of long term memory of information. These abstractions can be thought of as extreme points on a continuum that contains all real storage and communications systems.

It is useful to distinguish the two components, given their basic similarity, for two reasons. First, the autonomy property of the distributed system argues against treating the shared network as a long-term repository of shared information. Because the network is shared, it should do as little as possible for its users in order to reduce user interdependence. Second, in the message transmission mechanism, the tradeoff between reliability, cost and delay becomes very important because of the large physical distances involved, whereas in a local node, reliability can be achieved with relatively little cost and delay. Consequently, the reliability strategies for data storage systems generally achieve a high degree of reliability in the transmission of information from source to user, while a significantly lesser degree is generally provided by message communication systems.

Since in a distributed system, messages are used to request remote actions, the properties of the message system both in terms of reliability and synchronization have a serious effect on the ability to create actions that are composed of several subactions initiated at several nodes by messages. Taking no particular care to ensure reliability and synchronization of the delivery of messages, the behavior of such an action (its semantics) in terms of its effect at the multiple receiving nodes, and its interactions with other such actions that may be initiated concurrently, is extremely complex to describe.

To reduce the complexity of describing the behavior of such actions, a method based on numbering messages can be used to transform the problems of lack of integrity, variable delay, and duplication into a common problem, lost messages. The problem of coping with the unusual behavior of the message system is thus reduced to coping with the problem of coping with lost messages.

2.1 Reliability of Message Communications

Achieving integrity of the messages sent through the communications network is not usually a difficult problem. One can get quite a large amount of integrity by associating a checksum of an appropriate size with a message, checking upon receipt of all messages that the checksum correctly matches the data in the message. I am assuming that errors within messages are random. The result of the use of checksums is the transformation of all message content errors into lost message errors (thus transforming a question of the integrity of the data into a question of the instantaneous availability of the communications path between source and destination). By the use of encryption of messages, one can also treat attempts to modify messages in transit as random corruption of data in the unenciphered form of the messages[Kent76].

Messages are used either to communicate information to, or to cause actions by, a computation at some other node. In essence, then, it is unimportant where unavailability or lack of integrity occurs -- the important thing is that the system as a whole provide reliability from the source computation to the destination computation's use of the message. Any guarantee of reliability of the message system alone cannot ensure the reliable functioning of the system as a whole, unless we make the rather unreasonable assumption that the only unreliable component of the distributed system is the message transport mechanism. The reliability of the message system itself is much less important than the function the message system provides for coordinating responses to failures both inside and outside the message system.

To detect failure of a requested action, the standard mechanism is positive acknowledgment, i.e. when the action is performed, a message is used to inform the requester that the action has been performed. Of course, the need to wait for a positive response can lead to some rather serious problems. The basic problem lies in the knowledge that the requester has of the state of his action after a requesting message has been sent. If the requester receives a proper response, then it is sure that the action has been performed. However, if it has received no response, then the requester only knows that the request may not have been processed, not that it has not been wholly or partially processed. Achieving reliable control of remote actions requires some tricky design of the remote actions, so that a request may be repeated if no response has been gotten in an appropriate time, without causing errors due to running the request more than once (such requests are *idempotent*). Handling repeated requests will be discussed shortly.

As a basic assumption, I conjecture that the problem of unexpected loss of availability can be characterized by a request uncertainty principle, stated as follows:

> *Once a remote action has been requested, the requester cannot always determine, in a bounded time, whether or not it has occurred.*

A program that requests remote actions must thus always be prepared to somehow handle the case that it has initiated a remote operation, but cannot determine the status of its request. In non-distributed systems, this case is usually so rare that it is not explicitly considered in the design of software.

Unfortunately, if the requesting node fails, or chooses to give up after a while, it may be the case that it still does not know whether the request has been processed. It is important that the system give the requester the option of giving up without causing the possibly partially completed action to leave the system in an irrecoverable state. The option to give up on a request that has not yet been completed adds no difficulties that are not already present due to the possibility of a failure of the requester, and adds to the autonomy of the requesting node.

It is important to note that a certain part of the unreliability of the message system cannot be reduced by using more reliable components. The portion I refer to is that caused by autonomy. The likelihood that a node owner will disconnect or shut off his machine is independent of the innate hardware reliability. Also, it is often not economically feasible to provide complete reliability of the message system, especially where long-distance communication, with hazards of natural disasters, wars, etc., is involved in the system. For this reason, the unreliability of the message system must be taken for granted, and reflected in the application programming interface.

2.2 Synchronization of Message Communications

The primary problems with message communications from the point of view of synchronization of remote actions are duplication and delay. In most communications networks both of these problems arise normally, as a result of the internal structure of the networks. Even were the network design specialized to prevent duplication and varying delay on messages, however, protocols that attempt to ensure the reliability of message communications will introduce these factors anyway.

Duplication and loss of messages can be characterized quite simply. For every message sent in the system, that message will arrive at its intended receiver any number of times, from none on up. Delay can also be simply characterized for the purposes of the thesis -- the individual arrivals of the copies of a message may be at any times later than the sending of the message.

Duplication is a problem in the use of communications systems because messages are usually used to cause actions at the receiver. Depending on the kind of action, the repeated performance of the action requested by a message may be an error -- for example, a message that requests the receiver to subtract one from some integer cell will, if no attempt is made to prevent repeated execution due to duplicated messages, cause the cell to be decremented some number of times. One way to avoid problems resulting from duplication is to remember all messages ever received at the receiver, assuming that they are distinguishable. If the receivers of the system all ignore duplicated messages based on this information, then the behavior of the message system is simplified to the statement that for every message sent, it is received at the intended receiver either once or never.

Remembering all messages received is a quite expensive strategy in terms of the amount of memory needed at a node and the amount of time needed to verify that a message is not duplicated. Another strategy that does not require unbounded memory is to assign an identification number to all messages sent, where each receiver stores the largest number attached to any received message, and ignores any message that is received whose number is less than the number currently stored at the receiver. In this strategy, all duplicates are thrown away, but also non-duplicated messages that have a number less than the receiver number may be thrown away

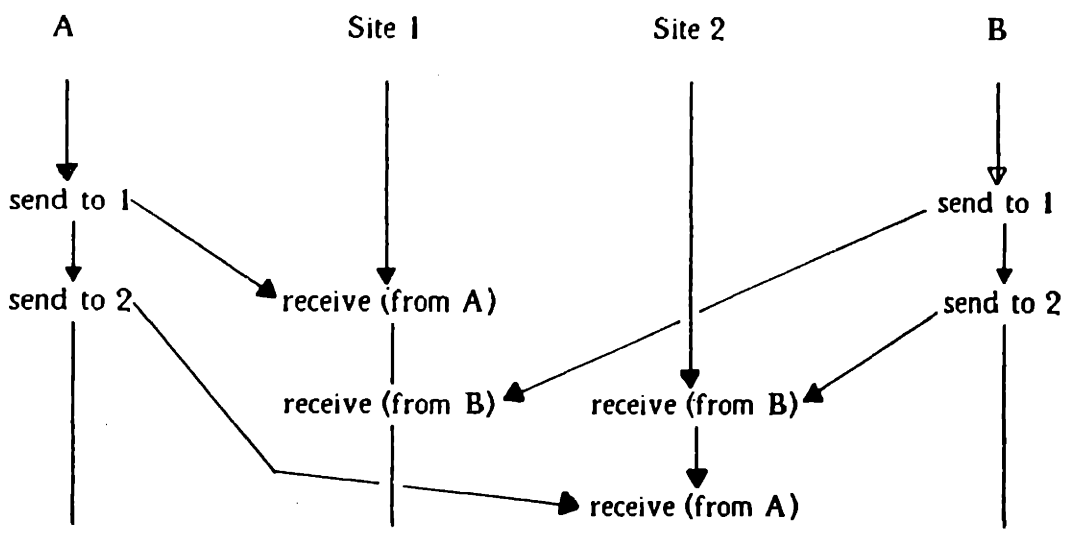
(some lost messages can never be resent in this strategy, since they may have identification numbers too low. A more expensive variant of the scheme is to have each receiver remember the highest message received from each source, so that the source can always retransmit the last message sent). To minimize the number of messages thrown away erroneously, the message identification numbers must be chosen so that messages are numbered in an order that ascends as the arrival time of the first copy. In most networks, the arrival time of the first copy is correlated strongly with time of sending, so by using the clock time of sending as the identification number, the number of messages falsely rejected as potential duplicates can be reduced.

Note well, however, that it is sufficient to number the messages *arbitrarily* to achieve duplicate rejection -- the use of clocks, and mechanisms that ensure that two different messages get different identification numbers, are only ways to ameliorate the false rejection problem.

I have enumerated these strategies for avoiding duplicate messages here because they form a basis for the mechanisms that handle duplication of requests in the system to be described in the rest of the thesis. An alternative approach to the one I have taken would be to eliminate duplicates in a low-level communications protocol, then build the system assuming that message duplications never happen. I have not taken this approach for two reasons. First, eliminating message duplications at a low level cannot help with the problem of requests duplicated as a result of retransmission in an attempt to handle a request whose status is uncertain. In the system to be described, duplicate requests are rejected in any case by mechanisms analogous to the mechanisms used for duplicate message rejection. Since the only objection to a duplicate message is that it may lead to a duplicate request, duplicate messages will be handled by the higher level.

Varying delay of messages can lead to another sort of synchronization problem. Messages can arrive in an order quite different from the order in which they are sent. The simplest example is a computation that sends two messages to the same receiver. If the first is delayed more than the second, then it may arrive after the second message has been received and processed. However, this example is rather tame compared with the one in figure 2 where each

Fig. 2. Reordering of messages by the message system



of two computations send messages to each of two receivers. At one receiver, the message sent by computation A arrives first. At the other, the message sent by computation B arrives first. This possible order of arrival can happen no matter what order each computation chooses to send the messages in. The result of this reordering of messages is that it is not at all simple to understand what the result of a set of actions requested by messages to remote sites will be. In the case shown in the figure, there are four possible outcomes (assuming that the requests have effects only

at their destination site, so that the relative ordering of a pair of requests destined for different sites can be ignored) -- 1) both of A's requests will be processed before both of B's, 2) both of B's will precede A's, 3) A will precede B on site 1, but not site 2, or 4) B will precede A on site 1, but not site 2. Given n computations, each sending messages to m sites, the number of possible arrival orderings is $(n!)^m$. Such a large number (if n=m=5, there are 25×10^9 orderings) of possible interactions among computations can be very hard to comprehend when writing a program that requests remote operations. Certainly some strategy is needed to make sure that under all possible arrival orderings, the proper result is achieved.

Fortunately, there are ways to overcome the complexity resulting from message reordering. The solution used in the thesis is based again on numbering messages, and accepting at a receiver only messages that have a larger number than the ones already received. If all messages intended for a particular receiver are guaranteed to have distinct numbers, then the possible orders in which messages can be received at a receiver are limited to subsequences of the sequence defined by ordering all messages according to their message number. Messages rejected at the receiver due to a too low message number are indistinguishable, from the sender's point of view, from lost messages.

Correlation between the order of message arrivals at several sites can be achieved with the same numbering mechanism. If in the example above, the two messages sent by A have the same number, and the two messages sent by B have the same number (without loss of generality, greater than the number used by A), then the possible orders of arrival of messages can be thought of as having A arrive before B at both receivers, and the subsequences that can result from loss of individual messages in that ordering.

As in the similar scheme that allows detection of lost messages by numbering messages, the choice of numbers may be arbitrary, subject to the restriction that different messages intended for the same receiver have different numbers. However, a completely arbitrary choice of message numbers can exact a heavy penalty -- many rejections of otherwise acceptable messages. In the example, if A's messages are generated and processed long before B even attempts to send his messages, yet B uses a number less than the one A used, then B will fail. By choosing message numbers so that they are chosen in an order that ascends in time, then the likelihood of such unnecessary failures will be reduced.

Here we lose a useful property if we "improve" the scheme so that each receiver remembers the highest message number received from each sender, and rejects those messages that arrive out of the order sent by it sender. The "improved" scheme cannot ensure a correlation among the messages sent by two different senders to two different receivers. Thus, if the two messages sent by A have the same number, and the two by B also have the same number, all orders of arrival are still possible, in contrast with the two choices (A first at both sites, or B first at both sites) achieved with the mechanism using a single highest number at each receiver.

The method of numbering request messages and accepting messages only in increasing order at a receiver is the basis for synchronization of remote actions in the system developed in this thesis. However, as pointed out above, using the method at the message level without knowledge of the requirements of the higher level is probably not as good as using the method at the request generation and processing level to organize synchronization. At the level of the system concerned with the actual semantics of the requests, the grouping of requests sent out with the same message number can be chosen to have exactly the right effect. Without semantic

knowledge, the best the message level can guarantee is that messages sent later will be processed later or not at all. A particular advantage of handling delay and duplication at the request level is that out of order, duplicated, and delayed messages do not always cause problems, depending on the semantics of the actions and the objects they act upon. For example, if one asks for the balances of two accounts at some database representing a branch bank, it makes no difference if the responses are processed in an order different than the order of the requests. Similarly, if one deposits two checks to one's account, it is, in the long run at least, irrelevant in which order the checks clear. The mechanism described in the thesis can often tolerate messages that arrive quite out of order. Duplicated messages that cause no "side-effects" at the receiver (such as pure queries) are always quite acceptable, and reordering such messages may often be acceptable. Requiring that such requests be processed in the order they are issued may cause a significant delay that is often unnecessary.

2.3 Reliability of Storage System

As noted earlier, the integrity of storage systems can be made quite high, at reasonably low cost, by using error correcting/detecting codes. Further, the availability of information stored on disk or other large scale secondary memory is usually as good or better than the availability of the node to perform computations. Failures of information to be available are usually transient (a disk pack off-line), and only very rarely will a node's storage system (taken as a whole, including whatever local backup mechanism keeps extra copies of the state on tape) lose information stored in it. Consequently, I will generally assume that a node never loses information once it is properly stored on disk. This assumption is not absolutely required -- it is possible to correct for loss of information by "turning back the clock" and repeating the actions needed to create the

information. However, the mechanism developed in the thesis cannot automatically correct for such loss of information, since once the information is lost, there is no way to regenerate it except by going outside the system.

If the basic storage mechanism is not reliable enough, replication of information to create redundant copies for the purpose of ensuring availability can be used. Two possible kinds of replication are possible, either multiple copies within a node, or multiple copies at several nodes. In the thesis, we assume that replication within a node is the primary means for achieving availability. However, in chapter five, a strategy for increasing the availability of a critical class of system objects, *possibilities*, by multi-node replication is described. In chapter six, a mechanism for encaching versions of objects to increase availability and decrease delay is also described.

Systems have both long-term and short-term storage. It seems to be the case in the real world, though it is not clear what the base cause is, that the more rapid accesses (stores and updates) are only possible from storage that tends to lose information upon failure -- core memory is more prone to failure than disk or tape.[1] Thus the storage used to hold the frequently accessed transient states of computations must be of the more volatile sort. Following the approach suggested by Lampson and Sturgis[Lampson76] we capture this idea by considering two kinds of storage in the nodes, stable storage and volatile storage. Stable storage is the kind of storage used to hold objects for a long time (across system crashes), while volatile storage is

1. Core memory is non-volatile, but it is randomly addressable. If a failure occurs in the addressing mechanism, it can destroy any part of the core memory. Tape and disk on the other hand, are not so randomly addressable, and have the property that only the portion of the tape or disk currently accessible can be damaged on a failure.

used to hold intermediate values created as part of computations. Volatile storage will be thought of as belonging to a computation that uses it, such that failure of the computation or the node running the computation will cause the volatile storage to detectably lose its values.

The best definition of volatile and stable storage is in terms of their interaction with failure. Once a stable storage record has been successfully written, succeeding reads are guaranteed to return the value stored. Upon a failure before the completion of an update is signalled, the updated storage location (record) contains either the old value, the new value, or an unambiguous indication that it is inconsistent. If an update signals its completion, the stable storage location is guaranteed to contain the new value. Our definition of stable storage is due to Lampson and Sturgis[Lampson76]. In contrast, once a volatile storage location has been written, it may lose its value (detectably) at any time.

2.4 Synchronization of Storage Systems

There are two basic problems of synchronization in the storage systems. First, there is the problem of making sure that the representation of data on stable storage correctly represents the state of the computations that are making changes to the storage. Second, there is the problem of making multiple changes to storage consistently, without other computations at the node being able to interfere by modifying data during the set of changes.

The problem of ensuring that the representation of data on stable storage is correct arises because of the common use of virtual memory systems to make secondary storage look like primary memory and because of sophisticated disk queueing algorithms. Basically, in many systems, a write to secondary storage may not occur immediately when it is logically requested.

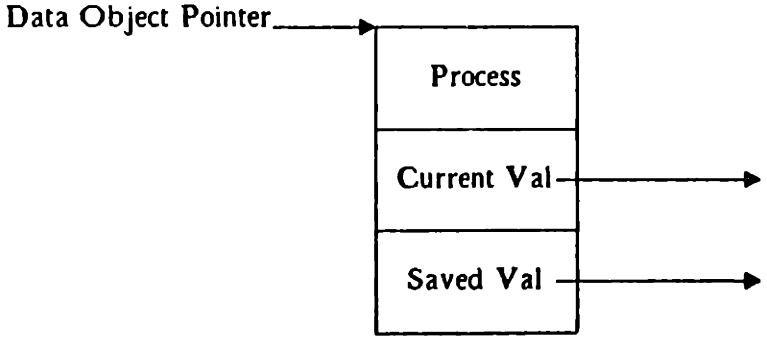
Given two successive writes to secondary storage, one at time t_1 and one at time t_2 ($t_1<t_2$), it is possible that the one specified at t_2 will happen first, and if a failure occurs, may be the only one to happen. In the virtual memory case, the reordering of writes arises because writes directly change the primary memory (volatile) copy, only later modifying the secondary storage copy. Thus, if the modified pages of primary memory are written out in a different order than that in which they were written originally, the changes to secondary storage will be made in different order. Note that this is only a problem when the system encounters failure, since accesses to objects belonging in stable storage always go through primary memory. In the case of optimizing disk queueing algorithms that reorder the write queue in order to minimize seek time on movable head disks, the same problem can occur because of the reordering. The solution in either case is to provide a mechanism within the system whereby one can ensure that a particular modification to a stable storage object has been completed to the point that the copy on stable storage has been modified. In the virtual memory case, a call on the operating system to "synchronize" secondary storage can be provided, whose semantics is not to return until the secondary storage copy is identical to the primary memory copy, using a forced disk write if necessary. In the case of optimizing queueing, a call on the operating system to wait until a queued write is completed is one way to provide the desired control. I will assume that such a mechanism is provided at each node, and is used to insure that writes to stable storage used in the algorithms executed at each node are done in the order specified by the algorithm.

The other problem of synchronization within a node is coordination between several computations that attempt to modify more than one data record on that node. This is the local node version of the general synchronization problem attacked in the thesis. I am going to assume that the solution provided in the rest of the thesis is used inside the local node for the general

synchronization case. However, there will be occasions in the construction of an implementation where the need to synchronize action on a single data structure composed of multiple records will arise. This is a simpler case, because there is no need to be prepared for synchronization constraints that may include any set of data objects on the local node. All that the nodes need to provide is a mechanism for creating atomic operations on individual data structures.

To this end, a node is expected to provide a very simple form of locking. Associated with a data structure of the sort I am concerned with, there will be a lock. Setting that lock prevents other computations from either reading or writing the data structure. Only one lock may be set at a time by a computation, so that there is no problem with deadlocks. Now the problem that must be solved is the interaction of failure with the locks. If the computation that has the lock set terminates without clearing the lock, then the data object must be presumed to have been only partially modified. What must happen is that the data structure being modified must be restored to its original state before proceeding, thus having the effect that the local computation that set the lock appears to have never run. A very simple mechanism that can be used to implement these lockable data structures is shown in figure 3.

Fig. 3. Lockable Data Structure



The new value constructed by a process that modifies the data structure is constructed by copying enough of the original value to avoid actually modifying the saved value. Since the saved value is not modified, resetting the data structure to the saved value upon salvaging correctly causes the data structure to appear as if it never was touched by the failed process.

The header contains three parts. The Process field is either null, indicating that the data structure is not locked, or set to name a process that has the data structure locked. If the Process field is non null, then if the named process exists, only that process can read or modify the data structure. If the process named does not exist (since it has failed), then the data structure must be salvaged. The current val field is a pointer to the current value obtained by the process that locks the data structure. The saved val field is set to be equal to the current val field whenever the process field is set to a non-null value. The salvaging that occurs when a process attempts to lock the data structure and discovers that the process field is non-null and names a failed process consists of copying the saved val field into the current val field.

An alternative mechanism would be to have the process copy the value to be modified into another storage area, modify it as needed, and then when done, switch a pointer to the modified version in an atomic operation. It is still necessary in this scheme to ensure that other processes updating the object do not simultaneously make their own copies and manipulate them, so a way of indicating an update in progress and a salvaging mechanism of some sort, or a way of preventing all but one such update are still needed in this mechanism.

The purpose of the locks just described is to provide minimal locking needed to ensure correct synchronization of actions at a node. It is intended therefore that the locks be set for as short a time as possible. A restriction to help ensure this is that no lock can remain set if the process waits for some external event, such as sending a message or receiving one. If a process does attempt to wait while it has a lock set, it will be terminated, thus effectively freeing the lock after the required salvage is performed.

2.5 Remotely requested actions

As noted earlier, actions are remotely requested by means of messages. A message that requests an action causes a computation to be started that performs the indicated request. The state of this computation is kept in volatile storage, and it can request one lock at a time as noted above. Due to duplicate messages, multiple instances of a requested action may run simultaneously. Actions may also run in an order different from the order in which the actions are requested. Further, the computations running actions may fail at any time.

Given actions that have such complex behavior, it is fairly hard to construct a working multi-node program. Further, defining interfaces to such programs that can be used to interconnect multiple programs is even harder. The remainder of the thesis is concerned primarily with the development of abstractions at the next higher level that simplify the task of constructing such programs and interfaces. The intent, to be realized in chapters five and six, is to define a set of abstractions that are realizable in a relatively straightforward way in terms of the behaviors of remotely requested actions that result from the complexity of the message and storage systems.

Chapter Three

Pseudo-time and Possibilities

In this chapter and the next, I describe a semantic interface for shared objects and the operations on them. The emphasis in these chapters is on the behavior of the interfaces, rather than on their implementation. For this reason, I will take an abstract view of the nodes, objects and computations in the system, such that the view represents fairly accurately what the "applications programmer" might imagine the system to look like. Briefly let me outline what will happen.

The key ideas of the thesis are presented in this chapter. Two key abstractions, *pseudo-time*, a way of relating and synchronizing the order of actions at multiple nodes, and *possibilities*, groupings of actions that must be done all at once or not at all, provide the tools by which a programmer can manage the effects described in the previous chapter. *Objects* that have indefinite scope and extent provide the mechanism for information sharing. Basic concurrency control, defining the interactions between independently executing computations accessing the same objects, is done locally at each object, resulting in defining the semantics of an individual object in terms of an *object history*. To achieve more global control of the interactions between computations, particularly exemplified by the notion of a *transaction*, pseudo-time is used to relate individual object histories to obtain a *system history*. The system history defines the notion of a consistent state of the system. Programs can then ensure that they observe consistent inputs and generate consistent outputs by the use of a *pseudo-temporal environment*, a dynamic naming

environment that functions to "stop the action" seen by the program, even though the system is not necessarily stopped.

As noted in the previous chapter, a key problem in distributed decentralized systems is dealing with the uncertainty resulting from an attempt to perform a remote action. The basic approach developed here to solve this problem is to make the changes involved in a remotely requested action in two steps. The first step tentatively performs the computation, such that no other independent computation can observe it. The changes made to each object are grouped into a set called a *possibility*. If the first step goes to completion without a hitch, the computation is confirmed by an act that converts the possibility into a "reality." If not, the possibility safely times out, eliminating the tentative changes from the system.

Creation of composite actions out of independently designed actions whose implementation details may be inaccessible to the programmer requires discipline in synchronizing the component actions and in managing failures, both on the part of the programmer doing the composition and on the part of the programmers who designed the actions being combined. By using the same pseudo-temporal environment and possibility with all the actions being combined, easy combination of parallel actions can be achieved. To allow the construction of such composite actions in such a way that they are not dependent upon the correct behavior of their user, the possibility concept is extended to include the notion of a ***dependent possibility***.

3.1 Objects

Objects are the means by which information is shared within the system between independently executing computations. An object is a named repository for information. Objects possess a state that can be observed and modified by the execution of programs that refer to the object as an operand. Objects are abstract, in the sense that while an individual object may in fact be represented in terms of a group of lower level objects, the representation is invisible to its users. The essence of objecthood is possession of a name.

In this thesis, I am concerned with objects used to store information for a long time (e.g., longer than the time between system crashes, or longer than the life of an individual console session) and that may be accessed by an unknown set of users (programs, people). The objects may be catalogued in a file system, such that the names used by the user programs will be character data, rather than pointers. The scope and extent of such names are not even deducible from within the system. Even if the objects are named only by unique identifiers (pointers), due to the decentralization of the system there may be no way to determine what references to an object exist. In a centralized system, by tagging unique identifiers stored within objects, mechanisms for finding all references to an object are possible (e.g. to garbage collect objects no longer referred to). In a system whose design was centralized (top-down by one person), all potential interactions are constrained at design time. Neither of these ways to limit interactions between computations will always be possible in a decentralized system whose design evolves in a decentralized way. The lack of constraints on scope and extent make the job of proper synchronization difficult, because the set of independent, asynchronous computations that could interact with an object may not be easily discovered.

The semantics of objects are defined locally to their implementation. In the abstract, the state of an object reflects the history of *operations* applied to it. The implementation of the operations in terms of the underlying representation of the object's state is solely managed by a program called a type manager that is common to all of the objects of a particular type. Hiding the implementation of objects within its type manager allows the implementation of objects to be modified and controlled locally, without the need for users of the objects to be aware of changes. CLU[Liskov78] and ALPHARD[Wulf74] provide this kind of implementation hiding, as do ACTORS[Hewitt76]. ACTORS in addition provide local control of synchronization between independently initiated operations on the same object[Hewitt77]. In this thesis, the basic synchronization of independent computations also is provided local to the type managers of individual objects. However, the abstractions defined in this chapter allow construction of synchronization behaviors that involve multiple objects, whereas Actors can handle multiple object constraints only by the construction of intermediate objects to mediate accesses to objects requiring synchronization[Atkinson78].

In the rest of the thesis, CLU syntax[Liskov78] is used to represent programs and objects. The objects of this thesis, however, are different from CLU objects in an important sense. They are shared by multiple users and are used to store information for a long time. NAMOS also allows arbitrary parallelism, while the current definition and implementation of CLU does not. CLU is used because the standard languages, such as Algol 60, PL/I, FORTRAN, etc. do not provide facilities for the construction of abstract types and operations, but we are very interested in exploring the interactions between abstraction mechanisms used to build modular systems and parallel execution and long-term storage of objects. Because the CLU type abstraction mechanism is used to describe types of objects that are stored for

long-term, shared use, we will have no syntax for describing the short-term, unshared types that CLU's object abstraction mechanism provides. This lack of syntax is not troublesome for the purposes of the thesis because we shall have little need to refer to such types in examples, but a real system should provide both mechanisms.

Message sending is implicit in the programs we use in examples. Executing some part of a program may require the sending of a message from the site currently executing the program to another node -- the mechanism by which this message is generated is not important. The programs themselves are written as sequences of steps with the exception being the two constructs, **either** and **all**, which execute a group of statements with no predefined order of execution, in parallel. The **all** construct is equivalent to a **parbegin** block, in that all statements must finish for the execution of the block to finish. The **either** construct finishes whenever at least one statement of the **either** finishes. The **all** construct is primarily useful for expressing that a group of statements need not be evaluated in any particular order, while the **either** construct is useful for expressing computations where not all of the branches are required to finish (timeouts are particularly important examples of **either**). Sequencing among expressions to be evaluated on different nodes may be implemented by a request and positive acknowledgment mechanism, such that calling a procedure on a different node involves sending the request and then waiting for a response signifying completion and providing results, if any, at the calling node. If an error occurs, interfering with either the request, the invoked procedure, or the response, then no response will get to the calling node. Such an error can be detected by a timeout.

When concurrent execution is introduced into a programming language (such as CLU), there are two possible ways of thinking about procedures (or operations of a data type). Since procedures in a von Neumann architecture are executed as a sequence of steps, we could view procedures as executing over a period of time, and unless otherwise prevented, all of the primitive operations ultimately composing the execution sequences of two concurrent procedure executions could execute in any arbitrary order. In this thesis, an alternative view is taken, viewing procedures as abstract "atomic" operations, whose internal steps are not interleaved with the execution of any procedure. The implementation of such procedures is one of the primary tasks of NAMOS. Either the traditional view of procedures or the alternative view taken by NAMOS degenerates into the same semantics, given a single execution point. What is missing in the NAMOS view is the ability to construct "sequencing control abstractions", such as a procedure that at intervals of 10 seconds increments its (by reference) parameter. Our thesis is that such sequencing control abstractions are best handled by a separate linguistic construct (perhaps called a subroutine?), leaving the procedure construct to build abstract operations.

Manipulation of objects is done by operations, requested by transmitting a message to a node capable of executing the type manager to which the operation belongs. A natural way to build objects that captures the physical decentralization of the nodes is to define the idea of an object *home*. The object home is a node that contains the type manager for the particular type and the mapping from the object's name to the component objects that make up its state. Implementing an object on a single node ensures that all operations that deal with the object go to a single place to do so, so that one can queue the requests in some order. There are, however, other implementations of objects that do not require that all accesses to an object go through a common system.

First of all, if an operation on an object does not change its state, but merely returns some function of its state, there is no inherent need to go to the home if a valid copy of the state of the object is obtainable at another node. Thus, copies of the object can be distributed by a kind of read-only encachement of the state of the object. The home is then needed only to make sure that operations that change the state of the object all go through the same place. There must be a means by which the consistency of an encached copy with the state of the object at its home can be guaranteed, however.

Second, there is no absolute need for a fixed home location. Consider, for example, an object whose purpose is to behave like an integer set, for which there are only two operations -- append, which adds an integer to the set, and member, which tests to see if an element is in the set. The append operation need only send a message to any one of several sites that maintain subsets of the total set object, whose union represents the entire set. The member object sends messages to all of the sites containing subsets of the set, requesting each node to test to see if the integer is a member. If any node responds yes, then the member operation returns true, while no responses from all nodes indicates that the member operation should return false. Other distributed implementations of objects exist. Johnson and Thomas[Johnson75] have suggested a strategy by which cells that contain scalar values can be implemented in a distributed fashion. In chapter five, a distributed implementation of a special kind of "object" called a commit record, used to implement the possibility abstraction, is described.

3.2 Object Histories

Common to the implementation of objects with fixed homes and the various ways of distributing implementations of objects is a fundamental requirement. Since an object has state, the results of one operation depend on the parameters to some set of other operations that are applied to the object. Normally, this dependency is derived from the time-ordering of operations applied to the object. In general, the effect of an operation may depend on all operations executed before the operation finishes execution. The fundamental requirement is that the implementation must execute the operations applied to the object in some order. With an implementation that defines a home node to be where all changes are made, ensuring that operations are applied in an order is trivial. However, consider an implementation of an integer cell that has two copies, each of which must be multiplied or added to independently. Then if site A tries to multiply the cell by 2, while site B attempts to add 4 to the cell, the messages to the two copies may arrive in opposite orders, giving different results. In this case, operations were applied to the cell object in no order (of course, the actions applied to the copies were ordered, but that doesn't necessarily lead to an ordering of the operations as a whole). If one were to try to read the value of the cell, one might get either answer, depending on the node that the read is attempted at.

Thus a major function of any implementation of objects is to assign an ordering to the operations applied to each object. Perhaps the simplest way to assign an ordering is to use the order of arrival of requests at a home node as the order in which requests are processed on the object. As noted in the previous chapter, the arrival ordering of messages may be very hard to control in order to achieve any form of synchronization among objects with different homes. The

possibility of unpredictable delay leads to lack of control of when requests will be executed, while the request uncertainty principle leads to the inability to stop a request once it has been requested.

Traditional approaches to synchronization of operations on objects attempt to control the arrival ordering in order to achieve synchronization. The only control that is usually possible is to delay computations originating requests, so that there is only one request outstanding to an object at any one time, and only after that request is known to have been processed can another request be originated. Use of locks or semaphores provide the mechanism for deciding when a computation should be delayed because a request is already outstanding. This approach is quite indirect -- in order to perform an atomic action on a group of variables, one must ensure that any other computation that might want to access those variables is stopped.

It is not necessary that the arrival ordering be the ordering used to process operations on an object. A strategy that gives the originator of requests more direct control of the order in which operations are performed would simplify the task of constructing programs that need to synchronize operations. To develop such a strategy, it is necessary to re-examine the concept of updating an object that has state.

Traditionally, updates have been thought of as modifying objects. Another possibility, almost unexplored, is to think of an object as a sequence of the states it has assumed and will assume as the result of all updates. The updates, then, simply create a new element of the group, but do not involve any notion of modifying any one of the states. Similarly, reading an object specifies some element of the group.

With this transformation of the view of objects, synchronization mechanisms become ways to bind the references that occur at different times in a computation to particular elements of the group of states of an object. Thus the synchronization problem is transformed into a naming problem. This viewpoint (which I find extremely fruitful) has led rather directly to the approach for synchronization in distributed systems presented here, and seems to be a rather nice way to think about synchronization in the design of programs.

To capture the notion of synchronization as naming, some concepts must be developed. An object can be thought of as a sequence of *versions*, the states that the object has assumed and will assume as the result of the computations that are applied to it. We will call this sequence the *object history*. It is a completely static picture of the object's behavior. Since the entire object history is not known until the system has finished execution, this is clearly a logical concept, rather than something that has a counterpart in the real system. Nonetheless, for the next few pages, we will suspend our practical judgement somewhat, and imagine that a program does execute in an environment where all the histories of all shared objects are known.

Accessing an object requires a complete specification of which version is to be accessed (either read or created). The way to think about this is that objects have two part names consisting of an identification of the object itself combined with information that uniquely selects a version in the object history. There are two issues that must be considered in designing these two part names. First, how are they used in programs? The ways that programs can generate and use names influences very strongly the utility of this approach to synchronization. Second, what is the structure of the mapping from version selectors to versions within an object history? As we shall see, by carefully choosing this structure we can make it easy to define a notion of

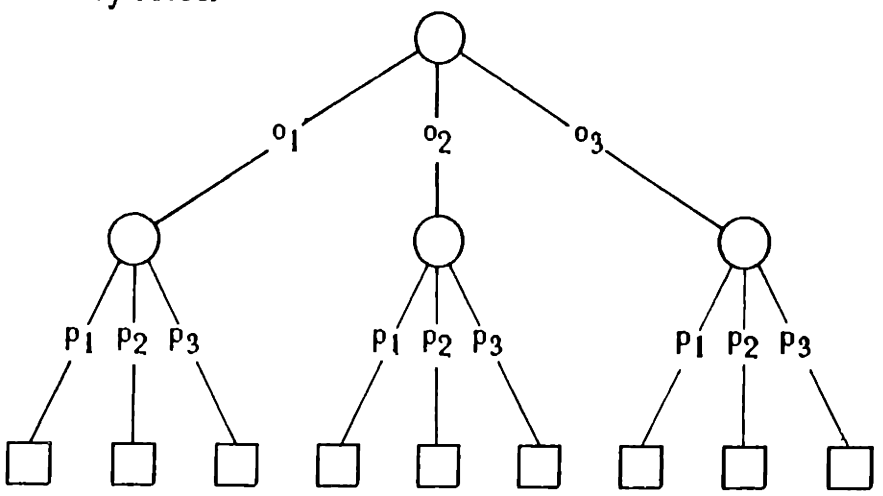
system global state that is not defined in terms of a snapshot of the distributed system at an instant of time.

Since the entire history of the object is known, the idea of updating an object in this static view is somewhat strange. Each version of an object is the result of an update. Executing an update may be best thought of as checking to make sure that the update was preordained to occur by the "god" that set up the system history. Eventually we will modify our view so that the static history is only partially known, so we can view updates as filling in gaps in the already known part of the static history.

3.3 Kinds of references

The model of the memory that holds shared objects is a mapping from completely specified names, called *version references*, to versions. A version reference is a kind of value supported by the system. It is a two-part name composed of an *object reference* and a selector called a *pseudo-time* (for reasons to be explained later). The mapping from a version reference **v=(o,p)** to the proper version **V** is done in two stages. **o** is used to select an object history out of the memory. **p** selects a particular version from that object history. An object history is just a function from pseudo-times to versions. Conceptually, then, the shared memory of the system can be thought of as a hierarchy with two layers of selection, as in figure 4. The entire structure of the memory represents the system's history for all time. The middle layer of nodes represent object histories. The leaf nodes represent versions of objects. The labels on the arcs represent object references (o_1, o_2, and o_3) and pseudo-times (p_1, p_2, and p_3).

Fig. 4. Shared Memory Model



Object references, pseudo-times, and version references are all types of data values that can be used by programs. Object references correspond to the pointer or reference data type. Pseudo-times don't have an explicit counterpart in any programming language that I know of, but their function, that of selecting a particular state of an object to act upon, is normally provided by the execution ordering of programs (they are analogous to array selectors, but have a different purpose). A version reference corresponds to a single access through a pointer or reference -- again, not a concept explicitly available to the programmer in existing languages.

Unlike pointers in ordinary languages, object references by themselves cannot be used for access to objects. Instead, a version reference is required to write or read a value in the memory. An object reference can, however, be combined with a pseudo-time to obtain a version reference. As we discuss later in the chapter, it is most convenient if the pseudo-time combined with an object is provided by an implicit mechanism we call a *pseudo-temporal environment*. Here we

describe the operations that manipulate version references and object references using explicitly specified pseudo-times. Converting a program from the implicit use of pseudo-times to the explicit form shown here is done by a process of "desugaring" the syntax, replacing references to objects with code sequences that choose the pseudo-times to be used and then call on the appropriate functions to create version references that refer to the proper versions. The functions about to be defined can thus be thought of as "hidden under the covers" from the programmer's point of view.

The only ways to use object references are as parameters to the following functions. In describing the implementation of these and succeeding functions, parameters are labeled by names indicating their type, so that or, or1, or2, etc. are object references, vr, vr1, vr2, etc. are version references, pt, pt1, pt2, etc. are pseudo-times, boolean is a boolean value, and value is some value of any type (integer, object reference, array of stacks, etc.). The result or results appear to the left of an assignment operation, and any error signals that can be generated appear in a list after the word "signals". The semantics of each operation is described in a few sentences following its parameter specifications.

vr := version_ref$freeze (or, pt)
This is a function that generates a version reference from an object reference and a pseudo-time.

boolean := object_ref$eq (or1, or2)
This is function that tells if two object references refer to the same object (in the sense of LISP's *eq* function or CLU's *equal* function).

We can think of a version reference as referring to a particular value. There are three operations specific to version references. Explaining them is somewhat tricky, because of our static view of memory. Obviously in a real programming language that can be executed, we must make sure that a version actually exists before we can return its value. In this description, however, we take the omniscient viewpoint that the entire system history is laid out before us. Later in this chapter the actual behavior of these operations will become much more clear.

version_ref$define(vr, value) signals(nonexistent_state, redefinition)
This operation creates the version specified by the version reference. It is used in the desugaring of an update operation on an object. The second parameter is a value that will be the value of the version referred to by the version reference, if no error is signalled. The nonexistent_state error indicates that an attempt to assign a version that never will exist (no version of the object exists for that pseudo-time). The redefinition error indicates that there was already a valid version associated with the specified version reference. The version_ref$define operation can be applied at most once to a version reference.

value := version_ref$lookup(vr) signals(nonexistent_state)
This operation gets the version associated with a particular version reference. It is used in the desugaring of an operation that reads the value of an object. The nonexistent_state error indicates that the version reference specifies a state that will never have existed.

or, pt := version_ref$decompose(vr)
This operation is just the inverse of version_ref$freeze.

In terms of the static system history, there are two ways in which we can think about the execution of version_ref$define. Since the system history is determined for all time, we can think of the version_ref$define operation as succeeding only when the version specified is the same as the one selected. Thus a failure of a version_ref$define operation implies that the version specified is already defined to be some other value created at an earlier pseudo-time. Conversely,

each value change associated with an object requires that an execution of version_ref$define must have happened.

Pseudo-times are ordered by a relation ⇒. The formula a⇒b is read "a precedes b". This relation is a total ordering, and in particular a⇒a (the relation is reflexive). We also a complementary relation, →, "strictly precedes", such that (a⇒b)≡¬(b→a). We require that an object must exist (never signal nonexistent state) for all pseudo-times between an initial pseudo-time of creation and some final pseudo-time of deletion. Thus for any object there is a time of creation t_{create}, and a time of deletion t_{delete}. The nonexistent_state error is signalled if and only if the pseudo-time component, t, of the argument to version_ref$define or version_ref$lookup does not satisfy ($t_{create} \Rightarrow t \Rightarrow t_{delete}$). The creation and deletion of objects is carried out by operations that specify the create and delete pseudo-times.

or := object_ref$create(pt)
This operation creates an object reference whose t_{create} is specified by the parameter.

object_ref$delete(or, pt) signals(bad_delete)
This operation deletes the object specified by the first parameter, by setting the deletion pseudo-time to the second parameter. The signal bad_delete indicates the delete operation was not performed because the specified pseudo-time was inconsistent with the history of the object. This could be because the pseudo-time preceded the creation pseudo-time or because a version corresponding to that pseudo-time exists, or because the delete pseudo-time has already been set to another value. As in the version_ref$define operation, only one such operation may ever be applied to the particular object reference.

More than one pseudo-time may refer to the same version. Essentially, if we think about all of the version ref$define operations that correctly terminate when applied to a particular object, they can be totally ordered by the pseudo-time contained in their version reference parameters. Call the set of such pseudo-times U, the *update history* of the object. We can then say that if we have two lookups on the same object at different pseudo-times p_1 and p_2 that give different values, then there must be a $p_3 \epsilon U$ such that $\neg(p_3 \Rightarrow p_1) \wedge (p_3 \Rightarrow p_2)$. In simpler words, versions only change as the result of version_ref$define.

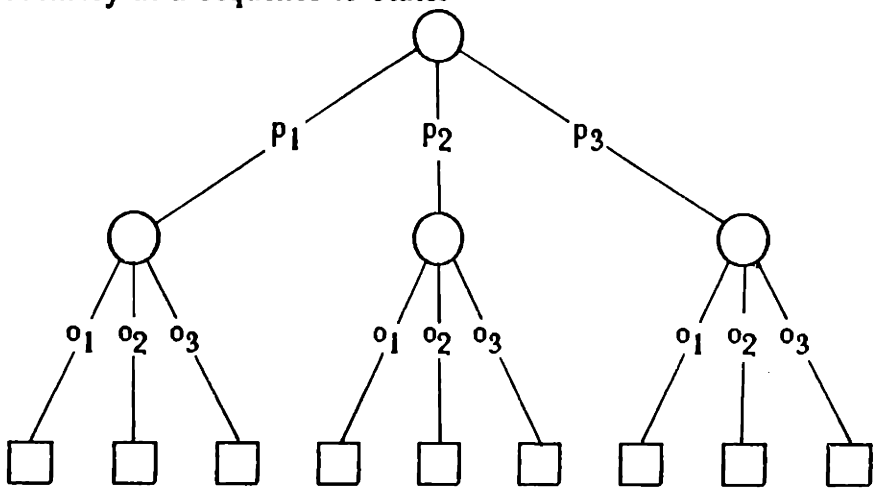
Thus, for all pseudo-times between the creation and deletion of an object, the object has a defined value. The definition of the value at any particular pseudo-time, t, is the result of a version_ref$define operation whose version reference contains a pseudo-time that precedes t.

The concept of describing the state transitions of a system by considering the sequence of states assumed by a variable was introduced by Van Horn in his Ph.D. thesis[VanHorn66] in order to compare the various possible execution sequences of concurrent programs in proofs. NAMOS incorporates the history idea into the set of mechanisms actually used by programs, rather than being reserved for use in proofs about programs.

3.4 Pseudo-time and consistency

Pseudo-time provides a very convenient way to define a consistent system state. We say that a consistent state of the whole system is the set of all object versions referred to by version references containing a particular pseudo-time t. This leads to a different view of our memory model shown in figure 5. Here, the difference is that the first selection is on pseudo-time rather

Fig. 5. Shared Memory as a Sequence of States



than on object. The pseudo-times can thus be thought of as *state references*, selecting consistent system states, from which object references can select the proper version of each object.

Since pseudo-times are objects that are used by programs, they give a tool for programming that allows explicit recognition of consistent states within the program. In contrast, traditional synchronization mechanisms, such as semaphores, locking, monitors, send-receive, etc. do not give a tool for representing or naming consistent states -- one can deduce the states assumed by the system by timing relationships among the execution of steps of programs.

A particular virtue of the model is that it requires that programs always generate version references to refer to objects. Thus programs that refer to shared objects must always be written with synchronization in mind. This has a positive effect, in that it may result in software that is less likely to fail when used in situations where unanticipated concurrency arises. A typical example of such a failure would be the use of text files implemented as segments in Multics.

Such files can be concurrently accessed by several processes, for example by a text editor and a printer being used to print the file. If the user editing does not know of the concurrent use of the file by the printer, it is possible that what gets printed may see the file while it is being modified. In a system based on NAMOS, we could queue a specific version of the file to be printed, so that future versions would not be printed, or alternatively, as we shall see shortly, the editor could arrange that the sequence of changes to the file are not observable by other processes by ensuring that the changes are made in pseudo-times that cannot be referred to outside the editor.

3.5 Programs and Pseudo-time

Since computations must always specify the pseudo-time in which shared objects are accessed, it is important that this specification not add a significant burden to the programmer's understanding of the program. One of the main arguments against the use of semaphores as a synchronization mechanism is that the use of semaphores (or other explicit locking techniques) just add to the complexity of specifying the algorithm the complexity of understanding the relationships between the data being accessed and the semaphores that must be set to prevent inconsistent results, along with the complexity of ensuring that deadlock does not occur and that an adequate degree of parallelism is achieved. We want to make sure that by explicitly including synchronization mechanisms in the language we don't add an enormous burden of complexity in the process.

What locking normally is used for is to insure that during the execution of some steps of a program, a particular object only changes state as the result of the actions specified in those steps. Thus for those steps, the program need not concern itself with interactions with other concurrent programs that refer to the same object. Locking is also used to hide inconsistency that

results during the process of attempting to make some coordinated change to a group of objects. For example, if two objects represent the state of bank balances, a transfer of money requires debiting one account and crediting the other. If these steps are taken separately, then it may be possible that an observer can observe the transient inconsistency that money has been either created or destroyed -- the sum of the balances isn't constant.

Giving the program the ability to make references to several objects with the same pseudo-time is the key to obtaining consistency. Pseudo-time is quite unlike real time in this respect. It is quite impossible for a program to guarantee to be able to access several distinct objects that may be implemented quite remotely from each other at the same real time.

Thus, for example, it would be possible to write the following program that accesses a consistent state of the system, adding up three objects to obtain their value.

```
obj1_vref := value_ref$freeze(obj1_ref,ptime)
obj2_vref := value_ref$freeze(obj2_ref,ptime)
obj3_vref := value_ref$freeze(obj3_ref,ptime)
sum := value_ref$lookup(obj1_vref)+value_ref$lookup(obj2_vref)+value_ref$lookup(obj3_vref)
```

The state to be accessed is specified by the value of the variable ptime. The three references to the three objects are named obj*i*_ref for particular *i*; these are used to generate three value references with corresponding names.

One problem with the example program is that it is rather clumsy to have to specify explicitly which pseudo-time is to be used for each access to an object. It would be much nicer to be able to write something like:

```
sum := obj1 + obj2 + obj3
```

where the default is that obj1, obj2, and obj3 are to be referred to with the same pseudo-time. Convenience thus leads to the development of the idea of a contextual mechanism for specifying the pseudo-times needed to resolve references to objects. The contextual mechanism is a *pseudo-temporal environment.*

The pseudo-temporal environment provides a mechanism whereby a program can ensure that the objects it refers to change only as a result of actions requested as steps in the program. Thus, we can write a rather ordinary sequence of program steps that refer to shared objects, reading and modifying them, and when executed in a particular pseudo-temporal environment, the sequence of steps will have the same effect on the state of the shared objects as they would have on non-shared objects. However, we must be careful not to write any object twice with the same pseudo-time.

It is quite convenient to be able to write programs that include multiple modifications to the same object -- at least because of the need for loops. So, for example, we might want to execute the following in a pseudo-temporal environment and gain the advantage of having exclusive control of the objects during the statements.

```
a0 := dequeue(queue)          % get what is in the queue.
enqueue(queue, a1)            % put a1 in queue
enqueue(queue, a2)            % put a2 in queue
a3 := dequeue(queue)          % take out whatever is in the queue (perhaps a1)
```

It is clear that the intent here is to make four references to the queue object queue. On careful thought, two of the four references are reads followed by writes (dequeue), and two are writes.

To execute this properly, we must refer to five distinct states of the queue object, the initial state and the state resulting from each of the four queue modifications.

If we want to make this sequence of four steps an atomic operation, so that the states in between the initial and final states are invisible outside the operation, we must prevent any other program from interfering with the queue during the execution of the operation. To say that no other program can interfere with the queue during the execution of the program means that the pseudo-temporal environment must provide a means to reserve a range of pseudo-times for exclusive use of the program, so that no other executing program can access the queue object in that range of pseudo-times.

We may imagine a pseudo-temporal environment as the generator of a sequence of pseudo-time values to be used in the execution of those program steps executed within its context. We must insure that in the sequential program above, each version.ref$define applied to a version of the queue object is applied at a different pseudo-time, and that they are ordered in a corresponding order to the order of the statements. In addition, the version.ref$lookup operations must be applied in pseudo-times that fall in the right place in that ordering.

In the way pseudo-temporal environments are generated and manipulated, one hopes to capture a modular notion of program construction. When the queue operation above is constructed, compiled, and executed, the programmer or user may not be able to provide any information about how many, if any, version.ref$define operations the enqueue or dequeue operations actually carry out when invoked. Thus, the actual execution will involve executing the enqueue operation's implementation in a subrange of the pseudo-times contained in the program's pseudo-temporal environment. We can think of the structure of the pseudo-times as being

hierarchical. The set of all pseudo-times, Ω, is broken up into subranges that begin and end in pseudo-times that are system-wide consistent states. One of these subranges that corresponds to the execution of a sequential program is further broken up into subranges that begin and end in pseudo-times that correspond to the states in between execution of operations that are separate modules. We can call all of these ranges (at all levels) pseudo-temporal environments. Ω is the "root" pseudo-temporal environment.

Thus we have the following operation that defines pseudo-temporal environments as subranges of other pseudo-temporal environments. In defining this operation, a new ordering relation is introduced. Two pseudo-temporal environments x and y are ordered if and only if all the pseudo-times in the range of x precede all of the pseudo-times in the range of y. This partial ordering is symbolized by the notation ~; the formula x~y may be read as "x strictly precedes y." This notation is also extended to the case where either x or y is a pseudo-time.

pte2 := pte$transaction(pte1)

The pseudo-temporal environment pte2 is a subrange of pte1. It is guaranteed that the result of two separate invocations of pte$transaction on the same pte will be two pte's x and y such that x~y or y~x. That is, x and y are non-overlapping subranges. Further, after executing w:=pte$transaction(z), w is a subset of z. pte$transaction can be applied repeatedly, in order to generate subranges of subranges.

Pseudo-temporal environments are then used to provide pseudo-times for version references used implicitly in computations. There are two ways to get pseudo-times from a pseudo-temporal environment. If one is needed for a version_ref$lookup operation, there is no need to get one that differs from the one last used for a version_ref$define operation. If one is needed for a version_ref$define operation, it must be strictly later than any one used previously to

define a version of the object. We ensure that by saying that the pseudo-time used for a version ref$define operation is later than the pseudo-time used for the last version ref$define or version ref$lookup in the same pseudo-temporal environment. Two operations are thus used to select the next pseudo-time to be used in a program. The first, pte$current is used only for desugaring reads into version ref$lookup operations, and the second, pte$next is used only for desugaring updates into version ref$define operations.

pseudo-time := pte$current(pte)
If **X** is the set of pseudo-times returned by pte$next(y) operations or contained in pseudo-temporal environments generated by pte$transaction(y) operations executed before a:=pte$current(y) is executed, then for all $x \in \mathbf{X}$, $x \Rightarrow a$.

pseudo-time := pte$next(pte)
If **X** is the set of pseudo-times returned by pte$next(y) operations or contained in pseudo-temporal environments generated by pte$transaction(y) operations executed before a:=pte$next(y), then for all $x \in \mathbf{X}$, $a \rightarrow x$.

Now we can give a procedure for desugaring a program that executes in a particular pseudo-temporal environment p, in order to convert from implicit use of a pseudo-temporal environment to determine the pseudo-time used for each reference to explicit code that selects a pseudo-time for each reference. We must have a notation to indicate that a particular pseudo-temporal environment is to be used to control the execution of a statement and any operations invoked in that statement. This will be the **in** statement:

in pte **do** *statements* **end**

where pte is some pseudo-temporal environment, and *statements* is a sequence of statements. To

execute the **in** statement means to execute the statements in its body, resolving object references to shared objects through pte.

Basically, what we do is to change every object reference used to read a value from x to version_ref$lookup(version ref$freeze(x,pte$current(pte))), every assignment to a shared object reference x:=y to version_ref$define(version ref$freeze(x,pte$next(pte)),y), and any invocation of a separate module m(params) to m(params,pte$transaction(pte)). In order to handle separately defined modules, such modules always get an implicit parameter that specifies the pte in which they are to execute. The entire text of a module is implicitly contained in an **in** statement that specifies the pseudo-temporal environment passed as parameter.

In this desugaring of programs, it is important to remember that pseudo-temporal environments and pseudo-times are *not* shared objects. They are merely objects local to the interpreter of the program (as is the instruction counter, for example).

As a brief example, the desugaring of the following program that sets the cell c to the sum of w and b, then calls a module d which divides the cell w by c,

```
example = proc(w,b,c)
        d = proc(x,y)
                x:=x/y;
        end d

        c:=w+b;
        d(w,c);
end example
```

is (using the dummy variable p to hold pseudo-times and dummies t_i to hold pseudo-times to clarify order of evaluation):

```
example = proc(w,b,c,p)
        d = proc(x,y,q)
                t4:=pte$current(q);
                t5:=pte$current(q);
                t6:=pte$next(q);
                version_ref$define(version_ref$freeze(x,t6),
                        version_ref$lookup(version_ref$freeze(x,t4)) /
                        version_ref$lookup(version_ref$freeze(y,t5)))
        end d;

        t1:=pte$current(p);
        t2:=pte$current(p);
        t3:=pte$next(p);
        version_ref$define(version_ref$freeze(c,t3),
                version_ref$lookup(version_ref$freeze(w,t1)) +
                version_ref$lookup(version_ref$freeze(b,t2)));
        d(w,c,pte$transaction(p));
end example;
```

When this program is executed, $t_1 \Rightarrow t_2$, $t_2 \rightarrow t_3$, $t_3 \rightsquigarrow q$, $t_3 \rightarrow t_4 \Rightarrow t_5$, and $t_5 \rightarrow t_6$. Given these relations among pseudo-times of reference, it is easy to show, given the memory model that the program has the effect of setting w on output to w/(w+b), and c on output to w+b. An important part of showing this is knowing that the only version_ref$define operations that might change w, b, or c are those executed with version references derived from the pseudo-temporal environment used in calling example. Since example is a module, by convention it is invoked with a pseudo-temporal environment that is a subrange of its caller's pseudo-temporal environment.

The traditional interpretation of procedures would allow any parallel computation to modify the shared cell object w in the example above between any of its references in the program. So in addition to the functional effect the procedure would have were there no parallelism and sharing, there are many other possible behaviors the procedure could have on the state of the objects it refers to when executed. For example, if w is changed between the two

statements of the outer procedure, almost any possible value could wind up in w, depending on the change made to w. In the interpretation of the program that is assigned by NAMOS, only the functional behavior, equivalent to dividing w by the sum of w's initial value and b and assigning to c the sum of w's initial value and b, can result from the execution of the example procedure.

3.6 Programs with internal parallelism

Not all programs are sequential. In fact in a decentralized system, doing remote operations sequentially (i.e. waiting for node A to finish its operation before starting the next operation, which is to be done at node B) may result in unnecessary and unsatisfactory delay -- thus writing parallel programs is encouraged. Although the pseudo-temporal environment concept is primarily intended for handling unanticipated interactions between computations run in parallel because they were designed and requested independently, the concept can also be quite useful in managing the interactions on shared objects resulting from designed-in parallelism. Two ways to create parallel computations are the **either** compound statement, which creates a set of parallel executions, one for each statement, and terminates whenever one or more of the executions terminate, and the **all** compound statement, which creates a set of parallel executions of the statements that terminates only when all of the parallel executions terminate. Termination conditions are important once we start discussing the interactions of failure with the synchronization mechanism.

In order to capture the essence of parallel execution, we have to extend the pseudo-temporal environment concept somewhat. It would be somewhat difficult to define the order of evaluation of pte$next operations in desugaring a parallel compound statement. By the definition of **either** and **all**, all of the actions on shared objects caused by the compound statement should deal with versions later in pseudo-time than those accessed by statements sequentially preceding the compound statement. Similarly, all of the actions following the compound statement must deal with versions later in pseudo-time than any dealt with by actions generated by the compound statement.

The individual parallel branches are designed knowing that they are executing in parallel with other computations. They must therefore specify explicitly within themselves how their accesses to shared objects are to be synchronized (it may be the case that no shared objects are accessed, in which case they need not be synchronized at all). This they will accomplish by using pseudo-temporal environments constructed by the pte$transaction operation. But in order to ensure that we don't have to think about how the pseudo-temporal environment is manipulated by concurrent pte$transaction operations, etc., we invent an operation that constructs parallel streams of pseudo-time.

pte1, pte2, ... := pte$paraction(pte0)

pte$paraction returns multiple values. The values returned are parallel streams of pseudo-time. If we execute a,b,c,d:=pte$paraction(e), the set X=a∪b∪c∪d is a subrange of e that is ordered with respect to all other subranges of e generated by pte$transaction(e) or pte$paraction(e). X is also ordered with respect to pseudo-times generated by pte$next(e) and pte$current(e). Further, to capture the notion of parallel execution, a∩b=∅ as do the intersections of any other pair of results. Finally, if f:=pte$transaction(a) and g:=pte$transaction(b) are executed, then f and g are non-overlapping: f~g or g~f. Thus f gives exclusive access with respect to the other environments.

Desugaring a parallel compound statement then just consists of preceding the compound statement with a pte$paraction(p) where p is the current environment, to produce as many results as there are parallel actions. Then each branch of execution is surrounded by an implicit **in** specifying the result of the pte$paraction corresponding to that branch.

We can also use the tool given by pte$paraction to specify the parallelism inherent in independent execution. Each user of the system can be thought of as executing by default in a pseudo-temporal environment that was initially created by an initial single execution of pte$paraction(Ω). Thus all independently initiated computations execute in fully parallel streams of pseudo-time in which transactions are guaranteed to be ordered with respect to transactions that happen in other streams.

3.7 But we can't know the entire history!

Now is the time to answer the question posed by our static view of system history. Pseudo-times have been shown to provide a nice way to think about the synchronization relationships among computations in the system. However, we have been assuming that any computation can reach out and get hold of or assign to an object version at any time. Clearly, if the execution of the the system is to be physically realizable, versions must be defined before they are used.

It is not really very difficult to ensure that a version is defined before it is used. The realization of a version_ref$lookup operation is simply to wait until the version is defined. Because certain versions may not be defined at a particular real-time instant leads us to define the concept of a *known history*. The known history of an object at any instant of real time

consists of the mapping between version references and versions that has been defined so far. Thus the known history of an object at any time is a subset of the object history. More importantly, as real time passes, more and more of the object history is contained in the known history. Thus at two different real times, the known history corresponding to the earlier time is a subset of the known history at a later time.

The known history changes as computations are executed in real time by a process called *eduction* (from the verb *educe*, meaning "to draw out, to elicit"). The eduction of a known history just consists of creating new versions and extending the range of pseudo-times that an already created version belongs to. There are two ways in which a known history is educed. First, a version_ref$define operation for a version reference not yet defined in the known history may be executed successfully. This results in creating the version associated with that version reference.

The second way in which a known history is educed is via the version_ref$lookup operation. If a version_ref$lookup operation is applied to a version reference not yet defined, there are two ways that the system can respond with a version. Either the operation can be made to wait before responding with a version until some version_ref$define operation completes, or a version that has already been defined at some earlier pseudo-time can be returned. Returning a version defined at an earlier pseudo-time requires that all version references for that object for pseudo-times between the pseudo-time of definition and the pseudo-time of the lookup refer to the same version. Any attempt made at a later real time to use version_ref$define with a version reference that refers to such an intermediate version must be rejected as a redefinition.

Figure 6 illustrates eduction of a known history by creating a new version. The labeled boxes are individual versions. The horizontal axis labeled "pseudo-time" represents the continuum of pseudo-times in increasing order. Versions are connected to pseudo-time by specifying their *ranges of validity* -- the set of pseudo-times for which the object is defined to have that version as its value. The object A, for example is the value of the object for pseudo-times between r and q inclusive. B is defined to be a new version of the object by version ref$define(version ref$freeze(object,p), ...).

Fig. 6. Eduction by creating a new version

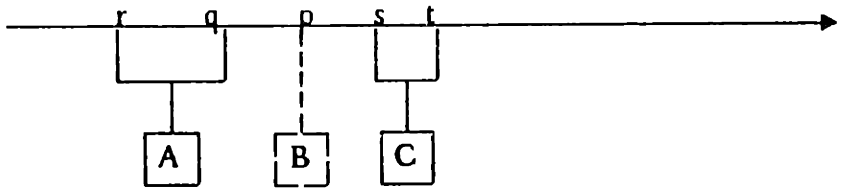
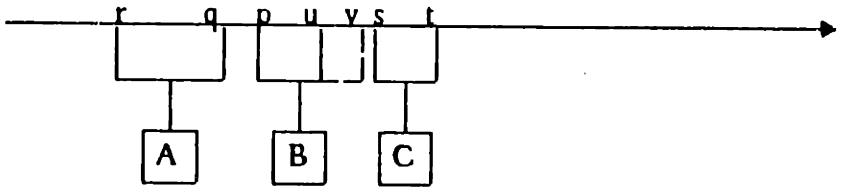


Figure 7 illustrates eduction as the result of a lookup executed at pseudo-time v that returns the version B, where the range of pseudo-times in which the version referred to was valid previously extended only to u.

Both kinds of eduction interact strongly with the failure management mechanism to be discussed shortly. Thus, I have been a little imprecise in the previous paragraph, but the imprecision will be corrected later.

Fig. 7. Eduction by lookup of existing version



Another fact of the real world is that maintaining the known history of an object would require the system to provide an ever increasing amount of memory as time goes on. Thus we would like to allow the system to maintain only a subset of the known history of each object. Consequently, a version_ref$lookup or version_ref$define operation corresponding to a region of pseudo-time for which the known history has been decided, but which is not maintained in the system at the time of the operation must be rejected for a new reason, unavailability. This rejection is signalled as the condition forgotten_state by those two operations. In chapter six, we consider further the mechanism for choosing the subset of versions to be maintained. However, the most important requirement is that the "current" version of an object always be maintained (the one whose pseudo-time is the greatest within the known history).

3.8 Generating pseudo-times and pseudo-temporal environments

Pseudo-times have been shown to provide a relationship among the histories of system objects. In a real system, however, it would also be useful to relate the system history to the history of the system's interactions with the real world. So far, we have not made any assumption at all about the relationship between the rate of increase of pseudo-times in independent,

concurrent processes. We have shown only that pseudo-time increases or stays the same within a single process, as a result of executing steps of that process. The simplest way to relate the pseudo-times in concurrently executing processes is to create a correspondence between the pseudo-times used and real time.

To understand the anomalous behavior that can result from not relating the pseudo-times to real time, imagine a system containing the data bases of a bank, allowing transactions that deposit, withdraw, and transfer money between bank accounts. When people step up to bank terminals, they invoke a transaction, which is executed in a pseudo-temporal environment whose output state is chosen at random. If we then construct a system history in which each transaction is executed, it will be, as I have pointed out, a serial schedule, and thus presumably will give correct results. But imagine the quandary of an individual customer. On Thursday May 25, 1978, he opens an account, depositing fifty dollars in cash (his transaction is executed in pseudo-times between 3276800 and 3276805). On Friday, he wants to withdraw ten dollars, so he requests a withdrawal (run as a transaction that reads the account at pseudo-time 151970), getting the response "no such account!"

The problem is obvious. Pseudo-time must be correlated with real time. First of all, the pseudo-times used for operations must be non-decreasing from the point of view of any user in the system. Further, since users can talk to one another outside the system about their interactions with the system, whenever two users are sure that a pair of interactions with the system were not simultaneous, the pseudo-times used in the interactions to specify output states must be ordered in correspondence with the real-time ordering.

Lamport[Lamport78] has already observed this problem. The system must assume that if two interactions with the system are not close together in time, then they must be executed in the order that users will expect. In order to generate values that are ordered in correspondence with real time, one must have at each node a way of creating a pseudo-time value that exceeds all previous values created at that node. The pseudo-time value created must also exceed all values created at other nodes at significantly earlier times. In chapter six, I will discuss how the correlation with real-time is easily achieved by using approximately synchronized clocks.

3.9 Failures and Recovery

In executing any real computation interacting with shared objects, failures must be expected. As noted in chapter two, the failures we are concerned about can all be modeled as failures to complete a requested action or group of actions. Thus, a program being executed may stop due to a crash of its associated processor, a requested operation may be rejected as inconsistent with the definition of the object (popping an empty stack, for example), a communications link needed to request a remote action may fail, preventing the action. Other occurrences that can be modeled as failures to complete a requested action are protection exceptions that result from a change in the requesting program's access rights to an object, rejection of a version.ref$define operation because the object referred to has already been defined for the specified pseudo-time, and rejection of version.ref$lookup or version.ref$define operations due to the forgotten state signal, resulting from deletion of out-of-date versions.

We would like to provide a basic mechanism to handle this kind of failure. None of these failures are preventable, so it is important that the mechanism be designed to handle such failures gracefully wherever they occur. Similarly, none of these failures can be expected to happen at "nice" places during the execution of a program. For example, if a program refers to an object several times in succession (as in the queue example), there is no guarantee that just because the first access succeeded, later accesses will also succeed. Further, once such a failure has occurred, it may be impossible for the program to take corrective action by *undoing* changes it has made -- if the node containing the program fails it certainly won't undo its changes, and if the protection status of an object, or the hardware communications leading to the object have failed, undoing the changes may be very likely impossible.

The primary problem of failure management is that meaningful operations on shared objects are constructed out of multiple operations at a lower level. Usually when executing a meaningful operation (such as a transaction), if some one of the component operations fails, the result will be some sort of inconsistency that may not be tolerable, because the level of abstraction at the operation's interface is compromised. For example, if in a bank transfer of funds from one account to another, if one account is incremented while another is decremented, and either the increment or the decrement fails, the bank will either lose track of some money or gain some money. This leads to the need to be able to manage groups of updates that must either be all done together or not be done at all.

Because it unlikely that a set of changes can explicitly by undone in response to a failure, we would like the system to behave as if they were never done if in fact a failure occurs. This means that there must be a mechanism for creating tentative changes in the state of objects that

cannot be observed outside the computation making the changes, such that within the computation making the changes, those changes will be observable.

We can solve the problem again by taking the omniscient viewpoint. What a program wants to know when it is about to do an assignment to an object is whether the group of assignments to which it belongs will properly finish, or whether some one of the assignments will fail. If we could send signals backwards in time, there would be no problem, since before the operation was begun, it could query the future about whether the operation will be allowed to complete. Then the operation would simply halt immediately with no effect if it were about to run into some kind of failure. In fact, sending signals backwards in time is not needed.

Instead, we use an old magician's trick for predicting the future. A magician produces a sealed envelope for inspection. He claims that written on the piece of paper inside is the name of the playing card you will pick. You carefully make sure that the envelope is sealed, by gluing one of about a hundred or so different postage stamps across the flap. Then the magician holds the envelope while you shuffle the deck and pick your card -- the Joker. You then take the envelope from the magician and unseal it. Inside, the paper says "Joker." Amazing!

Not really. Look at the envelope closely. There is an imprint in the paper of the front side that says "joker" as well. Look in the magician's hand. There is a small metal stylus taped to his right index finger. Inspect the paper from the envelope. Sure enough, it is carbonless copy paper that turns dark when pressed with a stylus. The magician's trick becomes clear. He merely waited for you to pick your card, then inscribed the name of the card on the envelope with a stylus, causing the paper inside to contain the name. He may be well schooled at writing in concealment with one finger, a noble accomplishment, but he hasn't predicted the future.

The essence of the trick lies in the envelope, because it prevented you from verifying that there was in fact the name of a card written on the paper. Given that you believed that something was written on the paper that you could look at any time you wanted to, you had to conclude that the magician knew in advance the card you would pick. But the magician knew that he could force you not to look at the paper until after you picked the card by placing it in an envelope.

We can use a similar trick to implement the kind of "backward" signalling in time that we want. We define a special kind of envelope called a *possibility*, that is defined to contain a piece of paper that either has an X written on it or nothing written on it. One of these envelopes is associated with each group of updates that is to be performed. We are told that the group of updates will all be executed without failure if there is an X on the piece of paper, but if the paper is blank, then none of the updates is to be executed. In fact, the system holds the envelope. But instead of executing the updates by actually making the assignments, the system makes a note at each object where an update is attempted that says, in effect, "If there is an X on the paper, the value is <newvalue>. Otherwise, the value is <oldvalue>." Actually the paper was created blank, but when we tell the system that all our updates are done, it inscribes an X on the paper, and says, "see what I mean, I knew you would be done." If you ask to see the paper before all the updates are done, the system opens up the envelope and says, "see, you aren't done." Once the system opens the envelope the paper inside cannot be changed, and any further updates are refused by the system, saying, "I'm sorry, I just can't do that update for you."

We represent the envelopes by entities within the system called *possibilities*. A possibility contains a boolean value (corresponding to the mark, or lack of it). This value is explicitly set to *true* when the associated set of updates complete. In the possibility mechanism, we also include a timeout, after which the possibilities value cannot be set to *true*. By including a timeout, we allow the system to bound the time that a group of updates may be in progress. Also, since prematurely opening the envelope aborts the group of updates by necessity, the timeout specifies a time before it is probably not a good idea to open the envelope.

Looked at in real time, there are three important states of a possibility. Before its value is decided, it is in the **wait** state, so named because an attempt to test the value will be forced to wait (at most until the timeout is past). If it is successfully set to true, it enters the **complete** state, so named because the group of changes made under it have been completed. If it is explicitly set to false or if the timeout is passed without it being set to true, it is in the **aborted** state.

The operations defining the possibility entity are as follows. The variable possi stands for some possibility, and time stands for some real time.

possi := possibility$create(time)
: The possibility is created with timeout equal to the parameter, and put in the wait state.

boolean := possibility$complete(possi)
: The possibility parameter is put in the complete state, unless it is already in the aborted state. The result is true if the possibility is either already complete, or is set complete.

boolean := possibility$abort(possi)
: The possibility parameter is put in the aborted state, unless it is

already in the complete state. The result is true if the possibility is either already aborted, or has been set aborted.[1]

boolean := possibility$test(possi)
The possibility parameter is tested to see if it is complete or aborted. If it is in the wait state, the operation does not return until the possibility is either complete or aborted. The result is true if the possibility parameter is complete, otherwise the result is false.

As with pseudo-temporal environments, it is easier to treat possibilities as implicit parameters to operations, rather than passing explicit parameters everywhere. Consequently, we need a statement that binds the possibility to be used in all computations initiated by the enclosed statement. For this, we have the **were** statement.

were *possibility* **do** *statements* **end**

The **were** statement binds the possibility parameters to all operations executed within the body to the specified possibility. All calls on modules are furnished with an extra parameter that is implicitly set by use of this possibility. After completion of operations run under a particular possibility, a program must **complete** the possibility in order to make the results of the computation available to other computations. The same possibility can be used in several **were** statements before executing a possibility$complete on it, although such usage seems unlikely to be common practice.

---

1. The abort operation is logically unnecessary, since a possibility will eventually enter the abort state after a timeout anyway. However, delay may be significantly reduced for operations that do lookups dependent on testing the possibility if the possibility is aborted as early as possible.

We have to modify the version ref$lookup and version_ref$define operations to explicitly represent the tentative nature of the version_ref$define operation. Both operations now have an added parameter, a possibility. When the version_ref$define operation is executed, a reference to the possibility is associated with the tentative version being stored. Such tentative versions are called *tokens* because of their use as place holders in the known history. A related concept, that of a write-once cell called a "token," has been described by Henderson[Henderson75].[1] One can think of a modifiable object's versions as a grouping together of many of Henderson's tokens that each represent an individual state of the object.

A token in NAMOS thus defines a potential state of an object. Once the state of the possibility associated with the token becomes known to be true (complete), the token becomes a version. Since the eventual state of the possibility is unknown, though, any attempt to access the object in a pseudo-time for which the token might eventually represent a valid version must be forced to wait until the state of the possibility is determined.

There is a problem, however, because a computation that creates a token may need to access that value later before the state of the possibility becomes set. Consider our queue example above. Suppose that all of the queue operations are executed in the same possibility. After the first queue operation the queue's state will be represented in terms of one or more tokens. The second queue operation should not be forced to wait until the state of the possibility is determined, because only after the fourth operation is completed will the possibility$complete operation be applied.

---

1. Except where explicitly specified otherwise, the word token in this thesis refers to the tentative versions, not to Henderson's tokens.

The problem is resolved by supplying a possibility parameter to version_ref$lookup. If this possibility parameter matches the one under which the token was created, then the token may be returned as the version requested. In this case, the range of validity in pseudo-time is extended just as if the token were a normal version.

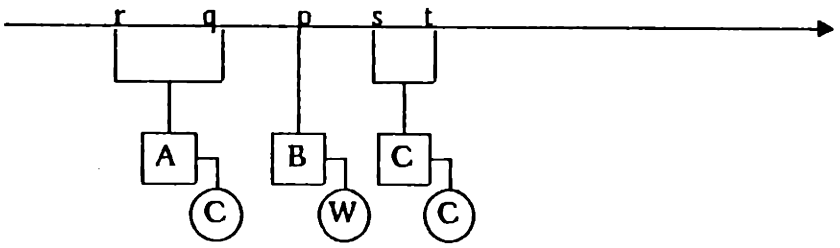
Thus the following redefinitions apply:

```
value := version_ref$lookup(vr, possi)
version_ref$define(vr, value, possi)
```

For any particular object, imagine that we can examine the entire set of version_ref$define operations ever applied to it, such that no error signal was made. Then the subset of version_ref$define operations whose possibilities are (eventually) true is the update history, defining the mapping from version reference to version for all time. However, the results of version_ref$lookup must be defined in terms of the entire set, whether the possibilities are true or not. Add to the subset whose possibilities are true the set of version_ref$define operations that have possibility parameters that match the possibility parameter to the version_ref$lookup operation, to get a new set. Then the result of the lookup operation is that version assigned by the version_ref$define operation whose pseudo-time is the largest pseudo-time not exceeding the pseudo-time of the lookup.

Figure 8 shows how a known history for an object looks, showing tokens and possibilities. In figure 8, possibilities are shown as circles, where the letter W, C, or A indicates whether the possibility is in the wait complete or aborted state. The value B is thus a token, because it refers to a possibility in the wait state, while the values A and C in the figure are versions. A

Fig. 8. Known History with Tokens and Possibilities



version_ref$lookup operation whose version ref's pseudo-time is between p and s might have one of several results. If the possibility parameter to the version_ref$lookup operation is the same as the possibility associated with token B, then B will be returned, whether or not B ever becomes a version. On the other hand, if the possibility parameter to the version_ref$lookup operation is not the same, then the lookup must wait until the possibility leaves the wait state. Then either B will be returned as the result if the possibility is complete, or A will be returned if the possibility is aborted.

3.10 Recoverability

Possibilities allow the construction of *recoverable* computations. The concept of recoverability was named by Gray[Gray77], although the idea has been kicking around in data base circles for a long time apparently[Davies73,Lampson76]. In this work, a useful concept is the *recoverable update set*. A recoverable update set is a set of changes to objects that is either made entirely, or never made at all, from the point of view of any computation observing the objects changed in the update set.

All of the changes made under the control of a particular possibility form a recoverable update set, since any computations viewing the objects will either never see the changes (if the possibility is aborted), or will see all of the changes eventually (if the possibility is completed).

If an operation makes all of its changes under the same possibility, it has particularly nice properties. In particular, if the operation finishes, it can be either totally aborted or completed, but there is no halfway state in which it can leave its changes. If the operation does not finish (e.g., because its node crashed), it can be aborted. Such an operation I call a *recoverable operation*. Not all operations are recoverable, nor should they be. In chapter four, I will discuss uses for operations that are partially recoverable.

An atomic transaction has the property that it either happens completely, or not at all, as well as the property that when it executes no other operation can see or change the states of shared objects that it accesses. With the **were** statement one can construct an atomic transaction. The simple form of an atomic transaction is:

```
in pte$transaction() do
      possi:=possibility(timeout);
      were possi do statements end
      possibility$complete(possi)
end
```

If the statement does not contain any embedded in or were statements, then it will be executed as an atomic transaction.

3.11 Modularity and possibilities

So far, we have not dealt with the problems of modularity in handling failures. A very common kind of problem that will occur is that a module executing at a remote site may fail in the middle of execution, leaving its changes half made. If the module's implementation is unknown to its users, it would be very unwise to allow the users to be able to complete the possibility that would enable its changes to become visible. A module must thus be able to locally ensure that the updates it makes locally are completed all at once or not at all.

Failures of this kind lead to a need for a kind of self-protection at the interfaces to operations. For example, consider that a bank may want to offer an interface to transfer money between bank account objects. To protect itself against losing or gaining money, it would like to make this operation an atomic transaction. Similarly, a user of the bank system (some bank customer) who also makes use of an independently programmed checkbook recording system will want to have an operation built using the two systems that transfers money between his checking and savings account, recording the fact in his checkbook records. Both the bank and the user want to assure recoverability, but the bank cannot completely trust the user to properly abort the bank transaction upon failure. If the failure in the bank transaction is a failure in the middle, leaving only one account changed, the user may choose to complete his possibility anyway, such that the bank transaction is completed even though only one of the accounts has been changed to reflect the transfer.

The user, on the other hand, does not wish the bank to proceed with the transaction unless his personal checkbook balance is also updated. Consequently, the agreement of both the user and the bank that the transaction is to be completed is required.

Again, the problem is that we would like to use modules in the construction of still larger modules, while preserving the level of abstraction provided by the lower level module interfaces. The designer of an operation, whether or not the module is to be used as a top-level module or many layers of modules down, would like to assume that the operation is recoverable (it is either completed or is not), in order to assure that the interface has simple semantics. An operation defined by a module has very simple semantics if it is, as a whole, recoverable (it either finishes correctly, or does nothing). However, operations are built out of smaller operations that also should be recoverable. If the smaller operations are designed without knowledge of their use, they cannot assume that their users will assist in ensuring recoverability. On the other hand, the smaller operations cannot make the decision to complete without the consent of the user operations.

By adding some power to possibilities, the problem of modularity in using possibilities as a failure recovery mechanism can be solved. The idea is to have the user create a possibility that controls the overall completion/aborting of the operation. The bank and the checkbook system each create a possibility specific to its own part of the operation. The bank and checkbook possibilities each depend on the user's possibility in the following way. If the bank's possibility is waiting or aborted either by timeout or abort operation, the user cannot make versions out of tokens created by the transfer operation by completing the user's possibility. If the bank's possibility is complete, then whether the new account changes become versions or not is wholly dependent on the user's possibility.

The mechanism added is a kind of possibility called a *dependent possibility*. Dependent possibilities are created by the possibility$dependent operation:

possi1 := possibility$dependent(possi2, timeout)

The result of the operation is a possibility that depends on possi2. The result, possi1, enters the complete state if and only if the possibility parameter is completed and possibility$complete is applied to the result (normally, possibility$complete is invoked on the resulting dependent possibility before the parameter possibility is completed). If the timeout elapses, possibility$abort is applied to the result, or the parameter possibility is aborted, then the value of the dependent possibility is false.

The bank example can then be built by having the bank construct a dependent possibility that depends on the one used to ensure recoverability of the whole action. In general, modules that interact with multiple shared objects will construct dependent possibilities that are used to control the entire group of actions.

The introduction of dependent possibilities requires a small fix to version_ref$lookup. The problem is easily seen in the queue example. If the queue operations are each executed in dependent possibilities that depend on the possibility that reigns over the whole program, then we have to be careful to define how successive queue operations see the result of previous operations. Once possibility$complete has been applied to the dependent possibility created for the first queue operation, the state of the queue should be made available to a queue operation that is applied within the overall (not yet completed) possibility. Within this other queue operation, a new dependent possibility that depends on the overall possibility is what is passed as parameter to the version_ref$lookup operation. The way the problem is handled is by properly defining what it means to have two possibilities match.

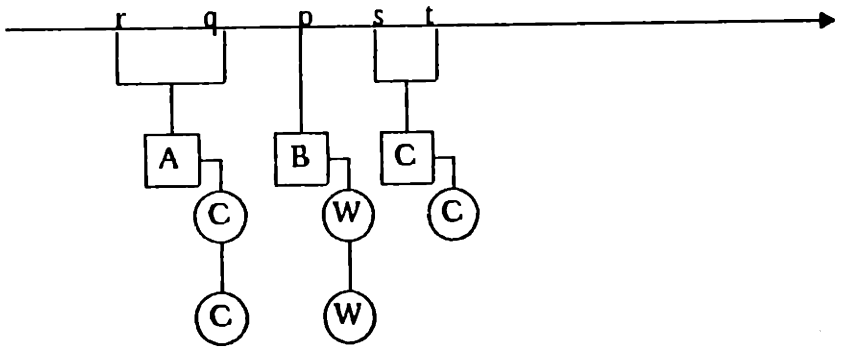
Call the possibility that a dependent possibility depends on its parent possibility. For a particular token, the possibility that must be matched is found by following the parent chain from the possibility used in creation of the token until the first possibility is found that has not had possibility$complete applied to it. If this possibility, the match possibility of the token, is the same as any possibility in the parent chain of the possibility parameter to version_ref$lookup, then the token can be returned as the value. Otherwise, the lookup must wait until possibility$complete is applied to the match possibility, and then the match is reattempted. (Note: if in searching the token's chain of possibilities, an aborted possibility is found, then the token is ignored for the purposes of the lookup, as before).

3.12 Known Histories Revisited

In figure 9, the pictorial notation for a known history is extended to show dependent possibilities. Dependent possibilities are shown as circles, with a further link to the possibility upon which they depend. The letter inside a dependent possibility is either W (indicating that no **complete**, **abort** or timeout has yet occurred to the dependent possibility), C (indicating that a **complete** has been applied before the timeout), or A (indicating that either a timeout or **abort** has occurred at the dependent possibility). The actual state of the dependent possibility can be determined by looking at the chain of possibilities upon which it depends. If any possibility in the chain is marked A, the dependent possibility is aborted. Otherwise, if any are still marked W, the state is wait, else the state is complete if all are marked C.

In figure 9, version A was created in a dependent possibility, and that possibility was successfully completed. Similarly, version C was created within an independent possibility that was completed. Token B is created within a dependent possibility, and so far, both that possibility and its parent are not yet either complete or aborted.

Fig. 9. Object Known History



3.13 Summary

In this chapter, I have described the user's and application programmer's view of the system mechanisms for synchronization, error recovery, and building composite operations. Pseudo-time, object histories, and computations that are recoverable are the basic ideas. These ideas are represented concretely in terms of pseudo-temporal environments, possibilities, and the known history of objects. The notion of an operation and its important special case, the transaction, have been described in terms of these concepts.

The construction of modular interfaces has been very important in the design. The concept of a dependent possibility and the hierarchy of pseudo-temporal environments allow the construction of modules whose internal implementation is not visible outside the interface, even should an unanticipated failure occur within the module or should some other operation interacting with shared objects used by the module be executing concurrently.

Chapter Four

Using the Mechanisms

In the previous chapter, a number of language features and mechanisms to be used for synchronization of access to data have been described. Although some examples were given there, I have not really shown how these facilities can be used to handle synchronization problems. In this chapter, I will develop some examples that show how the language facilities can be used for various kinds of problems. First, I will discuss the well known problem of controlling database transactions. A simple example of a bank with distributed accounts is implemented with the tools that NAMOS provides, to allow the creation of account transfer transactions and statistics gathering transactions. An important point made by the example is that arbitrary transactions can be constructed and run at any time.

Next, I will contrast the solution using my techniques with the solutions possible with locking mechanisms, and the kinds of solutions that are possible by use of synchronizing processes such as Hewitt's serializers. Both of these alternative approaches require both design-time planning to decide the class of transactions that may be run and careful discipline to ensure that the accesses made during transactions are properly synchronized.

The next problem discussed is the problem of user mistakes or failure resulting in irrecoverable loss of data. The idea of a consistent state defined by a particular value of pseudo-time allows easy definition of a backup mechanism. The backup mechanism provides

the capability of restoring the states of a set of objects from some earlier consistent state of the system. The problem of discovering what set of objects to restore to an earlier state is discussed, but no general mechanism is suggested to solve the general state restoration problem.

The problem of conversational transactions is then discussed, and is shown to be one of the problems best solved with a partially recoverable operation. Other problems, such as keeping metering information or "memoizing" (remembering in a cache the result of a hard-to-compute function of a particular set of arguments), are also shown to be cases where total recoverability is not appropriate, but partial recoverability is very useful.

4.1 Transactions

Perhaps the most basic and important synchronization problem is the achievement of multi-site transactions. The reason for the importance of such transactions is that they provide a convenient approach to defining the consistency of a database in the face of failures. Further, transactions are easy to design, since the programmer need not worry about the problems of synchronization of his transaction with others that manipulate the same database.

A transaction, according to Gray[Gray77], is a program that when run takes a consistent state of the system into a new consistent state. In other words, if a transaction is started with data that is consistent, the changes the transaction makes will leave the data consistent. During the transaction, after only some of the changes have been made, the data in the system may not satisfy any consistency requirements. Thus, running a transaction requires that:

1) The data referred to by a transaction all be from one consistent state.

2) Other computations that wish to see a consistent state be prevented from seeing the intermediate states of the system caused during the transaction.

3) Since a new state is built from a previous state by selectively changing some objects, but leaving most objects the same, the unchanged objects in the state containing the input data must still be the same in the final state constructed by the transaction.

Gray's definition of transaction is not complete. It only deals with what I call *internal* consistency -- consistency among the objects within the system. Another aspect of transactions is that they have a notion of external consistency -- consistency between the system as a black box and the history of inputs and outputs of the system. Imagine that upon completion of a transaction, the transaction printed out some date and time to indicate when the transaction was actually executed. It is useful to require that this date and time corresponds to some time between the initiation and completion of the transaction. Then the system will be externally consistent if it behaves as if the transactions executed were actually executed one at a time in the order indicated by their date and time values.

4.1.1 Building transactions in NAMOS

By convention, NAMOS is to be used such that a particular pseudo-time corresponds to a consistent system state as in Gray's definition above. Also, by convention, the ordering of transactions is by the ordering of the pseudo-temporal environments. Consequently, if some pseudo-time in the range selected by the pseudo-temporal environment were printed out as the date and time of execution, the system will guarantee that the internal state is externally consistent with the ordering of execution implied by the times printed out. Note that requiring

that date and time printed out correspond to some time between initiation and termination of the transaction places some constraints upon the way pseudo-times are chosen -- I will come back to this issue in chapter six. The pseudo-times chosen for "top-level" transactions, those initiated interactively by some user, must be reasonably close to real time.

As noted in the previous chapter, construction of transactions in this system is relatively simple. Any computation that manipulates shared objects can be executed as a transaction, by

a) Executing it in a pseudo-temporal environment constructed by the **pte$transaction** primitive, and

b) Making all of the updates conditional on a possibility that is completed only if no errors occur that prevent the computation from finishing.

Condition a) above ensures that the transaction reads consistent data, and that the only changes made to the state of the system during the range of pseudo-time composing the transaction are those that it initiates. Condition b) ensures that no matter what kind of error happens, the transaction never shows to the outside world of other computations any of the intermediate states that it creates as it proceeds to make changes.

Now consider an example. Consider a very simplified view of a distributed bank system.[1] The objects of the system will be account balances, and the transactions that I desire to implement are of two kinds. Transfers of money from one account to another are one kind of transaction, and another kind of transaction is a special one used by bank managers called the

1. NO attempt has been made to incorporate into this example any of the real-world aspects of banks. More precisely, I do not intend to imply that the simple bank system can be easily extended to incorporate the complexity of "float", the legal requirements applying to bank systems, or the user interface semantics provided by current bank practices.

MIS transaction. An MIS transaction takes some subset of the objects, and computes some statistics, such as means and standard deviations, based on the value of the account balances.

Suppose that we design the programs that implement these transactions in the obvious way. Let's say that each account balance is stored as an object implemented by the account cluster,[1] which defines abstract operations to credit the account, debit the account, and get the account balance. Each of these operations is implemented by a program that manipulates the representation of accounts; the account representation is not known outside of the account cluster. It is not necessary to understand the implementation of the account cluster that follows; it is included for completeness.

1. A *cluster* is the CLU mechanism for creating abstract types by specifying an underlying representation and the operations that are allowed to manipulate the underlying representation to provide the desired semantics for the type. Within a cluster, the reserved symbol **rep** indicates the type defined by the cluster, the special marker **cvt** is used to indicate a parameter to an operation whose type is the one defined by the cluster, but which during the operation is manipulated as the underlying type. Only the operations and procedures contained within the cluster have the privilege to observe and manipulate objects of the type via its underlying representation. For further details, see the references on CLU[Liskov77a,Liskov78].

```
account = cluster is debit, credit, get_balance;
      rep = record[credits, debits:int];

      balance = proc(acct:rep) returns(int);
            return(acct.credits-acct.debits);
            end balance;

      debit = proc(acct:cvt, amount:int) signals(insufficient_funds);
            if balance(acct) <= amount
                then signal insufficient_funds end;
            acct.debits := acct.debits + amount;
            return;
            end debit;
      credit = proc(acct:cvt, amount:int);
            acct.credits := acct.credits + amount;
            end credit;

      % get balance can be invoked by the shorthand account.balance

      get_balance = proc(acct:cvt) returns(int);
            return(balance(acct));
            end get_balance;
      end account;
```

Then a transfer between two accounts can be written as follows:

```
transfer = proc(acct1, acct2:account, amount:int) signals(insufficient_funds);
      dp:possi := possibility$dependent(2);  % create a dependent possibility
      were dp do
            account$credit (acct2, amount);
            account$debit(acct1, amount)
              except when
                insufficient_funds:
                  possibility$abort(dp);
                  signal insufficient_funds;
                  end;
            end
      possibility$complete(dp);
      return;
      end transfer;
```

When the transfer procedure is called it is executed in its own transaction pseudo-temporal environment, and with the caller's possibility. A new dependent possibility is created by the procedure to protect against a failure between the time account$credit is called, and the time account$debit finishes depositing the money transferred. If the transfer is successful, the new possibility is completed, otherwise it is aborted explicitly if there are insufficient funds, or implicitly by timeout on the possibility in the case of any other error or in the case that the computer executing the transfer stops in the middle.

Similarly, we can write a statistical transaction as a procedure, which again by default executes in a consistent unchanging pseudo-temporal environment. Since the statistical transaction is read-only, we need not be concerned with dependent possibilities. For an example, let us consider a procedure that computes the mean and standard deviations of the balances of a set of accounts specified by an array of accounts.

```
summarize = proc(accts: array[account]) returns(real, real);
% first result is mean balance, second is std. deviation

    mean, stdev:real;
    sum:int := 0;
    sumsq:int := 0;

    for a in array$elements(accts) do
        sum := sum + account.balance;
        sumsq := sumsq + account.balance ** 2;
        end
    mean := float(sum) / float(array$size(accts));
    stdev := float(sumsq) / float(array$size(accts)) - mean**2;
    return(mean, stdev);
    end summarize;
```

As in the transfer transaction, the summarize transaction again executes by convention in an environment that is unchanging.

Now let us consider how these various transactions can interact. There are two interesting cases. First, there might be two approximately simultaneous executions of transfers out of the same account (or into the same account, or one transfer in at about the same time as a transfer out). Second, there might be approximately simultaneous executions of a summarize transaction and a transfer affecting one of the accounts summarized. The following lemma is easily seen to be true based on the definitions in the last chapter.

> Two pseudo-temporal environments, p_a and p_b, are ordered (that is, $p_a \sim p_b$ or $p_b \sim p_a$), if both p_a and p_b were created via the pte$transaction operation and neither p_a is derived from p_b nor p_b is derived from p_a. A pseudo-temporal environment x is derived from y if x was created by a call on pte$paraction or pte$transaction whose argument was y, or x is derived from some pte z which was derived from y.

This lemma ensures that if we have two transactions executing, and one was not invoked on behalf of the other, then the pseudo-temporal environments in which they execute will be ordered.

Now let us consider the first case. If we have two transfer transactions executing at the same time, they will be executing in two ordered pseudo-temporal environments. Thus one of the transactions "happens first" in the ordering of system states by pseudo-time (although not necessarily in real time). Suppose transaction A runs in the "earlier" pseudo-temporal environment and B runs in the later one. Then we also know, by the lemma, that the credit and debit transactions executed by A and B are ordered such that A's credit precedes (in pseudo-time) A's debit which precedes B's credit which precedes B's debit.

Now consider the interaction in real time on the known histories of the account being debited by both transactions. If A performs its update on the account entirely before B accesses the account, there will be no problem. B will observe that the initial balance of the account is what was left after A finished his debit. However, if A's change to the account follows a successful read by B of the balance of the account, then B will have educed the known history so that the balance of the account cannot be changed in the range of pseudo-time available to A. A's attempt to perform an update will result in an error signal resulting from an attempt to redefine an already known version. This will prevent A from finishing the debit, and thus prevent the possibility from being completed. Consequently, the credit performed by A will be aborted, since the possibility controlling its incorporation into the system history is never completed.

The case of a summarize transaction in parallel with a transfer is similar. Either the summarize transaction precedes the transfer in pseudo-time or the transfer precedes the summarize in pseudo-time (by the lemma). If the summarize transaction precedes the transfer in pseudo-time, the interaction in real time is relatively simple. If the summarize executes first in real time, then the known history of all of the accounts it references are educed by extending the range of versions to pseudo-times from the summarize pseudo-temporal environment. Then the transfer further educes the range of its accounts' versions to the pseudo-times used to read the balance of the accounts, and creates new versions. If the summarize does not precede the transfer in real time, then the transfer will have first educed the versions of the accounts and done its updates. When the summarize is done, the pseudo-times used will refer to the versions of the accounts that were "current" prior to the transfer. Two possible results can occur. If the version of the account referred to by summarize has been kept as part of the known history, it can be

used for the computation of the balance. Thus, the summarize effectively executes as if the transfer has not happened. If the version referred to has been thrown away (to save storage), then the forgotten_state error is signalled, and the summarize fails. The normal case would be to keep old versions for a sufficient period of time to ensure that it is likely that all transactions that may refer to them have finished. Thus, we can trade off storage (becoming cheaper as technology improves) against the probability of aborting a read only transaction as the result of an update.

In passing, I note that in a system that does not retain old versions requires that a very large read-only transaction (one involving many sites and consequently much delay) lock out updates to anything it reads for a relatively long time. In particular, a system based on locking suffers from this problem. The use of naming for ensuring correct synchronization makes it easy to eliminate this from of lockout by retaining sufficient old history of objects likely to be involved in read-only transactions. The naming mechanism provides the method for properly managing the set of versions retained.

If a transaction is aborted because of a forgotten_state or redefinition error, it is quite reasonable to restart it in a new pseudo-temporal environment. Restarting transactions is the normal method of recovery for transactions that fail because of these synchronization errors. There are two problems with restarting transactions, however. First of all, there is no guarantee that the restarted transaction will not also be aborted because of a forgotten_state or redefinition. There is thus the possibility that a particular transaction may encounter *starvation*, never finishing because it is always aborted by new transactions accessing the same objects. Such a

possibility is unlikely if the likelihood of two transactions simultaneously accessing the same object is low. Nonetheless, it may happen.

Worse than starvation of an individual transaction is dynamic deadlock, where several transactions cause each other to be mutually aborted, and upon each restart, the timing of the transactions causes mutual aborting to recur. In dynamic deadlock, no work ever gets done by the transactions involved in the deadlock. Dynamic deadlock is also possible in NAMOS, although it is unlikely that the deadlock will persist in a distributed system because the exact timing that resulted in a transaction aborting another is unlikely to recur.

In chapter six, a mechanism involving token reservations is developed that can be used to reduce the likelihood of starvation and dynamic deadlock where needed, at the cost of requiring that a transaction "reserve" its resources in advance. Where a transaction can predict the resources it needs (the resources used must be knowable without knowing the values contained in any of the resources to be used), the token reservation mechanism is useful.

4.1.2 Constrast with locking and serializers

Most of the discussion of transactions is couched in the language and mechanisms of locking. Gray[Gray77] gives quite a readable summary of the use of the locking approach in managing a database. In essence, the locking approach requires that one seize exclusive control of the objects to be read and written for a period of real time. Exclusive control is granted by

gaining possession of a lock associated with the object, such that only one process has the lock set at any one time. Transactions may not read or write any data for which they don't hold locks.[1]

In order to guarantee that a consistent system state is observed during the execution of a transaction, there must be one point during the transaction where *all* of the objects touched by the transaction are simultaneously locked. In Eswaran, *et al.*[Eswaran76] this is called a two-phase transaction (the first phase is acquiring the locks, and the second is releasing the locks). Eswaran, *et al.* show that transactions that are not two phase can be executed in such a way that objects referred to by such transactions can be changed or observed by other transactions in the middle of the transaction. Further, there must be a mechanism that guarantees that all updates generated by a transaction are either completed or not done at all before any other transaction is allowed to read the updated objects. Thus, by the time any lock is released, all updates to be made must be known to the system.

If the objects to be updated lie on several nodes of a distributed system, ensuring that there is a point in time when all locks are held simultaneously can be very wasteful, because there must be at least one node that knows that all the locks are thus set -- implicitly then, the delay built into the locking scheme is at least the time for all nodes involved in the transaction to send messages to a common node that the objects at the sending node are locked, followed by whatever computation is to be done, followed by the time to signal to all nodes involved to store whatever changes that have been made, followed by a wait for commitment, followed by messages to all

---

1. As a refinement, some systems optimize the case of reads of an object by defining read-only and update locking modes. Any number of transactions may hold read-only locks for an object, but if any transaction holds an update lock for an object, no other transaction can hold any lock for that object. This refinement doesn't affect the arguments that follow.

nodes releasing locks and guaranteeing that all other nodes have committed. If a transaction involving a large number of nodes is executed with the locking scheme, it is clear that the period of time for which other transactions are prevented from looking at the objects it touches is potentially large. Thus, the likelihood of conflicts between transactions is increased, even under the assumption that it is rare that transactions involving the same data are requested from different nodes at the same time.

In contrast, in my system, I have traded off some space to reduce delay. The tradeoff is particularly clear when looking at the interference between read-only transactions like the summarize transaction above, and updates, such as the transfer transactions. Accessing a very large number of account balances to get statistics has little effect on transfer requests -- by the time the statistics gathering transaction gets to a particular account, the transfers accessing that account may have made a number of changes affecting later pseudo-times than the one used by the statistics gatherer, but since the statistics gatherer specifies what state of the system it wants to access, there will be no problem in getting the old version desired by the statistics gatherer. Of course, the implication is that multiple versions of an object have been preserved, rather than just the most current version, resulting in a cost in space. Quite often, though, space is much cheaper than the delay resulting from a statistics gatherer locking out all updates in order to get to a point where locks are set on all objects simultaneously.

I am not aware of any scheme that uses locking in a distributed system that can execute transactions reliably in the face of failures, while preserving the nice properties of transactions as described above. This is not to say that such a scheme could not eventually be worked out, just that I have not seen one that simultaneously handled all of the problems. The most unfortunate

kind of results of failures may be those that leave objects locked, when the transaction doing the locking has long since failed, or on the other hand, having a lock come unlocked in the middle of a transaction, without the node executing the transaction being aware that the lock was not locked for the whole of the execution of the transaction.

The use of locks also raises the specter of deadlocks that can result if some discipline is not used in the setting of locks. In a distributed system, the requirements of autonomy preclude setting locks all at once, and may preclude defining a lock hierarchy in order to preclude deadlock. The reason is that the objects touched by an operation involving objects at multiple nodes may not be known or knowable in advance because of the hiding of the implementation of the objects provided to assure autonomy. Consequently, some deadlock detection and correction mechanism must be developed that will work in the face of failures. My system has the advantage that there are no locks, and therefore no direct source of deadlock. Basically, deadlock is avoided by providing a time bound on the length of time a possibility can remain in the wait state. Thus, transactions may be aborted too often in my scheme, but there is no static deadlock that can arise. Dynamic deadlock can occur, however, as noted above.

Another problem with locking is that it is hard to add new transactions by defining new modules. If we have an existing transaction, it sets whatever locks are necessary inside the transaction. If we wish to create a new transaction that consists of executing a sequence of existing transactions, we cannot simply call them in sequence because that would not result in a two phase transaction when the locks set and released by each transaction are considered as a whole. Thus, we must make it possible to seize the locks needed by all of the combined transactions before executing any one of them, and release all of the locks needed after the new

composite transaction is completed. Thus, in building new transactions out of existing ones, the locks set by each transaction must be known outside the modules that require the locks to be set. It is thus hard to ensure that the locks needed by a module are set properly, since the module now depends on the caller to do it right.

Another approach to consistency is to design the system such that any two objects that can be simultaneously accessed by a transaction must be accessed by requesting the action through a guard process, whose job is to ensure that the data being accessed is accessed all in a consistent state of the system, and that the operation is completed before other requested operations involving the same objects are started. The serializer concept of Hewitt and Atkinson[Atkinson78] provides this mechanism as a basic synchronization tool. Two basic problems with serializers do not trouble NAMOS. First, with serializers there is no explicit mechanism whereby a serializer can handle the problems of errors that occur making the objects protected by the serializers inaccessible. In a sense, the serializer is in the right place to ensure order, but without the tools that allow ensuring the order. Second, in a system that has independently designed mechanisms with autonomous parts, it is unlikely that a common synchronizing process will have been designed in from the start, yet without such a common process, synchronization may be very difficult to achieve. Again, if we allow construction of new transactions out of previously existing actions, it may be that two existing actions to be used in a new transaction are not protected by a common serializer. Such a common serializer would have to be constructed and users of the objects protected by the new serializer would have to be changed to use the new serializer.

The primary problem with serializers as a synchronization mechanism in the kind of distributed system I am considering is that they require foreknowledge of the kinds of transactions that the user may want to achieve. If the bank system considered above were designed with serializers, it is fairly likely that someone coming up with a later requirement for some statistics not provided normally through the serializer interface would not be able to get those statistics reliably until the system had been reimplemented with a serializer that handled the new transaction type.

4.2 Backup

An interesting problem that is closely related to synchronization is that of implementing what I call *consistent backup*. Suppose that, as a result of user mistakes, some set of operations that has made changes to the system was in error. The user may have typed something at a terminal that was wrong, or whatever, or some processor was temporarily producing incorrect results. What would be desirable as a recovery action would be to restore part of the state of the system to an earlier time. Now this really consists of two parts -- first, finding out the part of the state that should be restored, and then restoring the state of the objects that must be restored.

Finding out the part of the system state that must be restored is often quite difficult, since it is possible that based on the changes made by the erroneous computation, a large number of other changes to the objects in the system have been made. Reversing those changes must be accomplished, as well, in order to completely restore the state of the system to a correct one. In order to do the reversal, the system would have to maintain a dependency graph, where an edge is present in the dependency graph whenever a version depends on another version. Such a graph could be maintained in NAMOS, if a list, called the depends-on-list, would be kept

associated with each version. Each entry of the depends-on-list would be the name of a version of another object that has depended on the version containing the depends-on-list. Finding the objects that must be restored based on a computation that has modified some set M of objects would then involve getting the transitive closure of the depends-on relation, and then finding those objects whose current version depends-on the set M.

Keeping the depends-on-list, however, may be quite difficult and expensive. An alternative would be to have a backup mechanism in which the user would have to guess what objects should be restored to their original state. It would be possible to design local depends-on-lists wherever the cost is justified, to help in discovering this. The result is that restoration of system state after this kind of error may not be correct, if the user or the modules that figure out the dependencies are incorrect.

Once the set of objects to be restored is chosen, however, a consistent restoration of the object states is a quite simple operation. State restoration simply involves making the version of an object that was defined in the pseudo-time corresponding to the system state to be restored the version that is valid in the current pseudo-time. We can define the restoration of an object in terms of the existing version reference operations:

```
version_ref$define(version_ref$freeze(obj1,pt2),
    version_ref$lookup(version_ref$freeze(obj1, pt1)), poss)
```

will make the version of obj1 at pseudo-time pt2 the same as that existing at the earlier pseudo-time pt1. If we execute a group of these operations all under the same possibility, we can make the state of all the objects the same as the state that existed at pseudo-time pt1, thus reversing all actions taken at pseudo-times between pt1 and pt2. In order that the code above

will work, it is necessary that the version as of pt1 still can be gotten without a forgotten state error resulting from the system having discarded it. It is also possible that the restoration will fail as a result of some simultaneous transaction defining the version at pt2 through a lookup with a later pseudo-time.

At the language level, it is much more convenient to define a per-type restore operation, such that *type*$restore(obj1,pt1) when executed in a transaction whose pseudo-temporal environment provides pt2 from its pseudo-temporal environment, will have the effect of restoring obj1 to the state it had as of pt1. In implementing *type*$restore for an abstract type, the definer of the type must make sure to restore the state of all objects in the representation of the object. The designer of the type must take into account the same issues he must deal with when defining a copy operation in CLU -- in particular, he must decide whether objects referred to in the representation are part of the object or not. If a stack that has been pushed is restored, it certainly should be popped, but the objects referred to by the stack probably should not be changed by a restore on the stack (unless the objects referred to were actually changed by the operation being undone). All primitive types that can be updated, such as records and arrays, will have restore operations built in, so that abstract types can use these restore operations to build their restore operations.

In addition to a restore operation for each type, we need at the language level the ability to obtain pseudo-times to pass to the restore operation. A simple interface would be to have an operation available at the language interface to obtain a checkpoint to which one can restore objects. The checkpoint operation would simply involve getting the current pseudo-time by

invoking pte$current on the regnant pseudo-temporal environment. We might write this as a statement:

checkpoint c;

where c is a variable whose type is pseudo-time. Desugaring the checkpoint statement simply results in an assignment into c from an invocation of pte$current on the regnant pseudo-temporal environment.

Of course, certain kinds of failures may still prevent state restoration. In particular, if the object history for each object is not maintained forever, there may be a certain point in the past before which the state cannot be restored. Choosing how much to keep of an object history, then, must be a carefully made design choice that properly trades off the cost of keeping the old versions against the value of being able to restore on errors.

As noted, this is only a partial solution to the problem of state restoration, but the ability to name individual versions of objects makes at least part of the problem quite simple. In a system without the autonomy constraints assumed in the thesis, constructing the depends-on relation may be more feasible on a system-wide basis. In such a case, a complete solution to the problem of consistent backup may be possible.

However, discovering the true dependencies between object versions is highly dependent on the semantics of the computations that generate versions. The system, in the absence of such information, would have to make assumptions that are worst case -- e.g. if some process has ever touched object X, all objects ever touched by that process may depend on X. For example if discovering a particular value in X is what causes some set of earlier updates to be **completed**

by completing some possibility, those values depend on version X. Just by seeing what the process actually did, one cannot discover that the earlier updates would have been completed in any case, independent of the value of X. Consequently, in the absence of any other information, the depends-on relation would be very bushy, requiring a much larger number of backups to recover from an error than might actually be the minimum required.

Consequently, I suspect that the best approach, whether or not the system has the autonomy requirements of the distributed systems I have considered, is to leave keeping up dependencies to a much higher level, and use a mechanism like mine to do the state restoration, once the set of objects to be restored is known.

4.3 Conversational System Interactions

Unplanned transactions are not the most difficult kind of interaction that the system may have to support. One factor that unplanned transactions have is that the code for the transaction is entirely within the system at the time of execution. In the case of conversational interactions such as those that might be supported by an interactive database query application, a multi-node debugger, or an interactive program library system used by multiple people, not even the program to be executed can be known by the system in advance. Nonetheless, there is a need in such interactive environments to obtain and act upon data from a consistent state of the system. Further, there may be a need for running a particular set of operations, checking to see that the result is correct, and then either aborting or completing the operations.

It is fairly easy to make a conversational interaction work in my system, with the use of the tools already present. The entire interaction ought to be carried out in a single pseudo-temporal environment, under a single possibility, just as in a transaction. The timeout specified when the possibility is created must be large enough to carry out whatever updates are contemplated. The interactive user is then free to pursue whatever interaction he desires to make. Upon finishing the interaction, the user lets the system know that the interaction is over, specifying whether or not to complete the possibility.

One basic problem with conversational interactions in any system is that the user must be prepared to have his interaction aborted. In my system, there are two kinds of failures - inaccessibility of a particular object version that is to be read, and aborting the possibility under which the interaction is carried out. Inaccessibility is a soft failure that the user may want to handle by ignoring it or by accessing another version. Aborting the possibility is important if updates are involved -- in which case the user can decide if he wants to try again or not. He must be aware that other values in the system may have changed, and so may want to inspect them again in the pseudo-temporal environment in which the retry is executed. I can imagine a simple kind of assistance that the system could perform, which would involve warning the user that his possibility is about to expire so that he can finish his updates, and which would also involve keeping track of those objects that are touched in the current input state, so that if the transaction is aborted, the values that change between that state and the retry state can be identified for the user.

Conversational interactions are much more difficult in a locking based scheme. One must be sure that all of the locks are set, and that the locks are properly cleared when the transaction is done. More importantly, since conversational transactions take a long time, the interference caused by reading of data with updates may become severe. Because the conversational transaction is unpredictable in what it will access, the only recourse for handling deadlock is some sort of deadlock detection and correction scheme. In a locking based scheme, aborting is at least as difficult as in my scheme.

Serializers allow for no conversational interactions other than the preplanned ones packaged up as part of the serializer requests.

4.4 Partially Recoverable Operations

In the previous chapter I noted that there is a need for operations that use multiple possibilities in their execution, making it more difficult to handle failures, since some of the actions carried out by a partially recoverable operation will not be reversed by aborting a single possibility. I would like to give several examples to justify this need, and to show how good use may be made of the ability to use multiple possibilities in an operation.

Perhaps the simplest example I can give is the keeping of metering information associated with an object. Perhaps it is desirable to keep track of the number of times attempts (whether or not aborted) to update an object are made. In order to do this, a way of keeping this count associated with the object without having it backed up when some failure having nothing to do with the object itself happens is needed. One simple way to build such a counter into the operations on the object is to update the counter under a brand-new possibility that is completed

immediately after the update. In the bank example, one could build into the transfer transaction the following code:

```
transfer = proc(acct1, acct2:account, amount, counter:int) signals(insufficient_funds);
      ip:possi := possibility$create(2);  % create an independent possibility
      were ip do counter := counter + 1; end
      possibility$complete(ip);      % the counter is now permanently incremented

      dp:possi := possibility$dependent(2);  % create a dependent possibility
      were dp do
            account$credit (acct2, amount);
            account$debit(acct1, amount)
              except when
                insufficient_funds:
                  possibility$abort(dp);
                  signal insufficient_funds;
                  end;
            end
      possibility$complete(dp);
      return;
      end transfer;
```

This code ensures that any attempt to transfer money results in the parameter counter being incremented.

It is important to note that the program uses multiple possibilities here, but did not use a new pseudo-temporal environment. Thus the change made to debit.count is only visible after the output state of the transaction, whether or not it is completed. Had a new pseudo-temporal environment been used, it is possible that for two transfers, the counting of the transfers would happen in a different order than the transfers actually happened, and might be actually not visible in pseudo-time till much later than either of the transfers.

Another important use of local possibilities is in the form of run-time optimization often called "memoizing." In this form of optimization, upon each use of an operation on some object, the results of that operation are remembered, along with the state of the parameters that led to the result. Then later, if the operation is re-requested with the same parameters, or some operation that has similar paramters is requested, the saved results can be used to more quickly compute the new results. If all of the updates made by an operation were undone by aborting the possibility, then such memoizing would only help in cases where the overall computation that generated the saved results was completed. An important use of such memoizing might be to reduce recomputation in the case of failure, however, by detecting the fact that he computation requested is the same.

By computing memoized results under a possibility that is completed once the saved results are available, the memoization can be completed independent of the completion of the enclosing computation. Since memoizing can be used in such a way as not to change the interface behavior of the object operations, the fact that the memoizing is not recoverable upon failure is not serious.

The common theme among these examples of partially recoverable operations is that the primary results of operations, that is the changes made by the operations that are important at the interface, are still recoverable, having been made under the possibility that is current at the invocation of the operation. Only secondary changes are made under possibilities that are completed independent of the overall computation, and in some sense the completion or non-completion of these secondary changes does not change the important semantic properties of the interface.

Chapter Five

Implementation of Possibilities and Tokens

A key notion in NAMOS is the idea of a possibility. So far, a possibility has been described as a somewhat special entity that has no history in pseudo-time. The purpose of the possibility is to represent the ultimate choice made by a computation of whether or not to make a particular hypothetical set of changes computed for shared objects real and known to other users of the objects. In this chapter, we discuss the realization of possibilities in a decentralized system of the sort described in the first two chapters.

An important problem in the implementation is dealing with multiple kinds of failures. Two basic kinds of failures are important here. Lack of availability due to communications failure or node failure (where refusal of a node to communicate is considered a "failure") is one sort of failure. The other sort is the failure of a computation somewhere in the middle. Both of these failures are really viewed as excessive delay -- the first sort being refusal to communicate for an overlong time, and the second sort being a long delay in the execution of a computation. Since long delays look like failures, one can never be sure whether a computation or communications path has stopped operating or whether it is still working, but slowly. Consequently, whenever a failure is discovered, the mechanisms that recover from the failure must not only recover in the case where the failing component is really dead, but they must also handle the case where the "failing" component is alive and continuing to attempt to operate on that part of the system state that has been the subject of recovery.

It is in the handling of failures that the implementation of possibilities and tokens becomes difficult. The chapter will first discuss the overall problem of "atomic commit" that is basic to the implementation. We motivate the idea of a *commit record* as the implementation of a possibility, and contrast it with some similar mechanisms that achieve solutions to the atomic commit problem. After this discussion, the mechanism by which tokens become versions or aborted versions is described, and the effects of failures are shown. We then discuss enhancements to the basic commit record mechanism. Use of the commit record to ensure atomic commitment of tokens introduces problems of delay and inaccessibility that can be reduced by encaching the state of commit records with the token. Further reduction of delay can be achieved by enhancing the commit record mechanism to automatically distribute changes of state at the earliest possible time.

The implementation of possibilities is then described, first for an implementation where the possibility is represented by a single commit record at a single site, and then for an implementation where the possibility is represented by multiple commit records "distributed" among several sites to provide a greater degree of availability. Finally, the storage management mechanisms by which the storage used to implement possibilities can be reclaimed will be discussed.

5.1 Atomic commit

An "atomic commit" mechanism is one that causes some set of actions to happen "simultaneously" as far as any outside observers are concerned. In the case of the decentralized system, the actions that are to be performed "simultaneously" are transformations of some set of tokens from tokens to versions. In fact, simultaneity is not the important factor, though it is often

discussed in this way. The requirement is that of the two possible results for each object token, version or aborted version, the same choice gets made eventually at all object tokens. A mistake leading to one token becoming a version and another one becoming an aborted version would lead to an inconsistency in the system.

The possibility is the abstract mechanism used for achieving atomic commit. All of the tokens created associated with a particular possibility form the set of objects affected by the atomic commit. The **possibility$complete** operation is an attempt to change all such tokens associated with the possibility into versions, while the **possibility$abort** operation is an attempt to change all such tokens into aborted versions.

Some computation in the system will be the *causal agent* of the atomic commit. That is, the causal agent is the computation that is intended to complete or abort the possibility. It is important that the system protect itself against the loss of the causal agent (or alternatively, excessive delay in deciding what to do, or a failure of communications between the causal agent and the set of actions to be performed). Thus, designed into the interface to possibilities is the idea of a timeout causing a default decision to abort the possibility. In the case where the causal agent either fails or takes an excessively long time to act, the possibility automatically goes to the aborted state. Then, no matter what action the causal agent may take (if it has not really failed) it cannot reverse the decision to abort.

In my system, an atomic commit mechanism is built out of a mechanism that is specialized to the purpose. This mechanism we call a *commit record.* A commit record is a piece of reliable storage that is created with a fixed initial state, then can be changed by an atomic action to either one of two other states, and can never again be changed. Basically, it is a piece of

non-volatile write-once storage that can be in one of three states. A commit record's state has one of three values at any instant that it is read -- call them waiting, complete and aborted, in analogy to the states of possibilities. Further, if any read of the commit record's state returns either the value complete or the value aborted (call these final values), then all other observations will return either that value or the value waiting. It is not necessary that reads be ordered with respect to writes, although in most real implementations once a final value has been returned, later reads will never return the value waiting.

Lampson and Sturgis[Lampson76] have described a mechanism for achieving atomic commit. Their mechanism differs in some detail from that used in NAMOS, and uses a slightly different basic mechanism. Their basic mechanism is what they call *atomic stable storage.*[1] The property of *atomic stable storage* is that it behaves as if any attempt to write the storage either finishes or it doesn't start (leaving the storage unchanged). Thus any read returns a value that was written by the most recent write to complete, rather than allowing the possibility that failing or concurrent writes will cause reads to obtain values that mingle bits stored by several different writes. Commit records have this property, but in addition are also write once, and have the range of values restricted.

Lampson and Sturgis describe a way of achieving atomic stable storage by using a disk with each record stored twice with an error detecting checksum. The atomic property is built by using a clever reading algorithm that reads both records, checking the checksum and comparing

---

1. Atomic stable storage is a further enhancement of the stable storage described in chapter two. The distinction is that with atomic stable storage there is no possibility that a modification to the storage will be left partially completed -- either the modification will never have happened, or it will have been completed.

the two copies. If the checksum on the first record is correct, but it differs from the second copy, then the first copy is correct, and the second copy is old, so it is copied from the first copy. If the checksum on the first copy is wrong, then the second copy is correct, and is returned. Otherwise, either copy will be correct. Their algorithm as described requires that writes be mutually exclusive, but this is easily achievable in a single computer system. The lock mechanism described in chapter two can be used to achieve this mutual exclusion.

This mechanism could also be used in the implementation of commit records, by simply ensuring that the commit record is only written once. However, other implementation strategies are possible.

Gray[Gray77] has described another mechanism for a centralized system to achieve atomic commit. Recovery from failure in his system depends on the fact that the whole system stops upon a failure, so that a recovery program can be run to back up the state of the variables changed before the failure, while no other computations can be observing the values created by the failing transaction. Because it assumes that the whole system is serviced by a centralized recovery algorithm that always gains control upon failure, and which can know how to restore all the states of objects touched by failing computations, it is not an acceptable approach for a decentralized system.

5.2 Tokens

The mechanism by which tokens become apprised of the change to a commit record is based on a passive mechanism. Stored with a token is information sufficient to designate and locate the commit record associated with the possibility under which it was created. Whenever it

is necessary to check whether a token should really be a version or an aborted version, the system containing the commit record sends a query message to the system containing the commit record. The response generated for such a query will be the state of the commit record -- waiting, complete, or aborted.

Since the commit record can never have been in both the complete state and the aborted state, if the answer to the query is either complete or aborted, all other queries referring to the same commit record will get either the same final answer, the answer "waiting" or no answer. A waiting response has basically the same information about the state of the token as no response at all. Since a waiting token is neither completed nor aborted, it is unknown what the final state of the token will be. Between the time the waiting response was generated and the time the site of the token receives the response, the commit record may have been set to one of the final states. However, once one token has received a final answer, all other tokens referring to the same commit record will receive the same answer if they wait long enough (assuming that the site containing the commit record and communications to it do not fail permanently).

The mechanism described so far has a built-in delay, since determining the state of a token always requires sending a message to the commit record and waiting for a response. In addition, the commit record must occupy storage for a long time, since it must always be present to decide for each token whether it is a version or not. We would like to be able to reclaim the storage for a commit record relatively quickly. Finally, for the period that a commit record is inaccessible, all tokens that refer to it appear inaccessible as well. Thus the likelihood of inaccessibility of a token may be very high, even after the token has become a version. We would like the extra inaccessibility resulting from the use of commit records to be as small as

possible.

To improve the commit record mechanism, reducing delay, storage requirements, and inaccessibility problems, we introduce the notion of encaching the commit state in each token. In addition to the location of the commit record, the last known state (initially waiting) is stored with each token. Call this part of the token the ECS (encached commit state). Whenever a response from a commit record is received, if the response is complete or aborted, that value is stored in the token's ECS. Thus, once the commit record signals a state of complete or aborted, no further queries are needed to ascertain the status of a token. In the absence of errors, each token need wait for only one query and response after the commit record is set to the final state. Delay is thus reduced, and for those tokens that have already copied a final state, any inaccessibility of the token is irrelevant.

There is a basic problem, however, in that there is always a period of time between the time the commit record enters its final state and the time that the tokens dependent upon it know what the final state is. Worse yet, due to the possibility of errors, this time cannot be bounded. Thus, even if the token is accessible some of the time, and the commit record is accessible some of the time, the token's state may be inaccessible for an unbounded period of time.

Is this a bug? Unfortunately not -- it is due to the same basic difficulty that makes the two generals unable to bound the time needed to make a decision. If were to accept a non-zero probability of making a wrong decision, the delay could be bounded. However, given that we always want to make the right decision, the maximum delay is unbounded.

Delay in encaching the information from the commit record can be further reduced by eliminating the delay due to the query sent by the token. If the response is labeled so that receipt of it before any query can be properly handled, then once the commit record reaches a final state, "unrequested responses" to queries about the status of the commit record can be sent to all tokens interested in the state of the commit record. These responses can be viewed as a pure optimization, since ignoring them does not cause incorrect operation of the system. However, if they are reflected in the ECS of each token that correctly receives the "unrequested response," then the delay due to querying the commit record after it is final will be non-existent.

Keeping track of where the tokens interested in such "unrequested responses" are can be done in several ways. It turns out that much the same information is needed to reclaim the storage used by commit records safely, so I defer the discussion of this to later in the chapter.

5.3 Possibility implemented as a single commit record

The simplest implementation of a possibility is based on storing the state of the possibility as a single commit record at a single site. This site serves as an arbiter, funnelling the manipulations of a possibility through a single queue that processes each request to manipulate a particular possibility in the order of arrival. What, then, does a commit record look like in such an implementation, and how is it manipulated?

The commit record is stored in stable storage at the site. Crashes of the site do not affect the value of the commit record. As noted in chapter two, such stable storage is, in practice, not perfectly achievable. However, the probability of failure of some devices, such as disks, optical

memories, and others, can be made quite low at reasonable cost by judicious use of redundancy local to the site.

The commit record contains at least the following fields:

commit_state: a two bit quantity, set to zero when the commit record is created. The first bit, commit_state.c, is set to one in a possibility$complete operation. The second bit, commit_state.a, is set to one in a possibility$abort operation, or upon timeout. The possible values are:

00 waiting. The initial state.

01, 11 aborted. Aborting takes precedence over completing.

10 completed.

timeout: a clock time accurate to some precision (say 1 millisecond), set to the time after which the commit state should be turned from zero to the aborted value, if not already completed.

The timeout is never changed, once the commit record is allocated. The commit state is write-once storage, that is written atomically. The atomic write property can be achieved in several ways. As an example, one might use write-once memory built by having a laser burn a hole in a piece of mylar as in the Ampex terabit memory for each one bit. Another possibility might be to use the same trick used by Lampson and Sturgis, storing two copies of the commit state on disk.

There are three types of operations that can be requested for a commit record -- **complete, abort,** and **test.** They are implemented quite simply. The only detail that I have not mentioned is that the clock used as the standard against which the timeout is compared is the local site clock at the site of the commit record. It is really quite unimportant what clock is used,

as long as the clock will reproduce the intended timeout reasonably accurately. If timeout happens early or late, the logical correctness of the system is not violated, but performance may suffer (early timeouts prevent normal completion of operations, while late timeouts result in delay when checking a token created by a failed operation). The algorithms are:

complete

0. set commit_state.c = 1

1. return **not**(commit_state.a)

abort

0. if commit_state.c=0
   then set commit_state.a=1

1. return commit_state.a

test

0. if timeout > time()
   then call **abort**.

1. return commit_state

In the implementation of these operations, mutual exclusion is necessary. Step 0. of the **abort** operation must be atomic with respect to other **complete** or **abort** operations. A simple way to insure the proper mutual exclusion is by having a lock that must be set to gain access to read or write a particular commit record. The lock is cleared at the end of operations, and cleared also whenever the site comes up after a crash or power-off. The locking mechanism for single objects described in chapter two is sufficient for this purpose. Since these operations are short, the overhead of locking need not be great, consequently an alternative scheme that uses a single lock for all commit records at a site may be quite acceptable.

If the storage to be used by commit records is to be re-used, the addressing scheme used for commit records should incorporate a mechanism for detecting dangling pointers. Consequently, the name used to designate a commit record at a remote site might include not only information sufficient to designate the storage address, but also a unique tag (generated by reading the local site clock, perhaps, if a clock of sufficient resolution is available). Another field in the commit record would contain the unique tag. If the tag in the name for a commit record does not match that stored in the commit record, then the commit record has been deleted. The meaning of a deleted commit record when discovered by a query from a token will be discussed when we discuss reclamation of commit records.

If a possibility is implemented by a commit record stored at a single site, the continuing accessibility of that site to the sites containing the created tokens may be a question of some importance to the nodes creating tokens, since the token may prevent transactions that manipulate the object containing the token from proceeding if the commit record is inaccessible. For this reason, a node may choose to refuse to create a token based on its lack of trust of the accessibility (or correct implementation) of the commit record based on its location. Nodes that provide highly accessible, correctly implemented commit records will be one of the necessary components of the system I propose. These nodes may either be provided as part of the network, or they may be provided by mutual agreement among those nodes that have resources that might reasonably be used together. Assuring the correctness of and availability of such nodes must be done by observations and agreements outside the system.

5.4 Dependent possibilities

There are at least two ways that dependent possibilities can be implemented. In one way, a dependent possibility is a commit record as above with an additional information field that refers to the possibility depended upon. The possibility$complete and possibility$abort operations on dependent possibilities would be exactly the same, manipulating the commit state field of the dependent possibility's commit record. The possibility$test operation, however, would be different. The test operation on a dependent possibility that has been completed must involve testing the state of the depended-upon possibility. If the depended-upon possibility responds that its state is completed or aborted then the the dependent possibility can return that value.[1] If the depended-upon possibility is waiting, then a new kind of response is sent, indicating that the dependent possibility is completed, and naming the depended-upon possibility. The depended-upon possibility's name then replaces the dependent possibility's name in the token, so that future tests after waiting bypass the dependent possibility.

An alternative way to implement dependent possibilities is to represent the name of a dependent possibility by a pair of names. The first element of the pair identifies the commit record that is modified by possibility$complete and possibility$abort operations, and the second element identifies the depended-upon possibility.[2] The possibility$test operation then consists of testing the commit record that is first on the list using the basic test algorithm defined in the previous section. If waiting or aborted, then the test finishes immediately, returning waiting or

---

1. To reduce delay on successive tests, the dependent possibility can also encache the state of the depended-upon possibility.
2. If the depended-upon possibility is a dependent possibility, then its name will be a pair also, and so on, until an independent possibility ends the chain.

aborted. Otherwise, the pair is replaced by the second element of the pair, and the test is repeated.

The second approach suffers from the need to store potentially rather long names for possibilities in each token. However, once the token is turned into a version, there is no need to store the name of any possibility, so the storage is only required during the window of time while the token has not yet become a version. The main advantage of the second approach is that the protocol for dealing with commit records is simple and uniform -- all of the complexity is localized in the algorithm executed at the token.

5.5 Determining the right to access a token

The representation of the name of a dependent possibility as a pair consisting of the name of a commit record and the name of a possibility (dependent or independent) is very useful in determining whether a computation may obtain the value stored in a token, however. Recall from chapter three that a computation that attempts to read a token may obtain the value if the possibility in which it is executing is "the same" as the one the token was created in, where "the same" has a very particular meaning. Algorithmically, the phrase "the same" can be understood to mean "search up the chain of dependent possibilities from the token until the first non-completed possibility is found. Then if that possibility is a member of the chain of possibilities under which the computation is executing, the token's value may be accessed by the computation." As a side effect of either implementation of dependent possibilities, the test operation causes the token to know the name of the first non-completed possibility. This name can then be checked against the name of the possibility under which the computation is executing to see if it is a member of the chain. If the possibility under which we are executing is

represented in the pair (or list) representation, this check is simple and local. If only the name of the topmost dependent possibility is known, then it would be necessary to inquire of that possibility whether the non-completed possibility matches any in the chain, possibly requiring a flurry of messages between sites containing the various commit records.

5.6 Possibilities implemented using multiple commit records as voters

In order to increase availability, one might desire to try to implement possibilities using several commit records at different sites. A number of algorithms that attempt to implement multiple-site updates of data stored redundantly at multiple sites are given in the literature[Alsberg76,Johnson75,Thomas76], however, these algorithms are too elaborate for the rather simple properties desired of possibilities. A modification of the majority rule approach developed by Johnson and Thomas[Johnson75] seems to provide an effective approach, however.

The problem is to implement an object whose representation does not require that a particular site be up to read from or store into it. The behavior of the object must, however, be the same as a commit record, in that the complete and abort operations should be atomic. The essence of the majority rule solution is as follows. The state of a commit record is a composite of the state of N *voters*, commit records stored at N different sites. The state of each voter is either waiting, completed, or aborted (just as before). When a majority of the N voters are in either the completed state or the aborted state, then the state of the possibility is the same as the state of the majority. In order to prevent a tie, the case where exactly half have voted for the completed state is defined to be a completed state of the possibility.

Voters are not allowed to change their vote -- thus they have a state that is write-once. Further, to handle the timeout, if a voter has not decided which state to vote for after the timeout has been exceeded according to the local site clock, the voter must vote for the aborted state. Thus, in behavior, a multi-site commit record is created out of data structures that look just like the single-site commit record described above.

Testing the state of a multi-site commit record involves taking a poll of the N sites. Because only a majority is needed to make a decision, the minimum number of sites that must be accessible to make the test will be $\lfloor N/2 \rfloor$ (in the case of a decision to complete) or $\lfloor N/2 \rfloor + 1$ (in the case of an abort). However, the maximum needed to be accessible to make a decision is N, in the unlikely circumstance that the vote is nearly a tie and the voters are queried in worst-case order. Thus, we want to minimize the likelihood of a tie.

I will generalize the notion of majority rule a little bit, since there is really not some magic property of N/2 that ensures that the multi-site commit record works. It is only necessary that there be some threshold value κ ($1 \leq \kappa \leq N$) for the decision whether the possibility is completed or aborted. If κ voters are completed, then the test says completed. If $N-\kappa+1$ voters are aborted, then the test says aborted. Otherwise, the state of the possibility is waiting. I call κ the *complete threshold* and $N-\kappa+1$ the *abort threshold* (if N=1, both the thresholds are required to be 1, so this case is the same as the single commit record implementation).

To execute a possibility$abort or possibility$complete operation on the multi-site possibility, the site requesting the abort or complete sends messages to all of the voters representing the possibility (some messages may be lost, of course). Each message is processed at the receiving site, and the voter is set to the completed or aborted state as required, unless it has

already been set. The requesting computation does not proceed until all voters have been contacted and have responded, or until a sufficient timeout has expired due to inaccessibility of one of the voters (the requesting computation retransmits requests to decrease the probability of lost messages).

In the case that the computation requesting a possibility$complete on the multi-site possibility fails in mid-stream, before sending some requests to voters, each voter independently times out, based on a local clock, entering the aborted state if no complete request has been received promptly enough. The timeout at each voter site also handles the case of a site that was inaccessible to a computation that requested a possibility$complete. The voter at the inaccessible site will eventually enter the aborted state.

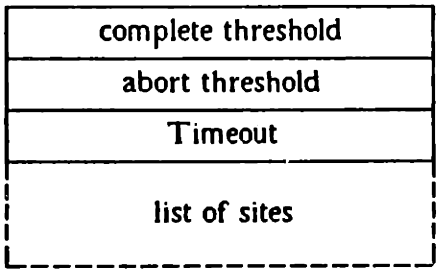
There are two cases that lead to a high likelihood of having a near-tie in the state of the voters. One case is that of near simultaneous abort and complete operations. If an abort gets to approximately $N-\kappa+1$ of the voters first, while a complete gets to approximately κ, a near tie occurs. In the use of the system, it is easy to avoid the circumstance. First, the abort operation is logically unnecessary because of the timeout, so it should only be used in order to speed up the process of aborting when it is *known* that no complete operation may be attempted. Second, in designing programs, it is normally the case that the execution sequences that explicitly abort will never go through an attempt to complete (as in all the chapter four examples).

The second case is when a complete operation happens to signal the voters at around the time of timeout. Thus, some of the voters may have timed out already when the complete arrives. The probability of this circumstance arising can be drastically reduced by a simple trick. If the clocks local to each site are reasonably synchronized (with some small probability of being

more than ϵ seconds out of synch), and if network delays are roughly bounded (with some small probability of being more than δ seconds), then if a complete is attempted within $\delta+\epsilon$ seconds of the timeout on the possibility, it should be refused out of hand at the requesting site, before sending anything to voters (in fact, it may send aborts just to speed up the aborting process). Thus the probability of a timeout overlapping a complete can be made small by making the timeout somewhat bigger than the expected time to complete the operation done under the possibility.

In order for these algorithms to work, the requesting site must have enough knowledge to locate each voter of a multi-site possibility separately -- so the name of a multi-site possibility may be somewhat larger than names of objects (that naturally have a single home). In addition, for the refinement just mentioned, the requesting site must be aware of the timeout set on the possibility. Also, the requesting site must know κ Thus a reference to a multi-site commit record looks like figure 10.

Fig. 10. Reference to Multi-site Commit record



In appendix A, an analysis of the probability of availability of a possibility implemented as multiple voter commit records is given. The basic conclusion is that as you increase N, two factors affect availability. First, if the probability of failure during a possibility$complete operation is low, the likelihood of a near-tie decreases as N increases. Opposing the first effect, however, is a second one. The probability of failure during a possibility$complete operation increases as the number of sites to be notified increases. This can increase both the likelihood that a near tie will occur, and the likelihood that the possibility can't be completed. In addition, the cost in terms of messages sent, load on the network, etc. increases as N increases. Thus, there is probably an optimal value of N which is relatively small, given normal site availability and network reliability.

If κ is about half of N, it is equally easy to find out that the possibility is aborted as completed, whereas as κ decreases, determining that the possibility is completed becomes easier and easier. Since the normal case is that operations are completed, smaller values of κ ought to increase performance overall, reducing the number of responses needed to determine the state of a possibility. Values of κ greater than one half of N don't seem to be useful, though.

5.7 Reclamation of commit records

So far, commit records have been considered to be permanently allocated. The cost of permanent allocation of commit records can be very large, since every transaction or other operation involving shared objects will require a distinct commit record to represent the possibility associated with the operation. Consequently a scheme to reclaim commit records when they are no longer needed is desirable.

Since the state of commit records is eventually encached within the versions and aborted versions stored as part of the object known history, it is clear that eventually there will be no references to a commit record. When there are no references from tokens still outstanding it is quite safe to delete a commit record. It is not a good idea to delete a commit record while it is still in the wait state, though, since some computation may still refer to the commit record through the possibility mechanism. That computation may yet create new tokens that depend on the possibility for realization.

The key issues in implementing a deletion mechanism for possibilities are:

1. It must be possible to detect or create the situation that no token exists that has a reference to a particular commit record.

2. Since computations may refer to commit records through possibilities, the use of possibilities that refer to deleted commit records must be prevented.

It is extremely dangerous to delete a commit record that has not had its state encached in all of the tokens referring to it. The result of such a deletion would be to make all such tokens permanently inaccessible, thus blocking read operations on the known histories containing those tokens. Consequently, any deletion algorithm should err only by not deleting unneeded commit records, never by deleting needed commit records.

Because of the modularity of programs, at the time a possibility is completed the program doing the completing may not be aware of the entire set of references to the commit record that have been generated by operations invoked in the possibility. For this reason, an explicit delete-commit-record operation that can be used by application programs is not a reasonable solution. Even if the program could figure out the set of references to the commit record that

exist in objects, the possibility of programming errors in using the explicit deletion would argue against such a strategy, the cost is quite high if such a mistake is made.

A commit record must thus contain enough information to allow determination of when it is safe to delete the commit record. Then the site containing the commit record can be responsible for ensuring that the commit record is deleted only when no references to it exist in tokens. If the commit record site behaves improperly, deleting it even when the site knows that references exist, tokens depending on the commit record will become permanently locked up. For this reason, a basic tenet of the system is that any site creating a token must trust that the site implementing the commit record associated with the token does so correctly. If a non-trustworthy site is proposed as the home for a commit record, the token may simply not be created, and an error response be generated.

When a token is created, it is necessary to know that the commit record on which it is to depend (a) still exists, and (b) is not completed or aborted. To insure (a) and (b), the site creating the token must first inquire of the commit record its state. If either condition is not satisfied, the token is not created, and an error is returned. Since a message exchange with the commit record is already needed, we can use the message exchange to tell the commit record that a token referring to the commit record is about to be created. Thus, when a commit record receives such a request, it can record the name of the token (object name together with pseudo-time it is *valid from*).

It is not too difficult for a commit record to obtain the list of all tokens that depend on it. To use the list, the commit record must be aware when a token on the list no longer depends on it. This can happen three ways.

1. A token depending on the commit record successfully encaches the state of the commit record.

2. A token depending on the commit record is deleted.

3. A token listed as depending on the commit record was in fact never created (the list is formed before tokens are created, so a failure after a token was entered on the list could prevent the creation of the token).

When a token encaches the commit record state or is deleted, a message can be sent to the commit record indicating that the reference is deleted. However, a failure in the sending of this message would result in not deleting the commit record even though it is deletable. Consequently, the primary means of checking the deletability of the commit record is by polling from the commit record. The commit record, once it reaches a final state can poll the objects named in its dependency list.

A trick that can be used is to encode an aborted commit record by immediately deleting it. Any request for the state of the commit record after it is deleted can be answered by saying "no such commit record exists." If the request was from a token, it knows that the token was not completed, and since commit records in the wait state are not deleted, it must be aborted. However if the request for the commit record comes as a result of an attempt to create a token, the token will not be created.

The messages between tokens and commit records are thus the following.

T->CR create-ref(CR-name, token-name): query to commit record whether token referring to the commit record may be created. CR-name is the name of the commit record queried, while token-name is the name to be used to refer to the token.

CR->T permit-create(CR-name, token-name, ok?): response to create-ref. If ok? is yes, then the commit record was in the wait state, and an entry was added to the list of tokens referring to the commit record. Otherwise, the commit record was in either the complete or aborted states. A deleted commit record also returns a "no" response. If no response is received, the create-ref may be retransmitted.

T->CR test(CR-name, token-name): message that is sent by a site containing a token to get the state of a commit record, in order to decide whether the token value can be returned as the result of a read or not.

CR->T state(CR-name, token-name, s): sent either in response to a test() or when it is desired to delete the commit record after it has been completed. In the case of a deleted or aborted commit record, s is "aborted". If the state of the commit record is complete, then s is "complete". If the state is waiting, no state(s) response is generated.

T->CR no-ref(token-name): sent when a state("complete") message is received, once the token encaches the fact that the commit record is complete or immediately if the token has already been deleted (see chapter six for details about deletion of tokens). The commit-record deletes the token named token-name from its list of tokens referring to the commit record.

The no-ref message allows the deletion of commit records. All of the other messages are needed already for the function of commit records, independently from deletion. Once a commit record enters the complete state, it immediately starts polling the dependent tokens by sending state("complete") messages to all the tokens on its list, until such tokens are all removed from the list by no-ref(token) messages. Once such tokens are all removed, the commit record can be

deleted. A nice property of this scheme is that it reduces delay in encaching the state of commit records into tokens, since an immediate attempt is made to broadcast the commit record state to all dependent tokens.

Multi-site possibilities can be managed by a modification of the same mechanism. However, individual commit records making up the possibility cannot be deleted until the possibility as a whole is aborted, or until the possibility is completed (as a whole) and that fact is encached in all tokens referring to the possibility. The fact that the possibility as a whole is aborted or completed is normally detected by some token as the result of a number of messages from individual voters specifying their state. The token will then encache the state of the possibility. In the single commit record implementation, the token would respond to the state() request with a no-ref() request to signal that no reference remained to the commit record from the token. In the single commit record case, the no-ref() request is not needed if the state of the possibility is aborted, since the commit record could be immediately deleted no matter how many references existed to it. However, since the token is the only entity with knowledge about the state of the possibility as a whole, the token must initiate the deletion of the commit record.

Consequently, in the multiple commit record implementation, a new request is required -- the overall-state(CR-name, token-name, state) message, sent from the token discovering the overall-state to at least one of the commit records (it may be sent to any number as an optimization). As a result of receiving an overall-state message, the commit record receiving it knows the overall state of the possibility. This knowledge is sufficient to initiate deletion. It also can be used to optimize further requests from tokens -- since the overall state is known at a commit record, that commit record can act in behalf of all other commit records and unilaterally

specify the state of the entire possibility. We can take advantage of this optimization by allowing two additional values to be returned by the state() request from the commit record to the requesting token -- one meaning "the possibility is completed overall" and the other meaning "the possibility is aborted overall". These messages have the effect at the token of immediately deciding the voting without further tallying of votes from other voters. Once a commit record knows that the possibility is aborted, it can delete itself (thus discovery of a deleted commit record in a possibility means that the possibility is aborted overall). However, if the possibility is aborted, all tokens referring to the possibility must be known to have encached the state of the possibility before any of the commit records are deleted.

The set of tokens referring to commit records representing a possibility is the union of the private lists maintained by each individual commit record as the result of "yes" permit-create responses, minus those tokens that have sent overall-state() messages to one or more of the commit records. A relatively simple algorithm for determining when the commit records can be deleted is to have the recipient of an overall-state message poll all other commit records to determine the entire set of tokens that have references to the possibility and to collect the set of overall-state messages so far received at other commit records. Then the recipient of the overall-state message can poll all of the tokens by sending a state("overall complete") message to each token. In response, each site will send back an overall-state("complete") message to confirm (note that this always has to be sent, even if the token has been deleted) that the token has encached the state and no longer will refer to the possibility. When the set of tokens referring to the possibility has been determined to be empty in this manner, then the commit record that determines it can signal all others to delete themselves and delete itself. If any of the messages are lost at this point the

others may not know to delete themselves. Since no tokens refer to the commit records any more, the commit records would never be deleted.

To guarantee that all of the commit records are deleted eventually, we need to have each commit record poll periodically to see what the state of the other commit records making up the same possibility is. For this purpose, we can organize the commit records into a "ring" so that each commit record knows about a left and right neighbor commit record. Periodically, each commit record polls both its left and right neighbors with a test operation. If it gets an "overall aborted", or a "deleted" response, it deletes itself. If it gets an "overall completed" response, then it marks itself as knowing the commit record is "overall completed" as well. Any other response can be ignored.

As noted above, when a token is created, it sends out create-ref requests to all of the voters. Each voter that receives such a request records the name of the token that might be created (the object id and the pseudo-time of creation) on a private list to be used in the deletion of the commit record. The permit-create("yes") response is returned by a voter if the voter is still in the waiting state, otherwise a "no" response is sent. In order for a token to be created, the possibility as a whole must be in the waiting state. Thus, a token may be created only if $\max(N-\kappa+1,\kappa)$ voters send back "yes" responses.[1] Each successful token creation will thus result in at least $\max(N-\kappa+1,\kappa)$ entries among the private lists of references maintained by the commit

1. Actually, if the permit-create request were modified to indicate whether the present state of the voter was completed, aborted, deleted, or waiting, then the number of responses needed to create could be somewhat reduced -- the token need only prove that the possibility is not yet out of the waiting state. This optimization is not important, since it only makes things better when the possibility is in the middle of changing its state to either the completed or aborted state overall, and the ability to more speedily create a token at the point after its possibility is being forced into its final state is of dubious value at best.

records making up the possibility. Thus, when taking the union of the lists to determine the set of tokens referring to the possibility, one need only include those tokens included $\max(N-\kappa+1,\kappa)$ times.

5.8 Summary

In this chapter we have discussed the implementation of possibilities in terms of commit records. Two basic strategies have been described -- a single commit record implementation and a multiple commit record implementation. The single commit record implementation is much simpler and more efficient, though it does have the drawback of decreasing the availability of tokens since the availability of a token depends both on the availability of the site containing the token and the site containing the commit record (and the communications in between). Although the availability decrease due to use of a commit record can be made smaller by broadcasting its state once it changes and encaching the state in the token, still, there is a window during which the availability of the token depends on the availability of the commit record, and the size of this window cannot be reduced.

The multiple commit record implementation increases the availability of the possibility as a whole during that window, at the cost of a more expensive (in terms of messages sent, storage, and CPU cycles) and more complicated implementation. We conclude that the multiple commit record implementation should only be used when availability is more valuable than the costs of implementing the multiple commit record implementation. It is possible for both implementations to coexist in the same system, with the multiple commit record implementation to be used only for those operations that deal with objects that have high availability requirements. It would be perfectly reasonable for a request to create a token for an object to be rejected by the manager of

the object on the basis that the possibility controlling the conversion of the token into a version has not been implemented with sufficient availability guarantees.

Chapter Six

Implementation of Objects: Known Histories, Versions, etc.

This chapter completes the description of implementation of the system. In chapter three, the behavior of objects was described in terms of known histories that provide a mapping from pseudo-time to versions valid in particular pseudo-times. Many issues that are important in a practical implementation were not discussed in chapter three, and will be discussed here. Some of the issues are:

- ★ maintenance of pseudo-time at multiple sites.
- ★ maintaining the set of versions in the known history.
- ★ creation and deletion of objects.
- ★ storage reclamation of versions that are out-of-date.
- ★ data objects that can have special implementations.
- ★ copying versions of objects (encachement).
- ★ handling related groups of objects, by "paging."
- ★ reducing the likelihood of dynamic deadlock.

First, the relationship of pseudo-time to real time, and the mechanisms of implementation of pseudo-time will be explained. The next few issues have to do with the representation and manipulation of objects at their home node. First a set of mechanisms are developed that can implement what I call a cell data type, or what might be also referred to as a mutable record

type. This is a universal data type, out of which any other type can be built. However, there are possible better implementations for particular data types -- two of these types in particular will be discussed, queues and accumulators. Finally, certain enhancements to the mechanism that can be used to optimize special performance problems are discussed -- a space and delay problem due to computations dealing with many small objects is solved by a grouping called "paging", a delay problem when dealing with primarily read-only data is solved by encachement of versions of objects, and a delay problem when there are computations that manipulate the same objects close together in time is ameliorated by an optional mechanism called *token reservations*.

6.1 Representation of Pseudo-times and Pseudo-temporal Environments

In chapter three, we discussed the properties required of pseudo-times and pseudo-temporal environments. Pseudo-times are values belonging to an ordered set. Pseudo-temporal environments are objects that keep track of the progress of a particular computation. We require that the pseudo-times used for updates executed as part of two sequential steps in a computation be ordered, such that if step A precedes step B, all pseudo-times used in A for updates precede all pseudo-times used in B for updates. In other words, the pseudo-times given out by the pte$next operations executed on a pseudo-temporal environment must be given out in increasing order.

A pseudo-temporal environment essentially is a way of keeping track of the largest pseudo-time used so far in its associated computation. The pte$next operation simply returns a still larger pseudo-time and changes the pseudo-temporal environment to reflect the new largest pseudo-time used.

There is further structure to pseudo-time and pseudo-temporal environments, however, due to the existence of concurrent computations. We wish to guarantee that two concurrent updates to primitive objects do not execute in the same pseudo-time, so that any two updates to the same object are totally ordered. More strongly, we wish to guarantee that the pseudo-times used for updates in two independently executing (possibly concurrent) transactions are ordered such that all pseudo-times used in one transaction follow all pseudo-times used in the other.

At the top level, there are a set of concurrently executing computations that are executing independently, that is, with no *a priori* relationship of the ordering between steps in distinct computations. Each of these computations has its own pseudo-temporal environment from which it gets pseudo-times, by pte$current and pte$next, and pseudo-temporal environments in which to execute multi-step transactions. For the moment, let us ignore the complication introduced by multi-step transactions, and think about a transaction as dealing with two pseudo-times, an input state reference which the transaction uses to read all of inputs, and an output state reference which the transaction uses for all of the updates it eventually makes. What we want is for a transaction executed in a top-level pseudo-temporal environment to be atomic with respect to all other transactions -- that is, the pseudo-times between the input state and the output state are not used for updates by any other transaction.

A top-level pseudo-temporal environment is a list of two integers, **last-time** and **pteid**. Each top-level pseudo-temporal environment is assigned a unique value of **pteid**. The **last-time** field is used to guarantee that the pseudo-times resulting from successive pte$next operations in a computation are monotonically increasing.

Pseudo-times returned by pte$next in a top-level environment are obtained by reading a clock (about which more will be said later), waiting until the clock value is greater than the value of **last-time**, then updating **last-time** to the clock value, and returning a top-level pseudo-time consisting of a list containing the new **last-time** and the **pteid**. The **pteid** guarantees that pseudo-times from pte$next in different top-level pseudo-temporal environments are distinct.

In general, a pseudo-time is represented by a list of integers. Two pseudo-times are ordered by finding the first position in which the pseudo-times differ (if the lists are of different lengths, then the shorter list is thought of as being extended with zeros to the length of the longer), and using the ordering of the integers at the position of difference. Thus, for example, two pseudo-times resulting from different pseudo-temporal environments by pte$next are ordered first by the times obtained from the clocks read, and if the times happen to be equal, the tie is broken arbitrarily by using the ordering of **pteids**.

Now, how are pseudo-temporal environments derived from top-level pseudo-temporal environments? All pseudo-times obtained from such a derived pseudo-temporal environment must belong to one tick of the top-level clock. The trick is to add "low order digits" to pseudo-time, such that the increases in pseudo-times returned from a derived pseudo-temporal environment are confined to these "low order digits". Thus, all the pseudo-times obtained by pte$next and pte$current from a pseudo-temporal environment derived using pte$transaction from a top-level pseudo-temporal environment are four element lists, where the first two elements are the same for all such pseudo-times, and specify the "instant" of top-level pseudo-time. The third elements of successive pseudo-times derived in this way are in increasing order. The fourth element is always zero for a transaction pseudo-temporal environment, but in one of a set

of parallel executions resulting from executing pte$paraction, the fourth element uniquely identifies the particular derived pseudo-temporal environment.

A derived pseudo-temporal environment has three parts, **last-time**, **pteid**, and **time-derived**. The new field, **time-derived**, specifies the "instant" of pseudo-time in which the computation carried out in the derived pseudo-temporal environment executes. For a pseudo-temporal environment derived from a top-level pseudo-temporal environment, **time-derived** is a list of two integers. The pte$next operation works as before, but returns a value that consists of **time-derived** concatenated with **last-time** and **pteid**.

We can handle any depth of derivation with this structure -- **time-derived** just is a longer and longer list. To execute pte$transaction in some pseudo-temporal environment, we get a new time, L, larger than the **last-time** field in the derived-from pte. The new pte is constructed by setting its **last-time** field to L, its **pteid** to zero, and making up a new **derived-time** by concatenating the **derived-time** field of the derived-from pte with (L-1) and the **pteid** of the derived-from pte. The pte$paraction operation constructs the derived pte's in a similar way, such that the derived pte's differ initially only in the **pteid** field.

As a practical matter, we can observe that individual steps in a computation are unlikely to execute more frequently than once every microsecond or so, on today's hardware, so a resolution of one microsecond on the integer clock values is sufficient. Similarly, it is unlikely that pseudo-times referring to the state of objects more than a few years in the past will be in use. So the integer clock times can easily be encoded in a fixed 48-bit field, by storing the time in microseconds since some fixed reference time, modulo 2^{48}. With the exception of top-level pseudo-temporal environments, the **pteid** field need be no more than 8 bits or so to distinguish

all of the parallel executions created by one paraction call. Top level computations can be uniquely identified by their time of creation, which can again be encoded in 48 bits. So a pseudo-time could be easily encoded in 96 + 56n bits where n is the number of levels of derivation. In a system that did not make use of derived levels, i.e. one that does not support the modular composition of transactions, a fixed size 96-bit allocation would suffice.

In this representation, if we ignore the modulo-2^{48} encoding (by limiting the lifetime of the system to 2^{48} microseconds), we can think of pseudo-times as simple binary fractions on the interval [0,1). Pseudo-temporal environments for two transactions such that one is not derived from the other can be though of as disjoint subintervals. Two parallel actions created by the same pte$paraction operation can be each thought of as the union of disjoint subintervals (corresponding to sub-transactions), such that the union of the parallel action ptes is a subinterval of the pte from which it was derived.

6.2 Maintaining the time - pseudo-time relationship

In conventional systems, operations make their changes always to the "current" state of the system; what is needed is to ensure that a notion of the current state be embodied in the construction of NAMOS. In NAMOS, the pseudo-times obtained from top-level pseudo-temporal environments function as the current state would in the more conventional system.

To achieve this behavior, a mechanism is needed for choosing the instant of pseudo-time in which a transaction is executed that is later than the pseudo-times of operations that execute before the current operation in real time. One possible mechanism would be to use a central

service that gives out pseudo-time values in increasing order -- essentially a central *sequencer* as defined by Reed and Kanodia[Reed78]. The problem with this approach is that the sequencer becomes a bottleneck in the system, both in terms of performance, and in terms of hanging up the system when it becomes inaccessible.

It is important to note that the mechanism that chooses pseudo-times need not perfectly follow the requirement that pseudo-times be given out in ascending order -- regardless of the order in which they are chosen, they may actually be used in a different order anyway. For example, transaction A may choose its instant of pseudo-time before transaction B does, but there may be some delay that prevents A from actually completing its operation until after B is completed. If A and B deal with disjoint sets of objects, this switching of execution order will have no effect on the the transactions or correct operation of the system, while on the other hand if some data is common to them both, the rules for educing known histories will prevent inconsistent results, perhaps aborting A to insure correctness. The system is thus highly tolerant of pseudo-times for transactions being chosen out of order.

Since it is my basic assumption that transactions that conflict on their set of objects to be manipulated will not be likely to happen close together in time, a mechanism that chooses output pseudo-times such that two such choices separated widely in real time will give pseudo-times that are ordered in correspondence with the real time ordering will suffice. The easiest way to do this is to use approximately synchronized clocks. By using a set of approximately synchronized clocks, one per node, one can ensure the property that any time one gets a value it will be greater than the values obtained at all other nodes at times sufficiently far in the past.

As noted above, the clocks at different nodes need not be in perfect synchrony. However, if one clock is consistently slow, it will have a serious effect on the performance of operations originating at that node. If operations originating at two different nodes encounter a conflict that causes one of them to abort, the one whose clock is slow is the one that will be likely to abort. If the clock at a site is extremely slow, it is possible that its updates will *always* abort, resulting in effectively preventing that site from having an effect.

Synchronizing the system clocks whenever they come up by using the operator's watch will usually get the system time accurate within a few minutes. Depending upon the rate at which clocks drift, and the likelihood of two operations running within a few minutes of each other attempting to update the same data, this may be sufficient synchronization. By taking great care, but without using particularly expensive technology, clocks that are synchronized within a few microseconds are possible, using broadcast signals such as those from station WWV as a time standard, and reasonably stable clock circuits.

Given that we are using these techniques, one can do still better by using a technique proposed by Lamport[Lamport78]. Essentially the mechanism is this. Whenever any message is sent between two nodes, it contains the time the message was sent at the originating node. The receiver of the message inspects the timestamp, and if the time is in fact greater than his own real time clock, one of two things is wrong -- either the source's clock is fast, or the receiver's clock is slow, in either case by an amount at least as great as the difference between the message timestamp and the receiver timestamp. Lamport proposes a strategy that amounts to adjusting the receiver's clock forward in such a case so that whenever a message is received that originated

at time t, the receiver node's clock is set to t if it is not already greater than t. This has the effect of synchronizing each clock to the rate of the fastest clock in the system.

There is a problem with this approach, however. If some clock advances much too fast, or if it is deliberately set far into the future, the node containing the clock will get priority in all operations, and more importantly, be able to lock out all updates from other nodes until those nodes advance their clocks. A modified version of Lamport's strategy, in which a message that has a timestamp very much greater than the receiver's clock will not be processed, helps to solve this problem. This technique will also help to solve the problem of incorrectly set clocks.

6.3 Known Histories

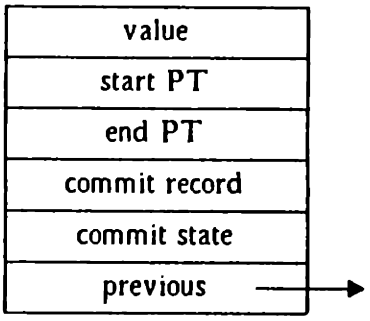
The known history is the mechanism that maintains the relationship between versions and their pseudo-time of validity. It is realized as a data structure stored at the home of an object. The primary purpose of the data structure is to keep track of the individual versions, tokens and aborted versions associated with the object, and to change as operations are applied to the object. Implementation of known histories is fairly straightforward, but it does involve some slightly tricky details.

It is assumed, as a basis of robust operation, that once an operation on the object has signalled its completion, that the changes made by that operation are safely stored away on stable storage. Thus, it is required that all of the versions, tokens, and the known history data structure itself be stored on stable storage before signalling the completion of an operation. Further, it is necessary that no matter where in the process of manipulating the known history the home node fails, the known history be left in a consistent state. The local locking mechanism described in

chapter two can be used to achieve this result, by keeping the previous consistent state of the known history such that it is automatically restored if the manipulation fails while the known history is locked.

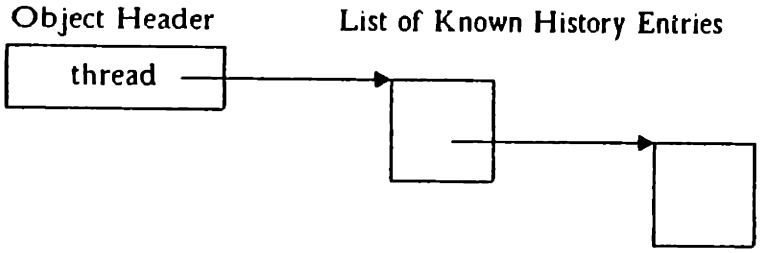
The known history data structure in its simplest form consists of a list of entries that contain a pointer to the representation of the value of a particular version, the start and end pseudo-times that define its validity, a reference (possibly to another node or set of nodes) to a commit record(or records) to specify the possibility under which the version is created, and the cache for the commit record's state (see figure 11).

Fig. 11. Known History Entry



These entries are threaded together as a singly threaded list in reverse chronological order of pseudo-times. This threading ensures that it is easy to add a new entry in between any pair of elements by a single atomic pointer swap. The object itself is then represented by a cell in stable storage, called the object header (see figure 12), that contains a pointer to the head of the list of known history entries.

Fig. 12. Known History Representation



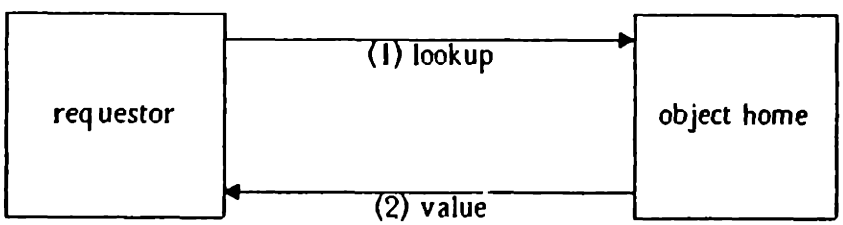
The two basic operations on the known history are the basic lookup operation, which attempts to find the version of the object that corresponds to a particular pseudo-time, and the new-token operation, which attempts to make a new token that is valid from a particular pseudo-time. I will consider these in turn.

The lookup and new-token operations provide basic tools that allow the creation of objects represented as cells that contain record values, where the value of the cell is loaded to inspect the state of the object and the value of the cell is changed by a store operation to change the state of the object. While cells are in a sense complete -- they can be used to construct any kind of mutable object, there are some kinds of objects that may be better represented in a slightly different way, taking direct advantage of the constraints provided by the restriction on the kinds of operations allowed on an object. In this sense, what is about to be described is a default implementation strategy that is known to work for all kinds of mutable objects. After this strategy is described, alternative implementation for special types of mutable objects will be discussed, with the example being queues.

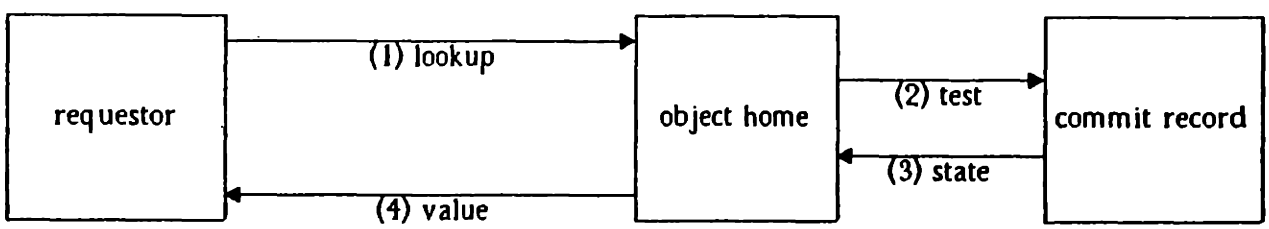
6.3.1 Lookup requests

The lookup operation is a search of the known history, looking for the version or token whose start and end pseudo-times bracket the pseudo-time searched for. If no such version or token exists, then the version or token whose start time comes closest to, but does not come after, the specified pseudo-time is found. If this process results in finding a token, then the state of the commit record must be queried to determine whether or not the token's data can be returned as the version looked for. If a version is found in the search, then it is returned, and its end time is extended, if necessary, to include the pseudo-time specified in the search. Figure 13 illustrates the message passing structures that can result from a lookup request.

Fig. 13. Communication in a lookup request



(a) The version referred to is not a token.



(b) The version referred to is a token.

The lookup operation is invoked as a response to a message from some remote user, in many cases, and there is a question with regard to how the waiting for a response from a token should be managed. One possibility is that the action on finding a token is that a query is sent from the home node to the site of the commit record, and another message is sent to the requesting node, indicating that it should try again later. This approach has the advantage that no memory of the pending lookup operation need be maintained at the node containing the object. Since no memory is allocated while waiting, if the response is lost, there is nothing to clean up. The requester recovers from an error by the simple mechanism of resending the request after a timeout. At some time in the future, a response will come from the commit record, and if that response is that the commit record is complete, then the token becomes a version that can be read later. This approach, while it simplifies recovery after a lost message, has the drawback that it may cause a rather large delay in the case where a token is accessed soon after it is updated.

The other possible approach for handling the waiting is to send the query to the token's commit record, but not respond to the requesting node until the commit record responds either positively or negatively. This mechanism requires remembering the request at the home of the object while waiting, to recall what node the lookup request came from, and what pseudo-time was specified as the target of the lookup request. However, the advantage is that as soon as the state of the commit record is known, it can be reflected back to the requester. There is a disadvantage, however, in addition to the memory required, in that the requester cannot distinguish the delay due to querying a commit record from a delay resulting because the original lookup request was lost before arriving at the object's home node.

Neither of these approaches is completely satisfactory, but a combination of the two can solve most of the problems of each. Upon receiving a request to lookup a specific version, the home node responds to the requester immediately, either with the version, or with a message meaning "I have a token to check on." If this latter message is sent, a query is sent to the token, and in volatile storage, an entry is made that remembers the request -- the requester node, the object, and the pseudo-time -- enabling the immediate forwarding of the response from the commit record back to the requester. When the commit record's response comes back, the volatile storage entry is used to find the particulars about the request, and if the token is now a version, the result of the lookup is forwarded to the requester. If the token is aborted, then it is deleted from the known history, and the lookup operation is reinitiated, resulting in looking at the next previous entry of the known history.

As a general rule in the implementation of NAMOS, we may use volatile storage to hold information that allows early forwarding of messages, or for other performance enhancements that are not strictly required for NAMOS to work correctly.

The combined approach has the advantage that if no error occurs, the response to a lookup that encounters a token is just about as fast as possible. On the other hand, because volatile storage is used to remember the information for forwarding a request, there is no problem in recovering the storage used to remember the state of requests involving tokens, should some failure occur. Let's consider the three kinds of failures that can occur. First, the commit record can become unavailable, either through loss of the query to the commit record, the loss of the response from the commit record, or loss of availability of the site containing the commit record. In this case, the requester node will get no response other than possibly the response that

says the home has received the request. After some period of time, the requester will time out and resend his request. Then, the home node will again query the commit record, and the request will proceed as before.

Second, the requester can fail after making its request. This can occur either because the requester times out, or because of a crash. Eventually, the home node may send a response to the requester, but there is no awaiting a reply from this response, so the requester cannot destroy the system by failing.

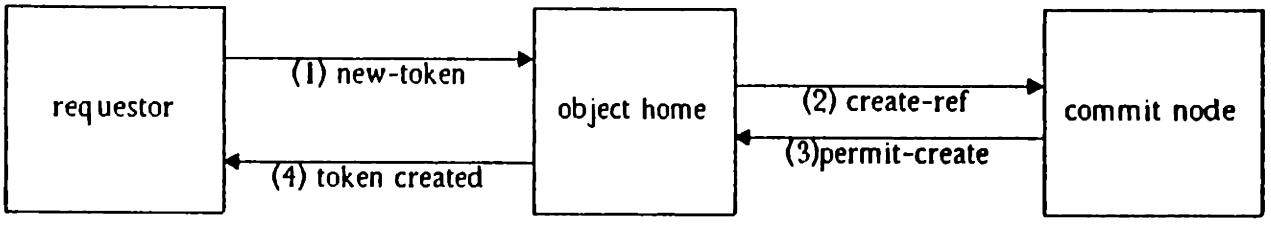
Third, the home node can fail, losing volatile storage. In this case, the response from the commit record may never get reflected back to the requester, who will then time out and retry. The response from the query to the commit record may come back at a time when it can be reflected in the state of the token of interest -- this will only optimize a retried request, or the response may come back while the home node is inaccessible, in which case, the retried request will perform as if the original request had not happened.

6.3.2 New-token requests

The new-token request is much simpler than the lookup request, since the new-token request does not depend on the state of tokens already in the known history. The purpose of the new-token request is to install a new token in the known history to reflect a pending change to the object. Parameters to the new-token request are the value of the new token, the commit record that represents the possibility under which the token is created, and the pseudo-time at which the new token is to be valid from. The basic action is to create (if possible) a new entry in the known history whose start and end times are equal to that specified in the new-token request.

As noted in the previous chapter, if the commit record must be reclaimed once it is no longer needed, it is important that before a token is placed in the known history, the commit record know about the token that may depend on it. Consequently, a **create-ref** message is sent to the commit record, and when a **permit-create** response is received, the fate of the token is decided. Figure 14 shows the pattern of messages that make up a new-token request.

Fig. 14. New-token request processing



Again, to handle the problem of failures at the home node, the requester is responsible for ultimately causing the retransmissions needed to handle lost messages and unavailable nodes. While the create-ref message is outstanding, no response is made to the requester. An entry in volatile storage is made to handle the **permit-create** response when it comes back. This entry contains the parameters of the new-token request.

The new-token request fails if there is already a version whose start and end pseudo-times bracket the pseudo-time the token is to be valid from. If such a condition is detected when the token is to be created, then no change is made to the known history, and an error response is sent to the requester. This condition can be detected either before or after the create-ref/permit-create exchange.

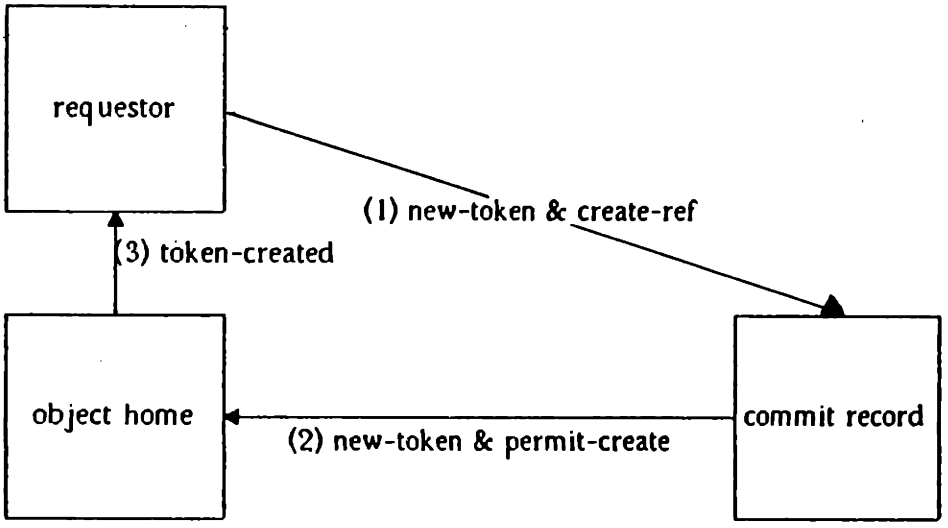
As noted in chapter five, we need not be concerned if the new-token processing fails after the create-ok message is sent; there is no problem if the commit record has a reference to a nonexistent token, since the query from the commit record to the object will generate a no-ref response indicating that no token exists that refers to the commit record.

If a token already exists that was created at the pseudo-time specified in the new-token request, then it may be the case that an earlier attempt to perform the new-token request failed after the token was entered into the known history, but before the response reached the requester; the requester may resend a duplicate request to recover from such an error. It is also possible that the new-token request was duplicated in the network. In either case, an error response should not be returned. Instead, the name of the possibility specified in the token's known history entry should be compared with the name of the possibility that is specified in the new-token request. If equal, then the message is a duplicate, and a response indicating the request was successfully performed is generated, since returning no response would not help the requester in the case of a failure that prompted retry of his request. If the possibility is not the same, then an error response should be generated, since this results from the case where a requester (or multiple requesters) try to create the same version under different possibilities -- a meaningless thing to do.

Having discussed the mechanism for implementing the new-token request, let us contrast it with an alternative mechanism that can be used to achieve the same effect, but perhaps with fewer messages and less delay. The observation that motivates the alternative mechanism is that a new token can never be created before polling the commit record to inform it that a possible reference is being created. This requires a pair of messages, increasing the delay significantly if

the commit record is not local to the node containing the object of the new-token request. One fewer message can be used by sending the new-token request through the commit record site on its way to the home of the object. When it passes the commit record, a reference to the new token that might be created is entered onto the commit record's reference list, and a notation that is equivalent to a create-ok message in information content would be added to the message before forwarding it to the home of the object. The new token could then be immediately added to the known history of the object, and the response could then be sent to the requester. The triangular pattern of messages that results is shown in figure 15.

Fig. 15. Alternative new-token processing



What is wrong with this approach? The basic problem is that it does not admit of any case where the requester is unaware whether or not the request he is sending involves the creation of a new token, or any case where the requester cannot know the name of the new tokens that are being created. If objects are implemented by an interface that hides their internal representation, then very commonly, the name used for the object by the requester will be quite different from

the name(s) used for the object(s) that is(are) a part of the object's representation. So, for example, an individual's bank balance may seem at some level to be a single object that can be changed by certain requests. However, internally, the balance may consist of several lower level abstract types of objects that are located at several nodes. A request to change the bank balance may involve updates to objects the requester never even dreamed existed, so there is no way to inform the commit record what objects may refer to it. In some implementations, such a request could involve creation and modifications of new objects, as well. These new objects would have to be created by the requester in order to make sure the commit record is properly informed.

There are, however, techniques for reducing the number of individual create-ref/permit-create exchanges that occur as part of a composite operation, and thus reducing delay as well as message traffic. These techniques will be discussed later as part of the section on "paging".

6.4 Non-cell object types

So far, with the new-token and lookup operations, we have the basic tools for implementing cells having load and store operations. Other types of objects, such as stacks, queues, accumulators, databases, etc. can be implemented using just cells as their basic tool for achieving mutability, but it may be that more optimal mechanisms can be used when the operations are more complex than simple load and store operations.

A general class of object types for which particularly nice implementation techniques exist are called *accumulator types*. The best example of this kind of object is an integer cell that has operations like adding a constant to the cell, subtracting a constant from the cell, and multiplying

the cell by a constant, none of which return a value, and another operation to obtain the value of the cell. Implementing such an object using the mechanism described so far requires that the operations on an object be performed one at a time, in the order of the pseudo-times assigned to the operations. Thus a delayed request to perform an operation on an object will be aborted if an operation at a greater pseudo-time was already executed.

It is possible to implement the cell very efficiently by storing just the operation (add subtract or multiply) and the constant operand in each entry of a modified known history. A new token in the known history contains the operation and operand with the pseudo-time at which the operation is to be performed. That token becomes a version when its associated possibility is set to the complete state. The actual computation of the values resulting from the operations can be deferred until a value is actually requested. When the value of the cell is requested, however, it will be necessary to actually perform the computations requested by the operations whose pseudo-times precede the pseudo-time at which the value is requested. First, all tokens must be either aborted or become versions. Then, starting with the oldest (in pseudo-time) version whose value has not yet been determined, the values of all versions whose range of validity precede the pseudo-time of the value request are computed. Since each version depends on the previous version, the range of validity of the previous version must be extended to close any gap in pseudo-time that may exist between successive versions.

This implementation of the accumulator allows update operations to be performed without aborting each other if they happen particularly close together in time. The only time at which the update operations will be aborted in this scheme is when a value for the accumulator is desired.

Then, any updates that have not yet been communicated to the accumulator home, but whose pseudo-times precede that of the value request, will be aborted.

In general, an accumulator type is a type of object for which there are one or more operations that change the state of the object without returning any results that depend on either the old or new state of the object. The technique just described can always be used to implement such operations. The advantage is always that the update operations do not delay or abort each other. Disadvantages include more complex implementations of such objects and longer delay whenever an operation whose result does depend on the state of the object is executed.

The accumulator implementation strategy is akin to a strategy often used to process updates like deposits to bank accounts -- the strategy of batching updates together and performing them during slack time on the system unless there is an urgent need to determine the effect of the updates before such a slack time (usually overnight) comes about. The strategy is also a special case of "lazy evaluation" or "call by need". What NAMOS does is provide a framework for the synchronization of updates to an object so that the arrival order of update requests for an object is not significant in assuring that results are consistent.

An interesting variation on the accumulator type is exemplified by the symbol table type. A symbol table has three operations, insert, which adds a new name and associated value to the symbol table, delete, which removes the entry corresponding to a name from the symbol table, and lookup, which returns the value associated with a particular name. To make the symbol table into an accumulator type, we must define the insert operation when the name already exists in the symbol table to change the associated value to the one specified in the insert; if an error were signalled in this case, the insert operation would have a result that depended on the state of

the symbol table (error/no error). Similarly, a delete operation on a name that was not present would just return with no error. The techniques used above could be used to implement the symbol table as an accumulator; however, another technique could also be used.

Suppose we were to choose a tree representation for the symbol table, with binary search to do the lookup. Imagine the symbol table as always containing all of the names ever to be used (inserted, looked-up, or deleted), and have a known history for each name, associating with the (name,pseudo-time) pair a particular value. Since we don't know in advance all of the names that will be used, we simply add to the symbol table tree whenever a new name is looked up, inserted or deleted. We need a way to symbolize in each known history that at a particular pseudo-time the name had no associated value (i.e., lookup signals an error). A reserved value is used. The insert, delete and lookup operations can be implemented with this representation in a rather interesting way. All three operations begin by finding the known history associated with their name parameter (creating it if it doesn't exist). The insert operation checks to make sure that another value is not already associated with that name at the insert's pseudo-time and if not, makes a new token with the new value (otherwise a redefinition error, like the one for version ref$lookup, is signalled). The delete operation is similar, except that instead of a new value, the reserved value indicating no value exists is to be placed in the known history. The lookup operation checks to see if either a normal value or the reserved value is associated with the pseudo-time of the request (extending the range of validity of an earlier value if necessary). If such a value exists, lookup returns it. If no version or token exists for the pseudo-time of the lookup or any earlier pseudo-time, the lookup request creates a token with the reserved value as its value, and then returns the fact that the name is undefined.

In this symbol table implementation, the arrival ordering of requests affecting different names is completely irrelevant. The only requests that can abort each other are requests with the same name. Were our original mechanism, or the accumulator enhancement just described, to be used, operations on different names could cause each other to be aborted. To do a lookup or insert in a binary tree composed of individual record objects whose updates are controlled by NAMOS, the states of all records above the record containing the name referred to would have to be known, so operations on those names could be interfered with.

A final example of interest is a FIFO queue type, with enqueue and dequeue operations. We assume that the enqueue operation does not return a value depending on the state of the queue (so a queue overflow error cannot be part of the interface). The queue object is an accumulator because of the enqueue operation, but, as with the symbol table, there is a better implementation.

The important observation to make about the two queue operations is that when the dequeue operation executed in pseudo-time t' returns the value enqueued in pseudo-time t, the only knowledge that can be deduced knowing the results of dequeue operations thus far is the set of values enqueued in pseudo-times up to and including t. In particular, nothing is known about enqueues that may be executed between t and t'. This observation and the fact that values are returned in the order enqueued lead to a nice implementation. To represent the history of enqueue operations we use the same structure as a known history, where each known history entry has as its value part the value enqueued at its start pseudo-time. The enqueue operation is performed by adding a new token to this enqueue known history in the usual fashion. A second known history represents the history of dequeue operations. The values in this known history

represent the pseudo-time at which the value dequeued was enqueued. Thus, the dequeue at t' above would result in adding an entry to the dequeue known history whose value was t, and whose start pseudo-time was t'.

A dequeue operation works by first doing a lookup on the dequeue known history, to obtain the pseudo-time the value previously dequeued was enqueued. Then the enqueue known history is searched to find the entry with the earliest pseudo-time after that pseudo-time. If no entry is thus found, the dequeue operation returns an empty-queue error, recording the pseudo-time of the dequeue as a new token in the dequeue known history. Otherwise, the entry found contains the value to be returned by the dequeue, and the pseudo-time at which it is enqueued is recorded in the dequeue known history as a token with a start pseudo-time equal to the pseudo-time of the dequeue request. Before returning, the dequeue operation extends the range of validity of the preceding version in the dequeue known history to close up any gap.

The resulting queue implementation, though complex to describe, allows enqueue requests to arrive in quite a different order than the assigned pseudo-times of the enqueues without aborting the enqueues. The queue type is particularly nice in this respect.

Figure 16 illustrates the effect of a sequence of transactions on a particular queue. For simplicity, the pseudo-times in which the queue operations are executed are rendered as a pair of integers. The first integer is the pseudo-time in which the input state of the queue would be observed in ordinary implementations of the queue operations, while the second is the pseudo-time associated with the modification to be made to the queue. The enqueue and dequeue known histories are shown after each operation. The changes are highlighted by underlining. It is fairly easy to see that the arrival order of the queue operations is not severely constrained -- only the

final enqueue shown is aborted because it arrives out of order with respect to its pseudo-time of execution.

Fig. 16. History of several queue operations

Operation	State (PT) Input	Output	Value	Enqueue Known History	Dequeue Known History
Enqueue	10	11	A	[11,11,A][0,0,-]	[0,0,0]
Enqueue	5	6	B	[11,11,A] [6,6,B][0,0,-]	[0,0,0]
Enqueue	20	21	C	[21,21,C] [11,11,A] [6,6,B][0,0,-]	[0,0,0]
Dequeue	21	22	B	[21,21,C] [11,11,A] [6,6,B] [0,5,-]	[22,22,6] [0,21,0]
Enqueue	24	25	D	[25,25,D] [21,21,C] [11,11,A] [6,6,B] [0,5,-]	[22,22,6] [0,21,0]
Dequeue	25	26	A	[25,25,D] [21,21,C] [11,11,A] [6,10,B] [0,5,-]	[26,26,11] [22,25,6] [0,21,0]
Enqueue	7	8	E	[25,25,D] [21,21,C] [11,11,A] [6,10,B] [0,5,-]	[26,26,11] [22,25,6] [0,21,0]

...fails, because there is already a version in the enqueue known history valid in 8.

Legend:

Known History Entry

[Start PT, End PT, Value]

Of our three examples, the symbol table example has perhaps the largest practical importance, since our example symbol table could be generalized to any associative lookup mechanism, for example a large relational database. All of the optimizations described in this section, however, rely on the fact that the object implementation can be exclusively locked for the duration of the execution of an operation, so these tricks can be applied only to single node objects.

6.5 Creation and Deletion of Objects

So far, we haven't yet discussed the mechanisms by which objects are created and deleted. In the following discussion, I assume that objects are explicitly deleted, so that a means for detecting "dangling references" must be provided. If some sort of automatic deletion of objects based on knowing that there will never be any references to an object after a particular pseudo-time can be provided, then the mechanism could be simplified by eliminating the need for detecting references to an object after it is deleted. Such an automatic deletion scheme would be hard to provide, however. First of all, NAMOS is intended to be used in the decentralized multi-node environment described in chapter one. It would be rather difficult, and certainly rather expensive to provide a distributed garbage-collection algorithm, although a scheme based on reference counts might be a good way to begin, were the second complication to be described not to exist. Second, and perhaps more importantly, the automatic deletion algorithm must know, for each object reference to an object, what pseudo-times are to be used with that reference in order to refer to a particular version. By taking the least pseudo-time, L, that can still be used (either because it is still to be past in some executing computation's pseudo-temporal environment, or because it is stored in some checkpoint as suggested in chapter four) with any object reference, the system could construct the transitive closure of the refers-to relation, where an object refers to another if some version of the object valid after L refers to the other object. All objects reachable from executing computations in this closure cannot be deleted. Any components of the closure that are unconnected to existing computations could be deleted. This strategy for garbage collection is both difficult to implement in a distributed system, and does not guarantee to find all deletable objects.

The important issue in creating and deleting objects is that the creation and deletion of an object has an effect on the observable behavior of an object in pseudo-time. Thus, the pseudo-time of creation of an object defines the earliest pseudo-time that the object can ever be assigned a value or read, while the deletion pseudo-time defines the latest such pseudo-time. Thus the known history as a whole must keep track of these times, once they are known. Consequently, the object header must specify the creation and deletion pseudo-times, once they are known.

It is possible that an object is created by a composite operation that later fails. Consequently, the creation must be mediated by a commit record. The protocol mechanism for this is simple. Creation of an object occurs in a particular state, by the creation of the first token of a known history. The first token is special, in that it must be committed before any other operation on the object can be performed. To allow later tokens to be committed would imply that the object existed, though it may never be created. This paradox can occur only if there is a way that the new name can be passed to another operation before the current operation finishes -- implying two possibilities, one for the object that contains the name, and one for the named object being created. Use of multiple possibilities is not the normal mode of operation, and as noted in chapter four, may lead to peculiar results if we are not careful.

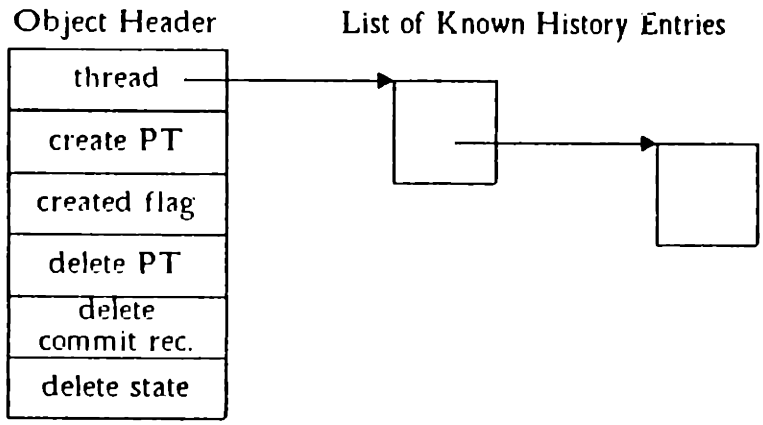
Thus, in addition to the creation pseudo-time, the object header must contain a flag indicating whether the object was really created. This flag simply encaches the commit state of the of the initial token. The encachement is necessary to allow the initial version to be eventually deleted, by the version deletion mechanism to be discussed.

Deletion of an object does not imply the freeing of storage of the object, since there may exist requests yet to happen that will attempt to refer to versions of the object that correspond to pseudo-times earlier than the pseudo-time associated with deletion. If storage were immediately freed, then those requests could not be satisfied, an unfortunate kind of behavior. Consequently, deletion simply prevents operations at a later pseudo-time from the deletion from being able to access the object. A separate mechanism for deletion of individual versions will enable the eventual freeing of storage for the object.

Deletion of an object requires care in implementation. Deletion of an object basically consists of freezing the evolution of the known history at a particular point -- however, the freezing must be mediated by a commit record. Deletion thus marks the object header as "potentially deleted," and saves the commit record pointer associated with the deletion and the pseudo-time of deletion in the object header. Later attempts to access the known history that would require that the cursor be advanced are delayed until the deletion is known to have happened (by the commit record entering the complete state) at which time the requests are denied. If the deletion fails (its commit record goes to the abort state), accesses past the pseudo-time of attempted deletion are permitted.

As a result of the modifications made to the object header to handle creation and deletion, it now looks as shown in figure 17. The object header can be freed once it is successfully deleted by the above mechanism and all versions have been also deleted by the mechanism described in the following section.

Fig. 17. Object header, revised to handle creation and deletion



6.6 Deletion of Object Versions

Object versions may be deleted form the known history, thus reducing the memory needed to retain the known history. Certain restrictions apply. The main restriction is that once a version has existed for an object, no other version can ever be made valid in the range of validity of the deleted version. Consequently, there must be some way to represent in the known history the ranges of validity of all versions that have ever existed.

An approach that saves some of the storage associated with versions is to delete only the value associated with a deleted version, while keeping the known history entries and marking them as deleted. A deleted known history entry would then just be place holder, preventing new tokens from being created there. This approach is still inadequate because storage required for an object that can be updated remains unbounded.

Essentially, the result of deleting versions is to save space at the cost of possibly causing certain operations that are still looking into "old" states to fail. Depending on the characteristics of use of data, some rather flexible strategies can be used. For example, one may implement many of the objects in a traditional large data base such that all versions and tokens (except the most recent version) whose range of validity strictly precedes N seconds before the current real time are deleted. This strategy will work quite well if transactions against the data base are executed in environments whose input state corresponds to the real time the transaction began, and if all the transactions have a high probability of completing in less than N seconds.

Implementing this strategy in the known history is quite simple. In the object header the value N would be stored, and any attempt to create a new token or lookup a value in a pseudo-time older than the current real time minus N would be prevented, giving an error message indicating that the version referred to no longer can exist. Versions and tokens whose ranges of validity end before the current real time minus N can be deleted when convenient. One other rule must be applied, in order to avoid losing the only version. The latest version and tokens with later pseudo-times may not be deleted, even if their ranges of validity lie before the current real time minus N, unless the object has been deleted. Thus we are assured that an attempt to get the current value (the value using the current real time as the pseudo-time of reference) will always succeed unless the object has been successfully deleted.

A more restrictive strategy is to keep only the most recent version and later tokens. In high update traffic where occasionally there are long executing transactions that attempt to do a coherent query of many data base records, this may prevent these large coherent queries from being very likely to succeed. However, this strategy is logically equivalent to a locking strategy,

in that one is only guaranteed to be able to access the "current" version. Except in abnormal cases, only one token will exist, so this strategy can be viewed as a complete methodology for the distributed locking problem. As I noted in chapter four, I am aware of no other methodology that handles the problem of maintaining consistency by locking in a distributed data base in the case where arbitrary crashes and loss of messages can occur. Consequently, if the advantages of maintaining multiple versions, and supporting interfaces that work in environments that are unknown at design time, were not considered important, the methodology of this thesis still can be of great help in designing distributed data bases.

In some implementations, old versions might never be deleted, but rather would be moved to some kind of write-once, low cost backup storage. Not only does this provide a complete audit trail of changes to the data base, but, as noted already in chapter four, it makes possible undoing later changes to parts of the data base where such changes may have resulted from user error. One can simply use the backup storage versions to retrieve old versions that are consistent in a particular state, and install them as the present state.

Once a version has been copied to backup storage, it doesn't really make sense for its range of validity to be changed. Consequently, to implement this strategy, the idea of using the value (real time)-N as a boundary before which tokens cannot be created and versions cannot have their ranges of validity changed is still important. This value is also used to select versions to move to backup storage. Versions are written into backup storage with associated information that specifies the range of validity. Since there may be holes between the versions put in backup storage, at the time each version is "frozen" for backup (when its end pseudo-time is less than

(real time)-N), its range of validity is extended to include any following hole, in order to make sure that a backup copy exists for any pseudo-time that lies in the range from creation to deletion.

If this strategy of keeping old versions in archival storage is used, it may be inappropriate to try to retrieve them for ordinary accesses because of excessive delay. Consequently, an appropriate mechanism must be used to distinguish references that are prepared to fetch from archival storage when necessary. A mechanism that is somewhat general is to specify how long a request is to wait for a value before it is declared inaccessible. The same mechanism would be useful for controlling the wait for a value that is on a node currently inaccessible due to a system crash or communications failures.

It is important to note that applying these implementation techniques blindly to the queue implementation described earlier will result in bad behavior. The version deletion mechanism for the known history that represents the values enqueued must recognize that there are references in the known history containing the last dequeue pseudo-time that will be left dangling if the (real time)-N strategy is applied blindly to the enqueued values known history. The proper approach is to delete only those versions in the enqueued value known history that are not referenced by the last dequeue pseudo-time known history.

6.7 Small Objects and "Paging"

An attractive way of handling storage and encachement algorithms for related groups of small objects is "paging". Simply put, in paging a group of objects are handled as a single object for the purposes of synchronization and storage management (including encaching as noted in the previous section).

In the model proposed here, it is possible to reduce the overhead of storing ranges of validity with object versions and keeping known histories by sharing ranges of validity across multiple objects in a "page." This strategy is particularly inviting from the space reduction viewpoint when composite operations that affect most or all of the objects in a page are common.

The basic mechanism is to maintain a known history for the page, consisting of *page versions* and *page tokens*. Every change to any object in the page causes the generation of a new page version (although changes to multiple objects in the page that are effective in the same output state with the same commit record will result in only one new page version -- how this is achieved is described shortly). Every attempt to change an object will generate a new page token unless the correct one already exists.

The only trick is managing the page tokens such that multiple changes to the page token are allowed. Our earlier protocol description for record objects would suggest that the second attempt to update the page token would be ignored as a duplicate request. In fact, we need a mechanism that allows multiple writes to the same page token if the writes are to disjoint objects. Basically, with the page token, we must keep a list of the objects for which new tokens are created, to which objects new update requests should be treated as duplicate messages. This list, like the pointer to the commit record, is not needed after the page token is committed.

Use of pages introduces some restrictions. For example, creation, deletion, and relocation of individual objects on a page seems to be possible, but allowing for it greatly complicates bookkeeping, and may not be worth the trouble to implement. However, creating, deleting, and relocating all of the page's objects as a group is quite simple.

Perhaps the greatest advantage of the "paging" strategy in this system is that in creating a page token and then doing all of the later changes, only one create-ref/permit-create exchange need be done. Thus, if a number of objects are used to represent a particular abstraction that are usually all changed at the same time, the message overhead of communication with the commit record on each new-token request will be drastically reduced.

6.8 Copying of Object Versions

Since object versions are immutable, they can be freely copied to nodes other than their home node. One might want to perform such a copy to optimize performance, where an operation needs to refer to a particular object quite frequently, yet the object changes (generating a new version) only rarely. There are, of course, other considerations that would argue against copying versions of objects -- in particular, the information hiding afforded by keeping the representation of the object only known at its home node, and the protection resulting from that information hiding that can restrict the ways in which the object can be inspected. However, given that copying is possible for the particular object, the system keeps enough information to allow the copying to take place.

We can view the copying of an object version as an encachement of the object, since the home of the object still contains the master copy of the versions of the object. When the object version is copied, it and the range of validity in pseudo-time of the object version are transmitted to the using system. At the using system, whenever a lookup request is requested for that object with a pseudo-time in the range of validity known to the using system, the copied version can satisfy the request.

A small detail of the mechanism for encachement may not be obvious, though. In particular, object versions can be copied when the end of their range of validity is unknown, either because there is a token following it that has not been committed, or because no read has extended the range of validity up to the time of the next version. In either case, the full range of validity of the version is not yet known at the time of the copy.

If a lookup request for the object is not in the range of an encached version, the encached version might not be useless. By augmenting the protocol for doing a remote lookup, it is possible to extend the range of validity of the encached version without the full cost of recopying the entire version. Basically, we add to the remote lookup request a parameter specifying the final time of the closest preceding encached version on the requesting site. The handler of the request may then need only to ship a special message that says, in effect, "your version that started at pseudo-time T is valid through time T'." Although this may not reduce delay, it certainly can help reduce the volume of network traffic.

In fact, this will be a very common case, where multiple read operations in successively later pseudo-times are issued at a site. An optimization that will reduce delay somewhat for this case would be to remember at the home site the locations of encached versions, to enable immediate transmission of the updated range of validity of encached versions. When new versions are created, the entire version could be shipped in the same way to the cache site. When a site no longer wishes to be a cache site, it should just forget about the encached object versions, and respond to updates with a no-cache(object) message, signalling the object home to remove from its memory the existence of the cache.

Some applications of the encaching strategy seem to be extremely useful. For example, the strategy seems to be just what is needed for distributing new versions of programs. NAMOS is so easily extended to include encachement because the information needed to synchronize the encached copy with the master copy is already needed for maintaining inter-object consistency and proper synchronization. Encaching writable data in a database using locking for synchronization would not result in such a uniform mechanism.

6.9 Reducing the amount of work aborted

When an operation is aborted due to the failure of a new-token request, the operation may have done a substantial amount of computation that will all have to be redone. This wasted work results from the fact that tokens are not created until it is known that a new value will be assigned to a particular object, so another interfering computation may perform a read that extends the range of validity of an earlier version, preventing the creation of the token.

If it were known in advance that a particular object will be updated by an operation, then a much better strategy would be to have the operation reserve the right to create a token in advance, and abort itself if that reservation cannot be made. It is not always possible, of course, to know what will be updated by an operation, so the case where such information is available should be treated as an optional optimization. A mechanism to make such reservations is fairly easily added to the system.

A new kind of entity, a *token reservation*, is the tool by which such reservations are made. A token reservation is represented by a known history entry that looks just like a token, but which has no value. A token reservation refers to a particular commit record, and may be

deleted from the known history at any time -- its purpose is to mark the fact that a token may be created at some later time, but it does not guarantee that the token can be created. Of course, deleting the token reservation right away without giving the operation a chance to actually create the token obviates the purpose of token reservations. The times when a token reservation ought to be deleted are (a) when the commit record enters either the aborted or completed state (if no token has been created by then, the operation will not ever attempt to create a token), or (b) after a sufficiently long time -- the timeout can be set to be some value greater than the timeout on the commit record, or (c) if the commit record has been deleted. Case (b) ensures that the token reservation does not depend on the accessibility of the commit record for its deletion, while case (c) ensures that the commit record need not record references to it from token reservations, thus eliminating the need for create-ref/permit-create exchanges at the time token reservations are created.

A token reservation is converted into a token when a new-token request specifying the same pseudo-time arrives at the home of the object. At that time, the create-ref/permit-create exchange of messages occurs. If a different commit record than the one under which the token reservation was created is specified in the new-token request, an error is signalled -- while this error signal is unnecessary (the token reservation is just an optimization), using different commit records is a misuse of the mechanism.

The lookup request is affected by the existence of token reservations as if the token reservation were a token, with two differences. Before querying the commit record, the local timeout on the token reservation is checked, and if expired, the token reservation is deleted, and the lookup continues with the next previous version or token. Also, if the result of the query is a

state("aborted") or a state("complete") message, the token reservation is deleted. Thus, a reservation is only held while the commit record remains in the wait state.

The use of token reservations can help reduce the likelihood of dynamic deadlock in the system, as well. By making all of the token reservations as quickly as possible, the likelihood of having two conflicting operations that run at nearly the same time is reduced, by our assumption that conflicting operations are rarely close together in time. An even more potent strategy that can be used to prevent dynamic deadlock would be to define an ordering of objects, and to ask for the token reservations in all operations in a sequence that is a subsequence of the order on all objects. As it does in the case where one is requesting locks, this strategy would guarantee that in any conflict situation, at least one operation would be allowed to finish, eventually breaking the dynamic deadlock.

I should reiterate here, however, that the use of token reservations is entirely optional -- if they are used correctly they will improve the performance of individual operations in some cases, perhaps at the cost of reducing the performance of other operations. To show how other operations could be adversely affected, consider the strategy of obtaining token reservations on all the objects that an operation *might* want to update, rather than just the objects that we are sure will be updated. Eventually, those objects that are not updated by the operation will delete the token reservations either due to timeout or detecting the completion of the operation's associated commit record. However, while the token reservations are around, they will slow down responses to lookups by forcing them to go look at commit records. Consequently, other operations that are later in pseudo-time than the one creating the token reservations may be delayed excessively as a result. Judicious use of token reservations is thus very important to good performance.

6.10 Summary

In this rather long chapter, a number of mechanisms involved in the implementation of objects in the system have been discussed. The really important part of the system design is in the management of clocks and the generation of pseudo-time, and in the management of the known histories of objects. Although we have seen a number of optional enhancements, such as the encachement of objects, special representations for special types, and token reservations, the system without optimizations is sufficient by itself to solve a wide range of synchronization problems.

Chapter Seven

Conclusions and Directions

In this dissertation, I have developed a new approach to the synchronization of accesses to shared data. In this chapter the key concepts, the range of applicability, and the limitations of the new approach are explored. Then possible directions for further development of this approach are suggested -- implementation, modification to remove limitations, and specialization of the approach for particular applications.

7.1 Concepts

There are several key concepts that I would like to reemphasize by listing them here. They include a new view of the semantics of updates, the notion that abstractions must be preserved properly in an environment where concurrency, sharing, and failures are unavoidable, the close tie between synchronization and error recovery, and the idea of defining a system-wide state without reference to an instantaneous snapshot of the entire system.

7.1.1 Synchronization of shared data

In the thesis we have concentrated our attention on one aspect of synchronization -- control of simultaneous accesses to shared data objects. It has been traditional to treat such synchronization with the same ideas and mechanisms as other problems of synchronization, such

as disk queue scheduling and interprocess control communication,[1] 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 usually have little to do with the very important case of 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 its data. The division of synchronization into two classes, data access synchronization and process (timing) synchronization, seems to be a useful and powerful division.

7.1.2 New update semantics

Our view that a data object really stands for a sequence of states and that accesses to the object (both read-only and update) 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 allowing programs to use 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 very little control, reducing the importance of timing in understanding the execution of programs simplifies the task of the designer.

---

1. Interprocess control communication (IPCC) is a generic term for mechanisms that allow one process to block itself when it has nothing to do, and be awakened by another process when something for it to do arrives. Although IPCC has been used in the implementation of mechanisms for controlling access to shared data, IPCC is more generally useful for controlling the timing of processes that must synchronize themselves for other reasons.

This view has also facilitated the ability to manage "out of date" states of data objects. By keeping states of a data object older than the current one, the system can allow more concurrency. In particular, it is not necessary for read accesses to an object to lock out subsequent writes. In a geographically distributed system, where delays are large, this locking introduces delay, since any write must delay sufficiently to determine that no remote computation has requested a read prior to the write. But since "out of date" versions are kept for a period of time, writes can be begun with confidence that read requests for an earlier version will not be interfered with.

The ability to manage "out of date" states of objects also enables the encachement of objects to support frequent remote reading that was described in chapter six, and the restoration of old system states for the purpose of recovering from severe errors.

It is interesting to note that our semantics 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[Dennis75]. Although the objects implemented in NAMOS 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 of objects 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 being an extensive memory with state changing operations, a view that seems to be better for inter-user sharing. Thus we may have gotten the "best of both worlds."

7.1.3 Abstraction and parallelism

Updates to objects shared among parallel computations make it very difficult to construct programs out of modular parts. In non-parallel computations, one can view the execution of a subroutine as an operation whose internal implementation is largely invisible to its user. For example, a sorting routine can be thought of in terms of a rather simple specification that relates the states of its input to the ultimate outputs. The implementation of a sorting routine, on the other hand, can vary quite a bit while satisfying the same specification.

In the case of parallel execution on shared data (unavoidable in the case of a multi-user database or file system), the semantics of a routine depends on the pattern of accesses made by the implementation of the routine, so it is possible to "look inside" an interface at the implementation by executing the routine in parallel with other computations that access the same data objects. Thus in the parallel execution of computations, the ability to construct abstract operations is severely curtailed.

A partial solution to the problem of constructing abstract operations in a parallel execution environment is to add explicit locking to the language, so that a routine can gain exclusive control of a set of resources to inhibit concurrent access to the data it accesses. The solution is only partial, however, since it becomes necessary to include in the interface specification of a module what locks it needs set in order to use that module as part of a larger module that also needs to set locks. By exposing the set of resources that must be locked outside the module, the implementation is exposed in a slightly different way -- one need not try running computations in parallel to discover the implementation, since much of the structure of the implementation may be deduced from the resources it accesses.

Our thesis is that it is both possible and extremely desirable to have system support for modules whose behavior does not depend on the behavior of programs accessing the same data concurrently. The designer and user of such modules need not concern themselves with parallel execution, locking, etc. in order to make sure the modules work as expected. In particular, the transaction pseudo-temporal environment and the dependent possibility support the implementation of such modules.

Preserving the degree of abstraction afforded by a module interface has required that certain problems be solved that are normally very difficult to solve in a system that uses locking for synchronization. In particular, our approach to synchronization can handle cases where the data objects to be accessed by an operation are dependent on the input values to the operation. Predicate locking, proposed by Eswaran, *et al.*[Eswaran76], attacks a similar problem, but still requires some care in specifying the class of objects that a program may access.

7.1.4 Synchronization and Recovery from Failures

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 provided by a module interface. Failures during the execution of a module may cause its implementation to become important to the users of the module, while failure of a module being used to implement part of a larger transaction must be reflected to the other parts of the larger transaction -- undoing any changes that may have already been made if the failure requires that the larger transaction be aborted.

Since both failure management and synchronization are tightly tied to the construction of abstractions, they must be carefully designed to work together. Most of the attempts to solve the problems of synchronization assume that the problems of assuring reliability are solved at a lower level. While it is possible with enough redundancy to construct a system whose reliability is high enough that the likelihood of hardware failures can be ignored, it may be that the cost of achieving reliable systems in this way is far higher than the cost of achieving reliability given knowledge about which parts of the implementation of the system must be reliable. Since we must already cope with unexpected loss of availability of objects and computational resources in a decentralized system (because of autonomy), handling failures of the communications system in the same way may achieve a cost saving by removing the need for a separate set of mechanisms to handle hardware failures.

7.1.5 System-wide state defined independent of time

By defining a correspondence between the states of individual objects, we have defined a concept of a system-wide state, such that the system goes through a sequence of such states as a result of computations executing in the system. The mechanism for defining the correspondence, using pseudo-time, has the advantage that it is only loosely coupled with the passage of time. A definition of system state that uses time to relate states of individual objects runs into serious problems for several reasons. First, there is the problem that in a geographically distributed system, such a definition of a system state requires that to observe all the objects in a particular state, one must take a snapshot of all of the objects at an instant of time -- since communication delays in such a system make precisely simultaneous observations practically impossible, system states thus defined are not observable inside the system.

A second problem is that the running of transactions concurrently in the system makes it very unlikely that the system state at any particular instant of time is consistent with some external ordering of actions. The result of this is that it is very hard to use that notion of system state in describing the behavior of programs to ensure that the system's behavior matches some external specification of correct operation. Such a definition of system state has led to the unsatisfying notion of defining a system as operating correctly in terms of its behavior after some "quiescing" action. For example, Thomas claims that his algorithm for synchronizing updates guarantees that if new requests to update are halted (the system is quiesced), the system will eventually stop in a state where all updates have been processed, in the proper order[Thomas76]. The problem with such a specification of a system is that the system may never be thus quiesced (an airline reservation system, for example, may always have several uncompleted seat reservation transactions for different flights in progress). In fact, a system may satisfy such a specification by entering a correct state only when the system is quiesced, but nonetheless processing all requests incorrectly!

Our definition of system state, since it is independent of time, allowed the definition of consistent system-wide states without having to talk about quiescing the system. A transaction in progress in the system does not preclude an observable consistent state within NAMOS, since the transaction does not destroy the existing consistent state it observes as its input state, it only creates a new, presumably consistent state.

7.2 Applicability of the concepts

In chapter one, we emphasized that the primary application of the ideas in the thesis is to the problems of synchronization and failure management in a decentralized, possibly geographically distributed, computer system composed of multiple, separately managed nodes sharing information via a communications network. Certainly the decentralized system application was the primary motivation for our work, but the ideas do seem to have wider applicability.

Our approach to synchronization has been motivated by a particular desire to support modular design and implementation of programs, and we believe our approach is a quite effective way to manage synchronization and error recovery problems in a system where program development is decentralized to the point that there is no one central designer who specifies the set of atomic transactions that the system will support. Even if the system were entirely confined within a single processor system, using shared secondary memory for the shared objects of interest, NAMOS seems to be a good way to support modular design. In such an application of NAMOS, the costs of managing individual object known histories, both in terms of space and time, can be controlled somewhat by an implementation specific to the single system case. Space can be conserved by storing only the most current version and at most one token for each object (requiring that an update arriving at an object with a token outstanding wait until the token is turned into a version or aborted). In the single system case, the complex protocols described in chapter five for recovering the storage for commit records need not be used because we can assume that the whole system is either up or down -- thus the commit record can immediately signal all tokens referring to it when it is completed, aborted or timed out. Similarly, in a single

system, the system usually knows when a process is terminated due to some error, so in normal failures the aborting of the commit record(s) associated with a computation can be made to happen sooner by reflecting the failure immediately.

The mechanisms of NAMOS can also be applied to solve synchronization problems where none of the problems of modular system construction exist. An example would be a "garden variety" database system used to support some sort of recordkeeping application such as accounting, payroll, inventory control, etc. In such applications, usually the whole system can be designed (by a "database administrator"), put into use, and will remain virtually unchanged for quite a long time. We certainly don't need the ability to add new transactions dynamically to the set of transactions that can be executed. The advantages of NAMOS over other methods in these applications are that geographically distributed data can be properly synchronized, that both synchronization and error recovery do not require as large a design effort as would be needed if explicit locking and checkpointing had to be designed in, and that later modifications to add features to the system, if needed, will not require existing synchronization code (probably there is some such code in each module of the system) to be modified.

For such a database system, NAMOS can be simplified to reduce its complexity. Since support for modules as abstract operations is not needed in this application, dependent possibilities are unnecessary, and all transactions are created at the top level. If the database system is not distributed, then the optimizations noted above for the single-processor case can be applied.

7.3 Limitations

We cannot claim, however, that the ideas described here are universally practical. We have made certain basic assumptions about the environment in which NAMOS is to be used, and where those assumptions are violated, NAMOS works poorly, if at all.

A key assumption we have made is that the likelihood of two transactions concurrently accessing the same data is reasonably small. If this assumption is not true, then transactions that read values out of objects will with high frequency cause transactions that perform updates on the same objects to be aborted. Although our assumption is true in many systems that allow sharing of objects, some systems do manage such objects that have high rates of contention; those systems may need additional mechanisms to support their concurrency control. Essentially, in the case of high contention, a more efficient way to do business is to schedule conflicting transactions so that one transaction completes its accesses to the shared data before any later ones are allowed to proceed. NAMOS's mechanism of token reservations, described in chapter six, serves as a kind of scheduling mechanism, but since it requires reservations to be made for all objects to be touched it may be quite costly in terms of communications overhead. An alternative, more centralized, scheduler could be designed that ordered transactions based on high-level knowledge about which transactions are likely to conflict. Of course, such a scheduler must be designed to deal with failure of a transaction, so that such a failure does not prevent later transactions from being scheduled.

The mechanisms of NAMOS will still be useful in a system with such a central scheduler. NAMOS's recovery mechanisms still provide a good way to deal with failure of transactions. The synchronization mechanisms of NAMOS can be thought of as dynamic verification that proper synchronization is being achieved by the central scheduler, such that if the central scheduler erroneously schedules two transactions acting on the same data at the same time, the NAMOS mechanisms will ensure consistency or abort one or both of the transactions. In fact, this suggests an interesting strategy for the design of a central scheduler. Instead of being conservative, always deferring one transaction whenever there is the slightest chance that two transactions will conflict, the central scheduler can be more optimistic, and only defer a transaction when it has a high chance of conflicting with one already executing. The basic mechanisms of NAMOS then ensure proper synchronization and recovery, while the central scheduler optimizes performance, increasing the chances of proper termination of transactions.

7.4 Directions for further research

So far, we have seen a relatively complete design for a mechanism to handle the problems of synchronizing and managing failures of accesses to objects. It is, however, only a paper design, so many of the aspects of its performance and usability can be characterized only qualitatively. Constructing a prototype implementation for the purpose of evaluating its implementability, performance, and usability seems to be the obvious next step.

The full NAMOS system need not be constructed for some of its attributes to be evaluated. A first step would be to replace the transaction synchronization and recovery mechanisms of some existing database system with the synchronization and recovery mechanisms of NAMOS. The objects of the system would be the data records or relation tuples maintained

by the data management system. The transactions of the data management system would be implemented as computations executing in a transaction pseudo-temporal environment. This would serve as a test of the performance of NAMOS. An extension along this direction would be to distribute the database (requiring a way to decide where a particular record or tuple is located), so that communications failures, and perhaps autonomy, can be incorporated into the substrate on which the NAMOS mechanisms operate.

A test on an existing database system would not show the usability afforded by NAMOS's support of modular construction methodologies. To test these features, a more interesting use of NAMOS would be required. One possible application of NAMOS would be in the construction of a decentralized network of "personal computers" where a strong need to share permanently stored data is needed. One kind of data that might be shared among such a set of computers would be useful user-written and maintained programs, such as sophisticated editors, compilers, etc. Small databases, such as files of papers in progress (i.e. journal articles or letters), annotated bibliographies, etc., might also be structured objects that are shared. It is reasonable to assume that such shared objects will occasionally be updated by the provider or users of the objects, so there will be problems in managing the concurrent accesses to the objects that are likely to occur. The mechanisms of NAMOS can be used to manage such sharing.

Another possible test of the ability to build modular systems in NAMOS would be to build a continually modified database system (such as, for example, a system like that the M.I.T. registrar uses to keep track of the subjects taken, grades, and other information it keeps about each student -- this system needs to be changed regularly as the rules and structure of M.I.T.

change) using NAMOS. One could then see if the tools provided by NAMOS lead to a simplification of the task of modifying the system to incorporate new features.

A direction for exploring the issues surrounding NAMOS that is more speculative is to incorporate the synchronization and failure recovery semantics of NAMOS directly in the computational model of some computer design. Such a direction is not without precedent. Randell has explored a similar direction in developing a hardware support for recovery blocks[Randell75], and numerous hardware designs have incorporated synchronization mechanisms such as semaphores. The advantage of directly implementing the NAMOS synchronization and reliability mechanisms in hardware is performance. The algorithm that finds the correct version of an object given a version reference composed of an object identifier and pseudo-time should execute as fast as possible, since it is used on every access to an object. Similarly, defining a new version must be as fast as possible. In a distributed system, where the time to access an object is controlled primarily by communication delay, accessing and updating objects using the mechanisms of NAMOS may not be very costly, but using NAMOS inside a single system for purely local operations might be rather costly if NAMOS is implemented entirely in software. The attributes of NAMOS, particularly the support of modular synchronization and reliability, make it otherwise attractive for synchronization of accesses purely local to a single system. Incorporating the basic operations of NAMOS in a computer design, using special hardware designed to optimize the naming of versions, would make NAMOS attractive in such a case.

Other future research could be directed at removing the major limitation of NAMOS -- that it can't handle a high degree of contention among the transactions acting on a particular object. We have suggested above a strategy that mixes together NAMOS and some sort of centralized transaction scheduling discipline. This approach must be explored in more detail to see if it does fulfill its promise of supporting high-contention situations. Another related idea that can be explored here is the idea suggested in chapter six of delaying attempts to read objects to decrease the likelihood of aborting an update not yet received.

Appendix A

Analysis of Availability of Multi-site Possibility

To show that the multi-site possibility provides more accessibility than a single site possibility, an analysis of the probability of availability is needed. Here I include an analysis based on some simple, but reasonable assumptions about the availability of sites and reliability of messages. I lump site availability and lost message probabilities together, and assume that all sites behave the same.

First, consider a test of the state of the whole commit record. If the test does not make a decision within τ seconds, the commit record is considered to be unavailable. Each voter site is assumed to behave similarly when its state is polled, providing a response saying whether the voter is complete or aborted within time τ with probability q. The value of q includes both the probability of site inaccessibility and the probabilities of transient loss of messages to and from the site.

When a complete or abort operation is attempted, we assume that there is no problem with interference from local voter timeouts (we assume the probability of this is negligible, because of the trick described above). If some individual site does not acknowledge an attempt to complete or abort it within time τ', the requesting site gives up. The probability that a site does not process a complete or abort sent to it within time τ' is p. The probability p again includes both the probability of message loss and the probability of site failure or inaccessibility.

Finally, we assume that the requesting site sends out the messages sequentially, and may fail in between any of the N messages. To characterize the probability of failure, I assume that with probability $1-\phi$ it will fail before sending the kth message, given that it has sent the $(k-1)$st. Thus we assume that the number of messages sent before a failure is represented by a geometric distribution. The value of ϕ will be quite close to 1 -- a site with mean time to failure of one hour and which takes one thousand instructions to send a message at one microsecond per instruction will have an ϕ of about 0.9999997.

Now let us consider the problem of accessibility. What we want to know is the likelihood that some attempt to test the state of a commit record will be unable to decide the state of the commit record within time τ. There are two cases, depending on whether there was an attempt to complete the commit record or not. If a complete was attempted, the probability that a tester will not be able to decide we call P_{cfail}, and if no complete was attempted, we call the probability of not deciding P_{afail}. For a single-site commit record, $P_{cfail}=P_{afail}=1-q$. For a multi-site commit record, the situation is more complex.

P_{afail} for a multi-site commit record is the easiest case. To decide that the commit record is aborted, at least $N-\kappa+1$ sites must respond. The probability of this is $\sum_{i=N-\kappa+1}^{N} \binom{N}{i} q^i(1-q)^{(N-i)}$. Thus, P_{afail} is the probability that at most $N-\kappa$ sites respond.

$$P_{afail} = \sum_{i=0}^{N-\kappa} \binom{N}{i} q^i(1-q)^{(N-i)}.$$

P_{cfail} is more complex. If an attempt to complete the commit record has been made, there is some probability distribution among the states that the voters may have reached that depends on p and ϕ. The probability P_{cfail} can be expressed as $1-P_C-P_A$, where P_C is the

probability that the tester will decide that the commit record is completed, and P_A is the probability that the tester will decide that the commit record is aborted (these are clearly mutually exclusive, so with P_{cfail} the probabilities sum to 1). Each voter can be viewed as an intermediate station in a message transmission from the complete request to the tester of the commit record. The probability that a message sent by the requester will get to the tester is pq. If we assume that the requester got to send exactly k messages before failing, then the probability that the tester will decide that the commit record was completed is $Q_k = \sum_{i=\kappa}^{k} \binom{k}{i}(pq)^i(1-pq)^{k-i}$. The probability that the requester got to send exactly k messages, once deciding to attempt to complete, is ϕ^N if k=N, and $\phi^k(1-\phi)$ otherwise. Thus,

$$P_C = \phi^N Q_N + \sum_{k=\kappa}^{N-1} \phi^k(1-\phi)\, Q_k.$$

Similarly, given that k messages were sent by the requester, the tester has a certain probability of determining that $N-\kappa+1$ voters are in the abort state. Now we have two cases. Among the k sites that messages were addressed to, some of the sites will enter the aborted state because of messages lost with probability (1-p), but will supply an answer to the tester. For any of the k sites, this will happen with probability (1-p)q. For the other N-k sites, the aborted state is guaranteed to be entered, and the probability that the tester gets an aborted response is q. Thus, the probability that the tester gets *exactly* i aborted responses is

$$T_{i,k} = \sum_{j=\max(0,i-k)}^{\min(i,N-k)} \binom{N-k}{j} q^j(1-q)^{N-k-j} \binom{k}{i-j} ((1-p)q)^{i-j}(1-(1-p)q)^{k-i+j}.$$

So, the probability of getting at least $N-\kappa+1$ responses that say abort, given k messages sent is

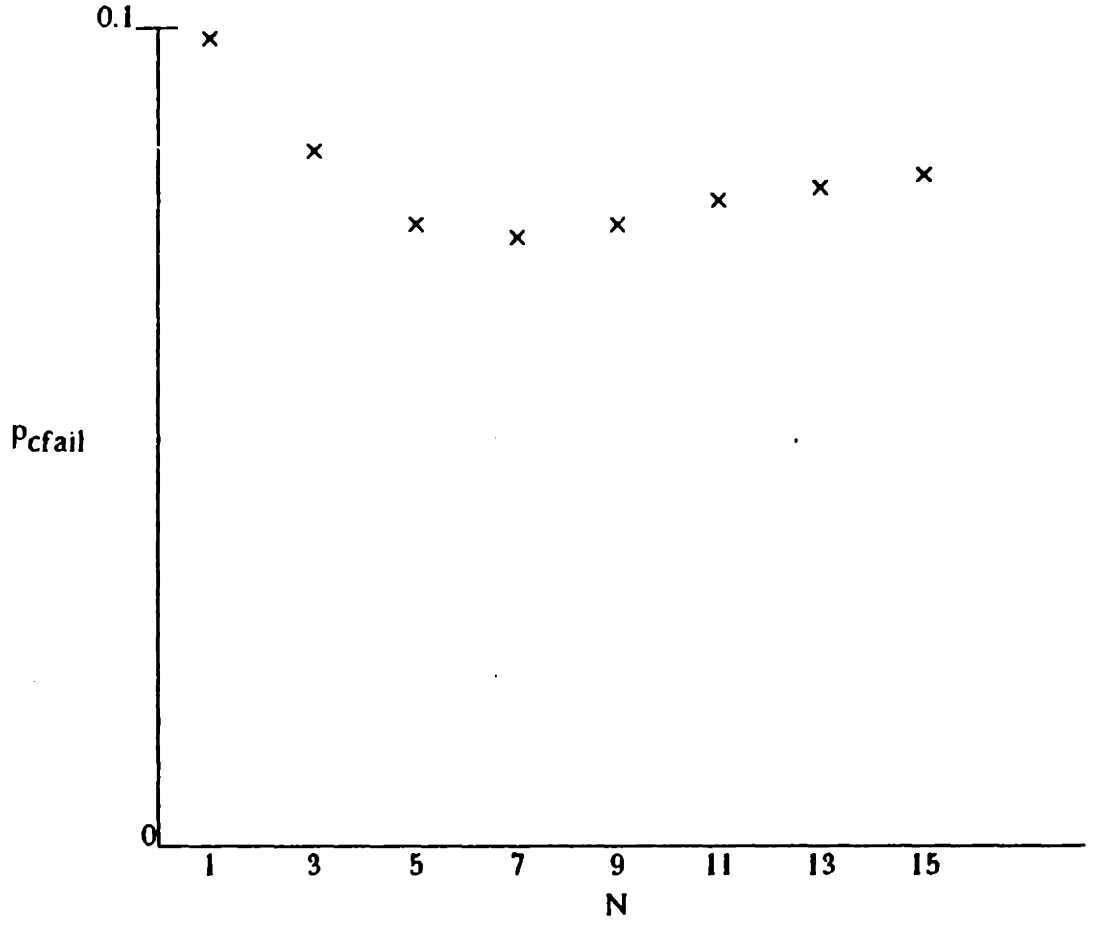
$$R_k = \sum_{i=N-\kappa+1}^{N} T_{i,k}.$$

Consequently, we can compute the probability of deciding aborted as

$$P_A = \phi^N R_N + \sum_{k=0}^{N-1} \phi^k(1-\phi) R_k.$$

The expression for P_{cfail} is thus quite complicated, and doesn't seem to be amenable to simplification. However, I have shown by experimentation with reasonable values of p, q, and ϕ that values of N and κ other than one (the single site case) can give either improvements in availability of the commit record's state, or can reduce availability. A sample graph of the variation of P_{cfail} as N varies over odd numbers of sites from 1 to 15, holding p, q, and ϕ all constant at 0.9, and with $\kappa=(N+1)/2$, is shown in figure 18.

Fig. 18. Graph of P_{cfail} as the number of sites varies



Improvements result because the expected number of sites to respond completed is about pqN (as it would be if ϕ were one), then as N becomes large, the probability that $\kappa=\alpha N$ (where $\alpha<pq$)

sites thus respond asymptotically goes to one. Similarly, as κ gets small relative to N, P_C increases. Reductions in availability result because as N gets large, failures at the requester begin to take over. These two opposing factors lead to the case that, at least in the simple model, there are optimum values of N and κ which minimize P_{cfail}.

P_{afail} is not so complicated. The state of a possibility for which no complete is ever attempted is more accessible as N gets large, and as κ gets large.

The overall availability of a possibility depends on the likelihood that operations that are controlled by it will fail to finish before attempting to complete. This likelihood is strongly application dependent, and cannot be determined by the system. If it were known, however, a reasonably optimum choice of N and κ could be made.

One factor ignored so far is the delay and cost of message traffic needed to achieve a particular level of reliability and availability. It is clear that one can improve p and q by increasing potential delay (by, for example, retransmitting requests periodically). Similarly, increasing N can increase the message traffic in the network, and possibly indirectly add queueing delay. Varying κ can reduce or increase the delay before deciding that a commit record is complete, while affecting the delay for deciding the same kind of commit record is aborted in the opposite direction.

By keeping $\kappa=\lceil N/2 \rceil$, and keeping N small (1,2, or 3), the message delays and costs can be kept small, and the common case of a completed operation will be optimized both in speed and availability.

References

[Alsberg76] Alsberg, P.A., Belford, G.G., Day, J.D., Grapa, E., "Multi-Copy Resiliency Techniques," CAC Document # 202, May, 1976.

[Atkinson78] Atkinson, R.A., and Hewitt, C.E., "Specification and Proof Techniques for Serializers," Draft, March 20, 1978.

[Backus78] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Logic of Programs," 1977 ACM Turing Award Lecture, published in *CACM 21*, 8 (August 1978), pp. 613-641.

[Bernstein77] 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.

[Bobrow72] Bobrow, D., et al., "TENEX - a paged time sharing system for the PDP-10," *CACM 15*, 3 (March 1972), pp. 135-143.

[Corbato65] Corbató, F.J., and Vyssotsky, V.A., "Introduction and overview of the MULTICS system," *Proceedings of the AFIPS 1965 Fall Joint Computer Conference*, Vol. 27, Pt. I, AFIPS Press, Montvale, N.J. pp. 213-230.

[Crocker75] Crocker, S.D., "The National Software Works: A new method for providing software development tools using the ARPANET," Proc. Meeting on 20 years of Computer Science, Pisa, Italy, July 1975.

[DOliveira77] d'Oliveira, C.R., "An Analysis of Computer Decentalization," M.I.T. Laboratory for Computer Science Technical Memo TM-90 (October, 1977).

[Davies73] Davies, C.T., "Recovery semantics for a DB/DC system," *Proceedings of the 1973 ACM National Conference*, New York (1973), pp. 136-141.

[Dennis75] Dennis J.B., "First Version of a Data Flow Procedure Language," M.I.T. Laboratory for Computer Science Technical Memo, TM-61, May 1975.

[Eastlake69] Eastlake, D., et al., *ITS 1.5 Reference Manual*, M.I.T. Artificial Intelligence Laboratory Memo AIM-161A, July 1969.

[Eswaran76] Eswaran, Gray, Lorie, Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM 19*, 11, November, 1976.

[Frankston74] Frankston, R.M, "The computer utility as the marketplace for computer services," M.I.T. Laboratory for Computer Science Technical Report TR-128 (1974).

[Gaines72] Gaines, R.S., "An Operating System Based on the Concept of a Supervisory Computer," *CACM 15*, 3 (March 1972), pp. 150-156.

[Goodenough75] Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," *CACM 18*, 12 (December 1975), pp.683-696.

[Gray77] Gray, J.N., "Notes on Data Base Operating Systems," *Operating Systems: An Advanced Course*, in Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978, pp.393-481.

[Henderson75] Henderson, D.A., "The Binding Model: A Semantic Base for Modular Programming Systems," MIT-LCS TR-145, February, 1975.

[Hewitt76] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," M.I.T. Artificial Intelligence Laboratory, A.I. Memo #410, December, 1976.

[Hewitt77] Hewitt, C. and Baker, H., "Laws for Communicating Parallel Processes," *Proc. IFIP 77*, August 1977.

[Hoare74] Hoare, C.A.R., "Monitors: an operating system structuring concept," *CACM 17*, 5 (October 1974), pp. 549-557.

[Johnson75] Johnson, P.R. and R.H. Thomas, "The Maintenance of Duplicate Databases," ARPANET NWG/RFC #677, January 1975.

[Kent76] Kent, S. T., "Encryption-Based Protection Protocols for Interactive User-Computer Communication," MIT-LCS TR-162, May, 1976.

[Lamport78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM 21*, 7 (July 1978), pp.558-565.

[Lampson76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center, Ca. November, 1976. To appear in *CACM*.

[Levin77] Levin, R., "Program Structures for Exceptional Condition Handling," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June 1977.

[Liskov76] Liskov, B.H., "A note on CLU," CSG Memo # 136, M.I.T. Laboratory for Computer Science, February. 1976.

[Liskov77a] Liskov, B.H., et al., "Abstraction Mechanisms in CLU," *CACM 20*, 8 (August 1977), pp. 564-576.

[Liskov77b] Liskov, B.H., and Snyder, A., "Structured Exception Handling," M.I.T. Laboratory for Computer Science Computation Structures Group Memo 155, December, 1977.

[Liskov78] Liskov, B.H., et al., "The CLU Reference Manual," CSG Memo # 161, M.I.T. Laboratory for Computer Science, July, 1978.

[Metcalfe73] Metcalfe, R.M., "Packet Communication," M.I.T. Laboratory for Computer Science Technical Report TR-114 (December 1973).

[Metcalfe76] Metcalfe, R.M., et al., "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM19*, No. 7, pp. 395-404, July, 1976.

[Randell75] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering, SE-1*, 2 (June 1975), pp. 220-232.

[Randell78] Randell, B., Lee, P.A., and Treleaven, P.C., "Reliability Issues in Computing System Design," *ACM Computing Surveys 10*, 2 (June 1978), pp.123-166.

[Reed78] Reed, D.P., "Synchronization with Eventcounts and Sequencers," to appear in *CACM*. (Presented at Sixth ACM Symposium on Operating System Principles, November 1977).

[Rothnie77] Rothnie, J.B., Bernstein, P.A., Goodman, N., and Papadimitriou, C.A., "The Redundant Update Methodology of SDD-1: A System for Distributed Databases," Computer Corporation of America Technical Report, February, 1977.

[Saltzer78] Saltzer, J.H., "Research Problems of Decentralized Systems with Largely Autonomous Nodes", *Operating Systems Review 12*, 1 (January 1978) pp. 43-52.

[Thomas76] Thomas, R. H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control," BBN Report #3340, July, 1976.

[VanHorn66] Van Horn, E.C., "Computer Design for Asynchronously Reproducible Multiprocessing," Ph.D. thesis, M.I.T. Department of Electrical Engineering. Also available as Project MAC Technical Report TR-34, from the M.I.T. Laboratory for Computer Science.

[Wulf74] Wulf, W.A., "ALPHARD: Towards a language to support structured programming," Carnegie-Mellon University Dept. of Computer Science, April 1974.

[Wulf75] Wulf, W.A., Levin, R. and Pierson, C., "Overview of the Hydra operating system development", *Proc. Fifth Symposium on Operating Systems Principles*, 131, November 1975.

Biographical Note

David Patrick Reed was born on January 31, 1952, in Portsmouth, Virginia. He grew up in Norfolk, Virginia, in Watertown, Massachusetts, in Kittery, Maine, in Jacksonville, Florida, in Edison, New Jersey, in Alexandria, Virginia, and in Hingham, Massachusetts. He graduated from Hingham High School, Hingham, Mass., in 1969. He received both the Rensellaer Medal Award and the Bausch and Lomb Science Medal at that time, and was a member of the National Honor Society.

From 1969 to 1978 he has attended the Massachusetts Institute of Technology, receiving a host of degrees. His undergraduate education was supported by a National Merit Scholarship and an American Water Works Foundation Scholarship. He received the B.S. degree from the Department of Electrical Engineering in June, 1973. His B.S. thesis was entitled "Estimation of Primary Memory Requirements for Processes in Multics." As a graduate student at M.I.T., he was supported as a research assistant in the Computer Systems Research Group of the M.I.T. Laboratory for Computer Science, and as a teaching assistant in the Department of Electrical Engineering and Computer Science. He received the S.M. degree from the Department of Electrical Engineering and Computer Science in August 1975 and the Electrical Engineer degree in January 1976. His S.M. and E.E. thesis was entitled "Processor Multiplexing in a Layered Operating System." He received the Department of Electrical Engineering and Computer Science Teaching Award in May 1975.

As an undergraduate student, Mr. Reed, along with a group of three other students, designed and implemented the Multics version of the MACLISP interpreter and compiler for the LISP language. He also participated in the design and implementation of the Multics operating system carried out at Project MAC at M.I.T. He has also worked at the IBM San Jose Research Laboratory during the summer of 1975.

Mr. Reed is a member of the Association for Computing Machinery, and its special interest groups on Operating Systems, Programming Languages, and Communications. He is also a member of the Sigma Xi scientific honorary society, and the American Association for the Advancement of Science.

In September, 1978, Mr. Reed will assume the rank of Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. He is married to Lynn Susan Reed, and has one son, Colin Alexander Reed.

Mr. Reed's publications include:

David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," to be published in *CACM*.

David D. Clark, Kenneth T. Pogran, and David P. Reed, "An Introduction to Local Networks," to be published in *IEEE Proceedings*.