MIT/LCS/TR-207

ROBUST CONCURRENCY CONTROL FOR A DISTRIBUTED INFORMATION SYSTEM

Warren A. Montgomery

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

This blank page was inserted to preserve pagination.

ROBUST CONCURRENCY CONTROL FOR A DISTRIBUTED INFORMATION SYSTEM

by

Warren A. Montgomery

December, 1978

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

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

ACKNOWLEDGEMENTS

Many people aided greatly in the production of this thesis. A complete acknowledgement would no doubt run longer than the work itself. My most sincere thanks to those who have not been included in this brief list, but none the less made my stay at MIT stimulating and productive.

I commend Professor L. Svobodova for her great endurance in reading many poorly written drafts of my early ideas and for her prompt response to each of my attempts to clarify those ideas. She was most helpful in that she could usually figure out what I meant to say, even when I was not sure myself, and yet still was able to read each draft as if she had not seen the work before. Dr. D. Clark was most helpful in providing key suggestions to clear up many of the vague areas, and his faith and interest in my early ideas were very much appreciated.

Professors F. J. Corbato and M. Hammer, my readers, provided many helpful suggestions for improving the thesis at and after my thesis examination. They were able to point out areas for improvement that were not so apparent to one who was caught up in the work.

I would like to thank Jim Gray of IBM Research, San Jose, and Dave Reed for several productive discussions of ideas in and related to my thesis. Such discussions were particularly helpful in providing early guidance on the merits of several of my ideas. Dave Reed must also be thanked for his help in creating many text-processing tools that were used in preparing the finished thesis.

I would like to thank all of the members of the Computer Systems Research group for providing an environment in which ideas were freely discussed and explored. They were also most helpful in being my agents after I had left M.I.T. I am extremely grateful for Allen Luniewski's help in relaying copies of the thesis between the printer at M.I.T., and myself in Illinois. My fellow workers and management at Bell Laboratories must also be thanked for their support in encouraging me to finish my thesis at a time during which it was most difficult to concentrate on the work.

My special thanks go to my wife, Carla, who endured five and a half long years of my career as a graduate student. She was always most understanding, even at times when her own thesis work was not going well. Her efforts in improving my writing have had a great impact on the quality and clarity of the final copy. Most of all, I would like to thank her for her unending confidence in my ability to finish, and for providing a strong incentive for doing so.

I would also like to thank the National Science Foundation for their support of my first three years at M.I.T. through the graduate fellowship program.

This empty page was substituted for a blank page in the original document.

ROBUST CONCURRENCY CONTROL FOR A DISTRIBUTED INFORMATION SYSTEM

by

WARREN A. MONTGOMERY

Submitted to the Department of Electrical Engineering and Computer Science
on December 1, 1978 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

Abstract

This dissertation presents a collection of protocols for coordinating transactions in a distributed information system. The system is modeled as a collection of processes that communicate only through message passing. Each process manages some portion of the data base, and several processes may cooperate in performing a single transaction.

The thesis presents a model for computation in a distributed information system in which the sites and communication links may fail. The effects of such failures on the computation are described in the model. The thesis discusses implementation techniques that could be used to limit the effects of failures in a real system to those described in the model.

A hierarchical protocol for coordinating transactions is presented. The accesses to be performed during a transaction are pre-analyzed to select the protocols needed to coordinate the processes that participate in the implementation of the transaction. This analysis can be used to guide the organization of the data base so as to minimize the amount of locking required in performing frequent or important transactions. An important aspect of this mechanism is that it allows transactions that cannot accurately be pre-analyzed to be performed and correctly synchronized without severely degrading the performance of the system in performing more predictable transactions.

A novel approach to the problem of making updates at several different sites atomically is also discussed. This approach is based on the notion of a *polyvalue*, which is used to represent two or more possible values for a single data item. A polyvalue is created for an item involved in an update that has been delayed due to a failure. By assigning a polyvalue to such an item, that item can be made accessible to subsequent transactions, rather than remaining locked until the update can be completed. A polyvalue describes the possible values that may be correct for an item, depending on the outcome of transactions that have been interrupted by failures. Frequently, the most important effects of a transaction (such as the payment of money) can be determined without knowing the exact values of the items in the data base. A polyvalue for an item that is accessed by such a transaction may be sufficient to determine such effects. By using polyvalues, we can guarantee that a data item will not be made inaccessible by any failure other than a failure of the site that holds the item.

A strong motivation for the development of these protocols is the desire that the individual sites of a distributed information system fail independently, and that a site or a group of sites be able to continue local processing operations when a failure has isolated them from the rest of the sites. Many of the previous coordination mechanism have only considered the continued operation of the sites that remain with the system to be important. Another motivating factor for the development of these protocols is the idea that in many applications, the processing to be performed exhibits a high degree of locality of reference, in that most operations involve only a small number of sites. By structuring the coordination mechanism to take advantage of this locality of reference, one can have protocols that are simple, efficient, and robust for the particular application.

keywords: distributed data bases, synchronization, message passing systems, reliability.

CONTENTS

FIGURES

Chapter 1
Introduction

Recent developments in electronic technology have made practical the interconnection of a large number of computer systems to form what I will refer to as a distributed information system. Each of the computer systems (or sites, as they are more frequently called) in the resulting system maintains some information and tools for accessing that information. The sites that make up a distributed information system may not be under the control of a single administrative authority. A distributed information system allows any user of any of the individual sites controlled access to the entire body of information managed by the system, while it allows each of the individual computer systems to control the use of the tools and information that it holds.

1.1 Reasons For Distribution

There are several good reasons for choosing such an organization for an information system rather than placing all of the information in a single large, shared computing facility. I will discuss some of these reasons briefly.

1.1.1 Autonomy

A very important reason for choosing a distributed organization for an information system is the autonomy of the individual sites. A recent study [DOliveira77] has shown that the ability to partition the authority and responsibility for information management in a distributed system is the most important reason for many businesses considering distributed

information systems. In a distributed system, each site has control over the information that it manages, and can set its own policies for controlling the availability of that information. As we shall see, autonomy has important implications for the assumptions that can be made about the cooperation of individual sites in the execution of processing operations, and for the protocols that can be used to coordinate such operations.

1.1.2 Reliability

A second reason for distribution is reliability. There are two ways in which a distributed information system can be made more reliable than a central facility. One way to achieve greater reliability in a distributed system is to replicate information, storing it at two or more of the sites in a distributed system. Replication increases the availability of information in a system with unreliable sites. A single failure does not make replicated information inaccessible. Unfortunately, modifying replicated information is much more difficult than modifying non-redundantly stored information. While a great deal of research has gone into the development of protocols to update replicated data, the problem remains difficult, and such updates are costly in that they require extensive communication between sites, reducing the economic advantage of distribution.

A second source of increased reliability, and one which I consider to be much more important, is the the failure of a single site or communication link does not necessarily make the entire system fail, while in a single, centralized system, the failure of a single component frequently interrupts all processing in progress. The individual sites in a distributed information system will be smaller and simpler than a single large computer system with storage and processing power equivalent to the total of that of the individual sites. This simplicity should mean that the sites in a distributed system fail less frequently than the single machine of a centralized system. Thus if a distributed system can be constructed so as

to limit the effects of a failure at one site to the interruption of processing that requires information at that site, the reliability of a distributed information system as seen by any individual user will be substantially better than that of a single shared machine.

1.1.3 Economics

A third reason for distribution is an economic advantage that makes a group of small computer systems less costly to manufacture than an "equivalent" single large machine. A single computer with a certain processing rate and storage capacity costs substantially more than a collection of smaller machines with the same aggregate processing rate and storage size. In addition to the computing hardware, communication and software development contribute to the cost of a distributed information system. Frequently, the information to be managed can be partitioned in such a way that most of the processing operations do not require information from more than one of the partitions. Each partition can be assigned to a small computer system capable of performing the processing required for the information in that partition. The cost of communication between sites in such a system would be relatively small. If the extra cost of developing software for a distributed information system can be kept small, a distributed information system may be substantially less costly than an equivalent central facility.

1.1.4 Flexibility

A fourth reason for distribution is flexibility. Changes in the amount of information to be managed by the system can require increasing or decreasing the storage and processing capacity. In a central system, this may require replacing the entire machine with one of a different capacity. In the distributed system, capacity changes can frequently be

accomplished by adding or deleting sites, with minimal impact on the sites not being changed.

Consider, for example, a corporation that has just acquired a subsidiary, and needs to modify its administrative information management system to manage the new subsidiary. Merging the information management systems of the parent company and the subsidiary into a single central facility could be very difficult. If the information management system being used by the corporation is distributed, however, the merger can be accomplished by adding one or more sites to manage the subsidiary.[1]

1.2 The Concurrency Control Problem in a Distributed Information System

Several problems must be overcome in order to make a distributed information system as easy to use as a central facility. The subject of this thesis, and what I believe to be the most difficult of these problems, is controlling the sequencing of user specified processing operations. The result of performing such processing operations concurrently should be the same as that obtained by performing them in some sequential order. Before this problem can be discussed in detail, we must have a more precise definition of the way in which stored information can be manipulated. For this purpose, I adopt terminology that has commonly been used in data base systems.

---

1. In rare cases, the existing information systems of the parent and the subsidiary may be compatible, requiring virtually no effort for the merger. Even if the information management system of the subsidiary must be substantially modified to fit into the parent's distributed system, this effort should be less than that required to merge both into a single shared facility.

The stored information consists of a set of individual data items, each of which represents some independently accessible piece of information. For each data item there is a current value that is the information that that item currently contains.[1] A data base state is a mapping from the set of items that makes up the data base to the set of values, specifying the current value of each item in the data base.

The high-level operations that are to be performed on stored information are known as transactions. A transaction can be viewed as a function mapping one data base state to another. Each transaction is performed as a set of primitive operations, called accesses, on individual data items. Some accesses to an item cause the current value of that item to be changed, and are known as updates. The set of items whose values are changed by the transaction are the output items of the transaction.[2] The new values produced by the transaction for these items are known as the output values of the transaction. Each transaction computes its output values based on the values of the items in the data base state that is the input to the transaction. The items that are used by the transaction in computing the output values are refered to as input items, and their values as supplied to the transactions are the input values of the transaction.

The user of a distributed information system views each transaction as a simple, complete operation, such as "deposit $50 in account number 13612". Each transaction "sees" the effects of previous transactions in the values that it obtains for its input items. A problem arises when several transactions are performed concurrently. Each transaction may see the effects of the others on the shared data items. In order to preserve the illusion that a transaction is a simple, complete operation, the transactions must be atomic, in that each

---

1. The term "version" has also been used for what I will refer to as a value [Reed78,Stearns76].
2. This has also been refered to as the write set of the transaction [Bernstein77].

transaction sees either all or none of the effects of each other transaction on the data items that it accesses. The definition of atomic will be made more precise in a later chapter.

The problem of insuring that transactions which are run concurrently are atomic is known as concurrency control and is common to both distributed systems and to centralized data base systems, where transactions are run concurrently to increase the utilization of resources. While there is a great deal of literature on this general problem, the particular characteristics of a distributed information system aggravate the problem of concurrency control, and make many of the mechanisms that have been developed to solve this problem in centralized data base management systems inappropriate for a distributed information system.

1.3 Basic Assumptions and Goals

Two common problems in evaluating a mechanism to solve a complex problem are understanding the goals of that mechanism and knowing the assumptions made about the effects of failures. This section sets forth *my* own goals and assumptions, to allow the reader to evaluate more precisely the mechanism proposed here. These assumptions and goals may not be appropriate for all applications, but *I* believe that they are most appropriate for many uses of a distributed information system as described above.

1.3.1 Implications of Delay

A characteristic of distributed information systems is that communication between sites is slower, more costly, and less reliable than communication within a site. An

implication of this characteristic is that unnecessary inter-site communication should be minimized, even if this requires more computation or more storage at each individual site.[1]

A second implication of communication delay is that no one site can readily obtain a view of the global state of all transactions in progress. State information from remote sites is delayed in communication and may be out of date. The lack of global state information makes concurrency control schemes in which some decisions (such as deadlock detection and backup) are made based on global information awkward for use in a distributed information system. Thus, ideally, the protocols used for performing transactions should allow each site to base its actions on its local state only.

A third implication of delay is that any operation involving several sites may be delayed for a long period of time before it can be completed. This means that the information should be organized such that frequent or important operations can be accomplished locally at some site. While I will not discuss the task of partitioning information in detail, I assume that the operations to be performed exhibit a high degree of locality of reference. Each operation requires only a small amount of the total information available, and the information can be partitioned so that very few operations require information from two or more sites.

This assumption is necessary to make a distributed information system practical. It seems quite reasonable for many applications, including management information systems, process control, and personal computing.

---

1. I am not addressing the concept of a "multi-microprocessor" distributed system consisting of a large number of small processing and storage elements linked with very high bandwidth communication

1.3.2 Partial Operability

As noted above, the individual sites in a distributed information system should fail less often than a single centralized system of equivalent processing power and storage capacity. If each site failure interrupts only those transactions which require resources at the failed site, then a transaction involving only a small number of sites should be less likely to be affected by a failure in a distributed information system than it would be in a centralized system. Thus as a goal, the mechanism for performing transactions should allow a group of sites that are functioning and can communicate with each other to perform transactions local to that group. I refer to this goal as partial operability. The most important aspect of partial operability is to allow any transaction that is entirely local to one of the sites to be performed whenever that site is operating and the request to perform the transaction can be communicated to that site.

This is a very different form of enhanced reliability from that achieved with replication, as described by Alsberg et al. [Alsberg76]. I believe that the goal of partial operability more accurately reflects the needs of most applications. We shall see later that both replication of data within one site and replication of data items at several sites fit naturally into the mechanism that I am proposing.

An implication of partial operability is that the dependence of one site on another to perform purely local transactions must be minimized. Protocols requiring a site to receive external authorization to perform local transactions, such as that used by Thomas in [Thomas76], should be avoided.

A more important implication of partial operability is that error detection and recovery are concurrent with the execution of transactions. Backward error recovery strategies [Randell78], which stop processing new transactions when an error is discovered and cause the data base state to be "rolled back" to a previously saved state known to be consistent, do not achieve the goal of partial operability. Because processing continues during error recovery, a site that encounters an error can "get behind" in that it may not be aware of recent transactions. For example, a site holding a copy of a redundant data base may discover that the values that it holds are out of date because they do not reflect transactions that were performed on other copies. The failure recovery mechanism must record any information sent to a site during a failure of that site, so that the site can be brought up to date on recovery.

1.3.3 Autonomy

As noted above, the autonomy of individual sites in a distributed information system is an important reason for choosing such a system over one with a central shared facility. One implication of autonomy consistent with the goal of partial operability is that individual sites should not be dependent on the system as a whole in that they should be capable of performing local transactions when not in communication with other sites. Thus we cannot assume that a site which is not in communication with any other sites stops all processing, as is done by SDD-1 [Bernstein77].

Another implication of autonomy is that each site controls the operations that can be performed on the data items that it holds. Thus any site may refuse to perform some operation at any time. One method of dealing with this possibility is to require that each transaction obtain permission to perform all of its component operations before any of these

operations is carried out. This can substantially increase the cost of performing some transactions, by increasing the need for locking (see Chapter 4).

For many transactions, the administrative policies of all of the sites that must cooperate are known in advance and examined in determining whether or not a site will cooperate in performing a particular transaction. Verifying that a transaction will not encounter access restrictions is similar in principle to verifying that a transaction preserves consistency constraints (i.e. that it always maps one consistent state to another). I will assume that even though the sites are autonomous, they will cooperate in performing a large class of common transactions. Thus in many cases, the acceptability of a transaction to be run can be simply verified before it is run, and will not interfere with synchronization. Dynamically changing access restrictions must be checked as a transaction is run, and will add to the cost of performing and synchronizing transactions.

1.4 Related Work

The work of this thesis concentrates in two main areas: concurrency control in data base systems, and reliability techniques. I will discuss the previous research in these areas separately first, and then relate it to this thesis

1.4.1 Concurrency Control

Several papers [Bernstein77,Gray75,Gray77,Stearns76] discuss the problem of controlling the concurrent execution of transactions so that each sees a consistent version of the data base. Gray et al. [Gray75] give definitions for four different levels of consistency and discuss locking strategies to achieve each. Atomic transactions as I have defined them maintain the highest level of consistency (level 3) defined in that paper. This is the level

that places the greatest constraints on concurrent execution of transactions.[1] The locking strategies presented by Gray are efficient, in that they allow the data base to be constructed so that a high degree of concurrency may be obtained with little locking overhead.

A second paper by Gray [Gray77] discusses a mechanism for concurrency control in a distributed system that makes use of the locking strategies described in the first paper. While this mechanism performs transactions correctly unless highly improbable failures occur, it fails to meet two of the goals outlined above. The locking strategy allows transactions to deadlock, requiring some mechanism to detect deadlock and abort one of the transactions involved in a deadlock in order to allow the others to proceed. Deadlock detection requires a view of the global state of all transactions in progress, violating the condition of making decisions based on local information.

The two-phase commit protocol used by Gray and others insures that a transaction is atomic, no matter what failure occur during its execution. If a failure occurs at the wrong time, however, one or more of the sites involved in a transaction may be obligated to hold onto locks set by the transaction until the failure is recovered, preventing the execution of transactions local to that site that set locks which conflict with those set by the transaction suspended by the failure. This violates our goal of partial operability.

---

1. While the authors claim that forcing all transactions to see level 0 or level 1 consistency allows transactions to be constructed to see higher levels of consistency, and may save locking overhead by allowing many transactions to run at the lower levels of consistency, they also point out that output values produced by a transaction reflect the level of consistency that that transaction saw. These low-level consistency values are propagated by any transaction that reads them, so that transactions desiring a high level of consistency can never read values produced by those observing a lower level. Thus low-level of consistency transactions would appear to have very limited use.

A study by Stearns and Rosencrantz [Stearns76] discusses a model for distributed data bases in which the data are partitioned among sites and each transaction is performed by a process that migrates among the sites that hold the values that the transaction accesses. Each site is responsible for controlling the execution of transactions at that site, and the sites communicate only when a transaction is moved and when a transaction is completed. The authors describe a class of control algorithms that work by assigning an order to the transactions to be processed and use that order to resolve conflicts between processes attempting to access the same data, possibly by aborting and restarting them. The necessity of restarting some transaction that has completed a substantial amount of processing is undesirable, but seems unavoidable in this model of concurrency control. Similarly, the protocols developed by Gray [Gray77] also require deadlock detection and backout.

Several papers [Bernstein77,Hammer78,Rothnie77] discuss the SDD-1 database system in which the set of transactions to be performed on the data base is analyzed to determine the amount of locking needed. Transactions are divided into classes by the sets of items that they read and write, and transactions in the same class are performed serially with respect to each other. Transactions in different classes can be performed concurrently. The conflicts between the sets of items read and written by different classes are used to select synchronization protocols to be used to coordinate concurrent transactions from different classes. Frequently, transactions can be run concurrently with little synchronization overhead.

The approach used in SDD-1 of pre-analyzing the set of expected transactions to minimize the synchronization overhead for the most common transactions seems to be very promising. The proof that this technique works, (i.e. that all transactions are atomic), however, is so long and complicated as to be unconvincing. Making SDD-1 robust in the event of failures also appears difficult. The synchronization protocols used frequently

involve waiting for messages that may be delayed by failures. The techniques used to insure that delayed messages do not cause excessive delay in the processing of transactions are extremely complicated, and may reduce some of the efficiency of this synchronization scheme by requiring additional message exchanges.

The reliability goal of SDD-1 is also somewhat different from that of this thesis. The goal in SDD-1 is to keep the system as a whole running, even if this means that sites that are separated from the network while involved in a transaction that spans several sites must stop. Thus SDD-1 does not achieve our goal of partial operability.

1.4.2 Reliability

The work in reliability is perhaps less developed than that on concurrency control. An important paper by Johnson and Thomas [Johnson75] describes an algorithm for updating redundantly stored data such that all copies converge to the same final value. The paper uses the notion of a timestamp, which expresses the order in which updates should be performed, so that all copies converge to the same final value, even if the updates are delayed, duplicated, or arrive out of order. Timestamps have been used in many protocols for reliable synchronization. This paper does not discuss the problem of synchronization for concurrent updates.

Thomas [Thomas76] proposed an extension of the ideas in that paper to provide synchronization. An algorithm was developed to allow updates to be performed as long as more than half of the sites were functioning. The algorithm is complex, and several flaws were found in the early versions. Another major problem with the Thomas algorithm is that it applies only to cases where the entire data base is stored at each site.

Alsberg and Day [Alsberg76] have developed a robust multi-copy update algorithm with a somewhat different approach. They designate one copy as the primary, and insist that all accesses occur through the primary copy. The other copies serve only as backups in case the primary fails. This strategy eliminates one of the major advantages of replication of data, that of greater concurrency in access. The algorithm does, however, seem applicable where the only concern is greater reliability, and not greater concurrency.

A forthcoming paper by Lampson and Sturgis [Lampson76] presents a general discussion of performing atomic transactions in a distributed system. The paper presents a method of storing and updating information in a single machine, such that it is preserved and updated correctly even if crashes occur during updates. This storage technique is useful for implementing an atomic update within one site.

The last part of that paper gives an algorithm for performing updates at several different sites atomically. A complicated protocol is used to distribute the updated values to each site, such that during most of the procedure, each site can independently decide to abort the update if messages are slow in arriving. There is still, however, a time window in which a site must wait for the arrival of message from other another site, and cannot decide whether or not to abort the update if such a message is slow in arriving. This algorithm is similar to the two-phase commit protocol described by Gray [Gray77] and that used by Reed [Reed78]. The Lampson and Sturgis algorithm makes the time window during which a site can not abandon a transaction interrupted by a failure quite small by insuring that all of the computation done by the transaction will be completed before any site is prevented from abandoning the transaction. This is accomplished via extra steps in the protocol and extra message exchanges. Chapter 5 discusses commit protocols in much greater detail.

Reed [Reed78] is also working in the area of robust synchronization mechanisms. He has developed a scheme in which each value assigned to an item can be named as a version of that item. The scheme allows a transaction to obtain a set of mutually consistent values for the items that it accesses by choosing the proper version names. This scheme is subject to the same limitations as the Stearns and Rosencrantz scheme, in that a transaction may need to be aborted to avoid deadlock. This problem is solved by having all of the updates performed by a transaction (by creating new versions) be conditional until the transaction has been completed.

This same mechanism of conditional transactions is used to solve the atomic distributed update problem. The mechanism is simple and convincing, but still leaves a time window in which a failure can cause delay in processing new transactions.

1.4.3 Relationship of this Thesis to Previous Work

This thesis presents a model for distributed computing that specifies the effects of failures on computation. The model is similar to the Actors model of computation [Hewitt76,Hewitt77]. The model describes computation performed by an unreliable system, in which components can fail and failures effect the outcome of the primitive operations of the model. The thesis discusses implementation techniques that can be used to insure that the actual effects of failures conform to their effects as described in the model. The techniques used build on the work of Lampson and Sturgis [Lampson76] and Gray [Gray77].

While much research has been done on the problems of synchronization in message based models of computation [Atkinson78,Halstead78,Hewitt77], much of this work has centered on developing primitive synchronization techniques that achieve mutual exclusion.

This thesis shows how to apply such techniques (the atomic process steps of the model) to a more complicated problem: coordinating transactions.

Most of the work on control of concurrent transactions has been on mechanisms that allow transactions to deadlock, using some mechanism to detect a deadlock situation and abort one of the transactions to resolve the deadlock and allow the others to proceed. The mechanism presented in this thesis instead avoids deadlock. This mechanism allows more concurrency than many other deadlock avoiding synchronization schemes, by postponing the actual locking of a resource until it is needed or must be locked to avoid deadlock with some conflicting transaction which needs that resource. This approach avoids unnecessary locking that restricts concurrency.

The mechanism used for synchronization in this thesis is based on control of the order in which messages are delivered. Implementations of the control algorithm that make efficient use of the kinds of communication networks frequently used to interconnect sites in a distributed information system are given. This approach is quite different from and can be substantially less costly than the approach taken by most of the work on synchronization in distributed systems which makes no assumptions about the communication network and implements synchronization constraints with higher level protocols.

The technique used to coordinate transactions involves an analysis of the access pattern of transactions that is similar to that used in SDD-1 [Bernstein77], but more fine grained in that the actual derivation of each output of a transaction from the inputs to that transaction is used in the analysis, rather than basing the analysis on the assumption that every output of a transaction depends on every input, as is done in SDD-1. This analysis shows how to structure the synchronization scheme so that frequent or important transactions can be performed with minimal overhead due to the synchronization.

The thesis includes a proof that it is impossible to solve the "atomic distributed update" problem for all cases in a way that achieves the goal of partial operability, given the semantics of the model presented here. The proof applies arguments advanced by [Gray77] and [Akkoyunlu75] to the model of distributed computing presented in the thesis.

A novel approach to the atomic distributed update problem is presented. This approach involves keeping several current values for some data items, and builds on the version naming synchronization schemes of Reed [Reed78] and Stearns et al. [Stearns76]. This approach is not limited to the particular synchronization scheme discussed in this thesis, but is applicable to any of the synchronization schemes discussed above.

To summarize, I feel that the important contributions of this thesis are:

A model for distributed computing in which the effects of failures are well specified and implementation techniques for meeting these specifications

A technique for coordinating what I refer to as an "atomic broadcast" that can be implemented efficiently in the kinds of computer networks currently used to connect sites in distributed information systems

A technique for analyzing a set of transactions to be performed to determine which ones can be performed without locking

A mechanism for locking data items at several sites in order to perform a distributed atomic update without allowing a failure to delay access to the locked data indefinitely, in most cases

1.5 Thesis Plan

Chapter 2 presents the process model of distributed computing that is used throughout the thesis. Techniques for implementing a distributed system that behaves as specified by the process model are discussed. The problem of synchronizing transactions is formulated in terms of this model. The chapter discusses several ways in which the order of execution of transactions can be controlled, and shows that only one of these achieves the goal of partial operability.

Chapter 3 discusses a simple synchronization problem that consists of coordinating what I refer to as an atomic broadcast. An atomic broadcast distributes a set of messages to a set of receivers such that the order in which any one receiver sees messages from several such broadcasts is consistent with the order in which the broadcasts are received by any other receiver. A simple mechanism to perform this task is presented. This mechanism forms the basis of the synchronization mechanism for concurrent transactions discussed in Chapter 4. Implementations of this mechanism that take advantage of the synchronization constraints imposed by the communication network are discussed. These implementations distribute the messages with very little overhead attributable to the enforcement of synchronization constraints.

Chapter 4 discusses the problem of synchronizing transactions. A technique for analyzing a set of transactions to determine what synchronization protocols are needed is discussed. This analysis is used to show that correct synchronization of all transactions cannot be accomplished with a protocol that achieves the goal of partial operability. Three different classes of transactions are distinguished, on the basis of their access patterns. A mechanism that builds on the atomic broadcast mechanism of Chapter 3 is presented to perform transactions. This mechanism can be tailored to minimize the cost of synchronizing

transactions that are expected to be performed frequently. The mechanism is general, however, in that any transaction, expected or unexpected, will be correctly synchronized. Unexpected transactions have little impact on the efficient operation of the synchronization mechanism for the expected transactions.

Chapter 5 considers the implications of the need for locking on the goals of partial operability and autonomy. These goals dictate that a site that has set a lock for some transaction should be able to decide to abort that transaction if a failure interferes with the prompt completion of the transaction or if the transaction violates the access policy of the site. I show that there is no protocol that can be used to insure that no failure can prevent a functioning site from promptly completing or aborting a transaction requiring locking.

As a solution to this problem, I propose a novel mechanism that allows locked data items to be made available to other transactions before the completion or abortion of the locking transaction is decided. This mechanism is appropriate for systems in which the ability to perform transactions in real time, without long delays waiting for locks to be released, is important.

Chapter 6 presents a comprehensive example showing how to apply the techniques of this thesis to a typical distributed information system. The example is an inventory control system described in a report on SDD-1 [Bernstein77]. The techniques of this thesis are used to develop a robust synchronization scheme for this example with little overhead due to the synchronization.

Chapter 7 summarizes the new ideas in the thesis and discusses areas for future research.

Chapter 2

The Process Model of Distributed Computing

This chapter presents the model for distributed computing that will be used in discussing synchronization in a distributed information system. The first section presents the model which includes specifications of the effects of failures on computation expressed in the model. Implementation strategies for limiting the impact of actual failures to the to the failure effects specified in the model are discussed. The second section poses the problem of performing transactions (as described in Chapter 1) atomically in the framework of the model. Various techniques that could be used for synchronization are discussed to show that only one of these can be used by a system that achieves the goal of partial operability.

2.1 The Model

Based on the assumptions and goals set forth in the previous chapter, I will now describe a model for computation in a distributed information system. In order to centralize the description, this chapter presents all of the model, even though some of the concepts will not be used until much later in the thesis. This model includes two forms of communication: message passing, and changes in state observable by later computations. Message passing may occur between sites or within one site. Communication through state changes, however, occurs only within a single site.

2.1.1 Definitions:

The basic unit of the model is a process.[1] A process can be viewed as the unit within which communication through state changes can occur. A process consists of a local state, a set of input ports, and a set of process step specifications. The computation performed by a process takes place in a series of process steps. A process step maps an input local state and a set of input messages into an output local state and a set of output messages. Each process step specification specifies the form of a process step, by stating:

> A set of input ports for the step. One message is received by the step from each port in this set.
>
> The output local state as a function of the input local state and the messages received.
>
> A set of output messages and their destination ports. Both the message contents and destination ports must be specified as functions of the input local state and the messages received.

An important point to note about a process step is that it *computes* its output messages and output local state. Thus a single process step can be used to perform computation on the local state of the process and the messages received, rather than simply retrieving information from the local state or storing information in the local state in response to messages. This capability of a process step to perform computation is used in the implementation of a transaction, as will be discussed in Chapter 4.

---

1. The word process has been used to denote a number of ill specified concepts in the literature. My use of the term process is not inconsistent with the common usage of the term, however the reader should realize that the term has a very specific meaning in this thesis. Other terms that have been used for very similar concepts are Actor [Hewitt76], and message handler [Reed78].

Conceptually, each process resides at one site, its home site. The home site of a process is the location of the process state of a process. The home site also is responsible for carrying out process steps. The fact that each process is implemented at a single site will be used in determining the effects of failures on the execution of process steps in this model.

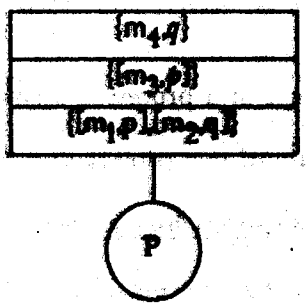
Each of the process steps of a process is atomic with respect to the other steps of that process. The output local state of one process step becomes the input local state of the next step in the sequence. The execution history of a process consists of the sequence of steps that have been performed by that process. For each process p, there is an ordering $<_p$ on the steps of p, such that $s_1 <_p s_2$ if s_1 preceded s_2 in the execution history of p.

The set of messages that a process has received in each of its process steps and the initial local state of the process form a complete description of its execution history. From the messages received at each step and the process step specifications, one can deduce the messages that are produced and the changes made to the process state. The input messages to each step can be represented by a set I of [message,port] pairs describing the messages received and the ports at which they were received.

Figure 2.1 shows an example of an execution history. The figure shows a list describing the input messages to the process steps of P. The first process step of P is represented by the bottom entry in the list, with subsequent process steps higher in the list. This list may be thought of as a log that records the messages received by P. When a process receives messages at a single port only, the execution history can be represented by a list of messages received, as each step receives a single message at that port.

Figure 2.1

The Execution History of a Process



The local state of a process is private to that process and can only be changed by process steps of that process. Similarly, for each port, there is a single process that receives the messages sent to that port.[1] The lack of "sharing" of ports or local state between processes greatly simplifies the implementation of processes in a distributed system.

The characteristics of a process stated above (sequential execution of process steps, each process at one home site, and no sharing of process state) describe the *behavior* that a process must exhibit, not its implementation. In practice any implementation that behaves as described above is satisfactory. In an implementation of processes, for example, process steps may be executed concurrently so long as the behavior is the same as that produced by sequential execution.

---

1. Note, however, that several processes may *send* messages to the same port.

One can view the execution of a process as being performed by an interpreter that carries out the execution of all of the processes in a system. This interpreter maintains a local state for each process and a set of messages for each input port. One cycle of this interpreter selects a process step specification of some process, selects a message from each of the input ports for that step, and carries out the selected step. The interpreter deletes the received messages from the sets of messages for the input ports, changes the local state of the process, and adds any output messages produced to the sets of pending messages for the appropriate ports.

The interpretation can be distributed (one interpreter for each process) because the only interaction between steps of one process and steps of some other process is the sending of output messages produced by process to input ports of some other process. This interaction can easily be accomplished by message passing between the distributed interpreters.

2.1.2 Effects of Failures in the Model

The process step specifications completely specificy any computation taking place in the absence of failures of the underlying mechanism that carries out the process steps. This section discusses the kinds of failures that can arise in a distributed information system and their effect on the execution of process steps and on message passing between processes. Two extensions of the process model to include a specification of the effects of failures on computation are presented.

2.1.2.1 Types of Failures

Two different kinds of failures can occur in a distributed information system: site failures, and communication failures. A site failure is local to some site, and can cause processing at that site to be suspended, or cause information stored at that site to be lost or damaged. A communication failure causes inter-process messages to be lost, delayed, damaged, or delivered to the wrong recipient. Many implementation techniques can be used to hide the effects of failures from a user computation.

An error detecting code, or checksum, can be used to detect messages that have been damaged or delivered to the wrong recipient. While it is impossible to detect all such errors[1] the probability of undetected communication errors can be made arbitrarily small by increasing the proportion of each message devoted to error detection. I will therefore make the assumption that all communication errors can be detected and henceforth ignore the arbitrarily small, but non-zero probability of an undetected error. If any message that is found to be in error is discarded, the possible effects of communication errors are limited to lost or delayed messages.[2]

The most common effect of a site failure is that computation at that site is temporarily suspended, and that some of the computation in progress at the time of the failure may be lost. If a site failure were to occur during the execution of a process step, that

---

1. No matter what kind of error detecting code is used, there is a chance that a communication failure will cause a message to be transformed so that it appears to be correct to the error detection mechanism but does not correspond to the original message.

2. Many communication systems exhibit another failure mode in which a message is duplicated. In designing a communication protocol, one has a choice as to whether to guarantee that all messages are delivered reliably, possibly delivering some twice, or to guarantee that each message arrives at most once, and that some messages may be lost. I have chosen the latter alternative. In the next section, I introduce the concept of robust sequenced communication which hides the effects of both of these kinds of failures.

step might be left partially completed, with the local state of the process corresponding to neither the input state nor the output state of that step. This can be prevented by using a robust storage management technique for storing the local state of a process. Such a technique allows a group of updates to be made atomically to information stored at one site, such that if a failure occurs either all or none of the updates take place. The atomic stable storage mechanism of Lampson and Sturgis [Lampson76] is such a technique. A description of all of the updates to be performed, known as an intentions list, is formed and written to permanent storage in a single operation before any of the updates are carried out. A failure occuring before the intentions list is generated or one interfering with the writting out of the intentions list causes none of the updates to be performed. Once the intentions list has been written, however, the error recovery mechanism can use it to insure that all of the updates specified will be made, even if the site making the updates fails after having partially completed them. The write-ahead-log protocol of Gray [Gray77] also provides the same capability for making a collection of updates atomically, by writing out a description of the updates to be made to a log tape *before* any of the updates are made.

Each process step can be implemented as an atomic update to stable storage. This implementation insures that a site failure leaves the local state of a process executing a process state either at the input state to that step or the output state of that step, and not some intermediate state or mixture of the two.

2.1.2.2 Two Ways to Include Failure Effects in the Process Model

By using the low level implementation techniques discussed above, one can constrain the way in which failures affect execution of processes. By augmenting the definitions of the process model to include specifications of the effects of failures, we can produce a model that describes computations in a "real" distributed information system in which site and

communication failures can occur. The choice of the specifications of the effects of failures should be made so as to reduce the effects of actual failures on the model, but also to be sure that an implementation of processes in which the effects of failures are limited to the specifications can be obtained. I consider two different failure specifications, one that is easy to implement and one that is harder to implement but limits the effects of failures more severely.

2.1.2.2.1 Simple Processes

Using basically the techniques described above, one can build an implementation of processes in which the effects of a site or communication failure are limited to lost or delayed messages. This is done by using error detecting codes to detect communication errors and discard messages that are in error, and storing the local state of each process in atomic stable storage. Some care must be taken in the implementation of a process step to insure that no possible failure causes messages to be apparently duplicated, by causing some part of a process step to be repeated. If a process step is restarted, after being partially completed, then it may send out the same message twice (once before being restarted and once after), or may modify its local state as if it had received the same message twice.

These undesirable effects can be avoided by performing a process step in three stages. First, delete any record of the input messages to the step so that a site failure occuring at this point would cause them to be lost. Then, perform the process step and update the local state of the process to reflect its completion. Finally, distribute the output messages of the process step to their destination ports. A site failure occuring before the local state of the process is updated can result in the process step not being performed, or an apparent loss of all of the input messages to that step. A failure after this point may cause output messages of the step to be lost. No failure causes the local state of a process to be

modified as if a process step were performed twice, or causes the messages produced by a process step to appear to be duplicated.

A less likely result of a site failure is that the information stored at a site in permanent stable storage is damaged. This can be detected, with high probability, through the use of error detecting codes. As with communications failures, however, it is impossible to detect all such errors. The local state of a process can be replicated within one site to decrease the probability that a failure will destroy all copies. A process step is implemented as an atomic update to all of the copies of the process state. Any copy of the local state of a process that survives a site failure can thus be used to become the current local state of the process.

To summarize, the effects of a site failure can be limited to lost messages (through a process step that was aborted after receiving messages), or delay of processes at that site. This is achieved by using atomic stable storage to represent the local state of processes, replicating local states, using error an error detecting code to detect damage to a local state, and indefinitely suspending any process for which no valid local state can be found.

Limiting the effects of failures to lost messages or delayed execution can easily be achieved without excessive communication or processing overhead. Many applications require a higher degree of reliability. In the next section, I discuss a different implementation of processes that gives a greater degree of reliability with greater overhead.

2.1.2.2.2 Robust Sequenced Processes

The effects of failures on simple processes are well specified, but still undesirable for most applications. For many applications, guaranteed delivery of all messages sent by a process to a port is desirable. This is a very difficult constraint to express, as indefinite delay of the delivery of any particular message by a communication failure cannot be prevented. In order to clarify what I mean by guaranteed delivery, I will introduce a constraint that I refer to as sequencing on the delivery of messages. Sequencing implies that messages sent from one process p to a port q are received at q in the same order in which they were sent by p. Robust sequencing implies in addition that no messages are lost.

For each port q define the ordering $<_q$ on the messages received at port q to be the total order in which those messages were received. For each process p the ordering $<_p$ on the process steps of p describes the order of occurrence of those steps. What I mean by robust and sequenced message delivery is that for any process p that sends messages to port q, the order $<_q$ in which the messages sent by p are received at q is exactly the same as the order in which the steps that produced those messages are ordered by $<_p$. This means not only that the messages are received at q in the same order in which they were produced, but also that there are no gaps in the sequence of messages received. Reception of message m sent by p to q can only occur after reception of any message m' for which $m' <_p m$.

Robust sequenced processes can be implemented by separating the execution of process steps from the actual communication of messages from one site to another. This can be done by maintaining a process database for each process. The process database for a process p contains the local state of p, an input message queue for each input port to p, and an output message queue for each port to which p has sent a message. Each output message queue contains a list of messages and a transmit sequence number (TSN). The

input message queue for a port q contains a list of messages and a set of receive sequence numbers (RSNs), one for each process that has sent a message to q. A process database is stored using atomic stable storage, so that a site failure during modification does not cause a process database to be left in some intermediate state.

A process step of p can now be implemented as an atomic update to the process database of p, which removes the messages received by that step from the input message queues, changes the local state of p, and appends the messages produced by that step to the output message queues. (If there is no queue for some destination port, a new one is created).

Messages can be transfered from an output message queue of a process p for a destination port q to the input message queue for q with a robust communication protocol using the sequence numbers RSN and TSN. Briefly, each site periodically attempts to send the first message in any non-empty output queue, attaching the TSN of that queue to the message sent. When the site holding port q receives a message sent from p, it verifies that the sequence number attached to that message is equal to the RSN of q for p, and if so updates the process database of the process associated with port q to add the message received to the end of the input queue for q, and to increment the RSN of port q for p. Whether or not the sequence number of the message received is correct, the receiving site sends an acknowledgement to the site holding p containing the RSN of q for p. This acknowledgement informs the sender of the most recently received message. The acknowledgement either acknowledges receipt of a message or informs the sender that retransmission of some message may be required. When the site holding p receives such an acknowledgement, it verifies that the sequence number in the acknowledgement is the same as the TSN of the message queue for q in the process database of p, and if so deletes the first message in that queue and increments the TSN.

I will not at this point explain how the message queues are initially set up when two processes first begin to communicate with each other. This is somewhat complicated and will be discussed at length in Chapter 3, where a use for robust sequenced processes is discussed.

This implementation of processes guarantees delivery of inter-process messages in sequence. The cost of the protocol is the extra messages (acknowledgements) used, and the storage required for the message queues and sequence numbers. This cost is small if each process converses with relatively few processes, and if messages in the output queues are promptly forwarded.[1] In the synchronization protocols used in this thesis, each process converses directly with relatively few other processes, thus the cost of robust communication is small.

2.1.3 A Justification for This Model

A number of models have been proposed for distributed computing. I feel that the model described above best reflects the goals and assumptions of the kind of distributed information system discussed in Chapter 1.

One semantic model that has been proposed for distributed computing is the object model [Liskov77,Saltzer78]. In the object model, information is represented by typed objects. For each type of object, there is a set of operations, such as *add*, *subtract*, *multiply*, and *divide* for integer objects, which can be used to manipulate objects of that type. In addition to primitive objects, such as integers or *booleans*, any user may define a new type of objects,

---

1. Any site that does not wish to devote space to large input or output message queues can refuse to execute a process step for a process with non-empty output queues, and can refuse to acknowledge a message sent to a port with a non-empty queue. These measures do, however, introduce the possibility of deadlock if the application requires buffering between processes.

describing the operations that can be performed on objects of the new type and a representation for objects of the new type. Computation is performed as sequences of operations on the sets of objects.

While the object model is a very natural one for many users, several problems arise in the application of the object model to distributed computing. The most serious of these problems is that it is unclear what the appropriate semantics for accessing remotely managed objects should be. Many suggestions have been made, including treating all object references uniformly, whether local or remote, treating references to remote objects specially and maintaining a local copy of the remote object, and disallowing references to remote objects, and instead using message oriented communication between sites. The first of these suggestions is difficult to implement, while the others violate the conceptual simplicity of the object model.

The uniform object model (in which a user computation does not distinguish between references to local and to remote objects) is difficult to implement reliably. Operations that involve objects at different sites can fail in different ways (due to the possibility of communication failures) than operations on objects all at one site. Hiding the different failure modes from the user is difficult or impossible, forcing the user to deal with the problem of determining what the outcome of a sequence of operations on objects will be if failures interfere with their normal completion.

Several similar semantic models based on message passing have been developed for distributed computing. These include Actors [Hewitt76], the μ-calculus [Halstead78], and data flow [Dennis75]. These models in their pure form all describe computation such that the only communication between primitive computation events is through explicit message passing.

The Actors model provides a uniform way of describing any computational activity as a group of events, each event being the reception of a message by an Actor. One problem with the Actors model is that exactly which Actors are primitive, implementing their effects directly rather than sending messages to other Actors to achieve ther effects, is left unspecified. Thus it is sometimes awkward to deal with the Actors model, as there is always another level of description "below" any level that you choose.

Side effects are introduced into the model with the notion of a cell, an Actor that remembers one of the messages that it has been sent and repeats that message on request. This primitive mechanism can be used for modeling computation in which the processing to be applied to some message is not known in advance and is dependent on some future event, such as storing a data item for later transactions. Cells are also used to implement events in which two or more messages are logically "received" (by using cells to store messages), as the Actors model does not allow an Actor to receive two or more messages in a single event.

The μ-calculus is similar in principle to the actors model. It, however, provides a mechanism for introducing primitive functions that are not implemented by message passing. Cells are used in this model as well, for storing values for later use. In addition, a mechanism called a token is introduced to provide a way for a pair of messages to be received in one event. While the token mechanism is more straight-forward than using a cell, it is still rather hard to understand. Moreover, the implementation of a distributed system based on the μ-calculus seems difficult (and in fact the protocols presented for one such implementation [Halstead78] are very complex), particularly in implementing cells and tokens.

Data flow schemas have frequently been used as a tool for describing repetitive processing, such as computing a Fourier transform. A data flow schema provides a natural mechanism for events in which two or more messages are received, unlike the above models. Unfortunately, computations in which the processing to be applied to some message depends highly on the contents of the message are hard to describe in data flow. Recursion and iteration are somewhat difficult to express naturally, and greatly add to the difficulty of implementation. A data flow description of computation where a lot of information is stored for later (unknown) use, such as a data management system, is awkward.

The process model previously described is an attempt to bring together some of the good features of the models described above, without the disadvantages. The two different forms of communication provided in the process model represent the properties of communication in a distributed system better than either observation of state changes or message passing alone. It is easy to specify the effects of a failure in a system based on processes, and to build an implementation of processes that meets the specifications. Distinguishing between intra-process and inter-process communication encourages the user to plan his application carefully so as to minimize unnecessary communication between sites, and to plan for site or communications failures.

The process model also captures the concept of autonomy. All stored data is represented by the local states of the processes. No process can be "coerced" into performing some function for any other process. All access to stored information is mediated by some process that can implement its own access control policy. This allows the problem of access control to be largely ignored in the model, as each process can provide its own access control policy. At the same time, the process step specifications of a process specify the access control policy of that process by stating what the process does in response to the messages that it

receives. Thus a user implementing some application can examine the process step specifications of processes providing services that he wishes to use and can in many cases determine whether or not access restrictions will be encountered in his application.

Also inherent in the process model is the notion that some processing activities can be performed simply by one site. Although each process has an overall specification of its operation, in a real system most processes will be implemented from smaller pieces. I will not specify what those pieces are, as the implementation of a process could be based on a message passing system, a conventional programming language, or the object model, depending on what is deemed most convenient. Within one process, however, one need not deal with the special problems of a distributed information system, as each process is executed solely at one site.

The mechanism used to specify a processing event that logically receives messages from two or more sources (multiple ports) seems much more natural in the process model than the mechanisms using cells or tokens. As events in which two or more messages are received are common in many applications and can be constructed from the primitives in the Actors or μ-calculus models, there seems to be no reason not to include this important special case in the model. Inclusion of this capability identifies for the implementor of the system the cases where two messages are being received by what is logically one process step. This makes it simpler to construct an efficient and robust implementation than if the multi-port receive were simulated using some more general mechanism.

The association of several independently named ports with one process is a very useful feature of the process model. It can be used to group several independent processing activities that wish to communicate via a shared data base in a single process. Such processing activities can be implemented as independent process step specifications of the

same process, each of which receives its input messages through a different set of input ports. This use of processes is similar to a monitor [Hoare74], or a critical section.

A second, more important feature of ports is that they provide a way to classify the messages sent to a process *before* messages are received. One application of this capability would be a process with several queues of pending messages that are serviced with some priority algorithm, not necessarily in the order in which the messages arrived. Ports also allow a process to temporarily ignore one class of messages while exchanging messages with other processes to complete some processing activity. This use of ports will be demonstrated by the locking strategy discussed in Chapter 4.

The differences between my model and the others are a reflection of different goals. My model is an attempt to provide a way to express applications for a distributed information system clearly, such that the effects of failures are well specified. Others have been more concerned with formality and minimization of the primitive concepts.

2.2 Atomic Transactions Revisited

This section examines the problem of performing transactions atomically as expressed in the framework of the process model. We show how the simple definitions of transactions given in Chapter 1 can be stated in terms of the process model and show how to express the property that transactions are performed atomically as constraints on the order of execution of process steps. Several mechanisms that could be used to control this order of execution to achieve atomic transactions are discussed.

2.2.1 Expressing Transactions in the Process Model

Recall that a transaction is a set of accesses to stored data items. The definitions of a data management system given in the previous chapter can be mapped onto the process model by using several processes (at least one for each site) which I refer to as data managers. Each data manager maintains some of the data items as components of its local process state. The process steps of a data manager perform the accesses to the data items held by that manager. If a data item is replicated, with several sites having copies, then several data manager processes maintain copies of that item.

A transaction in the process model consists of a set of process steps of the data manager processes which together carry out the accesses needed to perform the transaction. Each data manager may perform several steps in carrying out a single transaction. If communication between managers is required to perform a transaction, then the output messages of some of the processes steps that perform that transaction will be used as input messages in some of the other process steps performing the same transaction.

In addition to the data manager processes, which implement accesses to data items, there are transaction processes, which perform the function of translating from a high level description of the transaction to a set of messages to be sent to the data managers. These messages direct the data managers to perform the necessary accesses to carry out the transactions. A more detailed description of the function of the data managers and transaction processes is given in Chapter 4, which discusses mechanisms for performing transactions.

2.2.2 Performing Transactions Atomically

Intuitively, a transaction is atomic if either all or none of its effects are visible to other transactions. There are two ways in which one transaction may observe the effects of other transactions: messages sent by steps of one transaction that are received by steps of another transaction, and the modifications of local process states that are made by steps of one transaction and later observed by steps of another transaction.

The first method of observation, direct message passing, rarely occurs. This is because a transaction is a complete, independent processing activity and does not in general communicate directly with other transactions. The exception to this case is that the user who submits a transaction (by sending a message into the distributed information system) may know of other transactions by having received messages sent from other transactions. Controlling sequencing of transactions so that the order of transactions as perceived from explicit message passing is consistent with their order as perceived from observations of modifications to local state is relatively simple. For the moment, I will present a definition of atomic transactions that ignores this method of observing ordering. Chapter 3 discusses this problem further.

The second source of communication between transactions, the local process states, is much more important in most applications. Recall that for each process p there is an ordering relationship $<_p$ that defines the relative order of occurrence of process steps of p. These local ordering relationships can be used to define an ordering of the transactions as follows:

> Transaction $T_1 < T_2$ iff there is a process p and process steps s_1 and s_2 of p such that s_1 is a part of T_1, and s_2 is a part of T_2 and $s_1 <_p s_2$.

Thus two transactions are ordered if both contain steps of the same process. This ordering is a reflection of which transactions may have directly observed effects of which other transactions. A transaction T_1 can also observe the effects of some transaction T_3 indirectly if there is some transaction T_2 such that $T_3 < T_2$ (because of the order of process steps of some process p) and $T_2 < T_1$ (because of the order of process steps of some other process q. Indirect observation can occur because the effects of a transaction may depend on the values that that transaction saw.

The condition that we require for a transaction to be atomic is that either all or none of its effects be reflected in the data values seen by other transactions. If a group of transactions is performed atomically, then the effects of those transactions (modifications made to the values of data items and messages produced by the transactions) are the same as if the transactions were performed serially in some sequence, with each transaction being entirely completed before the next transaction in the sequence is begun. This requirement can be expressed as a condition on the $<$ ordering resulting from the execution of transactions as follows:

> Transaction t is atomic with respect to a set of transactions T if there is no sequence of transactions $t_1, ..., t_n$ in T such that $t_i < t_{i+1}$ for $1 \leq i < n$, and $t_n < t < t_1$. Equivalently, a set of transactions is atomic if the transitive closure[1] of the $<$ ordering, $<^*$, is a partial order on that set of transactions.

In order to insure that a set of transactions is performed atomically, we must insure that the $<$ ordering resulting from any concurrent execution of those transactions is cycle free. One way to insure this is by choosing the assignment of data items to data managers such

---

1. Throughout this thesis, I will use the superscript $*$ to designate a reflexive transitive closure (i.e. $x <^* x$ for any x).

that for each transaction there is a single data manager that can perform that transaction. Thus each transaction is seen by only one data manager, and no cycles in the < ordering can arise.

This approach can be rejected because the assignment of data items to managers is not solely under control of the system designer. The autonomy of individual sites dictates that certain items must be managed by processes at certain sites. Some transactions may need to access data items from several different sites. Because each process must be executed at one site, there is no way to have one data manager process perform a transaction at several sites.[1]

Perhaps a more serious objection to this proposal is that it makes the addition of new transactions, which access items in patterns that were not planned, difficult or impossible. Adding a new transaction may require complete redesign of the system so as to allow new transaction to be performed by a single process.

We therefore must show how to coordinate transactions that involve process steps from several different processes. This can be accomplished by controlling the order in which the data managers perform the process steps which perform accesses of transactions. The next section discusses four primitive mechanisms that could be used in coordinating the process steps of the data managers.

---

1. Recall that the specification of the effects of failures on the execution of a process was greatly simplified by the fact that each process is executed at one site. Therefore, we do not wish to abandon this assumption.

2.2.3 Primitive Synchronization Mechanisms in the Process Model

There are several mechanisms available in the process model that could be used to constrain the order in which processing operations are performed by processes. These mechanisms could be used to construct a solution to the problem of performing transactions atomically, in much the same way as mechanisms such as Semaphores [Dijkstra68] or Monitors [Hoare74] are frequently used to construct solutions to other synchronization problems.

To achieve the goal of partial operability, the synchronization scheme for transactions must allow a transaction that is purely local to one data manager to be performed whenever a request to perform that transaction is sent to the data manager. Thus synchronization mechanisms that do not allow such transactions to be performed promptly should be avoided. The goal of partial operability will thus serve as a guide in selecting synchronization techniques for transactions.

One synchronization technique that has already been introduced is the sequencing of messages sent between processes. Sequencing consists of guaranteeing that messages sent from one process to a port are received at that port in the same order in which they were produced by the process. As we shall see in the next chapter, robust and sequenced message communication is sufficient to provide proper synchronization of many kinds of transactions.

Sequencing alone does not compromise the goal of partial operability. The only case in which the constraint of sequencing prevents a message sent from a process p to a port q from being promptly received and acted upon is the case in that there is a previous message from p to q that has not yet been received at q. Using the implementation of robust sequenced processes described earlier in this chapter, this situation is quickly remedied

whenever it occurs. Unfortunately, as we will demonstrate in Chapter 4, sequencing alone is not sufficient to perform all transactions atomically.

A second technique that could be used to control the order of execution of transactions is one that I call explicit locking. Explicit locking consists of postponing the reception of some class of message by a process until some other message has been received. Chapter 4 will discuss locking in greater detail and will introduce a mechanism for explicit locking into the process model.

A synchronization scheme using explicit locking does not achieve the goal of partial operability. Using explicit locking, a data manager could postpone the reception of a request to perform some local transaction until that data manager had received other messages. Explicit locking could cause the local transaction to be delayed indefinitely.

Sequencing and explicit locking both control the order of processing operations by controlling the order in which messages are received by processes. Another approach to the control of the order of execution of processing operations is to control what action is taken by a process on receiving a message. The following two synchronization techniques use this approach.

One way in which a process can postpone the processing operation requested by a message that that process receives is to record the message in the local state of the process. The stored message can be retrieved and acted on in a later process process step. One could call this technique squirreling.

Using squirreling, a transaction local to one data manager can be delayed indefinitely because the request to perform that transaction can be squirreled away indefinitely by that

data manager, pending reception of some other message. Thus squirreling does not achieve the goal of partial operability.

Another mechanism that can be used to postpone the processing requested by a message is to have a process that receives a message that the process should not yet act on send the message to another process. That other process would either act on the message, or pass it on again, possibly back to the first process. This technique could be refered to as buck passing. Buck passing also does not achieve the goal of partial operability, as a request to perform a transaction could be deferred indefinitely by being passed from process to process.

Both buck passing and squirreling are what could be called implicit locking (because request messages are not explicitly postponed, but the requested processing is postponed). Implicit locking is characterized by the fact that two or more process steps of the data manager receiving a request are used to perform the processing requested by a message.

When two or more process steps of a single data manager are used to carry out a transaction, the goal of partial operability is not achieved. If two or more process steps carry out accesses for a transaction, then other transactions that access the items accessed in those steps may have to be excluded from occurring between the two steps. If a failure delays the second step of a data manager performing accesses for a transaction, then transactions local to that data manager that must be excluded may be indefinitely delayed. If only one process step (of two or more) of a data manager performs accesses for the transaction, then some condition must be preventing those accesses from being performed by the first step of the data manager. The manager must in effect be waiting for some message before it will perform the accesses for the transaction. That message could be delayed indefinitely, delaying the transaction.

To summarize, the sequencing mechanism is the only one of the techniques for controlling concurrency in the process model that achieves the goal of partial operability. In Chapter 3, we will demonstrate a mechanism that uses sequencing to provide control for many processing operations. In Chapter 4 I demonstrate that sequencing alone is insufficient for coordination of all possible transaction, and show that some mechanism in which two process steps of some process are used to perform one transaction is needed.

2.3 Summary

This chapter presents a semantic model for a distributed information system in which the effects of failures are well specified. The model combines features of Actors, Data Flow, and the Object Model. The model makes a strong distinction between two forms of communication: inter-process messages, and intra-process communication through shared state information.

Two different classes of failures in a distributed information system were discussed: site failures and communication failures. We showed two ways in which the process model could be extended in order to include a specification of how computation is affected by such failures. One extension (simple processes) was easy to implement, but allowed failures to have relatively severe effects. A second extension (robust sequenced processes) limits the visible effects of failures, but requires more overhead in its implementation. The remainder of this thesis will make use of robust sequenced processes in developing algorithms for performing transactions.

The problem of performing transactions atomically is translated into the terminology of this model, and a plan for an implementation of a distributed information system based

on the model is given. A condition for determining whether or not a transaction is atomic is expressed in terms of the ordering relationships of the model.

Finally, techniques for controlling the order of execution of process steps were discussed. One of these techniques (sequencing) was shown to be consistent with our goal of partial operability. Other techniques allow a failure to delay the completion of a local processing operation indefinitely, but as we shall demonstrate in Chapter 4 are necessary for coordination of some kinds of transactions.

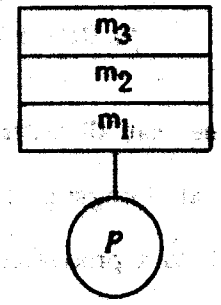
from the execution history of p. For a process that receives messages at one port only, the execution history can be represented as a list of messages received, as shown in Figure 3.1. This representation can be viewed as a log, recording each message received by p as it is received. The most recently received message in the execution history is at the top of the list.

I define a broadcast message to be a set of messages and destination ports. A broadcast message B can be represented by a set of pairs, $\{[m_1,p_1], \ldots, [m_n,p_n]\}$ such that m_i is a message, and p_i is the name of the port (and process) to which m_i is sent. The individual messages that make up a broadcast are referred to as the components of the broadcast. The order in which a group of receiving processes receive a group of broadcast messages can be derived from the order in which the components of those broadcasts are received by the individual processes. The order of two broadcast messages B_1 and B_2, is defined as $B_1 < B_2$ if B_1 contains a message m_1 sent to a process p, and B_2 contains a message m_2 also sent to p, and $m_1 <_p m_2$. This definition is completely analogous to the definition of the $\ll$ ordering

---

Figure 3.1

The Abbreviated Execution History of a Process



$m_1 <_p m_2 <_p m_3$

Chapter 3
Atomic Broadcasting

Many transactions performed by a distributed information system can be decomposed into independent component operations, each of which is performed at one site and does not depend on any other site. In the model of the previous chapter, each component of such a transaction is performed by a single process step. All of the messages that form the inputs to these process steps can be constructed in advance, before any step is performed. The ordering of such a transaction relative to other transactions is controlled by the order in which these messages are received.

In this chapter, I introduce a mechanism for atomic broadcasting, which distributes a set of messages to a set of destination ports so that they are received atomically with respect to other such sets. If an atomic broadcast is used to distribute the input messages for a transaction with independent components, that transaction is performed atomically. Atomic broadcasting is a simpler problem than that of coordinating arbitrary transactions.

3.1 Definitions

For convenience, I assume that all messages and all ports are uniquely identified. Many processes receive messages at a single port only. For such processes, I will use one identifier such as p to refer both to a process and the port at which that process receives messages. Recall that for each process p there is an ordering $<_p$ on the messages sent to p that reflects the order in which those messages are received. Each message m is included in the order $<_p$ when it is received by p. The ordering $<_p$ for a process can be determined

on transactions. Similarly, a broadcast message M is atomic with respect to some set of broadcast messages if the < ordering on those messages is cycle free.

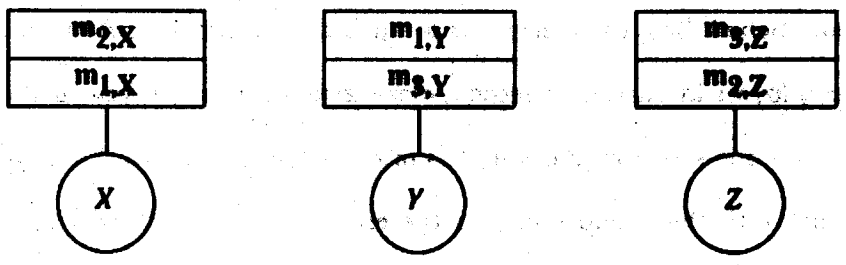
Figure 3.2 illustrates the reception of three broadcast messages that were not atomic. B_1 includes two component messages $m_{1,X}$ for *X* and $m_{1,Y}$ for *Y*. Similarly, B_2 and B_3 contain components for *X* and *Z*, and for *Y* and *Z* respectively. In this example, *X* receives a component of B_1 before one from B_2, *Y* receives a component of B_3 before one from B_2, and *Z* receives a component of B_2 before B_3. These ordering relationships constitute a cycle.

A second way in which two broadcast messages could be considered to be ordered is if the sender of one message was one of the receivers of the other. For the moment, I will ignore this kind of ordering relationship. A later section extends the notion of an atomic broadcast described here to include such relationships.

Figure 3.2

Non-Atomic Broadcasting

$B_1 = \{\langle m_{1,X}, X\rangle, \langle m_{1,Y}, Y\rangle\}$
$B_2 = \{\langle m_{2,X}, X\rangle, \langle m_{2,Z}, Z\rangle\}$
$B_3 = \{\langle m_{3,Y}, Y\rangle, \langle m_{3,Z}, Z\rangle\}$



$B_2 > B_1$ $B_1 > B_3$ $B_3 > B_2$

3.2 An Illustration of Atomic Broadcasting

The independence of the process steps to be coordinated in an atomic broadcast (the steps that receive the messages that make up the broadcast message) makes coordination of atomic broadcasts simpler than coordination of more general operations. A simple real world analogy may help to illustrate this point. Consider an office in which all communication is through interoffice memos. Sending some important notice to all employees about a change in working procedures is an instance of an atomic broadcast. The notice should be sent atomically, so that employees working on and communicating about the same project receive the notice at the same point in their work. This can be accomplished relatively easily through the office mail system. At one instant, all of the notices are entered into the mail system and take their places in the queues of mail waiting to be delivered to and read by the employees. After that, each employee will find the notice at the same point (relative to other mail) in his list of messages. It does not matter that some employee on vacation may not see the memo for a month or more, as he will eventually see it in the proper sequence relative to other mail.

Compare this situation with that of a group project, which requires a joint discussion by a group of employees. To complete such a project atomically with respect to other work in progress effectively requires that each group member set aside a certain time for the discussion. Scheduling the meeting is a much more difficult problem than placing a notice in each employees in basket. A second, more serious problem is that if the meeting has to be suspended for some reason, the members of the group can not work on any other project that may conflict with the group effort, as the effects of such work will not be known to other members of the group.

This analogy is crude, but gives a feeling of the differences involved. The distribution of the memo as an atomic act is easy, because there are no constraints on when the recipients actually read the memo. It is sufficient to place the memo in the correct sequence in each employee's mail.

3.3 A Mechanism for Atomic Broadcasting

In this section, I present a mechanism for coordinating atomic broadcasting that uses robust sequenced communication between processes to distribute the component messages of a broadcast message to their destination ports. The solution uses processes that I refer to as message forwarders to distribute these messages. Each message forwarder receives messages at a single input port. A message forwarder has a single process step specification which can be described by a function $f(M) = \{[m,p]\}$, mapping each message received to a set of output messages and destination ports for those messages.

The messages received or sent by a message forwarder each contain a set of component messages and destination ports. The components of each such message form a subset of the messages that comprise some atomic broadcast. Each process step of a message forwarder receives some input message and partitions the components of that message among the output messages that it produces. For each such step, the output messages together contain exactly the same set of components as the input message to that step.

The protocol for atomic broadcasting organizes all of the processes in the system, (message forwarders, transaction processes, and data managers), in a hierarchy. Each process p has a unique parent f in the hierarchy. I will also describe this relationship by saying that p is a child of f. I say that p and q are relatives if either p is the parent of q or q is the parent of p. In the hierarchy used for this protocol, each process f that is the parent of some

other process is also a message forwarder, and there is a single message forwarder r which is the root of the hierarchy, and is an ancestor of all other processes. The transaction processes and data managers form the leaves of this hierarchy. Any hierarchy of message forwarders can be used to perform atomic broadcasting. As we shall see, however, the organization of the hierarchy determines the number of messages that must be sent to distribute each broadcast, and should be made with some knowledge of the expected communication patterns.

In order to send an atomic broadcast, a process formulates a single message containing a set of components, each of which specifies a message to be sent and a destination port. This single message is sent to any message forwarder that is above all of the destinations in the hierarchy. Recall that each step of a message forwarder partitions the components of the message received among the output messages produced. Each message forwarder sends output messages only to its children in the hierarchy. On receiving a message, a message forwarder partitions the components of that message such that each component is sent to the child that is above the destination port of that component in the hierarchy.

Figure 3.3 illustrates the operation of this protocol in distributing the three broadcast messages shown in Figure 3.2. The processes are organized in a three-level hierarchy, where f is the parent of processes Y and Z, and r, the root, is the parent of f and X. Figure 3.3 a shows the orderings for all processes after B_1 and B_2 have been received by r and B_3 has been received by f. Figure 3.3 b shows an intermediate state in the distribution of messages to X, Y, and Z. Figure 3.3 c shows the final state when all components of all three broadcasts have been received.

Figure 3.3

Coordinating Atomic Broadcasts with Message Forwarders

$B_1 = \{[m_{1,X}, X], [m_{1,Y}, Y]\}$ **3.3 a**

$B_2 = \{[m_{2,X}, X], [m_{2,Z}, Z]\}$

$B_3 = \{[m_{3,Y}, Y], [m_{3,Z}, Z]\}$ **The Initial Execution State**

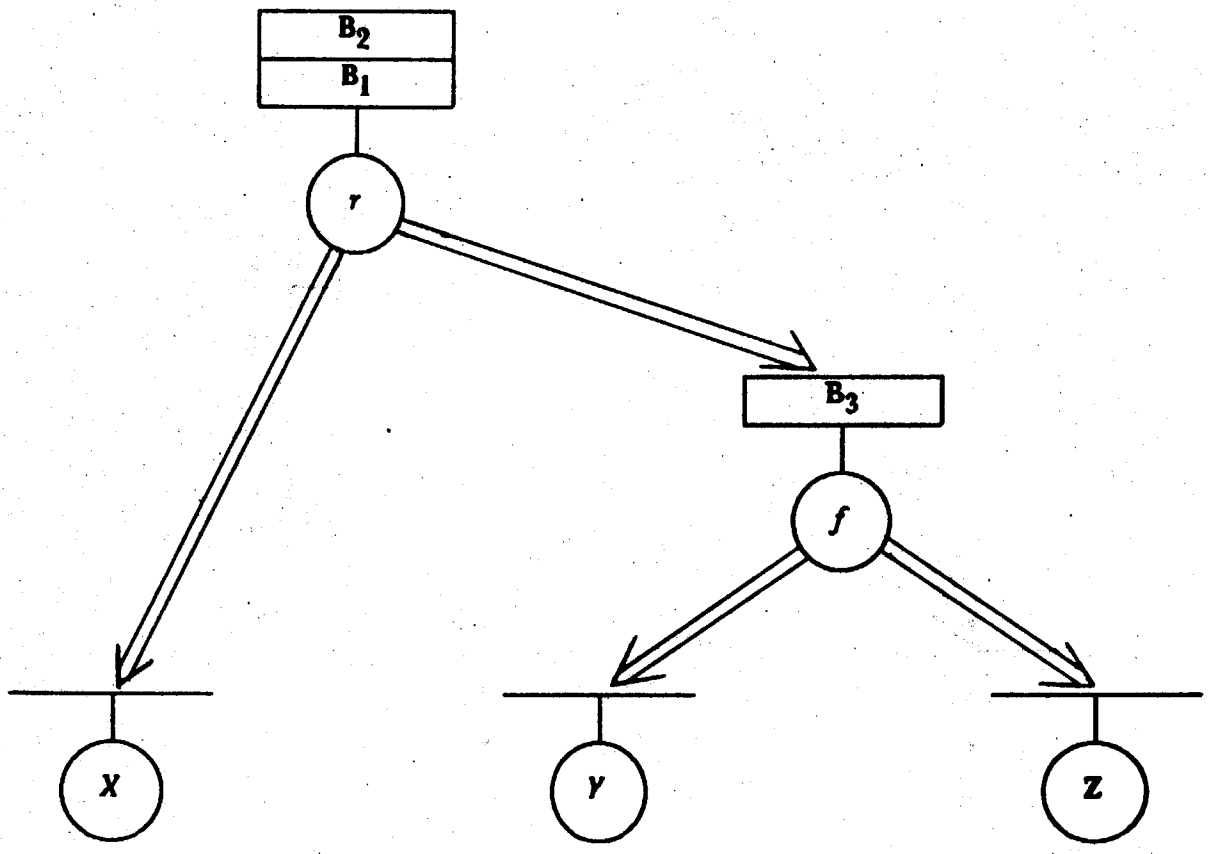


Figure 3.3b

An Intermediate State

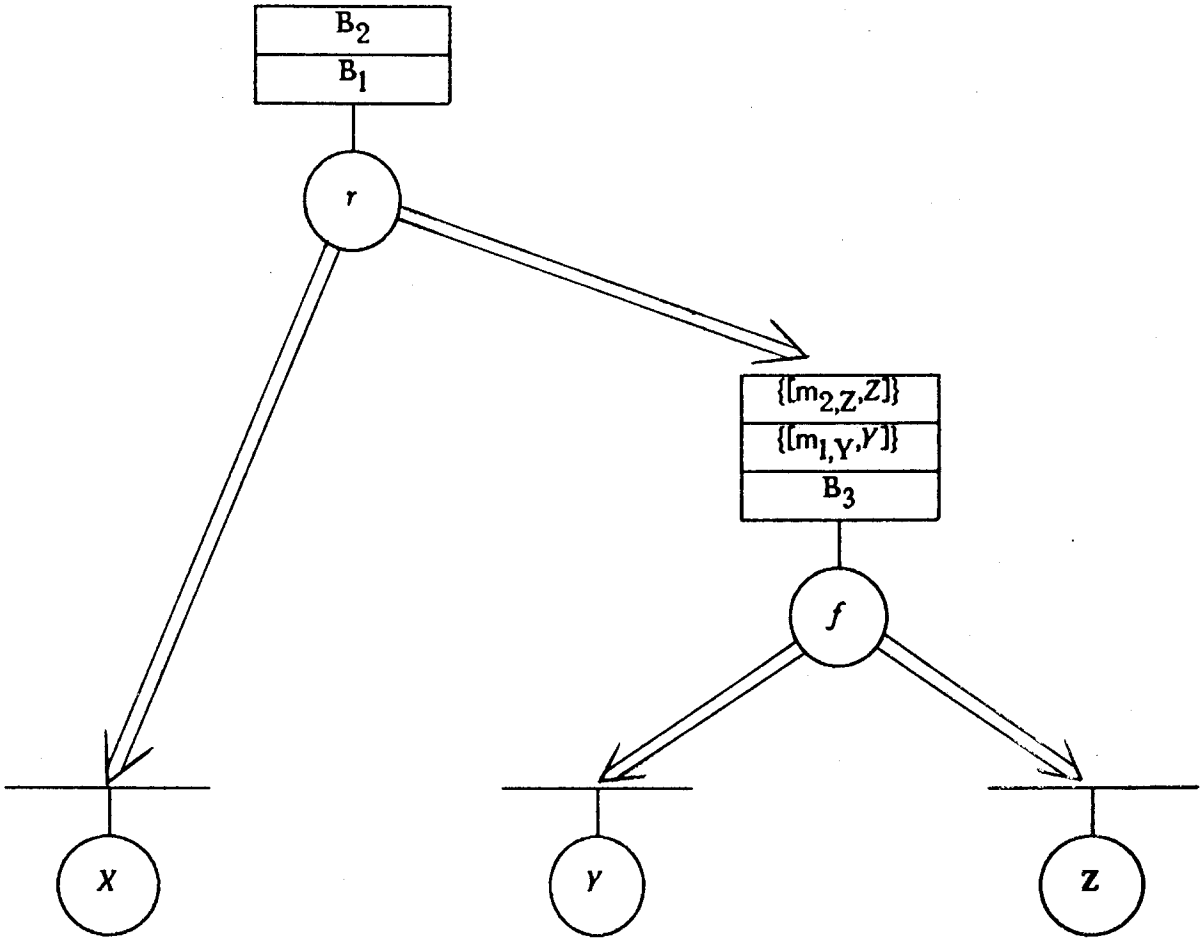
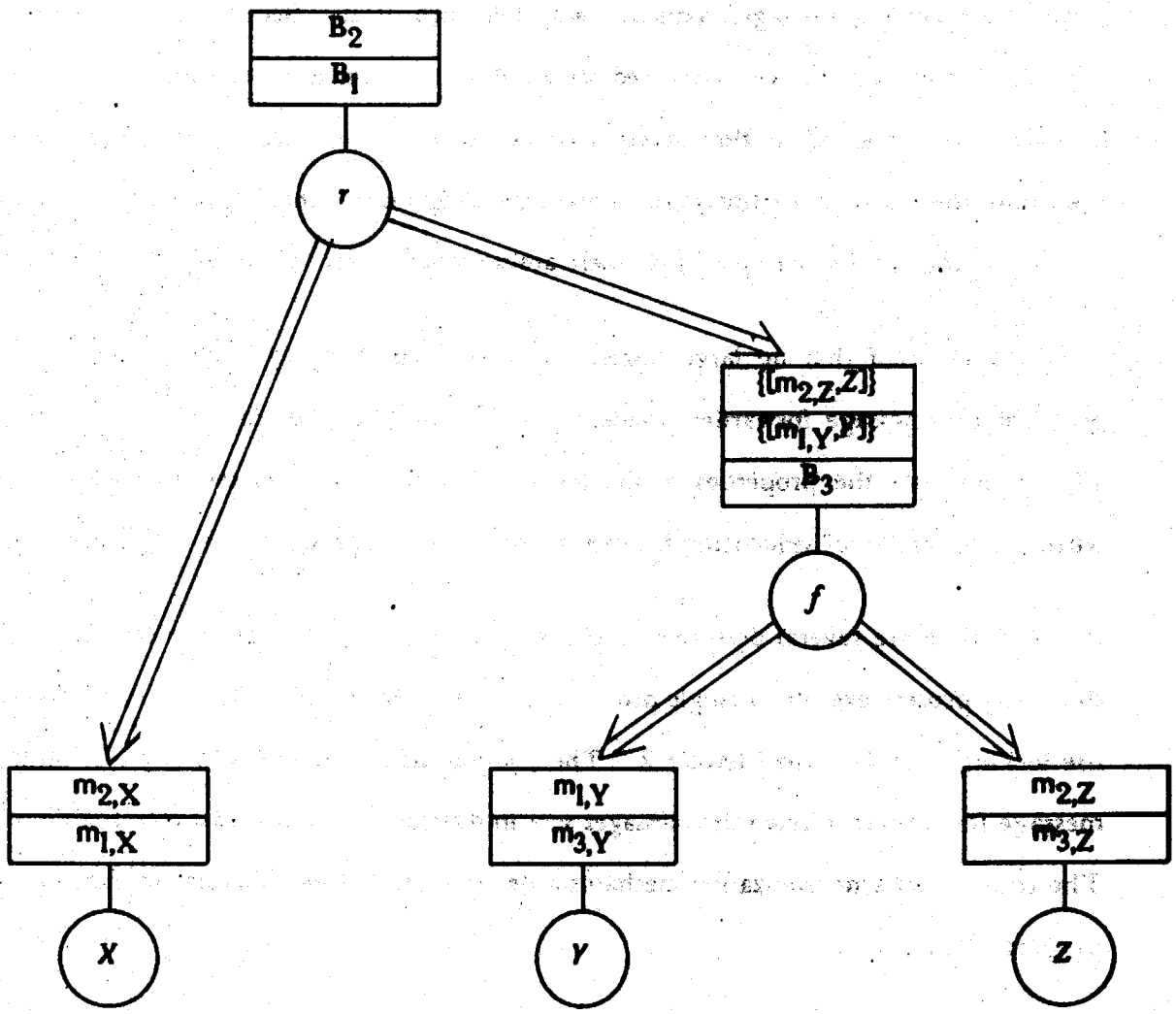


Figure 3.3c

The Execution State After Delivery of B_1, B_2, and B_3



A brief argument for the correctness of the solution is given here. A more detailed and more formal proof appears in an appendix to this thesis. Any two broadcast messages B_1 and B_2 are initially sequenced by one message message forwarder, the highest message forwarder receiving messages connected with both broadcasts. Because the message stream between any two processes is sequenced, the order of two broadcasts sequenced by a message forwarder is preserved as the messages connected with those broadcasts travel down the hierarchy toward their destinations. This sequencing insures that no pair of broadcasts in the $<$ ordering can form a cycle. (i.e. there are no messages for which $B_1 < B_2$ and $B_2 < B_1$).

The proof that no larger cycles can arise is substantially more complicated. The proof of the message forwarder protocol given in the appendix covers cycles of all sizes. This proof uses the properties of the hierarchy to show that no cycles can be achieved without a violation of sequencing between a process and its parent in the hierarchy.

There are several desirable properties of this protocol that are not obvious. One is that each process executes a single process step for each broadcast. This mechanism does not use locking as defined in Chapter 2. The solution insures that *all* processes, including the message forwarders, receive the messages sent in distributing a broadcast message atomically. The transaction synchronization mechanism described in the next chapter makes use of this property.

Another point to note is that the protocol works for structures of message forwarders other than hierarchies. I will use the term **synchronization network** for the logical organization of processes used in the protocol. A synchronization network is simply a directed graph that describes the parent - child relationships among processes. The proof of the message forwarder protocol given in the appendix depends on the synchronization

network only in that it requires that there be at most one path in the synchronization network between any two processes. This property is, of course, satisfied by a hierarchy.

A second requirement that must be imposed on the synchronization network used in the distribution of a broadcast B is that the destinations of the components of B must have a common ancestor in the synchronization network. If this were not the case, there would be no way to distribute B using the protocol, because there is no process to which B can be sent initially. If we are designing a synchronization network capable of coordinating any broadcast message involving a group of processes, then we must insure that all of those processes have a single common ancestor. This requirement, taken together with the requirement that there be at most one path between processes in the network, means that such a synchronization network must be a hierarchy. If, however, the set of broadcast messages (or at least their destinations) is known, and a synchronization network is being designed specifically to distribute those messages, then it is possible that a non-hierarchical network could be used. This is illustrated by the example in Chapter 6.

3.4 Other Ordering Restrictions on Broadcast Messages

The above protocol insures that each process receiving a component of a broadcast receives that component in the same order relative to those of other broadcasts. Thus a broadcast is atomic as viewed by the receivers. Recall, however, that there is another way in which the processes may perceive ordering among broadcasts, in that the sender of one broadcast may have been a recipient of other broadcasts.

In general, each process step that produces a broadcast may have received some knowledge about other broadcasts. This potential knowledge can be described by the **should follow** relationship among messages described below. Each message m sent by a process p in

a process step s **should follow** a message m' received by p whenever:

> a) There is a message m" received by p in process step s or in a step that preceded s, and m' and m" are components of the same broadcast.
>
> OR
>
> b) There is a message m" received by p in step s or in a step that preceded s, and m" should follow m'.

This relationship describes ordering constraints among messages that must be enforced in order to prevent the system from behaving anomalously if the correct interpretation of a message m sent by a process p to a process q depends on q having received messages containing information that was derived from broadcasts received by p before p sent m.

For example, if I could in one atomic broadcast send my paycheck to be deposited at the bank and checks drawn on my account to pay monthly bills, it would be disturbing to me if when one of those checks was sent to my bank to be cashed, it arrived before the deposit. This kind of behavior does not violate the definition of an atomic broadcast given above. In this example, there are two separate actions: my distribution of the deposit and payments, and my creditor's sending of the check to the bank to be cashed. Each of these could be sent in a separate atomic broadcast, however they cannot be part of the same atomic broadcast, as the debtor's action is not known until the check is received. Nothing in the definition of atomic broadcasting prevents these two broadcast messages from being sequenced in the apparently anomalous order, because the causal relationship between the two events that produced these broadcast messages is not recognized.

Unfortunately, the protocol described above allows such anomalous sequencing to occur. Consider the hierarchy shown in Figure 3.3 . A message m sent to both X and Y must be initially sent to the message forwarder r. It is possible for X to receive its component, and construct and send a message to Y, and have this new message received by Y, before the component of m sent to Y is received at Y.

A simple extension of the message forwarding system described above provides correct sequencing. Each broadcast B must initially be sent to a message forwarder f in the hierarchy that is an ancestor of the sender of B as well of as all of the processes associated with the destination ports of B.

Notice that if a component of some broadcast B has been received at any port, then any component of B that is destined for a process p and has not yet been received by p must be awaiting reception at the input port to some process that is an ancestor of p. The extended protocol prevents anomalous sequencing by insuring that a message B enters the hierarchy *above* all of the messages that B should follow. The sequencing of messages between the message forwarders then insures that any message that B should follow will be received at its ultimate destination before B. A more detailed proof appears in the appendix.

This solution to anomalous sequencing is very simple (though the proof that this solution works is somewhat complicated), and easily implemented. Therefore, I will only consider the implementation of the more complete solution.

3.5 Implementation

In this section, I will present two simple implementations of the synchronization protocols described above, one using point-to-point communication, and one taking advantage of communication technology that makes distribution of one message to several receivers relatively inexpensive. There are many optimizations that could be used to improve these implementations. I present them merely to show that such a system could easily be implemented, and that I have not ignored any difficult problems by making unreasonable assumptions about the implementation of message forwarders.

3.5.1 Atomic Broadcasting Using Point-to-Point Communication

In chapter two I presented a simple implementation of robust sequenced communication. This implementation can be extended to implement message forwarders. Robust, sequenced communication insures that messages sent from a message forwarder to some port arrive in the sequence in which they were produced, and are not lost. In addition to proper sequencing, we must show how the hierarchy of message forwarders can be maintained.

A serious problem of the protocol described above for message forwarders is that each process that sends a broadcast message must know the location, in the hierarchy, of each of the destinations of the components of that broadcast. This knowledge is necessary to select a message forwarder that is above all of the destinations. A second problem is that each process may send messages to a large number of ports. This is expensive using the

implementation of processes described in Chapter 2, because message queues must be maintained for each such port.

I solve these problems by changing the protocol for distributing the components of a broadcast slightly so that each process need only communicate with its parent and children in the hierarchy. This is accomplished by changing the process step specification of a message forwarder so that if all of the components of a message received by a message forwarder are bound for descendants of that message forwarder, the message is partitioned among the children of the message forwarder as before. If, however, the destination of some component is not a descendant of the message forwarder, the message is sent, intact, to the parent of the message forwarder.

To send a broadcast using this modified protocol, a process formulates a message containing a list of the component messages of the broadcast, and sends that message to its parent in the hierarchy. This message rises in the hierarchy until it is above all of the destination ports of the components of the broadcast (as well as its sender). When the message reaches a message forwarder that is above all of the destinations, it is distributed as before. Each process communicates directly only with its immediate neighbors in the hierarchy, thus the number of message queues needed to maintain robust sequenced communication is small.

We can now consider how the necessary information about the hierarchy could be maintained. Each message forwarder f must know which processes lie below each of its children in the hierarchy. This knowledge could be built into each message forwarder, or be built into the structure of process names. If the life of the hierarchy exceeds the usefulness of

individual processes, however, we must expect that processes will be created, deleted, or even moved in the hierarchy, and that these changes must be reflected to the message forwarders.

In showing how to add a process, I will assume a mechanism for generating unique process names, and will not attempt to solve the problem that a process being added to the hierarchy may already be part of it. I also assume that it is possible to determine from a port name which process receives messages at that port. This knowledge allows the message forwarders to determine the destination *process* of a message from its destination *port*.

To add a process p to the hierarchy, some message forwarder f is selected to be the parent of p. Process p informs f of this choice by sending a "request for adoption" message. This message establishes the message queues and sequence numbers for sending messages from p to f. Message forwarder f can reply to p either by accepting or rejecting this request.

If the request is accepted, the mechanism for sending messages from f to p is established with the sending of the reply, and p can begin to send and receive atomic broadcast messages. Message forwarder f sends a message to its parent which is propagated up the hierarchy informing all processs that are now ancestors of p of the presence of p. Before any message can be sent to p, the sender must be informed of the existence of p. Any message that could inform a process of the presence of p must either have been sent by p or should follow (as defined in the previous section) some message sent by p. The messages that inform the message forwarders of the presence of p will always precede any message sent by p (and therefore any message that should follow a message sent by p) at the ancestors of p, because of the sequencing. Thus any message forwarder encountering a message with a component sent to p is guaranteed to know whether or not p is one of its descendants.

Special care must be taken when the request for adoption is rejected. Communication failures can cause either the request for adoption or the reply to that request to be lost. We must be sure that loss of messages cannot cause p and f to become confused such that one thinks that the request was successful while the other does not. Such confusion is particularly likely if the request is re-transmitted by p if f does not respond promptly. This problem is similar to that of initiating a connection in a communication protocol, such as TCP [Cerf74] or DSP [Reed76]. The solution that I am using is similar to that of DSP.

When a message forwarder rejects a request for adoption, it may be doing so because it has insufficient resources to establish communication with a new child. If this is the case, we do not wish to burden the message forwarder with the task of remembering that it has rejected a request. Therefore, we must keep in mind that if a message forwarder f is sent a request for adoption several times (because the sender of the request re-transmitted the request when f did not reply promptly with an acceptance), then f may first reject the request and subsequently accept it. (Once a request has been accepted, however, the message forwarder can know to accept any subsequent re-transmissions of that request). This means that a process that has sent a request may not negotiate with another potential parent if it receives a rejection or no prompt reply. If the original request (or a re-transmission of the original request) were later accepted, this could allow one process to have two parents in the hierarchy. Thus we require that if a process receives a rejection (or no reply at all), it must either keep trying (re-transmitting its request) until it is accepted, or choose a new unique name and attempt to establish communication with another parent.

This approach may result in a message forwarder adopting a process that no longer exists (because that process has chosen a different name), but this does not cause a problem. No messages will ever be sent to or from such an abandoned process. An abandoned process

will be detected and deleted from the hierarchy through the same mechanism that deletes processes that is described in the following paragraph.

A process can remove itself from the hierarchy by sending a message to its parent notifying its parent of its intention to leave the hierarchy. When a message forwarder receives such a request, or a message forwarder can reliably determine that one of its child processes no longer exists then that message forwarder can reclaim the message queues for that process and inform the ancestors of the message forwarder of the disappearance of the process. Once a process has left the hierarchy, it must choose a new name in order to re-join unless it can be determined that no process remembers the old name.

A process p can be moved from one location in the hierarchy to another location in the hierarchy in a series of small steps of the form shown in Figure 3.4. Each such step changes the parent of p from f to g, where f is the parent of g, or g is the parent of f. Both cases are essentially similar, so I will only describe the former.

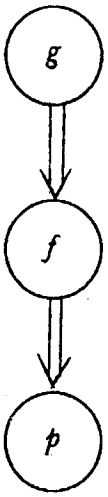
> 1) p sends a message to f (the current parent of p) requesting movement, and stops sending messages to f.
>
> 2) f receives this request, and performs an atomic modification of its state with the following changes: A request to close is put at the end of the output message queue for p, p's request is put in the output message queue for g, and f's view of the hierarchy is changed to reflect p's movement.
>
> 3) p receives the request for close from f, drops its now empty queue of messages for f. p now sends a request for adoption to g.
>
> 4) g receives the request from f, establishes a queue for p, and accepts p's request for adoption. g now updates its view of the hierarchy to include p

The last two steps take place in either order, depending on the relative timing of the

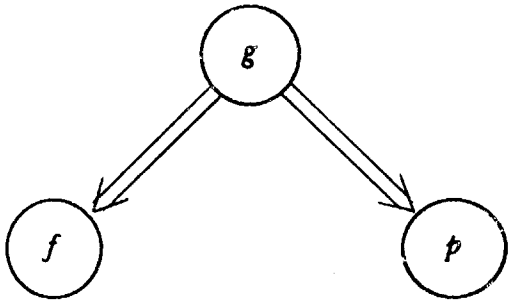
Figure 3.4

Moving a Process

From:

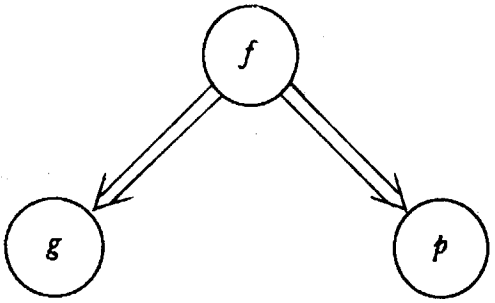


To:

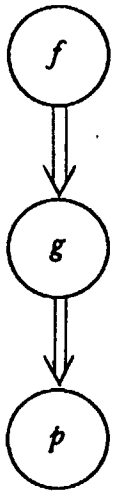


Or

From:



To:



messages sent from f to p and g. No knowledge of the move must be propagated beyond f and g. Note also that the transfer should be negotiated in advance to prevent g from refusing the request for adoption.

3.5.2 Atomic Broadcasting with a Broadcast Medium

In many communication architectures, it is no more costly to send a message to a set of receivers than to a single destination. A broadcast network such as a ring network [Farber72] or an Ethernet [Metcalfe76] has this property, as does communication through shared memory on a single site. Our scheme for atomic broadcasting can be modified to take advantage of this ability to distribute component messages of a broadcast to several receivers.

In the absence of errors, a broadcast network acts like a message forwarder. Each site presents its messages to the network. The network receives one message at a time, and distributes that message to the intended receivers. Messages sent through the network are totally ordered, just as messages sent through a forwarder.

To send an atomic broadcast message to a set of receivers on the same network, all of the component messages of that broadcast are packaged into a single message for the network. If the packet size of the network is too small to hold all of this, the contents of each component message can be pre-distributed to its intended receiver. The sender picks a unique identifier for the broadcast and attaches it to each component message, sending the component messages singly or in groups to the intended receivers. When such a component message is received, it is saved by the receiver and not placed in the stream of incoming messages. When all components have been distributed, the sender sends a message containing the unique identifier to all receivers using the broadcast capability of the

communication network. The unique identifier is used by the receivers to identify the component message that was pre-distributed and insert that message into its stream of incoming messages. The broadcast networks designed thus far all have a packet size large enough to accomodate such a unique identifier.

If very large messages are sent, it would seem that we are not obtaining any benefit from the availability of the broadcast mechanism over the point-to-point scheme described above. Note, however, that if we were to use the point-to-point scheme for coordinating atomic broadcasts among the processes executing on sites connected by a broadcast network, then in general each component of a broadcast message would have to be transmitted over the network twice, once to reach the common ancestor of the recipients, and once to reach its destination. The protocols of this section transmit each component of a broadcast message exactly once,[1] thus saving extra message transmissions.

If the network and sites were completely reliable, including all components of an atomic broadcast in a single message would be sufficient to distribute the component messages atomically. Unfortunately site failures or simple lack of buffering can cause a site to miss a message from the network. To solve this problem, there must be a mechanism that uniquely orders the broadcast messages, even if failures occur during the transmission of a broadcast. Such a mechanism would allow each site to know the order in which incoming broadcast messages should be processed by that site, even if failures cause some of the transmissions to the site to be lost or to arrive out of sequence. A mechanism must also be provided to allow a site that has missed a message to obtain a copy of that message.

---

1. This excludes re-transmissions necessitated by errors.

This can be accomplished by appointing one site as the coordinator of the broadcasting. The coordinator has the responsibility for arbitrating the broadcast messages on the network, and does so by assigning a sequence number to each. To send an atomic broadcast, a site assigns a unique identifier to that broadcast and transmits the components of that broadcast in one or more transmissions on the broadcast network. Each transmission is identified with the unique identifier, so that the receivers can identify the transmissions that are used to distribute an atomic broadcast.

These transmissions are seen by every node on the broadcast network, including both the recipients and the coordinator site. Each recipient receives and stores its component of the broadcast from the transmissions used to distribute that component. This stored component is not yet included in the input message queue of the receiving process at that site. The coordinator receives and stores all of the components. When the coordinator has received all of the components of an atomic broadcast, the coordinator assigns a sequence number to that broadcast and transmits a message to all sites containing the sequence number, the name of the sending site, and the sending site's unique identifier for the broadcast. This message informs all receivers of components of that broadcast of the proper sequence in which that broadcast is to be included with components from other broadcasts. The message from the coordinator also serves as an acknowledgement to the sender of the broadcast that the broadcast has been distributed and the sender can delete it from its output message queues.

It is relatively simple to see that this scheme works if no errors occur, as it is essentially the same as the scheme for distributing large broadcasts in the absence of errors described above, with the exception that the coordinator distributes the single message that demands all receivers to include the broadcast in their input message queues, rather than

having the sender of the broadcast do this. To see how this scheme also works in the event that messages are lost on the broadcast network, let us consider the possible errors.

One error that can occur is that one of the transmissions used to distribute the components of the broadcast is not received by one or more sites. If it is the coordinator that misses one of these transmissions, then the coordinator will never detect the broadcast as being complete, and will not send the sequence number message. After a suitable timeout interval, the sender of the broadcast can detect that something is amiss (because it does not receive the message from the coordinator) and can retransmit the components. Any site that received the components correctly the first time can identify and discard the retransmission because of the unique identifier assigned by the sending site.

If one of the receivers fails to receive a component correctly, but no other errors occur, then eventually the coordinator will transmit the sequence number for the broadcast. The receiver will discover that it has not stored the component for the broadcast identified in the message sent by the coordinator, and can request re-transmission of that component by the coordinator. Thus the coordinator also acts as a backup for obtaining copies of lost messages.

Another error that can occur is that the the message sent by the coordinator may be missed by one or more sites. If the sender of the broadcast does not see this message, it will begin a needless retransmission, which again can be discovered and discarded by the receivers. The coordinator can retransmit its message to acknowledge the distribution of the broadcast to the sending site.

If one of the receivers misses the coordinator's message, this may not be immediately detected. The receiver will detect that it is out of date when it next receives a message from the coordinator. That receiver can then request retransmission of the messages that it has missed from the coordinator.

The protocol described above for atomic broadcasting using a broadcast communication network is relatively simple, makes efficient use of the network if no errors occur, and works correctly if messages are lost or duplicated by the network. There are several points about this protocol that must be clarified before it can be used as the basis for a practical implementation of atomic broadcasting.

The coordinator site must record all of the broadcast messages, and must keep each broadcast until it knows that that broadcast has been received by all receivers. In order to avoid having to save broadcasts forever, we can have each site periodically send a message containing the sequence number of the most recent broadcast that that site has received correctly. The coordinator can use these messages to determine when it is safe to delete a saved broadcast message, and when a site is out of date and should be sent information about one of the saved broadcasts. The message sent by the coordinator must identify which of the sites are receivers of the broadcast. This information can be determined from examining the components of the broadcast, and can be encoded in the message by using a bit vector with one bit for each site indicating whether or not that site is a receiver of the broadcast. The bit vector is used by a receiving site in order to determine whether or not that site should have received a component of the broadcast. This in turn tells the site whether or not it missed the transmission of the component by the sender.

Each site must keep track of the most recent sequence number sent by the coordinator that has been seen and correctly processed by the site. In a typical application of this protocol, it might be the case that each site is a receiver in relatively few of the atomic broadcasts. If this is the case, it may be necessary to filter the messages sent by the sender and by the coordinator in the receiver's network interface in order to avoid interrupting the receiver unnecessarily. This could be done by maintaining a register in a site's network interface, which contains the sequence number of the most recent broadcast that that site has correctly processed. When a message from the coordinator is seen by the network interface, it examines the message to determine whether or not the sequence number in that message is one greater than that in the register in the network interface. If this is so, and if the message does *not* describe a broadcast in which the site is a receiver, then the register is incremented, and the receiving site is not interrupted. If a message from the coordinator does not meet these conditions, then it is reported to the receiving site, which either detects that the sequence number indicates that the receiving site is out of date, or that the message pertains to the receiving site. If the message pertains to the receiving site, then the receiving site incorporates the broadcast described by the message into its input message queue (in stable storage), and then updates the sequence number in its network interface.[1] Otherwise, the receiver requests retransmission of the missed message(s) from the coordinator.

---

1. Notice that if a second message comes in before a message received by a site has been incorporated, the sequence number in the network interface of the site may be out of date. This causes no problem, as the site detects that it has missed the second message and immediately obtains it. The sequence number cannot be updated before the message has been recorded in the input message queue, as a failure of the site may cause the message that has been received but not yet recorded in the queue to be lost.

3.5.3 Use of Broadcast Networks and Point-to-Point Communication Together

The schemes for providing synchronization of atomic broadcasts using a broadcast network can be used in conjunction with the message protocols for point to point communication in a network with a number of different physical communication media. To do so most efficiently, all of the processes running at sites linked by a broadcast network should be made children of a single message forwarder representing the network. Other broadcast networks and sites are linked through message forwarders representing gateways connecting networks.

To see how this is done, consider the physical communication topology shown in Figure 3.5. The physical configuration is three broadcast subnetworks, with sites *F* and *G* acting as gateways between *Net1* and *Net2*, and between *Net2* and *Net3* respectively. One possible efficient hierarchy for this network is shown in Figure 3.6. This figure is a skeleton hierarchy showing one message forwarder for each site. The processes at a site would be descendants of the single message forwarder shown for that site. Consider a broadcast message sent by a process at site *g* to processes at sites *D*, *E*, *H*, and *I*. Site *G* would use the broadcast network *Net2* to distribute components to sites *D*, *E*, and *F* This message forwarders at *D* and *E* would route their components to the proper destination processes. The message forwarder at *F* would use *Net3* to distribute the messages for *H* and *I*.

3.6 Evaluation

The algorithm described here for coordination of an atomic broadcast is only one of many that could have been used for this purpose. The desirability of this algorithm as opposed to the others depends mainly on the extent to which the hierarchy of message

Figure 3.5

A Physical Communication Topology

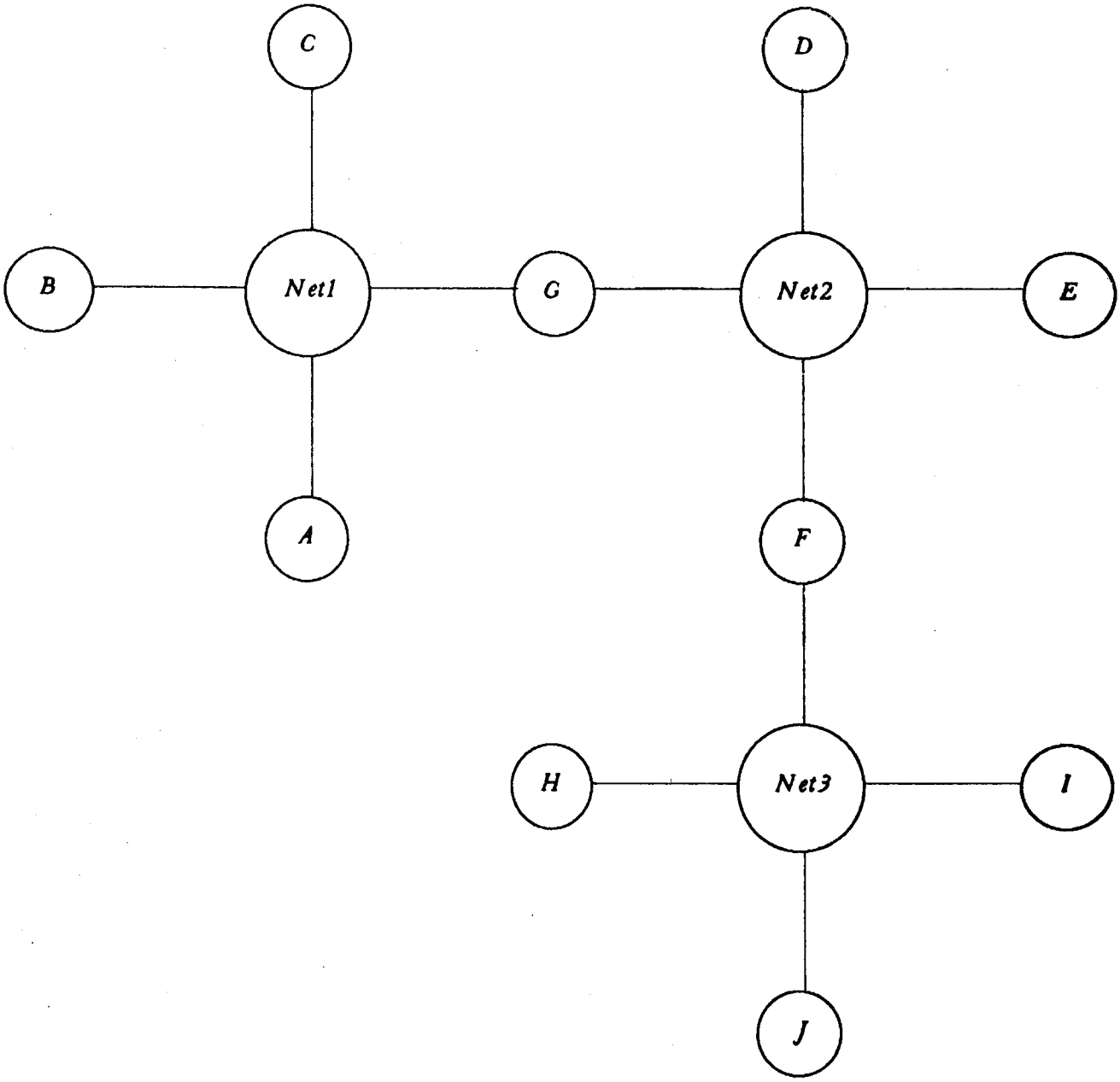
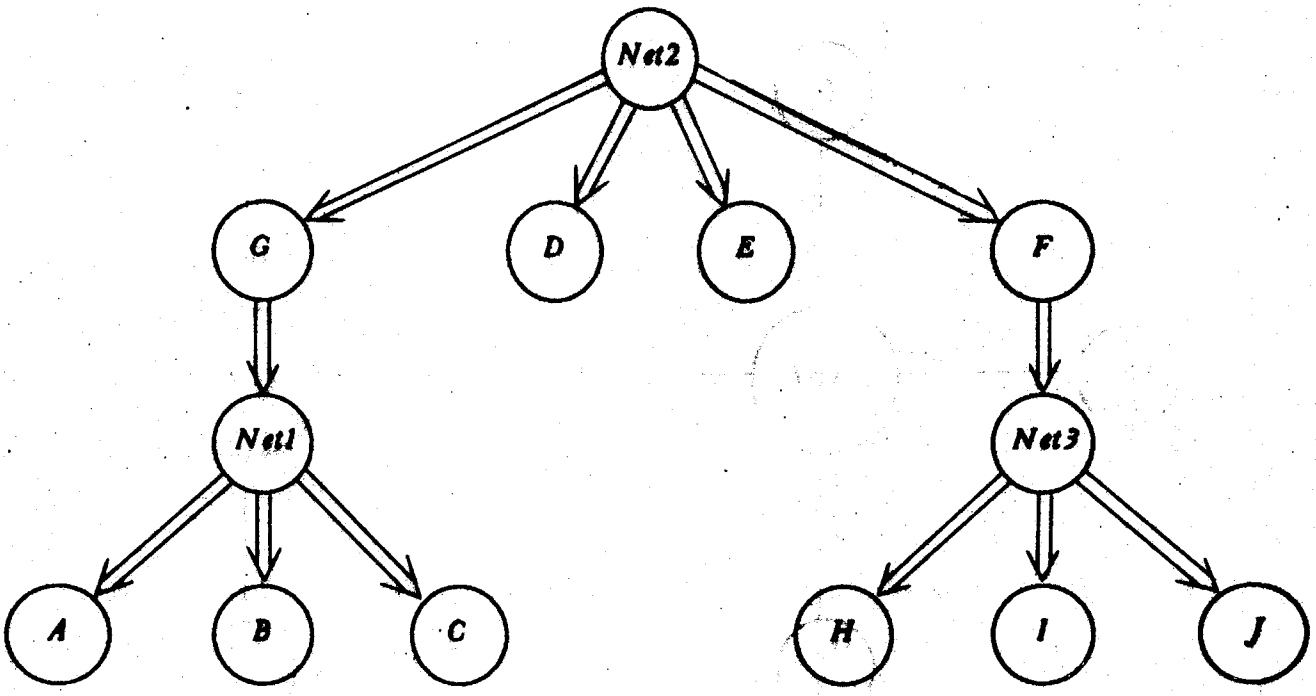


Figure 3.6

A Logical Topology for the Network of Figure 3.5



forwarders reflects the logical and physical communication paths in the distributed information system.

I have already argued that many applications for a distributed information system exhibit a strongly hierarchical organization. This is a reflection of hierarchical management policies. If the hierarchy of message forwarders is chosen so that processes that need to communicate frequently are nearly always children of the same message forwarder, the message forwarder scheme involves little extra message passing beyond direct communication between processes.

This is particularly true if the physical communication network is also hierarchical. If the physical communication network is hierarchical, (counting broadcast networks as a single node in that hierarchy), then the atomic broadcasting mechanism described here is as reliable as any other communication mechanism. Each message follows the shortest path in the hierarchy between its source and destination. Two transmissions take place for each link in the hierarchy that a message traverses (one carrying the message and one carrying an acknowledgement). This is the minimum number of messages needed to deliver a message reliably, and the synchronization adds no extra messages.

If, however, the physical communication network is strongly non-hierarchical, with many alternate paths between any two sites, imposing a logical hierarchy may cause communication between some sites to be very inefficient, where a direct link between those sites exists. This problem can be alleviated to some extent by sending the contents of all large messages over the shortest possible route, and sending a message header through the hierarchy to designate when the pre-distributed message contents are to be included in the incoming message stream of the receiver, as was done for broadcast networks with small packet sizes. This technique reduces the communication overhead due to the hierarchy, but does not reduce the vulnerability of the hierarchy to failures. If much communication is local, however, this vulnerability may not be a problem.

The message forwarder scheme has several advantages over other synchronization mechanisms for distributed systems that could be used to coordinate atomic broadcasting. One advantage is its simplicity. The message forwarder protocols can all be described by simple statements. Each step is deterministic, and the only source of non-determinism is the order in which two messages sent to the same process are received.

The inability of two processes to determine reliably whether or not some message sent between them was received does not cause a problem in the message forwarder scheme. Using the protocols described above, once a message has been sent, the sender assumes that that message could have been received, and does not take any action inconsistent with that assumption. Once a message has been sent, it is re-transmitted indefinitely until it gets through.

Another interesting feature of this solution is that the sender of a broadcast need not participate in the completion of a broadcast. Once the broadcast message has been delivered to a message forwarder, it will eventually be delivered to all receivers, even if the sender crashes. The sender of a broadcast cannot, however, know when that broadcast will be delivered, as that depends on the availability of the message forwarders and receiving ports, and on the order in which messages are received by those ports. The broadcasts from one sender are, however, delivered in the same order in which they were sent.

A third distinctive feature is that the order in which a broadcast is received relative to other conflicting broadcasts is not determined in one decision. The decision is distributed among the message forwarders through which the messages of one broadcast pass, each of which performs some arbitration. In a scheme using timestamps to arbitrate between concurrent messages, once a timestamp has been assigned to a message its order relative to other messages has been fixed. Postponing this decision by distributing it among the message forwarders provides greater flexibility that can be important in some circumstances.

Even after some of the component messages of a broadcast have been received by their destination ports, other messages from the same broadcast may still be held by the message forwarders, and thus other non-conflicting broadcasts sent later may be received earlier. This flexibility is important if the communication network connecting ports

partitions, in that broadcasts local to one or the other of the partitions can continue to take place, even if there are messages from more "global" broadcasts that have not yet been delivered. The extended protocol and the implementation discussed above guarantee that this flexibility does not allow messages to arrive out of order, in that any port p receiving a message B will have received any message that the sender of B could have been aware of before receiving B.

The message forwarder scheme takes advantage of "locality of reference" in communication more effectively than many other synchronization schemes that could be applied to atomic broadcasting. Some schemes, such as those using timestamps, require periodic communication among all of the sites. Such a scheme would be expensive for a distributed information system in which most operations involve only one or a few sites. Sending an atomic broadcast using message forwarders, in contrast, requires only the participation of the sender and receivers, and possibly a few additional sites holding message forwarders.

One point that remains to be explored is to determine exactly what kinds of operations can be performed using an atomic broadcast. This question will be answered in the next chapter.

3.7 Summary

This chapter has discussed one simple synchronization problem in a distributed information system: that of sending a set of messages to a set of destinations "atomically". A mechanism was developed to provide the proper synchronization by using message forwarders to distribute atomic broadcasts to their receivers. The mechanism was extended

to prevent anomalous behaviour if correct interpretation of one message depends on prior reception of some message.

Two implementation strategies for message forwarders were presented. One implementation that was independent of the physical communication network, using robust sequenced processes was developed. The protocols allow processes to be added, deleted, or moved within the hierarchy of message forwarders. A more efficient implementation that takes advantage of a broadcast communication network was also outlined.

Chapter 4

Atomic Transactions in the Process Model

In this chapter, the problem of performing transactions atomically in a distributed information system described by the process model of Chapter 2 is considered in greater detail. A method is presented for describing the data flow that a transaction causes among the items that it accesses. The difficulty of coordinating transactions to be performed atomically is shown to be dependent on the interaction of their data flow descriptions.

A synchronization scheme consistent with the goals set forth in the first chapter is developed. This scheme makes use of the hierarchical mechanism for atomic broadcasting described in Chapter 3. The mechanism is simple, efficient, and frequently avoids locking.

4.1 Analysis of Transactions

The techniques needed for synchronizing a set of concurrent transactions are dependent on the data flow among data items caused by performing the transactions. The set of input items to each transaction and the way in which those inputs are reflected in the updates made by that transaction affect the way in which transactions interact. I will use an abstraction which I refer to as a transaction graph to describe the data flow between items caused by performing a particular transaction.

A transaction graph is a directed graph in which the nodes are the data items in the data base. These arcs show how the output items of a transaction are derived from the input items to that transaction. The transaction graph for a particular transaction T

contains directed arcs pointing at each item that is updated by T. For each such item i, there is an arc running from each item j such that the new value given to i by T depends on the value of j seen by T.

Figure 4.1 shows the transaction graph for a simple banking transaction. This transaction modifies the values of three items, x, y, z. The transaction could represent a bank's action on cashing a $50 check for a customer, where x represents the amount of cash disbursed by the bank, y represents the account balance, and z represents the customer's "overdraft protection" loan account.[1]

Figure 4.1

A Simple Transaction Graph

The Transaction T:

```
Set x = x-50;
If y < 50 then do;
    Set z = z + y - 50;
    Set y = 0;
    end;
else Set y = y-50;
```

---

1. In this simple example, we assume that the overdraft protection is unlimited and ignore any other bookkeeping that must be done by a "real" banking system.

The transaction graph depicts the way in which the outputs of the transaction are computed. The arc from y to z in the transaction graph of T reflects the fact that the value for y must be obtained *before* the value produced by T for z can be determined. Such arcs describe constraints on any implementation of a transaction in that the access to an item that is the source of some arc must be performed *before* the access to the item that is the destination of that arc.[1]

In the process model of a distributed information system, a transaction is carried out as a set of process steps. A transaction graph can be used to construct a similar abstraction, which I refer to as an activity graph, describing the data flow among the process steps that implement a transaction. Two points cause an activity graph for a transaction to differ from its transaction graph:

> 1) Several of the items accessed by a transaction may be held by a single data manager, allowing all of the accesses to those items to be performed in a single process step.
>
> 2) Some data items may be replicated, with copies held by several sites. This means that one access in the transaction graph may be performed by several process steps in the activity graph.

The nodes of an activity graph are the processes that participate in performing the transaction. For each arc from an item x to an item y in the transaction graph for T, the activity graph contains one arc pointing to each manager that holds a copy of y from some manager holding a copy of x. Arcs connecting a process to itself are not shown. If an item x which is the source of some arc in the activity graph of T is replicated, then we have a choice of which copy of x to use in computing the output of T dependent on x. This choice

---

1. Note that if a transaction graph contains a cycle, this means that some item in the cycle must be accessed at least twice in any implementation.

is reflected by the arc in the activity graph of T connecting *some* process holding a copy of x to the process that holds an item whose new value depends on x. If transactions are run atomically, then all copies of a replicated item seen by a transaction have the same value, and thus the choice of which one to use will not effect the output values produced by the transaction.

Figure 4.2 shows the activity graph for an implementation of the transaction depicted in Figure 4.1, in which each of the items is replicated at two of the three data managers. The graph indicates several decisions made about the implementation of T. M_1 holds copies of items x and y. The new values produced for these items depend only on their previous values, so a decision has been made so that M_1 is to compute the new values for its copies of these items from their previous values at M_1. Similarly, M_2 is to use the old values of the copies of y and z that it holds to compute their new values. M_3, however, holds a copy of z, but no copy of y from which to compute the new value of z. A decision has been made that M_3 is to obtain this information from the copy of y held by M_2.

Notice that in this example, all three managers participate in the computation of the outputs of the transaction. This results in some duplication of effort, as, for example, both M_1 and M_3 compute new values for x. We could have centralized the computation of the outputs of the transaction in one of the three managers and distributed the results to the other managers, which would have lead to a radically different activity graph.

The model of a transaction used in this thesis, in which various parts of a transaction are performed in parallel, is different from the model used by many other researchers in which the accesses required to carry out a transaction take place in some well defined sequence. Allowing for parallel execution of various parts of a transaction not only allows the transaction to be completed faster, but also simplifies the task of synchronization

because the synchronization mechanism can choose the order in which two parts of a transaction that are logically independent (such as those performed by M_1 and M_2 in this example) are performed.

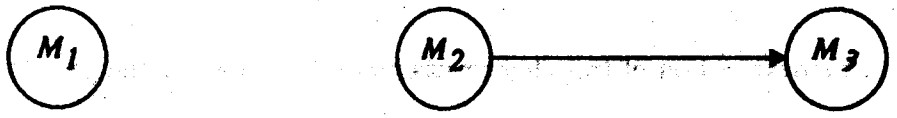
The arcs in an activity graph represent constraints on the order in which the process steps used to perform a transaction can occur. Some step of a process that is the source of one of these arcs must be completed before some step of the process that is the destination of that arc. Recall that performing a transaction atomically with respect to other transactions also constrains the order in which process steps occur. The difficulty of coordinating a group of transactions to be performed atomically depends on the interaction among their activity graphs.

For a group of transactions, we can construct a joint activity graph, which is a merger of the activity graphs of the individual transactions. The joint activity graph contains an arc between two processes whenever the activity graph of some transaction in the

---

Figure 4.2

An Activity Graph For an Implementation of T



Assignment of Items to Managers

M_1: x,y $\qquad$ M_2: y,z $\qquad$ M_3: x,y

group contains such an arc. Each arc is labeled with the names of the transactions that contribute that arc. Figure 4.3 shows an example of such a graph for three simple transactions.

Each of the three transactions is responsible for one of the three arcs in their joint activity graph. This is because each transaction transfers information for an item held by one manager to an item held by some other manager.

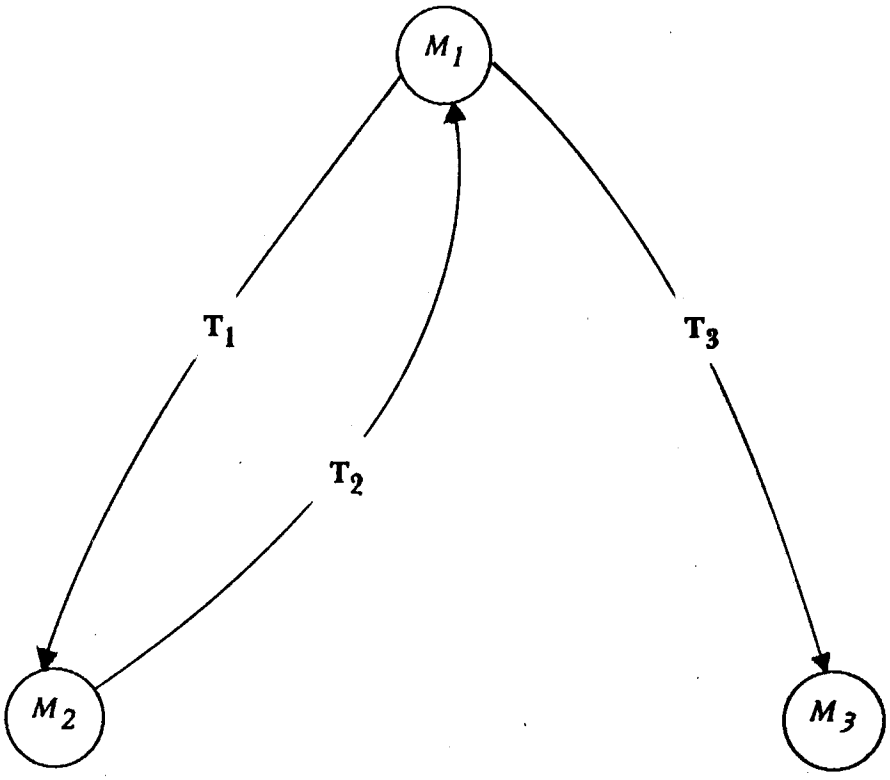
Activity graphs and joint activity graphs can be viewed as finer grained versions of the L-U graphs used to describe transactions in SDD-1[Bernstein77]. The analysis of transactions in SDD-1 does not examine the derivation of outputs from inputs, but instead assumes that each output of a transaction may depend on any of the inputs. In fact, each output may depend on only a small subset of the values read, a fact that is represented in activity graphs.

Activity graphs provide a simple way of describing the way in which input values seen by a transaction affect the output values produced. The arcs in an activity graph also describe ordering relationships among the process steps that carry out a transaction in that the process step at the source of some arc must be completed before the process step that is the destination of that arc.

The next section of this chapter examines the impact of the patterns of accesses of a group of transactions, as described by their joint activity graph, on the synchronization techniques that must be used to coordinate those transactions.

Figure 4.3

A Joint Activity Graph



Transactions:

T_1: Set B = B + A

T_2: Set A = A + B

T_3: Set C = C + A

Assignment of Items to Managers:

M_1: A
M_2: B
M_3: C

4.2 A Simple Approach to Transaction Synchronization

In the previous chapter, I presented a simple mechanism to distribute a set of messages to a set of receivers as an atomic broadcast. This mechanism could be used to distribute a set of input messages to the process steps of a transaction. In this section, I explore the applicability of the atomic broadcast mechanism to the problem of performing transactions atomically. I show that that mechanism can be used only when the joint activity graph of the group of transactions to be performed does not contain a cycle.

The simple synchronization scheme developed for atomic broadcasting cannot be used directly to coordinate a transaction that has an activity graph containing an arc connecting two processes. This is because there is no way to describe such a transaction in a set of independent messages to be delivered to the data managers as an atomic broadcast. The process step at the source of an arc must be completed before a message describing the access to be performed by the process step at the destination of that arc can be formulated.

One might expect that the atomic broadcast protocol could be modified somehow in order in to synchronize a group of transactions using sequencing of messages between processes to control the order of process steps. If the joint activity graph of the transactions does not contain a cycle of arcs, then this can be done, as will be shown in the following section. If, however, the joint activity graph of the set of transactions to be performed contains a cycle, any protocol for coordinating the transactions must use some form of locking, as will be shown subsequently.

4.2.1 Synchronization of Transaction Groups Without Cycles

First I will show how to coordinate a group of transactions whose joint activity graph contains no cycles. The approach I will use is to modify the message forwarder scheme of the last chapter to allow a process to act both as a message forwarder and a data manager. Such a process receives a message and produces a group of messages for its children in the hierarchy in each process step. The messages produced need not be a simple partitioning of the message received, but can depend on the local state of the receiving process.

One can perform the transaction depicted in Figure 4.2, for example, by making process M_2 the parent of both M_1 and M_3 in the hierarchy. The transaction could then be performed by sending a message describing the transaction to M_2 using the atomic broadcast protocol described in Chapter 3. This message propagates through the hierarchy until it reaches M_2. When this message is received M_2, that data manager performs the specified updates to its copies of y and z. In the same process step, M_2 forwards the portion of the request relevant to M_1, and sends a message to M_3 describing the accesses to be performed on x and z for M_3. M_2 includes the current value of y in the message sent to M_3.

Including some of the data managers as message forwarders in the hierarchy allows some of the process steps of a transaction to be performed before the input messages sent to other steps are constructed, while retaining the hierarchical structure of message sending. Recall that the message forwarder protocol of Chapter 3 insures that *all* of the processes, message forwarders and data managers alike, see a broadcast as atomic. The request to perform a transaction in this scheme is treated like an atomic broadcast, and thus is seen as atomic by the data managers.

A group of transactions can be performed atomically with the modified message forwarder scheme whenever a hierarchy of message forwarders and data managers can be constructed so that the arcs in the joint activity graph always run from a process to one of its descendants. This can be done whenever the joint activity graph contains no cycles. In a later section, I show how assignment of data items to data managers can be chosen so as to eliminate cycles from the joint activity graph of any *expected* group of transactions.

4.2.2 Synchronization of Transactions with Cycles in the Joint Activity Graph

If there is a cycle in the joint activity graph of a group of transactions, then there is no way to construct a hierarchy so that a process that is the source of some arc is always an ancestor of the destination process of that arc. Thus the message forwarder protocol cannot be used. The following paragraphs give an argument to support the claim that any protocol that correctly coordinates a group of transaction whose joint activity graph contains a cycle must use locking.

Consider first a group of two transactions that form a cycle, such as T_1 and T_2 in Figure 4.3. The execution of a transaction consists of a set of process steps. The arcs in the joint activity graph indicate that T_1 must be performed by a set of process steps in which a process step of M_1 precedes a process step of M_2. Similarly, in performing T_2, a process step of M_2 must precede a process step of M_1.

To perform the two transactions atomically, either both steps performing T_1 must precede both steps for T_2, or vice versa. To perform the transactions without locking, recall from chapter two that at most one process step of each data manager can be used for each transaction, and that the sequenceing of messages between processes is the only restriction that can prevent a message that has been sent from being received promptly. We must

therefore prevent, somehow, the situation that the process step of T_1 at M_1 and the process step of T_2 and M_2 are both completed before either transaction is completed. This can be shown to be impossible by demonstrating that this undesirable situation can be forced to occur in an execution of any protocol for the synchronization of T_1 and T_2 that does not use some form of locking.

Consider the state of the system during the execution of T_1 in which M_1 is performing its process step of T_1. If T_2 were begun at this point, the synchronization protocol must prevent the execution of the process step of M_2 related to T_2 from preceding that which accomplishes the completion of T_1. Without using locking to control the order in which messages are received, the only way to control the order in which M_2 receives the messages pertaining to the two transactions so that the undesirable order is avoided is to have both messages sent by M_1, and use sequencing of messages between M_1 and M_2 to force the messages to be received in the correct order.

Thus to force the execution of the process step of M_2 that completes T_1 to precede that that begins T_2, both of these process steps must be triggered by messages sent from M_1. This means that the execution of T_2 must include two process steps of M_1, one that precedes the step of M_2 and one that follows that step. Using two steps o of one process to perform one transaction is a form of locking, therefore it is impossible to coordinate the cycle of two transactions without locking.

This argument can be extended to cycles of any size by demonstrating that unless locking of some form is used, then it must be possible to reach a state in the execution of the system in which each transaction has completed a process step in one of the processes in the cycle, making it impossible to complete the execution of the transactions atomically. We are left with the conclusion that some other mechanism must be needed in order to synchronize a

cycle of transactions. The locking mechanism used in this thesis is explicit locking. This locking mechanism consists of delaying the reception of a message until some other message *from some other process* is received.[1] Locking is to be avoided wherever possible, because a failure of the sender of the expected message, or of the communication network, may delay processing of messages from other sources. This violates our goal of partial operability, as now a group of functioning sites cannot necessarily carry out a transaction purely local to those sites, because one of the processes involved in the transaction may be locked, waiting for a message from some other site. This problem will be considered in greater detail in Chapter 5.

The particular mechanism that I will use for locking in the process model is to place a pre-requisite on the process step specification of a process step. A pre-requisite is a predicate that may include variables in the local state of the process. A process step is not performed unless the pre-requisite for that step is satisfied. By placing a pre-requisite on all process steps that receives messages from one of the input ports of a process, one can inhibit the reception of messages at that port until some condition is met.

With this locking mechanism, we can now extend the transaction synchronization mechanism in the previous section to coordinate arbitrary groups of transactions.

---

1. Note that in sequencing, it is possible that the processing of a message is postponed, but only until a message sent from the same sending process is received.

4.3 Classes of Transactions

On the basis of the activity graph of a transaction, we can group transactions into three classes. These classes reflect the difficulty of correctly synchronizing the transactions, in terms of the mechanisms needed.

4.3.1 Transactions with Independent Components

The simplest class is that of transactions whose activity graphs contain no arcs. I refer to these transactions as transactions with independent components. A transaction with an activity graph with no arcs can be performed atomically using only sequencing by using the hierarchical protocol described in the previous section. Such a transaction places no constraints on the organization of the hierarchy, as any hierarchy can be used. The hierarchy can be chosen to optimize locality of reference, without concern for introducing the need for locking in these transactions.

An example of such a transaction would be a transaction which adds 5% interest to all of the savings accounts in a bank. The new value of each account depends only on its previous value. No matter how the accounts are distributed among data manager processes, each manager can compute the new balances of the accounts that it holds solely from their previous balances.[1]

It is instructive to see just how large this class of transactions is. All "query transactions", which do not perform any updates to the data base, fall into this class. A query transaction can always be performed by sending out a set of requests to the data

---

1. In this simple example, I have deliberately ignored other processing that such a transaction may be required to perform in an actual banking system, such as accumulating a total of the accounts or of the interest paid.

managers as an atomic broadcast in order to obtain a consistent "snapshot" of the data base.[1] Such requests can be sent as an atomic broadcast, using the mechanism of Chapter 3, in order to obtain a snapshot that reflects either all or none of the effects of any other transaction. The sender of the requests can then gather the replies and use them to satisfy the query.[2]

A second class of transactions that always have independent components are transactions that only make updates to the database. If the new values that a transaction gives to items are completely independent of the previous state of the data base, such a transaction has independent components.

A third class of transactions that always have independent components are transactions in a fully redundant data base, such as that considered in [Rothnie77,Thomas76]. Many of the protocols that have been developed for synchronization of transactions in a distributed data base work only for the fully redundant case. This point suggests that synchronization of transactions in a fully redundant data base may be somehow *easier* than synchronization in a data base in which each site holds only a partial subset of all of the data items. In fact, the fully redundant case is easier, because all of the transactions in a fully redundant data base have independent components, allowing synchronization to be accomplished without locking.

---

1. If the data needed to satisfy a query cannot be accurately predicted in advance, this may be a very large set of requests. An example of such a query would be "tell me the value of the record that this record points at."
2. Alternatively, the requests can ask the managers to make copies of the data items involved in the query, and the copies can be processed to satisfy the query in any efficient manner.

Actually, a fully redundant data base violates the assumption made about locality of reference, as all transactions that update the data base involve all of the sites. All such transactions must be sequenced by the root node of any hierarchy used for the hierarchical synchronization scheme.[1] A much more interesting case is that of a data base that is not completely redundant, but still has the property that all of the input items to a transaction exist on any site that holds an item updated by that transaction. All transactions in such a system have independent components, and may also exhibit locality of reference.

4.3.2 Transactions With Predictable Data Flow

A second class of transactions based on activity graphs is those with activity graphs with well defined arcs. I call this the class of transactions with predictable data flow. Some of the process steps that perform such a transaction must be completed before the input messages to other steps can be produced. A transaction in this class cannot be performed atomically using the atomic broadcast scheme in every hierarchy, but instead requires that each process that is the source of some arc in its activity graph be an ancestor of the process that is at the destination of that arc.

An example of such a transaction would be the simple check cashing transaction depicted in Figures 4.1 and 4.2. Any implementation of this transaction requires that an access to the item y precede the access that updates the value of z.

---

1. Note, however, that query transactions can always be implemented as being local to one site and run efficiently without locking.

4.3.3 Unpredictable Transactions

A third class of transactions, partially distinguished from the second, is those for which it is impossible to predict which items will be accessed and how until some of the accesses are performed. If we were to construct an activity graph for a transaction whose access pattern is completely unpredictable, that graph must include an arc between each pair of managers. Such a transaction would cause a great many cycles in when included in a joint activity graph, even though the *probability* that each arc is used in any particular invocation of the transaction would be small. This suggests that such transactions need special consideration so that they do not add to the cost of performing more predictable transactions.

An example of such a transaction would be a transaction following a linked list of records, performing some processing on each entry in the list. For such a transaction, it is impossible to predict which records will be accessed before the transaction is run. The transaction could potentially access any record in the file containing the linked list, and might transfer information from any of those records to any other record.

It should first be noted that unpredictability comes in degrees. Frequently, one can limit the set of items that a transaction could access, for example to the records in a particular file. Even relatively crude bounds can reduce the number of arcs in a transaction's activity graph to the point where it could reasonably be treated as predictable. The assignment of data items to managers can greatly affect the impact of unpredictable transactions. If all of the items that could be targets for accesses of such a transaction are under the control of a single manager, the unpredictability disappears. Thus if "unpredictable" transactions are frequent, the choice of the assignment of data items to managers should be made with this in mind.

The three classes of transactions discussed above are a categorization of transactions according to the difficulty of performing them atomically. I am assuming, and this assumption appears to be consistent with current practice, that the most frequent transactions will be those of the first two classes. In fact, in many current applications of distributed information systems queries are much more frequent than updates, making the transactions with independent components the most frequent. With this assumption in mind, I have designed a mechanism to provide correct synchronization for all three classes of transactions that is substantially more efficient and robust for transactions in the first two classes. This mechanism is the subject of the next section.

4.4 A Hierarchical Scheme for Transaction Synchronization

In this section, I present a mechanism for synchronization of transactions in a distributed information system that makes extensive use of the ideas developed above and in Chapter 3. The mechanism is described in terms of restrictions on the patterns of message passing that can occur during the execution of a transaction. In the next section, I consider the implementation questions in greater detail.

The mechanism that I will use for synchronizing transactions is an extension of the message forwarder mechanism described in Chapter 3. The processes are organized in a hierarchy including both data managers, which hold items, and message forwarders, which merely relay messages. Some of the data managers may act as message forwarders as well. Each process in this hierarchy now has two types of ports, a front door port, and some back door ports. The front door ports are used for receiving requests pertaining to new transactions, while the back door ports provide a mechanism that allows a process to receive additional messages pertaining to the current transaction *without* enabling reception of

requests from new transactions. This mechanism, together with the use of pre-requisites on process steps, will be used for locking.

A transaction can be initiated by any process by formulating a message describing the accesses to be performed. This message invokes a set of process steps that together perform the intended transaction. Some of these process steps are invoked by messages received at the front door of some process, while others are invoked by back door message reception. Messages sent to the front door of some process must follow a similar protocol to that used in atomic broadcasting:

> Messages sent to the front door may only be sent to the relatives (in the hierarchy) of the sender.
>
> A process receiving a message from one of its children through the front door may either send the message intact to the front door of its parent, without modifying its local state, or it may perform any processing desired on the message and generate messages for the front doors of its children.
>
> A process receiving a message from its parent through the front door may perform the desired processing and send messages to the front door ports of its children.

Messages sent to the front door follow the direct route in the hierarchy between the process that initiates a transaction and the data managers that perform the transaction. The same argument that was used to prove that the hierarchy of message forwarders correctly synchronizes atomic broadcasts can be used to show that the transactions are atomic as ordered by front door message receptions. Not all transactions can be performed entirely with front door message receptions, however. A back door message is required whenever the process step to be performed by one process depends on data held by some other process that is not one of its ancestors. In order to prevent the steps invoked by back door messages from

introducing ordering relationships that would make transactions non-atomic, several restrictions must be applied to back door messages:

> Any process involved in a transaction may send a message to the back door port of any other process involved in that transaction.
>
> The reception of a message at the back door of a process in conjunction with some transaction must be preceded by the reception of some message concerning that transaction at the front door of that process.
>
> No steps receiving messages at the front door of a process can occur between the step that receives a message at the front door about the transaction and the steps that receive messages at the back door about the same transaction.

These restrictions taken together insure that all of the steps of a process related to a particular transaction are consecutive and that the first step of each process related to a transaction is invoked through the front door. Thus the ordering of transactions as observed through all message receptions is the same as that observed only through the reception of messages at the front door ports, and thus the transactions are performed atomically.

The restrictions on back door messages require advance planning before a back door message can be sent. A process must know to expect a back door message from a transaction so that it will not receive any messages from other transactions before getting the back door message. Thus the message sent to the front door of a process that will subsequently be sent a back door message must contain a lock request, which causes that process to stop receiving messages at its front door until the expected back door message is received. The next section describes how the messages are constructed and routed to achieve this effect.

4.5 Implementation of Hierarchical Locking

This section discusses several details related to the implementation of the hierarchical locking scheme. First, I show how the messages needed in the implementation of a transaction are constructed from the description of the transaction. This is the responsibility of the transaction process, though the individual data managers must also send messages as outputs of the process steps that they perform. Another issue discussed is the coordination of messages sent to the back door ports to conform to the rules described in the previous section. An efficient implementation is described in which lock requests can be sent to a large number of processes *without* actually delivering the requests in most cases. This implementation makes it practical to run unpredictable transactions using this mechanism. Finally, I discuss the problem of choosing the hierarchy of processes. This hierarchy should be chosen so as to conform closely to the physical communication network topology, to reflect "locality of reference" in the transactions to be run, and to minimize locking.

4.5.1 Constructing the Messages

We must now show how a transaction process can perform its function of translating from a high level description of a transaction to be performed into a message that will eventually cause the desired transaction to be performed in accordance with the synchronization rules described above. I will first consider the class of transactions with predictable data flow, for which it is possible for the transaction process to know the set of accesses to be performed (or at least the manager processes that perform those accesses) in advance. Later, I will show how the scheme can be extended to transactions with unpredictable flow.

For a transaction with predictable data flow, the transaction process can know in advance what accesses are to be performed and thus could formulate a message to be distributed to the managers describing those accesses. Two steps must be carried out in formulating the set of messages. First, the accesses to be performed must be derived from the high level description of the transaction, and then the set of messages to be distributed to the managers must be constructed. The first of these tasks is performed by the transaction process.

The second task requires knowledge of the hierarchy as well as knowledge of the transaction. We could require each transaction process to have this knowledge, allowing it to formulate the set of messages described below, however it seems more natural to delegate this task to a process in the hierarchy that is above all of the data managers that are to participate in the transaction.

The transaction process formulates a description of the transaction that describes the accesses to be performed. This description may be similar to a transaction graph for the transaction. The transaction process then sends a message containing this description to its parent in the hierarchy.

Each process receiving such a description of a transaction to be performed examines the description to determine whether or not all of the data items accessed by the transaction are held by data managers that are below the receiving process in the hierarchy. If not, the description is forwarded intact to the parent of the receiver. If all of the data managers that are to participate in the transaction are below the receiving process in the hierarchy, the receiving process has the knowledge to generate the messages necessary to perform the transaction. The receiving process formulates a description of the transaction in a set of

components, one for each manager that participates in performing the transaction. These components are constructed as follows:

> Each manager is given a description of the accesses that it is to perform. This description may be at any level that is understandable to the data manager, the transaction process, and the process constructing this description.
>
> Each manager that is to receive back door messages is in addition given a lock request, which describes the nature of the expected back door messages.
>
> Each manager that must produce back door messages is given a description of what messages are to be produced and their destinations.
>
> Each manager that must produce input for its descendants in the hierarchy (because of an arc in the transaction graph) is given a description of the input to be produced.

The process constructing this description then treats it like a message that it has received through its front door containing component requests to be distributed. Each such message is processed as follows:

A process *M* receiving a message through its front door post examines that message to see if it contains a component for *M*. If not, the message is partitioned according to the message forwarder algorithm of Chapter 3 and distributed to the children of *M*. If the message contains a component for *M*, then *M* takes action on the message.

The action taken depends on whether or not the component of that message destined for *M* contains a lock request. If it does not, then *M* performs whatever access is specified (it is guaranteed to have sufficient information to do so), possibly modifies the other components of the message to include data values to be passed to its descendants, and distributes the components of the message to its children according to the atomic

broadcasting protocol. Any back door messages to be sent by *M* are also sent by the same process step.

If the message contains a lock request for *M*, then *M* cannot perform all of its accesses until it receives additional information. Some of the accesses to be performed by *M* depend on receiving additional information from some other process. *M* distributes the components of the message to its children (possibly modifying some of the components to include values of data items held by *M*), and sends any back door messages that are requested. *M* then stops receiving messages at its front door until necessary back door messages are received. When *M* receives all of the back door messages associated with the transaction that sent the lock request, it can perform the specified accesses and re-enable message reception through the front door.

Some care must be taken with back door messages to avoid confusion. The back door messages of several concurrent transactions for some process may become intermingled, causing a back door message to arrive at a process before the corresponding lock request. The simplest solution to this problem is to use a separate back door port for each transaction. The transaction process initiating a transaction choses an identifier for each transaction. This identifier can be combined with a process name to form a unique back door port for each transaction and each process involved in that transaction. A process that has received a lock request can then enable message reception only through the back door for the particular transaction being performed.

4.5.2 Coordination of Unpredictable Transaction

Two problems must be overcome in applying this mechanism to transactions with unpredictable flow. Each process that could receive a back door message in performing a transaction must be sent a lock request. Because the set of data managers involved in a transaction cannot be predicted until some of the transaction has been performed, all potential participants in a transaction must initially be sent lock requests. This allows the accesses that cannot be predicted in advance to be performed by sending messages to the back door of the appropriate manager when the access to be performed is known. The component request sent to a manager may cause that manager to send and receive back door messages dependent on the items that the manager holds and the information it receives in the back door messages. Any transaction can be performed in this way.

The second problem comes in determining when a transaction has been completed, so that the data managers sent lock requests can release those locks. Because the set of accesses to be performed is not known in advance, a process that has received a lock request does not know when it has received all of the messages connected with the transaction that it will ever receive and thus when to release its lock. Each manager must remain locked until it has received all of the messages that pertain to the transaction that sent the lock request.

A simple solution to the problem of determining when an unpredictable transaction has been completed is to have some process monitor the progress of that transaction. When the transaction has been completed, the monitoring process can send out back door messages to all of the recipients of locks to release the locks. This strategy may sound very inefficient, however the next section discusses the problem of distributing the lock requests, and describes an efficient implementation of locking for unpredictable transactions based on the approach of this section.

The progress of an unpredictable transaction is monitored by having each process that finishes some portion of the transaction report to a monitor process. Any process can act as the monitor for a transaction, however communication will probably be minimized if the highest process in the hierarchy involved in performing the transaction performs the monitoring function.

Each message (front door and back door) sent in performing a transaction carries a completion weight. A process initiating a transaction arbitrarily assigns completion weights to the messages that it sends so that these weights sum to one. Each process step redistributes the completion weight of the message that it receives among the messages produced by that step. No message is ever assigned a completion weight of zero, and every message sent by each process is given some completion weight.[1] If a step produces no output messages for other processes involved in the transaction, it instead produces an output message for the monitor containing the entire completion weight received at that step. Thus completion weights are gradually returned to the transaction monitor process as the various process steps of the transaction are completed. The transaction is done when the completion weights in the messages sent to the monitor sum to one.[2]

---

1. An optimization of this scheme would be to recognize the special case of a a message containing only lock requests. In performing an unpredictable transaction, many of the lock requests that are sent may be completely unnecessary, and need not be delivered before the transaction is completed. We can speed up the recognition of the completion of the transaction by assigning any message containing only lock requests a completion weight of zero. If the locks in that message are necessary, some other message with a non-zero completion weight will be forced to wait until the necessary locks are received. The next section describes a scheme in which the number of messages that must be sent to perform an unpredictable transaction can be substantially reduced by postponing or eliminating delivery of unnecessary lock requests.

2. The arithmetic on completion weights must be done carefully so as to avoid loosing or gaining completion weight due to round off error. One could maintain completion weights as rational numbers rather than as floating point solution so as to avoid round off error.

through the hierarchy, being forwarded only when "pushed" by subsequent messages or the completion of the transaction setting the lock.

This can be accomplished by slightly modifying the implementation of the processes in the hierarchy. Recall that each such process maintains a queue of messages to be delivered to each of its relatives in the hierarchy. The implementation described in Chapter 2 attempted to forward message from each of these queues whenever they were non-empty. One could instead construct the implementation so that messages are forwarded from a queue only when the queue contains a message which is not purely a lock request.

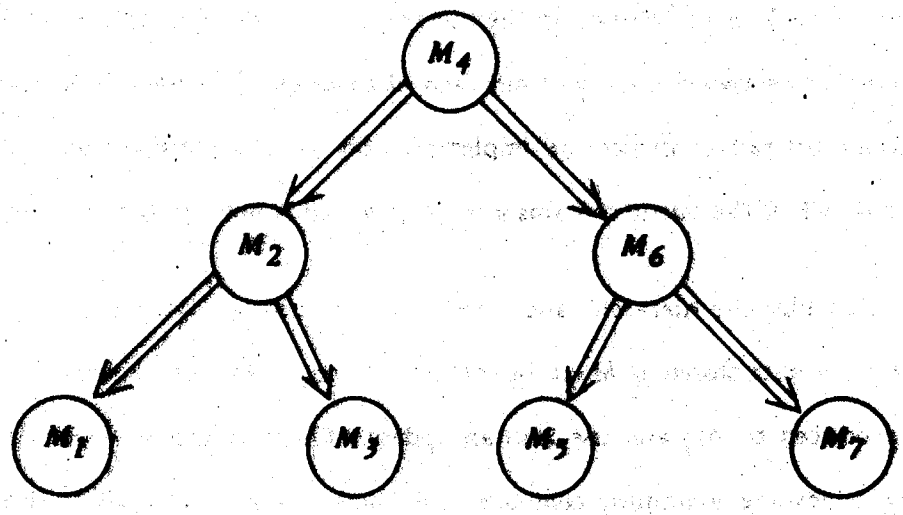
Consider the hierarchy and transaction depicted in Figure 4.4. The transaction uses the values of data stored at M_1 to update data at M_5. M_5's only participation is to take the value produced by M_1 and use it in an update. This transaction would be implemented by sending a message containing components for both M_1 and M_5. When this request reaches M_4, these components are separated. The component for M_1 travels quickly down the hierarchy to its destination. The component for M_5, however, contains only the lock request, and will not be sent from M_4 to M_6 until pushed by additional requests. Thus it is likely that while M_1 is computing the value to be sent to M_5, the lock request will be held up awaiting delivery to M_6. This allows M_5 to continue to participate in transactions local to the right hand half of the hierarchy while T is being performed at M_1.

This example raises another problem that must be solved: that of insuring that the lock request for a transaction does arrive eventually, and arrives before the back door messages of the transaction. This problem is partially solved by using a unique port for receiving back door messages related to each transaction, as once the back door message arrives, it will wait at its unique port until the corresponding lock request arrives. It would

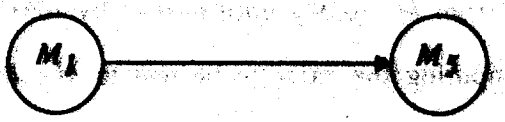
Figure 4.4

A Transaction Using Delayed Locking

The Hierarchy



The Activity Graph of T



be desirable if the lock request could be delivered promptly once the transaction has produced a message for the back door port. This can be achieved in one of two ways.

The implementation of a process could notice when a message is waiting at the back door port and send a request up the hierarchy to forward any lock requests. This strategy would be effective, but may requires additional message sending. If the communication network topology closely corresponds to the synchronization hierarchy, a second strategy may be more effective.

If the communication network topology closely parallels the topology of the hierarchy, then any message, including the back door messages, must essentially flow along the arcs in the hierarchy to reach its destination. We can take advantage of this fact to provide for prompt forwarding of lock requests when appropriate. Each process that is not a leaf node in the hierarchy now has a third type of port, a pass through port. Each such process is always ready to receive messages at its pass through port, and pass them on to one of its relatives in the hierarchy. The pass through ports provide a mechanism to send back door messages from one process to another in the hierarchy through intervening processes. Each such message is identified with its ultimate destination, and sent to the pass through port of the parent of the sender. When a process receives a message at its pass through port it passes it on either to the pass through port of one of its relatives or, if the ultimate destination is a relative of the receiving process, to the ultimate destination port.

The pass through ports provide a mechanism to allow each process in the path of a back door message to notice its progress. A process can match a lock request with back door messages that it is also forwarding by the unique port ID of the destination back door port. When a process has a back door message to be forwarded to one of its relatives, it checks its queue of front door messages to be forwarded to the same relative for a corresponding lock

request. If such a request is found, then the back door message can be combined with that request and replaced in the queue of front door messages. This will cause that request to be promptly forwarded, as it is now not solely a lock request.

One point of caution should be noted. If the lock request is contained in a message that has been forwarded but not but not yet acknowledged, it cannot be combined with the back door message and both must be forwarded independently. This is because the process to which the lock request had been sent may have already received it, so that it is too late to modify it. Thus each process must keep track of messages that have been forwarded but not yet acknowledged.

In the example of Figure 4.4, when M_1 has finished computing the value that it sends to M_5, it sends it as a back door message. This message propagates up the hierarchy through the pass through ports of M_2 and M_4. When M_4 attempts to forward this message to M_6, it notices the corresponding lock request. It combines the back door message with the lock request, and sends the combined message to the front door of M_6. When M_5 receives the combined message, it performs the specified update, and realizes that its role in the transaction is completed, setting and releasing a lock in the same process step. If, however, M_5 were required to receive additional messages in carrying out the transaction, it would remain locked until these messages were received.

Pass through ports also provide a mechanism to optimize the execution of unpredictable transactions. In an unpredictable transaction, a great many processes may be sent lock requests, and later lock releases and not participate in performing the transaction. Using the scheme for forwarding lock requests described above, most of these requests will not be delivered until the transaction has been completed, and will await forwarding at some level of the hierarchy. Thus while an unpredictable transaction may send out a great many

locks, few will actually be received. When the transaction is completed, however, lock release messages will be sent out for all of the participants in the transaction. Because these are sent out as back door messages, the processes forwarding the lock release messages will attempt to combine them with the lock requests still awaiting forwarding. When a lock request is combined with a lock release, it is known that the lock is unnecessary and both messages can be discarded.

Using this implementation, it is likely that most lock requests will be retained at a high level in the hierarchy. Most of the unnecessary lock requests will be canceled at a high level, before much effort has been expended in delivering them to their destinations. This implementation makes it practical to run transactions that are very uncertain and must lock a large number of managers but in fact perform very few accesses. If, as assumed throughout this thesis, most of the transactions involve managers with a common parent at a low level of the hierarchy, then running a transaction that sets many unnecessary locks interferes very little with the execution of most of the transactions, as the lock requests that are not needed never reach the level in the hierarchy at which they would interfere with the more frequent transactions.

4.5.4 Choosing the Hierarchy

Several considerations should guide the choice of a synchronization hierarchy for a distributed information system. The hierarchy should reflect the patterns of locality of reference in the expected transactions. There are frequently natural boundaries of the application, such as the local and regional offices of an organization using a distributed information system for inventory control, which can guide this choice.

A second point in guiding the choice of hierarchy was described in Chapter three. The hierarchy should be chosen to reflect the underlying communication network. This allows much more efficient implementation of the protocols in terms of true amount of communication required. If the hierarchy exactly matches the network, the communication required for these protocols is minimized.

In many cases, the topology of the communication network closely parallels the patterns of locality of reference. This is because it makes sense for the sites that must communicate most frequently to be directly connected, or to be connected to a shared broadcast network. One notable exception is networks in which the sites are connected through a common carrier, such as the ARPANET, or TELENET. In these cases, many independent distributed information systems share the same communication network. The patterns of locality of reference in the individual applications are lost in the load seen by the communication network as a whole. Thus the topology of such a network is not likely to resemble the pattern of locality of reference in any single application.

A third factor in the choice of the hierarchy is the capacity and reliability individual sites. Given some reasonable approximations for the expected transactions, one can estimate the volume and importance of the message traffic through each site. These should be used in evaluating whether or not a particular organization is suitable, by insuring that each site has sufficient capacity to handle the expected message traffic, and that very important transactions do not depend on the availability of a site that is known to be unreliable.

Another factor to be considered is the desire to avoid locking. Locking is undesirable both because it increases the number of messages that must be sent (the lock request messages), and violates the goals of autonomy and partial operability. A process that has received a lock request is dependent on other processes to complete the transaction and

release the lock before it can continue processing other transactions. In the next Chapter, I will present a mechanism that provides a solution to this problem, allowing a process that has received a lock request to continue processing other transactions before the outcome of the transaction sending the lock is known. It is still desirable, however, to reduce locking, and to choose the hierarchy so that frequent transactions do not require locking, and processes managing frequently used data are rarely locked.

4.6 A Rejected Alternative Solution

This solution is of course only one of many that could be used for the problem of controlling transactions. There are several solutions that provide correct synchronization with simpler protocols. In this section, I discuss briefly one of these alternatives and the reasons for its rejection.

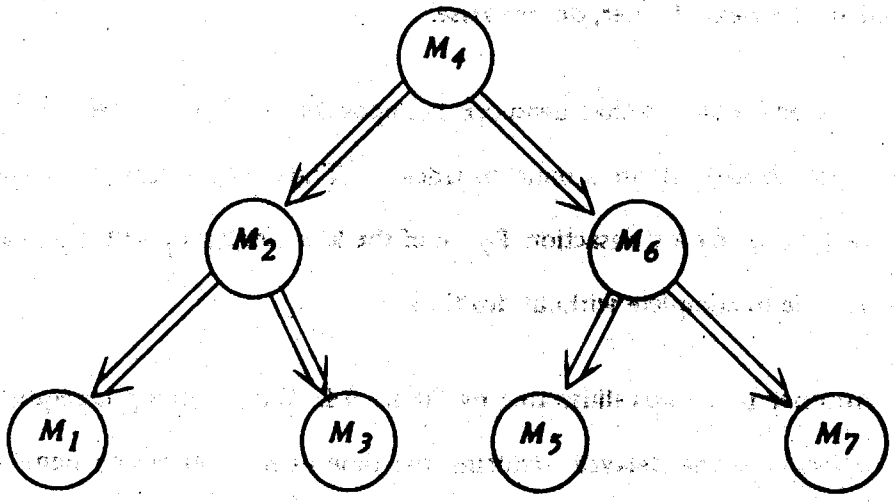
Considerable complexity is introduced into the scheme by the ability to begin a transaction at any level of the hierarchy. If we had required all transactions to begin with a request sent to the root of the hierarchy, it would be easy to lock a large portion of the hierarchy in order to perform some transaction. This could be done as follows:

When a process receives a message, it performs whatever action is required of itself, and passes on the components of that message to its children as before. If the process requires input from one of its children to complete its requested access, or if one of the requests forwarded cannot be completed solely based on the information in that request, the process sets a lock and stops receiving new messages until it can complete its requested action and distribute all of the necessary information to each of its children. Each process makes a local decision about locking and there is no difficulty detecting when a transaction has been completed.

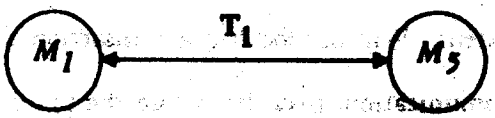
Figure 4.5

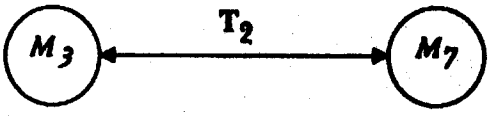
Concurrency Restrictions Due to Hierarchical Structure

The Hierarchy



A Joint Activity Graph of the Transactions





4.7 Conclusions and Summary

Several points about this solution should be noted. One is that a large number of transactions can be performed without locking. The hierarchy can be arranged so that the transactions expected to be most frequent do not require locking. Without locking, the problem of deadlock detection and prevention, and the "distributed atomic update problem" described in the next chapter, do not arise.

A second point is that deadlock is impossible in this scheme. The locks are set in messages distributed in an atomic broadcast. Thus if any lock set by a transaction T_1 precedes a lock set by a transaction T_2, all of the locks set by T_1 will precede those of T_2 and T_1 will be able to complete without deadlock.

Another point was illustrated by Figure 4.1. When locking is required, frequently the setting of locks can be delayed, reducing the time interval in which items are locked. The scheme presented does this by delaying ordering decisions, and distributing the decision of a transaction's order relative to other transactions through the hierarchy.

The hierarchical scheme for performing transaction achieves the goal of partial operability to some extent. Without locking, a transaction can be performed as long as all of the processes and communication links that lie on the paths between the processes that must communicate in performing the transaction are functional. While this does not completely achieve the goal, as it is possible that two processes will be prevented from performing some transaction because of the unavailability of their parent in the hierarchy, the hierarchy can be tailored to make this circumstance unlikely.

Locking introduces the possibility that a process will be prevented from performing local operations because a failure has delayed the transaction setting a lock. In the next chapter, I present a mechanism to deal specifically with this problem.

This chapter introduced a method for analyzing the patterns of the accesses performed by a transactions, namely transaction graphs and activity graphs. Using these transaction graphs, we demonstrated that sequencing of messages between processes is not itself sufficient to provide synchronization for some sets of concurrent transactions. Three classes of transactions were discussed. Many of the transactions that we expect to be performed in information system fall into either the class of transactions with independent components, or the class of transactions with predictable data flow. (The transactions in the example system discussed in chapter 6 are nearly all in the first class.) These are the simplest transactions to synchronize.

A mechanism was presented to coordinate concurrent transactions using the atomic broadcasting mechanism developed in Chapter 3. This mechanism correctly synchronizes transactions of all three classes, but works most efficiently (in terms of the number of messages needed) on transactions in the first two classes. The mechanism can be optimized to perform those transactions that are known to be important at the time of the design of the system.

The implementation of this mechanism was considered to show how the messages are generated from a description of the transaction, and how the processes are implemented. This implementation demonstrated techniques to reduce the amount of overhead caused by transactions requiring locking. Finally, the important properties of this solution were summarized.

Chapter 5

Polyvalues: A Mechanism for Performing Atomic Updates to Distributed Data

In this chapter, I consider the implications of using locking on the problem of achieving the goal of partial operability. First, I show that no system that uses locking can achieve this goal. A mechanism is presented that solves this problem, by allowing a process that is participating in a transaction and has set a lock to install the results of that transaction conditionally, so that it can release the lock and continue processing other transactions before knowing whether or not the transaction setting the lock will be completed.

5.1 Motivation (The Trouble with Locking)

In the previous chapter, it was demonstrated that some form of locking is necessary for synchronizing certain groups of transactions. Unfortunately, locking compromises the goal of partial operability, as a site that has received a lock cannot perform local transactions conflicting with that lock until the locking transaction is completed. One could imagine a solution to this problem in which a site that has received a lock could abandon that lock, aborting the transaction setting that lock. This must be done in such a way that if a lock is abandoned, all of the sites participating in the transaction which set that lock will decide to abort that transaction.

To achieve the goal of partial operability, each site must be able to decide whether or not to complete the transaction without consulting other sites. In this chapter, I refer to the decision of whether or not a transaction has been completed as the outcome of the

In order to achieve the goal of partial operability, the locking must not exclude transactions local to process X or to Y indefinitely. Failure of X or of Y or of the communication network connecting them may, however, delay any message sent between the two indefinitely. This means that each process must at any point be able to decide whether or not to abort the transaction in progress without communication with other processes.

A protocol of message exchanges between X and Y that decides the outcome of a transaction can be viewed as series of process steps in each process. Each of these steps is triggered by a message that may be delayed indefinitely, so that after each step, each process must be prepared to decide whether or not to abort the transaction. This decision must be based *only* on the information that that process had before beginning the protocol and the information gained from messages received while performing the protocol. Both processes must make the same decision at any point in the protocol.

If a failure delays messages after the first step of the protocol is performed in each of the processes, at least one of the processes must decide to abort the transaction. This is true because the transaction being performed requires locking, and a transaction requiring locking cannot be performed with a single process step in each process. Therefore, after each process has performed one step of the protocol, at least one of the processes must have insufficient information to complete the transaction, and thus must decide to abort.

If the execution of the protocol is not delayed by a failure, a step must be reached after which one process would decide to complete the transaction if the next message in the protocol were delayed by a failure. Let the first step of either process after which that process decides to complete be known as the commit point of the transaction and assume that it is a step of process X. After the commit point, X would decide to complete the transaction if a failure delayed the completion of the protocol.

Now consider the decision made by process Y if a failure were to prevent communication immediately after the commit point step. Because the commit point was not a step of Y, Y cannot be effected by the completion of that step, and hence must make the same decision before the commit point as after. This is would be a contradiction, as Y must either decide to complete before the commit point, violating the assumption that the commit point was the first step after which either process decided to complete, or Y must decide to abort after the commit point, resulting in an inconsistent decision.

This argument applies to any number of processes attempting to perform some transaction requiring locking, and shows that there is no way to achieve the goal of partial operability while performing transactions requiring locking. The argument depends on the property of the process model that the immediate effects of performing a process step are limited to one process, and that the observation of the completion of a process step by any other process may be delayed indefinitely.

5.1.2 Approaches to the Problem of Abortable Locking

There are several approaches that can be used to reduce the probability that a failure during the execution of a transaction requiring locking will cause indefinite delay. These approaches provide only a partial solution to the problem of achieving the goal of partial operability because a failure or combination of failures during the execution of a transaction can cause indefinite delay of transactions that are completely local to a functioning site, or cause the transaction to be performed inconsistently.

5.1.2.1 Accepting Inconsistency

One possible solution that has not been extensively used is to accept a small probability that a transaction requiring locking will not be performed atomically if a failure occurs at the wrong time. This approach is not appropriate for all applications, as strange, inconsistent results may occur. If the consequences of not being able to perform some transaction promptly are worse than the consequences of a synchronization error, (as would be the case for a transaction controlling the landing of an airplane), then it may be desirable to use a protocol in which a failure at the wrong time cause a transaction to be partially performed, or may cause the transaction to be incorrectly sequenced with other transactions. This kind of strategy has been used in image processing systems, in which the data base has a great deal of redundancy that allows any observer to tolerate an inconsistent state. To my knowledge, there are no distributed data management systems that use this approach.

5.1.2.2 Avoiding Locking

Another approach is to use synchronization protocols that minimize the need for locking. The protocols presented in Chapter 4 of this thesis and those used by the SDD-1 distributed data base system[Bernstein77] are two examples of this approach. In Chapter 4, I examined the problem of organizing the data base so as to reduce the amount of locking required. Locking cannot be avoided entirely, however, unless the data base is replicated so that each site has a complete copy. Such replication eliminates locking, but makes all transactions that update the data base require the participation of all of the sites, eliminating transactions that are local to one site.

5.1.2.3 Minimizing the Window of Vulnerability

The approach most frequently taken to locking in a distributed system is to minimize the time interval during which a failure causes indefinite delay. One example of this approach is the two-phase commit protocol described by Gray [Gray77]. Each site participating in a transaction goes through two phases, a lock phase in which locks are set and the site computes the results of the transaction, and a wait phase during which the site has completed the computation related to the transaction and has the information necessary to make the updates requested by the transaction without further input from other sites, but does not yet know the outcome. If a failure delays the completion of the lock phase at a site, the site can decide on its own to abort the transaction, and all sites will eventually decide on their own to abort or be told of the decision to abort. If a failure delays messages during the wait phase, however, a site must wait until it receives a message indicating the outcome of the transaction.

Figure 5.1 gives a finite state machine description of the action of one of the sites in this protocol. The figure shows four states of the execution of the site. The allowed transitions are shown by the arcs, each of which is labeled with the message received to begin this transition, and the message sent in making this transition in italic type.

In the lock state, a site waits for messages containing the information necessary to complete its portion of the transaction by determining the new values for the items at that site updated by the transaction. After these have been received, the site enters the wait phase and sends an acknowledgement message indicating this fact. If a failure delays the reception of messages by a site in the lock phase, that site can abort the transaction by sending an abort message and entering the abort state, discarding any computation done by the transaction. In either the abort or the done states, the site is ready to accept new

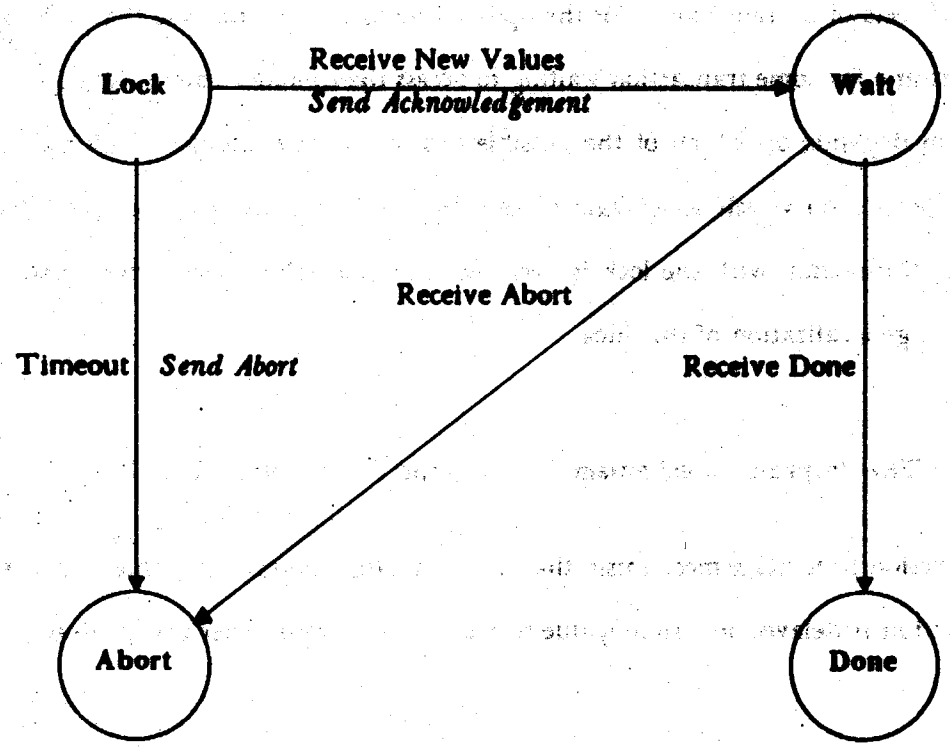
transactions. The acknowledgement and abort messages sent by the sites are accumulated by a coordinator for the transaction until either all sites have responded acknowledge, or any site has sent an abort. The coordinator then generates done or abort messages for all of the participants.

The motivation behind this protocol is that the time that each site spends during its lock phase computing the results of the transaction is likely to be longer than the time spent during the wait phase. This is not necessarily true, as one site may take much longer than

---

Figure 5.1

A Two-Phase Commit Protocol



the others to complete its portion of the transaction, causing the other sites to remain in the wait phase for a long period of time.

Lampson and Sturgis [Lampson76] present another commit protocol that includes an extra round of message exchanges to avoid this problem. In their protocol, no site enters its wait phase until all of the computation of the transaction has been completed at all sites.

5.1.2.4 The Polyvalue Approach

The motivation behind preventing a transaction from holding on to a lock indefinitely is to be able to run other transactions that need to access the data that has been locked without indefinite delays. Frequently, the results produced by a transaction depend only loosely on the input values seen by that transaction. If the outputs to be produced by a transaction holding a lock are known but the outcome of the transaction is uncertain, there are two possible sets of current values for the updated items. One could use these two sets of values to determine for some transaction waiting to access the updated values, whether or not that transaction depends on which of the possible sets of values is correct. Any transaction that does not depend on which set of values are used can be run using either set *before* the outcome of the transaction with the lock is decided. The polyvalue scheme described in the next section is a generalization of this idea.

5.2 The Polyvalue Mechanism for Avoiding Delay Due to Locking

This section presents a mechanism that in many cases solves the problem of insuring that no transaction is delayed indefinitely due to a lock set by some other transaction.

5.2.1 The Polyvalue Concept

If a two-phase commit protocol is used to perform a transaction, a site that has reached the wait phase knows output values of the transaction. If those values could somehow be conditionally installed, such that a transaction accessing one of the updated items would see both values, then the locks on the updated items could be released. This can be accomplished by installing what I refer to as a polyvalue for each updated item. A polyvalue is a bookkeeping tool for keeping track of several potential current values for an item, depending on the outcome of currently pending transactions.

A polyvalue is a set of pairs, <v,c>, where v is a value and c is a condition, which is a predicate on a set of identifiers for transactions. The pair <v,c> in a polyvalue for some item I specifies that I has value v whenever c is true when c is evaluated in a model where transaction identifier T is *true* if T has been completed. The conditions in a single polyvalue must be disjoint (no assignment of truth to the transaction identifiers makes two conditions in the same polyvalue true) and complete (for any assignment of truth values, one condition is true).

Each transaction is assigned a unique transaction identifier. When a site that has reached the wait phase for a particular transaction T cannot determine quickly whether T will be completed or aborted, that site installs polyvalues for all of the items that T is trying to update. The polyvalue installed for an item I has two pairs, <v',*T*>, and <v,¬*T*>, where v was the value of I before the execution of T, and v' is the value produced by T. This polyvalue describes the possible values that could be the current value of I, depending on the eventual outcome T.

Before installation, each polyvalue is simplified in three steps. First, individual pairs are expanded. Any pair $\langle v,c\rangle$ where v is itself a polyvalue is replaced by a group of pairs. This group contains one pair of the form $\langle v_i, c_i \wedge c\rangle$ for each pair $\langle v_i, c_i\rangle$ that was in v. Next, redundant pairs are coalesced. The pairs $\langle v_1, c_1\rangle$ and $\langle v_2, c_2\rangle$, where $v_1 = v_2$, are replaced by the single pair $\langle v_1, c_1 \vee c_2\rangle$. These redundant pairs can arise because it is possible that several different possible outcomes of the pending transactions could produce the same value for an item. Finally, the condition attached to each pair is simplified, and any pair $\langle v,c\rangle$ for which c is logically *false* is discarded.

This simplification procedure reduces the polyvalue constructed to one in which each pair has a simple value, and the number of pairs is minimized. A polyvalue with a single pair $\langle v,c\rangle$, must have a condition c which is logically *true*, and is indistinguishable from a simple value. Thus the procedure for constructing polyvalues for the results of a pending transaction can be described without treating the cases where the new or old values of the updated items are themselves polyvalues as special in any way.

5.2.2 Performing Transactions on Polyvalues

A transaction operating on polyvalued input items is known as a polytransaction. A polytransaction produces polyvalued results, each of which specifies the real output value produced under any possible outcome of pending transactions. The following is a simple discussion of polytransactions.

I will first describe the computation phase of a polytransaction, in which input values are read and outputs are computed. Each polytransaction T consists of a set of alternative transactions T_c, each of which performs the same transaction on a different set values for input items. Each alternative transaction T_c is tagged with a condition c, which is derived

from the conditions on the input values read by T_c. Each polytransaction begins with a single alternative transaction T_{true}, which begins to access items in performing the transaction. When an alternative transaction T_c accesses an item whose current value is a polyvalue $v = \{<v_i,c_i>\}$, T_c is partitioned into a set of alternative transactions, $\{T_{c \wedge c_i}\}$, each of which has the same history as T_c, and each of which accesses one value v_i from v and acquires the corresponding condition c_i, in addition to the previous condition, c, on T_c. If $c \wedge c_i$ is logically *false*, then $T_{c \wedge c_i}$ can be abandoned, and not computed.[1]

Thus the number of alternative transactions grows as a polytransaction T is run. Each of these alternative transactions runs up to the wait phase (i.e. each runs until the outputs have been computed and distributed to all of the appropriate sites). Each site receiving outputs of T constructs a polyvalue for each item I to be updated. This polyvalue contains the pairs <v,c> where v is the value produced by T_c for I.

If all alternative transactions of T produce outputs for some item I, then this set of pairs will be complete and disjoint.[2] If, however, there are some alternatives of T which do not produce a value for I, then the conditions of the alternatives which do produce values for I will not be complete. This can happen if the decision of whether or not T updates I depends on the input values seen by T. Under any outcome of pending transactions for which T will not produce a new value for I, I would retain its previous value. Therefore, if the conditions on the alternatives of T which produce a new value for I do not form a

---

1. As will be shown, outputs produced by a alternative transaction with a condition that is logically *false* will never be used.
2. T begins with a single alternative with condition *true*. As the computation phase of T progresses, alternatives of T are partitioned according to the conditions on the polyvalues that they access. Because the conditions on the pairs of any individual polyvalue are complete and disjoint, the conditions on the alternatives of T are at any point complete and disjoint.

complete set, another pair <v',¬c'> is added where v' is the previous value of I, and c' is the logical OR of all of the conditions on the new values for I.[1] The wait phase of a polytransaction proceeds as described above. Should a communication failure interfere with the wait phase, a polyvalue can be produced from the outputs of the polytransaction and the previous values for the items updated by the polytransaction.

5.2.3 A Simple Example

Let us consider a simple example involving three items at three sites, and three transactions on those items. Let A, B, and C be the items, and let the transactions be:

T_1 = If A $\geq$ 100 then {A = A - 100; B = B + 100}

T_2 = IF B $\geq$ 100 then {B = B - 100; C = C + 100}

T_3 = if B > 10 then B = 1.05*B

Now assume that before the transactions are run, each item has value 100. If a failure occurs during the wait phase of T_1 preventing the site holding B from learning the outcome of T_1, then that site gives B a polyvalue of {<200,T_1>, <100,¬T_1>}. If T_2 is now run, it will be run as a polytransaction, because of the polyvalue of B. T_2 would produce new values for B and C of {<100,T_1>, <0,¬T_1>} and 200. If a failure occurs during the wait phase of T_2 again preventing the site holding B from learning the outcome of T_2, then after simplification, B receives a polyvalue of {<0,¬$T_1 \wedge T_2$>, <100,($T_1 \wedge T_2$)$\vee$(¬$T_1 \wedge$¬T_2)>, <200,$T_1 \wedge$¬T_2>}. Now, if T_3 is run, it is performed as three alternative transactions. Two of these alternative transactions produce updated values for B, while the alternative for ¬$T_1 \wedge T_2$ does not, because the input value for B read by that alternative transaction is too small. Thus the

---

1. One could alternatively always add this pair, and rely on the simplification procedure to discover that ¬c' is logically *false* when the other conditions are complete.

polyvalue assigned to B by T_3 after simplification is $\{<0,\neg T_1 \wedge T_2>, <105,(T_1 \wedge T_2) \vee (\neg T_1 \wedge \neg T_2)>, <210,T_1 \wedge \neg T_2>\}$.

This example shows the mechanics of manipulating polyvalues, to perform transactions, even after the occurrence of improbable failures. From this example, it is hard to see what has been gained, as one cannot determine from inspection what the values in the data base are, or what transactions have been completed.

The answer is that in many cases, a polytransaction will produce simple output values. This is true of many query transactions, which attempt to determine whether or not the value of some item falls in a certain range. In many cases, a query about an item can be answered without knowing the exact value of that item. A polyvalue can provide all of the information necessary to answer common queries. Consider, for example the test made by T_2 on B. The decision made by this test is the same when applied to both components of the polyvalue for B.

Another area where polyvalues are useful is that of transactions that have real world effects, such as authorizing transfers of money, or allocating a real world resource, like a seat on an airplane. For such transactions, it is frequently more important to know what the real world effect is than to know what the eventual values in the data base are. If such a transaction is run on an input set containing polyvalues, then the real world effect can be accurately determined when all alternatives produce the same effect. In many applications, important real world effects can be determined without knowing the exact values in the database.

Consider, for example, a transaction which is to withdraw funds from a savings account for which the current balance is represented by a polyvalue. The important effect that the transaction must decide quickly is whether or not the customer is to receive the cash from the withdrawal. Computing exactly the new balance in the account need not occur rapidly. The transfer of funds depends only loosely on the balance in the account in that it need only be determined that that balance is, under all possible outcomes of pending transactions, greater than the amount withdrawn. Thus in most cases the withdrawal can be quickly authorized.

5.3 Recovery of Pending Transactions

The mechanism described above installs polyvalues for the results of a transaction T delayed in the wait phase by a temporary failure. When that failure is recovered, the wait phase of T can be completed, determining whether T is to be completed or aborted. Thus the value of the transaction identifier for T appearing in conditions in the pairs of polyvalues can then be determined.

A site learning of the completion or abortion of a transaction T can reduce its polyvalues by re-evaluating any condition that depends on the outcome of T, substituting either *true* or *false* for T depending on whether T was completed or aborted. This substitution simplifies conditions that involved T, and upon simplification, some of these conditions may become logically *false*. Thus knowledge of the completion or abortion of pending transactions can be used to reduce the number of possible values which a polyvalue represents. Eventually, if the outcome of all pending transactions is known, each polyvalue will have only one pair with a condition that is not logically *false*, and thus can be reduced

to a simple value. Some mechanism must be provided, however, to propagate the knowledge of the outcome of a transaction T to sites holding polyvalues with conditions involving T.

Such a mechanism must insure that all sites that hold a polyvalue with a condition dependent on a transaction T will eventually learn of the outcome of T. We also desire that knowledge of T be deleted when it is no longer necessary (i.e. when no condition involves T). The record of the completion or abortion of a pending transaction is similar to a commit record [Reed78] for that transaction. Unlike a commit record, however, knowledge of the outcome of a transaction may still be needed even after all of the output values of the transaction have been installed. Any polyvalue could potentially refer to any pending transaction.

One could have each site maintain a table of outcomes of pending transactions, and use a system-wide garbage collection strategy to delete entries that are no longer relevant. While this scheme would work, it would be inefficient in the case that dependence on the outcome of pending transactions does not in general spread very far. Most sites do not need to know the outcome of most pending transactions.

Another possible mechanism is to give a site that creates a polyvalue for a pending transaction the responsibility of maintaining a record of the outcome of that transaction until such a record is no longer necessary. When a site wishes to reduce a polyvalue, it must ask all of the sites that are responsible for maintaining a record of the outcome of the transactions appearing in that polyvalue of those outcomes. To do so, information must be passed along with the polyvalue to determine the relevant sites to ask. This scheme is similar to that used with possibilities by Reed [Reed78].

There are two main problems with using this scheme for keeping track of pending transactions. One problem is that it would be difficult to determine when the record of the outcome of a pending transaction is no longer needed, and some form of garbage collection may be necessary. A second problem is that the messages sent to inquire about the outcome of a transaction may place a burden on the communication network, as the inquiring message may be sent many times (once for each attempted access to the polyvalue) before the outcome of the transaction is determined. The scheme described below overcomes these problems by distributing the responsibility for maintaining the outcome of a pending transaction among the sites that have polyvalues dependent on that outcome.

Each site maintains a table, referred to here as the polyvalue table, listing the items that it holds that currently have polyvalues. This table is used to locate all of the polyvalues that can be reduced when the site receives a message indicating the outcome of some pending transaction. A second table maintained at each site, known as the transaction table, keeps track of the spread of knowledge of pending transactions. Each entry of the transaction table contains a transaction identifier, its outcome, (completed, aborted, or pending), and a list of sites to which this site has sent information dependent on the outcome of that transaction.

To maintain its transaction table, a site must make an entry for each transaction identifier that appears in a condition of a polyvalue at the time that that polyvalue is installed.[1] When a site sends a message containing a polyvalue to some other site, it must

---

1. No action is required if the site already has a table entry for that transaction.

record the name of the site to which the polyvalue was sent in the transaction table entry for each transaction identifier that appears in a condition in that polyvalue.

The information in the transaction tables in the various sites is used to control the distribution of knowledge of transaction outcomes. Each site that receives a commit or an abort message for a transaction that it previously knew as pending can update its table entry for that transaction, and reduce any polyvalues that depended on that transaction. A site is responsible for informing all of the sites that are listed in its transaction table entry for the transaction of the outcome. This list was constructed to include all of the sites that were given information dependent on the outcome of the transaction, and therefore may hold polyvalues dependent on that outcome. Once all of these sites have been informed, the table entry for the transaction can be deleted.

With this scheme, knowledge of a pending transaction propagates only to those sites which have received polyvalues dependent on the outcome of that transaction. If a great deal of computation has been based on the outputs of a pending transaction, then informing all of the appropriate sites of the outcome of that transaction may require many message exchanges.[1] If the outputs of a pending transaction are not used, however, only the sites that hold those outputs need be informed of the outcome of the transaction.

Figure 5.2 shows how this scheme works in the example described above. Let T_1 and T_2 be the two transactions described earlier on items A, B, and C. Assume that these items are held by sites *A*, *B*, and *C* respectively. The figure shows the values of these items

---

1. In fact, if the polyvalues depending on a pending transaction are used frequently, a site may have to be informed of the outcome of that transaction several times. It is possible for a site to receive a polyvalue dependent on the outcome of a transaction after that site had been informed of the outcome of that transaction and had forgotten that outcome. A site does not need to remember transaction outcomes indefinitely.

and the tables of pending transactions in the sites maintaining these items at several stages: initially; after T_1 is suspended; after T_2 is suspended; after T_1 is eventually completed; and after T_2 is eventually aborted.

5.4 Use of Polyvalues in the Hierarchical Locking Scheme

The discussion of polyvalues thus far has been at a relatively high level, so as to be applicable to any distributed system in which locking without unbounded delay is needed. The polyvalue mechanism described above could easily be incorporated into most of the distributed update algorithms that appear in the literature. I shall now consider how to apply these ideas specifically to the distributed locking scheme described in the previous chapter.

Recall that in the locking scheme of the previous chapter, any process producing outputs to a transaction depending on inputs obtained from another process which is not one of its ancestors in the hierarchy is sent a lock request message. The lock request message causes the process to refuse to receive any new messages pertaining to other transactions until the transaction issuing the lock is completed. The processes involved in the transaction exchange messages until each locked process has sufficient information to produce its outputs and release its lock. In order to apply the concept of polyvalues, this locking strategy must be modified so that each locked process goes through two phases, a computing phase in which the process could abandon its lock and cause the transaction to be aborted, and a wait phase in which the lock cannot be abandoned, but the output values are known.

I will first consider the case of a predictable transaction, where the set of processes making updates is independent of the data values seen by the transaction. This assumption simplifies the task of deciding when a transaction can be completed, as each process making

Figure 5.2

Recovery of Pending Transactions

Initial State

A	B	C
100	100	100

Transaction Tables:

(empty)	(empty)	(empty)

After T_1 is Suspended

A	B	C
$\{<0,T_1>, <100,\neg T_1>\}$	$\{<200,T_1>, <100,\neg T_1>\}$	100

Transaction Tables:

T_1,pending,{}	T_1,pending,{}	(empty)

After T_2 is Suspended

A	B	C
$\{<0,T_1>, <100,\neg T_1>\}$	$\{<0,\neg T_1 \wedge T_2>, <200,T_1 \wedge \neg T_2>,$ $<100,(T_1 \wedge T_2) \vee (\neg T_1 \wedge \neg T_2)>\}$	$\{<100,\neg T_2>, <200,T_2>\}$

Transaction Tables:

T_1,pending,{}	T_1,pending,{C} T_2,pending,{}	T_2,pending,{}

After T_2 has been completed, and A and B have been notified.

A	B	C
0	{<100,T_2>, <200,¬T_2>}	{<100,¬T_2>, <200,T_2>}

Transaction Tables:

(empty)	T_1,done,{C}	T_2,pending,{}

After C has been notified of T_1
And T_2 has been aborted.

A	B	C
0	200	100

Transaction Tables:

(empty)	(empty)	(empty

---

updates knows when it can complete those updates, and the set of processes making updates is known in advance.

For each transaction, one process serves the function of transaction coordinator. The transaction coordinator has the responsibility for determining when all of the processes involved in the transaction have reached the wait phase. To begin the transaction, messages must be sent to each of the processes involved in the transaction in a single atomic broadcast. The protocol of Chapter 4 must be slightly modified to send every process that is to perform an update a lock request message. Recall that the protocol of Chapter 4 sends lock requests only to those managers that cannot complete the transaction in one process step. The extra locking is needed in implementing polyvalues because we wish to be able to abort the

transaction if the completion of the transaction is delayed, and thus cannot allow any manager participating in the transaction to complete its portion of the transaction before the decision to complete is made.

Each process performing an update thus receives a lock request, along with any other instructions for completing the transaction. When a process has enough information to perform its update, it sends a "ready" message to the coordinator. (For any process whose update can be made without inputs from other processes, this happens immediately). Before sending the "ready" a process can decide to abandon its lock at any point and cause the transaction to be aborted. After sending the "ready" message, a process enters its wait phase and cannot abandon its lock. When all of the processes that were sent lock requests have answered "ready", the coordinator decides to complete the transaction and sends "complete" messages to the back door ports of the processes which received locks. Upon receipt of the "complete" message, a process completes its update and enables reception of new requests. If too much time elapses before the coordinator receives "ready" messages from all locked processes, the coordinator can abort the update by sending "abort" messages to all. The "ready", "complete", and "abort" messages must all be identified with a unique identifier for the transaction (probably assigned by the transaction process that initiated the transaction), so that delayed messages do not cause confusion.

Each process in this protocol goes through two phases, a lock phase before sending the ready message, and a wait phase after sending that message. After having sent a "ready" message, a process knows the new values that some items in its local state will take on as a result of completing the update. The process can, instead of waiting for a "complete" or "abort" message, decide to install polyvalues for these items. Each data manager process acts like a site in the polyvalue scheme described in the first part of this chapter. Messages sent

from one process to another containing data items or results computed from data items can contain polyvalues as well.

Two problems must be overcome in extending this scheme to arbitrary transactions. First, the coordinator must be able to know when the transaction can be completed, as the set of processes making updates is not known in advance. Second, each process that participates in the transaction must be able to determine when it has received all of the messages that it will receive as a part of the transaction, so that it knows when to enter the wait phase.

The first of these problems was discussed in Chapter 4, and the completion weight mechanism was introduced for its solution. The transaction coordinator can also act as the transaction monitor, which can determine when all of the computation of a transaction has been completed.

We can modify the completion weight scheme to allow uncertain transactions be performed with a two-phase protocol. Each process step of an uncertain transaction which prepares a set of output values to be installed must return some completion weight to the coordinator *whether or not it also sends messages to other processes.* The coordinator thus receives messages containing completion weight from each process that has updated items to be installed. When the completion weight sent to the coordinator reaches one, the coordinator sends out lock-release messages as before. These lock-release messages are distributed as described in Chapter 4.

In this protocol, each manager can at any point decide to abandon its lock and continue processing other transactions. To do so, a manager installs any updates that the transaction has made as polyvalues, and simply ignores any further messages about that transaction (except for the lock release or abort messages from the transaction coordinator).

This action may or may not cause a transaction to abort, depending on whether or not that transaction requires further participation by the manager which has abandoned it. If the manager abandoning the transaction is not needed to complete the transaction, then eventually, the completion weight returned to the coordinator will sum to 1, assuming no other manager decides to abort. If, however, the manager deciding to abandon the transaction must perform additional processing to complete the transaction (either by supplying more inputs, or making updates, the transaction will not complete, because the portion of the transaction dependent on the abandoning manager can not be completed. Eventually, the coordinator will decide to abort the transaction.

This scheme allows the polyvalue mechanism to be applied to the execution of uncertain transactions. In this scheme, each update made by the transaction goes through two phases, a lock phase before it is computed, and a wait phase after it has been computed, and the manager holding the updated item has replied to the coordinator.

Another point that should be noted about the use of polyvalues in the locking scheme of Chapter 4 is that the protocols that allow abortable locking described above may require that more lock requests be sent than the simple protocols of Chapter 4. Note, however, that any transaction that does not require locking with the simple protocols still does not require locking, we are only increasing the number of locks sent for transactions that already require locking.

5.5 Restricting the Spread of Polyvalues

The polyvalue mechanism is expensive in that polyvalues consume a great deal more space than do simple values, and a polytransaction may require a great deal more computation. The simple analysis of the polyvalue scheme and a simulation of the protocol reported in an appendix to this thesis demonstrate that in fact the expected number of polyvalues in a distributed information system is quite small. Should further control be necessary, any site can prevent the propagation of polyvalues by not installing polyvalues as results of some pending transaction, and instead waiting until it knows the outcome of the transaction, or by refusing to perform some transaction with polyvalue inputs and instead waiting until it can reduce the polyvalues. These decisions save resources at the expense of possibly delaying important transactions that could have been performed promptly on polyvalues.

In a system with real time response requirements, it is not unreasonable to expect that the set of transactions that must be performed in order to produce needed results at the proper time will be known. It is precisely these transactions that should be performed as polytransactions, so that if possible, the needed results can be obtained despite uncertainty in the database values due to the presence of pending transactions and polyvalues.

Consider a system controlling some manufacturing operation in which several computers are used to control the manufacturing and are physically located near the components that they monitor and control. Several different kinds of transactions act on the data base. There are data entry transactions that are run periodically to enter data about the operation being controlled into the database. There are also monitoring transactions which are run periodically to determine whether or not the database values indicate any potentially dangerous conditions requiring immediate corrective action. The monitoring

transactions are structured so that many examine only values local to some site in order to insure that a communication failure cannot interfere with monitoring.

In addition to these two kinds of transactions, there are control transactions that direct the completion of specific manufacturing tasks. There are also transactions that implement administrative decisions to change the manufacturing process by modifying items representing parameters to the control and monitoring transactions, and transactions that allow the state of the manufacturing process to be examined. The monitoring transactions need to be performed in real time in order to prevent failures in the physical components of the manufacturing process or "bugs" in the control transactions from creating a hazardous situation. These monitoring transactions examine the values produced by the data entry transactions and the parameters of the process to detect problems. Any normal set of parameters and data inputs will not trigger corrective action.

In order to insure that the monitoring transactions function in real time, polyvalues should be used for any data items that might be read by the monitoring transactions. The control effects of the monitoring transactions should be independent of the exact value of the data items describing the process, as long as these data items reflect normal operation. The transactions which direct specific manufacturing tasks and the transactions implementing administrative decisions may involve updates to data items at several sites, and thus may require locking. The locking performed for such transactions should allow the creation of polyvalues for their outputs if some failure prevents the locks from being quickly released.

Transactions representing administrative control of the manufacturing process or control of specific functions may be deemed less important, and may not be executed as polytransactions if necessary. Any process holding items accessed by the monitoring

transactions, however, must be prepared to install polyvalues for those items to insure that the monitoring transactions are not delayed.

5.6 Summary

This chapter has been devoted to a discussion of the "distributed atomic update" problem. It was shown that it is impossible, given the failure semantics of the process model, to construct a protocol which performs a distributed update atomically while not delaying access to the updated items indefinitely at any functioning site. Several strategies were discussed to avoid the distributed atomic update problem in a transaction synchronization scheme.

The remainder of the chapter presented a concept refered to as a polyvalue, which may provide a practical solution to this problem in many cases. Polyvalues allow an update to be performed conditionally, such that both the updated and non-updated values are presented to subsequent transactions. In an information system where the most important effects of transactions depend only loosely on the exact values stored in the data base, the polyvalue scheme allows these important effects to be determined quickly, even when the exact values of items in the data base are uncertain due to transactions that have been started but not yet completed.

This chapter presented some simple examples of the mechanics of manipulating polyvalues and discussed a possible application of polyvalues in a process control system.

Chapter 6

Application of the Techniques to the Design of a Distributed Information System

The past four chapters of this thesis have presented various aspects of an overall approach to the problem of robust synchronization in a distributed information system. In this chapter, I present an example of a distributed information system and show how the techniques that I have developed can be applied to derive a synchronization scheme that satisfies the goals set forth in Chapter 1. This solution is compared with those using other distributed synchronization schemes.

6.1 The Problem

The chosen example is an inventory control system for a chain of supermarkets. The problem is adapted from an example given in [Bernstein77]. The data base is used to keep track of the quantities of various products (cans of beans, paper napkins, etc.) on hand, on order, or in transit at each individual market and at the warehouses that supply the markets. The supply chain of the supermarkets is hierarchical, with groups of markets supplied by local distributers, groups of local distributers supplied by regional distributers, and so forth. The following sections describe the data and the transactions to be performed.

6.1.1 The Data Items

For each location (warehouse or supermarket) the data base contains a set of data items describing each product. These are:

> Quantity on Hand (QOH) -- The quantity of that product stored at that location.
>
> Desired Quantity on Hand (DQOH) -- The goal of how much of the product to try to keep at this location to satisfy demand (purchases by customers or orders from lower distributers or supermarkets).
>
> Re-order Quantity Threshold (RQT) -- A minimum quantity of the product to keep on hand. When QOH falls below RQT, an order is submitted to bring QOH up to DQOH.
>
> Quantity on Order (QOO) -- The amount of the product that has been ordered from the distributer for this location, but has not yet been delivered.
>
> Quantity in Shipping (QIS) -- The amount of the product that has been shipped from the distributer for this location, but has not yet been delivered.

The data items pertaining to each of the products are independently examined and updated (i.e. there is no single transaction that accesses input items pertaining to two or more products), so I will consider only the items pertaining to a single product. In fact, a typical supermarket may stock a total of 10,000 different products, and thus independent sets of these items exist for each of these products.

The five items are maintained for each location, market or warehouse. To distinguish between items describing different locations that are used by the same transaction, I will use subscripts, such as QOH_0 to designate the level of the distribution hierarchy to which an item pertains. Level 0 designates the local markets, while increasing

subscripts designate more global distributers. This is sufficient to distinguish the items because each transaction accesses items pertaining to at most two locations: a location and its supplier.

6.1.2 The Transactions

The transactions in this system serve both to reflect real world effects, such as the unloading of a truck, to the data base, and to determine when some real world action should be performed to keep supplies of all products available. For each product there are four different kinds of transactions: Point of Sale, Re-Order, Shipping, and Receiving.

Point of sale transactions (P transactions) update the quantity on hand to reflect a customer purchase. P transactions take place only on the QOH for the locations corresponding to supermarkets, and not on those for the distributors. For a typical supermarket, there are about 25,000 P transactions per day.

Re-order transactions (O transactions) generate new orders for merchandise which has been depleted. An O transaction examines the QOH, QOO, RQT, and DQOH for some location and produces a new value for QOO. For each location, approximately 2000 O transactions are performed per day to determine which products must be ordered.

Shipping transactions (S transactions) reflect action by a distributer to fill an order. A shipping transaction examines the QOH of the distributer and the QJS and QOO of one of its customers in order to decide how much of the product to ship to that customer. The S transaction updates the QOH of the distributer and the QOO of the customer to reflect the shipping decision. S transactions are performed at the rate of about 15 per day per location.

Receiving transactions (R transactions) record the arrival of shipped goods at a location. Each R transaction adds the amount received to QOH, and subtracts it from QJS and QOO. About 15 R transactions take place for each site each day.

These transactions are summarized in Table 6.1. In the paper which is the source of this example, the authors were unconcerned with the details of how each transaction derives its outputs from its input values. I have therefore made some "educated guesses" in deriving a more complete description of the transactions.

Note in particular that the receiving transactions are presumed to take as a parameter the amount of the product received, and to use that amount to update the items QOO, QJS, and QOH. An R transaction always has independent components, because the new value of each of the items updated depends only on its previous value and on the parameter Q. Another possible interpretation would be to use the value QJS to determine the amount received, thus making the new values of QOO and QOH depend on QJS. I

Table 6.1

Transactions for Inventory Control

Transaction	Description	Frequency
P	$QOH_i := QOH_i\text{-}(Q)$	25,000
O	$QOO_i := O(QOH_i, QOO_i, DQOH_i, RQT_i)$	2,000
S	$QJS_i := S_1(QJS_i, QOO_i, QOH_{i+1})$ $QOH_{i+1} := S_2(QJS_i, QOO_i, QOH_{i+1})$	15
R	$QOH_i := QOH_i\text{+}(Q)$ $QOO_i := QOO_i\text{-}(Q)$ $QJS_i := QJS_i\text{-}(Q)$	15

believe that my interpretation more closely resembles what would happen in a real inventory control system, as the parameter Q represents the amount actually received, and may not correspond to QIS for variety of reasons.

Having a complete description of the data base and the transactions to be performed, we can now proceed to analyze the system using the tools developed in Chapters 4 and 5.

6.2 Analysis of the Transactions

In this section, I present transaction graphs for the transactions to be performed by the inventory control system. These are analyzed to explore the ways in which the transactions interact with each other. This analysis is used to determine the protocols needed to perform the transactions using several different organizations of the data base (choices of which items are held at each site). The choice of the synchronization network for each of these organizations is discussed. Finally, I discuss the use of polyvalues in this distributed information system.

6.2.1 Transaction Graphs for this Application

The transaction graphs for typical transactions from these four classes are shown in Figure 6.1. The P transactions are the simplest, as each P transaction accesses and updates a single data item. P transactions will have independent components in any organization of the data base.

The R transactions are somewhat more complex, as they access and update three different items. As noted in the previous section, however, the new value of each of these items depends only on its previous value. Therefore the transaction graph of an R transaction does not contain arcs interconnecting the three updated items.

The O transactions update a single data item (QOO for some location), but do so based on several inputs. As shown in Figure 6.1, the transaction graph for a O transaction has arcs connecting QOH, RQT, DQOH, and QOO to QOO.

The S transactions are the most complex. Each S transaction updates two items (QJS for some location and QOH for its supplier), based on the previous values of three different items. The transaction graphs for these transactions thus contain cycles of arcs, connecting QJS, QOH, and QOO to both QJS and QOH. These cycles indicate that S transactions are likely to require locking in any organization of the data base.

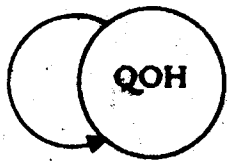
The four kinds of transactions taking place at various levels of the hierarchical organization of the locations interact. This interaction is indicated by Figure 6.2, which shows a joint transaction graph for the transactions taking place at three levels of the hierarchy. The joint transaction graph is constructed from the individual transaction graphs in the same way that a joint activity graph is constructed from individual activity graphs. To distinguish between the transactions taking place at different levels of the distribution hierarchy, each transaction identifier is given a subscript to indicate the level that that transaction pertains to.

The interactions among the transactions suggest that the processing to be performed exhibits a high degree of locality of reference. If the items are grouped so that all of the items pertaining to a single location are maintained by a single manager, the only transactions that require the participation of more than one manager are the S transactions. These transactions represent .05% of the total volume of transactions to be run (though they probably represent a higher proportion of the processing, because they are more complicated than the more frequent transactions).

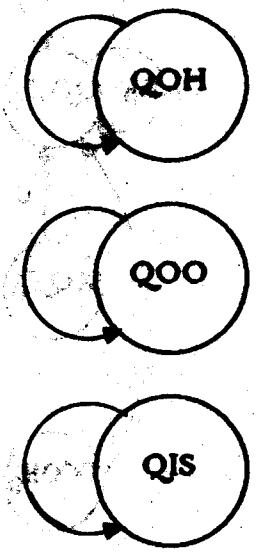
Figure 6.1

Transaction Graphs for Inventory Transactions

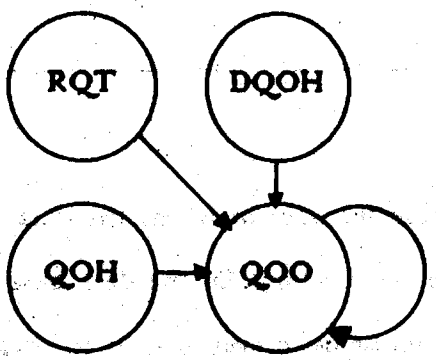
6.1a P Transactions



6.1b R Transactions



6.1c O Transactions



6.1d S Transactions

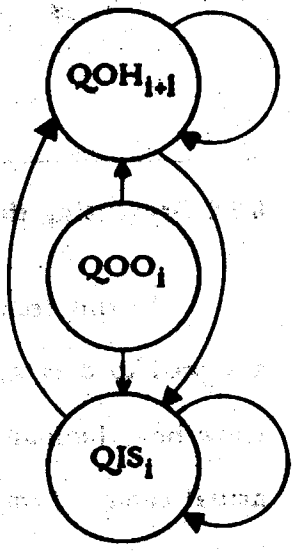
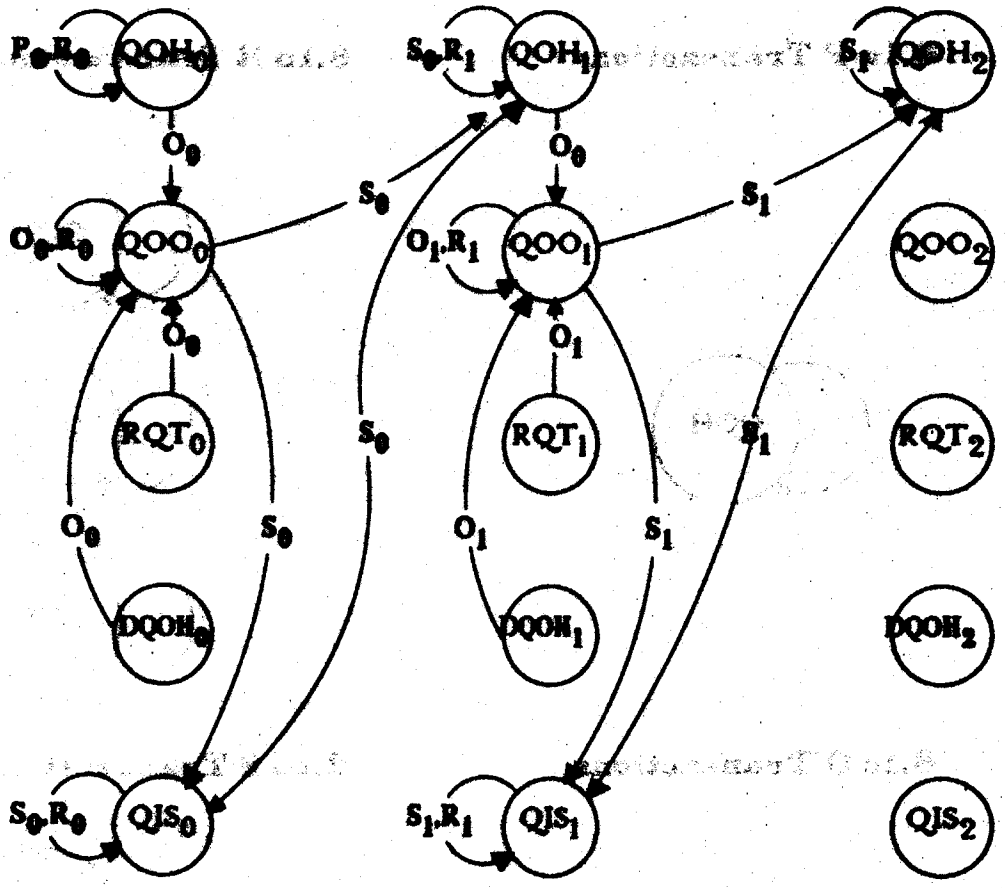


Figure 6.2

A Joint Transaction Graph of The Inventory Transactions



6.2.2 Organizing the Data

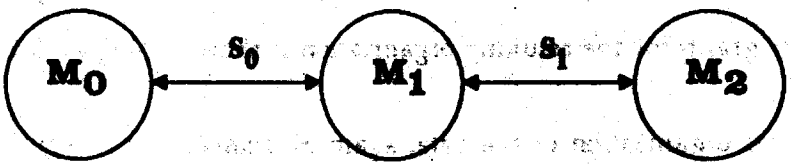
In this section, I consider several different ways in which the data items could be assigned to data managers. Each of these organizations for the data base is discussed to show how the four types of transactions would be performed in such an organization. The actual choice of implementation would be based on the desired level of availability for the

data items as well as the cost of performing the transactions and the processing and storage capacities of the sites holding the data manager processes.

A simple organization for this data base would be to assign all of the items pertaining to one site to one data manager process which executes at that site. A joint activity graph of the four transactions as performed in such an organization is depicted in figure 6.3. The graph shows that in this organization, the only type of transactions requiring communication between data managers are the S transactions. All of the other transactions can be performed by one of the managers alone, because all of the items involved in any of the other transactions are under control of a single data manager.

Figure 6.3

An Activity Graph for a Simple Data Base Organization



Assignment of Items to Managers:

M_0	M_1	M_2
QOH_0	QOH_1	QOH_2
QOO_0	QOO_1	QOO_2
RQT_0	RQT_1	RQT_2
$DQOH_0$	$DQOH_1$	$DQOH_2$
QIS_0	QIS_1	QIS_2

In this organization, most transactions would require no inter-manager synchronization at all, and the rare S transactions would require locking with any synchronization network of the data manager processes (because of the cycle involved in each S transaction). Because none of the items in this organization are replicated, this organization requires the minimum possible storage space.

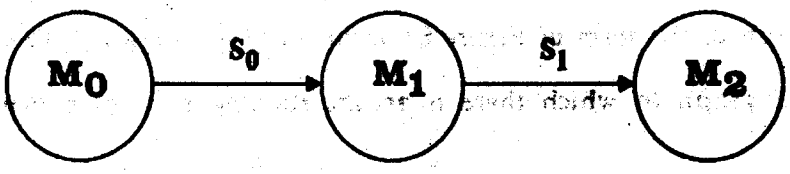
While the S transactions are infrequent, the necessity of locking while performing an S transaction is undesirable. Locking makes the two sites involved in performing an S transaction vulnerable to failures during the execution of the transaction. Many strategies can be used to reduce this vulnerability to an acceptable level, such as using polyvalues (as will be discussed in a later section), or running the S transactions at a time when there is little other activity, such as after the stores have closed. We can avoid the necessity of locking for the S transactions by reorganizing the data base.

The arcs from M_1 to M_0 and from M_2 to M_1 result from the necessity to update the QJS items at one manager based on the QOH items at the higher level manager. We can avoid this dependency, by moving the item QJS_1 from manager M_1 to manager M_{1+1}. A joint activity graph for the resulting organization is shown in Figure 6.4.

In this organization of the data, again all transactions except for the S transactions are again completely local to one of the data managers. The S transactions are performed by two of the managers and require communication. Unlike the previous organization, however, this communication is one-way, such that if a hierarchy of managers is chosen such that M_{i+1} is always a descendant of M_i, then the S transactions can be performed without locking.

Figure 6.4

An Activity Graph for a more efficient Organization of the Data



Assignment of Items to Managers:

M_0:	M_1	M_2
QOH_0	QOH_1	QOH_2
QOO_0	QOO_1	QOO_2
RQT_0	RQT_1	RQT_2
$DQOH_0$	$DQOH_1$	$DQOH_2$
	QJS_0	QJS_1

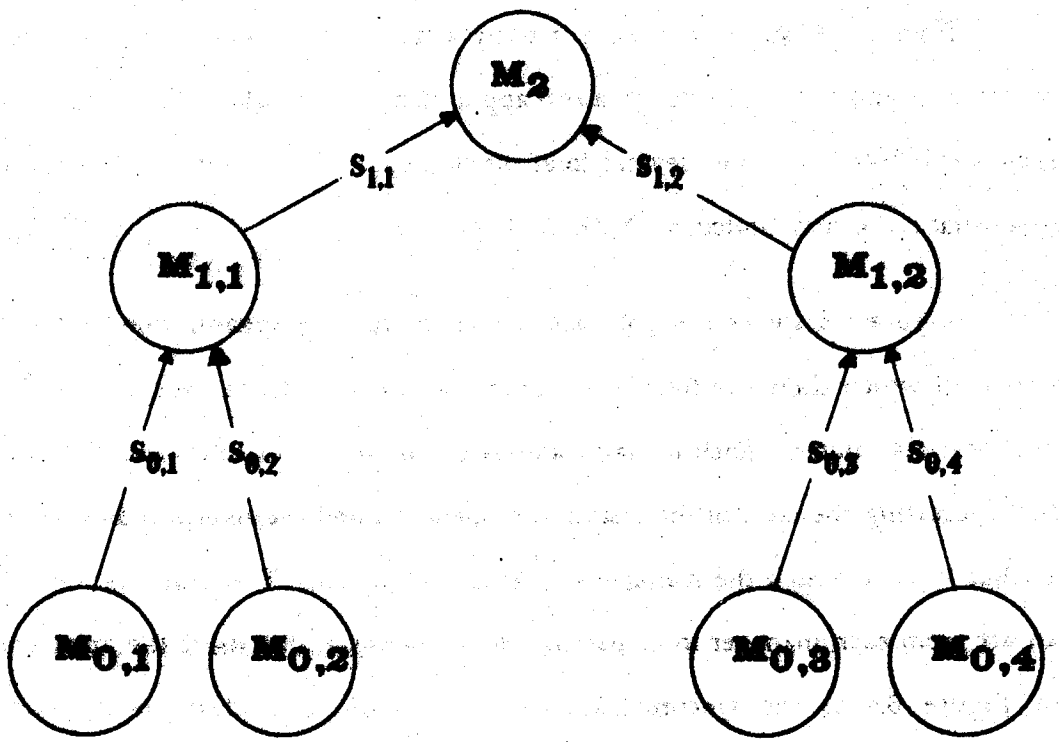
Figure 6.4 shows the joint activity graph for 3 locations in the hierarchy of distributers and supermarkets. In a real application, there would be several supermarkets for each local distributer, and several local distributers. This makes the joint activity graph somewhat more complicated, as shown by Figure 6.5.

Figure 6.5 shows the joint activity graph for this organization of the data, for a system in which there are four supermarkets (Thus four M_0 managers) being supplied by two local distributers. Each manager and each transaction is labeled with two subscripts, the first indicating the level in the distribution hierarchy and the second indicating the location at that level to which the manager or transaction pertain. The graph is hierarchical, with an arc from each manager to its parent. Notice, however, that using the apparent hierarchy in Figure 6.5 as the synchronization network would not allow the transactions to be

performed without locking. Each manager (Except for the M_0 managers) must obtain information from its children in the hierarchy to perform the S transactions. If the arcs in Figure 6.5 were reversed, then the transactions could be performed without locking by using the hierarchy in the joint activity graph as a synchronization network. I will refer to an activity graph of the form of Figure 6.5 as an inverted hierarchy to distinguish it from a hierarchical graph in which there is an arc running from each manager to each of its children.

Figure 6.5

A More Complete Activity Graph



Given this organization of the data base, we must chose a synchronization network that allows the transactions to be performed with the protocols of Chapter 4. While the four classes of transactions described here do not involve any transactions that access a large number of items, presumably in a real inventory control system there would be other transactions much less frequent than those in the four classes which perform functions such as changing the parameters DQOH and RQT, or allowing a user to obtain a snapshot of the quantities of some item in the various locations. In order to provide the ability to synchronize *any* possible transaction on the data, the organization of data managers must be hierarchical.

Any hierarchy of data managers that is consistent with the inverted hierarchy defined by the arcs in the joint activity graph must be some linear ordering of the nodes. The conditions that $M_{2,0}$ be a descendant of all managers, and that *some* process be an ancestor of all managers, and that there can be only one path between any pair of managers force a linear ordering. This is not a very desirable organization for synchronizing the transactions, because the message sent from some manager M_l to M_{l+1} in performing an S transaction may have to be routed through many other managers that do not otherwise participate in that transaction. This makes S transactions expensive and vulnerable to failures, however a failure occuring during an S transaction does not unnecessarily delay other transactions, because there is no locking.

Another alternative is to abandon the ability to perform any arbitrary transaction and restrict the synchronization mechanism to acting on the four classes described above. If we are only interested in performing these four transactions, then the only communication among managers that is needed is that described by the joint activity graph of the four transaction classes. A non-hierarchical synchronization network could be used to coordinate

the transactions. The inverted hierarchy of the joint activity graph meets the requirements for a synchronization network outlined in Chapter 3, (it has only one path between any two processes), and provides all of the necessary communication paths to carry out transactions from the four classes.

Thus the data managers could be logically organized in an inverted hierarchy. All of the transactions except the $ transactions would, as before involve only one data manager. To perform an $ transaction, a message would be sent to the manager M_i holding the item QOO_i. This manager would obtain the value of this item and send it to the manager M_{i+1} which holds QOH_{i+1} and QIS_i. This manager can then update these items to complete the transaction with no locking.

This organization of the managers is clearly most efficient, in terms of the amount of locking and the number of messages sent, but has sacrificed the ability to coordinate other kinds of transactions. One would probably not want to choose an organization of data managers in which it is not possible to synchronize any arbitrary transaction, as this may make it very difficult to implement new kinds of transactions. Still, in a situation in which the transactions and the data base are permanently fixed, such as a distributed rocket guidance system, choosing the logical organization of the data managers without considering the kinds of new transactions that could be desired is not a problem.

6.2.3 Replicated Organizations of the Data Base

The two organizations of the data base described above have a single copy of each data item. In this section, I consider organizations in which some of the data items are replicated. Replication could be desired either for greater robustness (less chance that a

transaction will be delayed due to a site being inaccessible), or in order to eliminate locking by making more of the transactions have independent components.

One could, by replication, make all of the transactions have independent components. This effect could be achieved by making sure that whenever a manager holds a copy of some item I, it also holds copies of all of the items needed by the transactions that update I in order to make that update. For each item I held by a manager, that manager must also hold copies of all items from which arcs in the joint transaction graph point at I. Thus, in effect, each manager holding a copy of I must hold copies of all items that are linked to I by a chain of arcs in the joint transaction graph. The joint transaction graph of Figure 6.2, indicate that a site holding a copy of the items QOH_i or QOO_i must also hold copies of the items QOH_j, QOO_j, RQT_j, $DQOH_j$, and QIS_j for all $j \leq i$, because of the chain of arcs linking these items to QOH_i and QOO_i. This presents an awkward problem, as it means that in order to make all transactions have independent components, a single site must hold copies of *all* of the items and therefore must participate in all of the transactions. It therefore does not seem practical to avoid locking through this approach.

This particular application appears to have little need for replication to increase the availability of data items. The transactions that are most critical to perform quickly are the P transactions and the O transactions. While we could replicate the QOH_0 items in order to increase the availability of these items, there seems to be little point in doing so. Because the P transactions are by far the most frequent, replicating the QOH_0 items would add greatly to the amount of communication and possibly the amount of computation required. A more appropriate approach might be to make the sites which hold the QOH_0 items highly reliable. Another approach that could be used is to use several sites to hold the data items pertaining to each supermarket, partitioning the items so that for each product, there is one

site that holds all of the items that pertain to that product. This approach may allow the individual sites to be smaller, simpler, and more reliable than a single site managing all items for a supermarket.

It would also, presumably, be important that O transactions be executed promptly, to insure that supplies of products remain adequate. In order to make the O transactions less vulnerable to failures, we could replicate the items accessed by the O transactions. Unfortunately, the O transactions at each location access many of the items for that location, including the QOH items. Thus, replicating items to make O transactions more reliable would make many of the transactions more expensive, due to the necessity to update the copies of the QOH items.

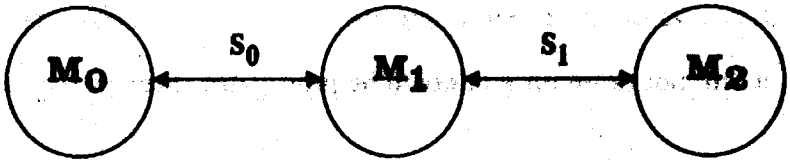
Another organization of the data that might be used is to replicate the QJS_i and QOO_i items so that M_i and M_{i+1} each have copies of both items. Figure 6.6 shows a joint activity graph for this organization. In this organization, M_i and M_{i+1} each have copies of the items pertaining to orders sent from location i to location i+1. This organization does not provide any reliability advantage over the first organization considered in performing the four transactions. Having the QJS and QOO items replicated may, however, allow the human managers in charge of shipping and receiving at the sites to determine the status of orders more easily, even if a failure interrupts communication between locations.

6.2.4 The Use of Polyvaues

Another way of increasing the availability of the data items in the event of a failure is to use the polyvalue mechanism described in Chapter 5. As noted above, the P transactions are the most critical. While locking in all of the above organizations of the data

Figure 6.6

An Activity Graph for a Redundant Data Base Organization



Assignment of Items to Managers:

M_0:	M_1	M_2
QOH_0	QOH_1	QOH_2
QOO_0	QOO_1	QOO_2
RQT_0	RQT_1	RQT_2
$DQOH_0$	$DQOH_1$	$DQOH_2$
QIS_0	QIS_1	QIS_2
	QJS_0	QJS_1
	QOO_0	QOO_0

base is rare, it is possible that a failure during one of the S transactions could delay access to the items used by those transactions. This could in turn delay other transactions.

By using the polyvalue mechanism described in Chapter 5, we can avoid this delay. Two factors suggest that the polyvalue mechanism would be effective in eliminating unnecessary delay of transactions local to one site by failures of other sites. First, many of the transactions depend only loosely on the actual data base values. The O transactions, for example, make a decision of whether or not to order that depends only loosely on the items read. Second, very few of the transactions require locking, thus the probability that a transaction requiring locking will be interrupted by a failure is small.

Notice also that most of the transactions would not propagate polyvalues that have been introduced into the data base. The P transactions and R transactions do not propagate information among items in the data base, while the other two types of transactions may propagate a polyvalue to at most one new item. This means that if a polyvalue is introduced, it will not cause uncertainty to be propagated through the data base.

Whether or not the polyvalue mechanism should be used for this application depends on the actual cost of implementing polyvalues (in terms of the extra checking that must be performed in the course of performing a transaction to handle the possibility of polyvalue inputs), and the concern for reliable operation. The cost of implementing polyvalues is not likely to be high, but the benefits are likely to be small, as so little locking is performed in this implementation as to make it unlikely that a polyvalue will ever be produced.

6.3 Comparison with Other Mechanisms

Several other mechanisms could be used for performing synchronization of the transactions in this example. This section briefly compares some of the other mechanisms that have appeared in the literature with the solution described above.

6.3.1 Comparison with SDD-1

As this example is derived from one used for the SDD-1 system for synchronization of distributed data bases, it seems natural to begin any comparison with SDD-1. This discussion presumes that the reader is basically familiar with the SDD-1 mechanism and the solution to this problem using SDD-1.

Using the analysis and protocols of SDD-1, one concludes that the P transactions and the O transactions can be performed by the simplest (p1) protocol. This protocol requires no locking and in fact closely resembles the protocol used to perform these transactions in the solution described above. The other two transaction classes, however, require the p3 protocol of SDD-1. This protocol performs locking, by forcing the various data managers to perform transactions in time-stamp order. Thus SDD-1 locks for two of the four transaction classes, while my mechanism locks for only one. The reason that two of the transaction classes require locking in SDD-1 is because the analysis techniques used by SDD-1 do not recognize that the R transactions actually have three independent components. While these components must be performed atomically with respect to other transactions, there is no flow of information among the three components, thus they can occur in any order with respect to each other. The more fine-grained analysis used in the mechanism of this thesis discovers this fact, which allows these transactions to be performed without locking.

The locking protocol used by SDD-1 is similar in cost to that used in this thesis, if the SDD-1 mechanism is implemented simply and without regard to failures. Both involve sending a message to each of the data managers which will be involved in the transaction, requesting that the data manager set aside some time to perform the transaction, and then performing the transaction with as many additional messages as are needed to move the data.

The robust implementation of the SDD-1 protocols [Hammer78], however, is very complicated, and may involve many extra messages. It appears to be quite difficult to be sure that the transactions are performed in timestamp order without allowing a single failure to stop all transaction processing.

The robust implementation of the SDD-1 protocols attempts to minimize the probability that a failure will make data inaccessible through the use of abortable locking, and a voting strategy to determine when a transaction passes its commit point. Using these techniques may greatly increase the number of messages that must be sent and the processing needed to implement transactions using the locking protocols. In contrast, using the polyvalue mechanism to increase the availability of data that must be locked does not greatly increase the cost of performing transactions if no failures occur. Only after a polyvalue has been created does this mechanism begin to be more expensive.

In summary, the protocols used in this thesis are likely to be slightly less costly (in terms of processing power, and messages sent) to achieve a comparable level of robustness than the protocols of SDD-1. This does not mean that my mechanism is always less costly than SDD-1, as the cost of both mechanisms depends strongly on the application.

6.3.2 Comparison with Gray's locking strategies

Another distributed concurrency control mechanism is described in a set of notes by Gray [Gray77]. These notes describes a mechanism in which a set of sites, each of which is capable of synchronizing local transactions, can communicate to perform multi-site transactions atomically. This mechanism could be used for this example, by assigning the items in the data base to various sites.

The protocols used by Gray require locking whenever two or more sites are involved in one transaction. Thus the S and R transactions would require locking in this example. The locking mechanisms proposed by Gray are a mechanism for recording the locks held by each transaction at each site, and a deadlock detection mechanism to escape from settings of

locks in which no transaction can proceed. The cost of setting the locks needed to perform a transaction is similar to the locking mechanisms of SDD-1 and this thesis.

The problem of deadlock detection, however, adds to the cost of Gray's scheme. Deadlock detection requires analysis of the sets of locks held by all transactions at all sites, and may be quite costly in a large system. Gray suggests that deadlock detection can be partitioned, so that deadlocks among small groups of sites can be detected more rapidly and with less computation than deadlocks involving a large number of sites. This strategy is likely to work reasonably well in this application, as each transaction involves only a small number of sites. Deadlock detection still represents an additional cost in operating the system, over that of using the protocols of SDD-1 and this thesis which use pre-analysis of the transactions to avoid deadlock situations.

6.4 Summary

This application (the distributed supermarket inventory system) is typical of the kinds of distributed information systems which this thesis addresses. The analysis shows that the transactions exhibit a strong degree of locality of reference, and that most of the transactions can be implemented without locking. The choice of which of the data base organizations and synchronization hierarchies to use for this application depends on the concern for reliability, and the desire to maintain flexibility to perform transactions other than those initially planned. The overhead of synchronization in the organization in which the hierarchy parallels the hierarchical organization of the locations is very small, as very few transactions require locking, and no extra messages are used for the synchronization of transactions which do not require locking.

The polyvalue mechanism described in chapter five can be used in this application to minimize the probability that a failure will delay transactions.

The implementation of this application using the techniques of this thesis was compared with two other distributed data base concurrency control mechanisms. This comparison pointed out that the techniques of this thesis were likely to be less costly than the other mechanisms in achieving a comparable level of robustness.

In conclusion, the synchronization mechanism of this thesis can be seen to be efficient and robust for this application. At the same time, the mechanism allows a great deal of flexibility, letting the system designer trade off the desire for reliability and efficiency against the ability to incorporate unplanned transactions easily.

Chapter 7

Conclusions and Areas for Further Research

This thesis has presented a model of synchronization of transactions in a distributed information system, and several mechanisms for providing such synchronization. This chapter summarizes the important contributions of the thesis to this field, and suggests some areas for further investigation.

7.1 Summary of Thesis Work

The work of this thesis has concentrated in two areas: development of a model of computation in a distributed information system, and development of specific mechanisms for concurrency control in such a system. The major ideas of the thesis in each of these areas are summarized below.

7.1.1 A Model for Distributed Computing

The process model of distributed computing presented in Chapter 2 is a framework in which computation in a distributed information system can be discussed. This model specifies that the effects of site failures or communication failures are lost or delayed messages. The thesis discusses techniques that could be used to provide an implementation of the concepts in the process model for which the effects of failures are confined to these specifications.

I developed two basic strategies for synchronizing transactions described in the process model: locking and sequencing. Sequencing achieves the goal of partial operability defined in Chapter 1, while locking may allow a failure of one site to delay a transaction that is local to some other site. In Chapter 4, I demonstrated that locking was needed to correctly coordinate some groups of transactions. Chapter 5 presented an argument for the claim that in any implementation of locking, a transaction local to one site may be indefinitely delayed by a failure at some other site. Taken together, these results demonstrate that it is impossible to achieve the goal of partial operability while correctly synchronizing all transactions, given the possible effects of failures as specified in the process model.

7.1.2 A Hierarchical Concurrency Control Mechanism

Chapters three and four of this thesis present a hierarchical mechanism to coordinate transactions. This mechanism has several interesting properties. First, it is quite simple to describe, and relatively simple to prove correct. Many of the synchronization mechanisms described in the literature are quite complex, and correctness proofs for these mechanisms are very long and complicated. The simple implementations of the protocols described in Chapters 3 and 4 suggest that my mechanism would be relatively easy to implement in an actual distributed information system.

A second important property of my scheme is that it performs well when the patterns of accesses to items in the distributed data base show a strong locality of reference. The mechanism can be tailored so that frequent transactions require little overhead for synchronization. The mechanism can also be designed so as to avoid locking whenever possible. The thesis describes analysis techniques that can be used to assess the cost of performing the most frequent or important transactions. This analysis can be used to choose an organization of the data and the synchronization network so that these transactions are

performed efficiently and reliably. The mechanism provides correct synchronization for all transactions, even those not anticipated in the design, however unanticipated transactions may be much more costly to perform and more likely to be delayed by failures.

Chapter 5 of the thesis presents a novel solution to the problem of unavoidable delays caused by failures during the execution of a transaction using locking. The polyvalue mechanism in many cases allows a transaction to be run even if the values in the data base accessed by that transaction can not be determined exactly, due to a failure. With this mechanism, important transactions that must be performed promptly are, in many cases, not delayed by the locks set by other transactions. The protocols presented for manipulating polyvalues again are most efficient if most of transactions are local to one or to a small number of sites. This assumption of locality of reference appears to be true of many applications.

The model and mechanisms of this thesis shed some light on what is a very poorly understood area of computer science. They do not by any means provide a complete solution to the problem, and in fact suggest several interesting research problems.

7.2 Areas for Further Research

There are a number of ways in which the work of this thesis could be extended to provide a better understanding of synchronization in distributed systems. These include the investigation of the applicability of the process model to real physical systems, further investigation of applications, better techniques for constructing the synchronization hierarchy, and implementation of the protocols.

7.2.1 The Applicability of the Process Model

The results of this thesis are based on the semantics of failures in the process model. In particular, many of the results are based on the notion that there is no single event detectable by two processes simultaneously. While I strongly believe that this is true of any physical system, there may in fact exist ways of implementing communication in which the sender of a message can know for sure whether or not that message was received. If this is the case, one might be able to implement abortable locking as described in Chapter 5, contrary to the arguments advanced in that chapter and in several other papers in the literature.

Another related area for investigation is that of ways of including the effects of failures in a model of computation. In the process model, I assumed that a failure could delay any message indefinitely. It is possible that some less pessimistic assumption about failures would lead to a workable model for a distributed information system. One might, for example, assume that no more than N sites fail concurrently. While it would be impossible to implement a system so as to conform to this assumption, if the probability that the assumption is violated is sufficiently small, then a distributed information system based on the assumption may be acceptibly reliable, and may be simpler to implement.

7.2.2 Applications

This thesis makes extensive use of the assumption that applications of a distributed information system will exhibit locality of reference in their use of data. This assumption appears to be true of some planned applications, however more careful statistics of actual applications may be needed to confirm the validity of this assumption. We may in fact

discover that the flexibility of a distributed information system will encourage different organizations of information that do not exhibit the same patterns.

7.2.3 Analysis of Transactions

The thesis presented techniques for determining the cost of performing a transaction (in terms of the number of messages required) using various logical organizations of the data. Guidelines for choosing the synchronization network and data base organization, given a description of the most frequent transactions to be performed, were given. These guidelines are not, however, detailed algorithms that design the synchronization mechanism. Considerable effort and ingenuity may be needed in choosing an optimal, or near optimal synchronization network, and in choosing the assignment of data items to data manager processes. These problems are similar to many others that occur in managing resources in a computer system, and it would seem likely that good algorithms for designing a distributed information system using the hierarchical synchronization mechanism of this thesis could be derived.

7.2.4 Implementation of the Protocols

Finally, the thesis presented only a few simple implementations of the synchronization protocols. Improvements on these implementations can no doubt be made. One area that seems of particular interest is using the hierarchical synchronization protocols in a computer tailored to managing data. The hierarchical synchronization mechanism presented here fits well with the proposed models of memory for a data base machine. This mechanism may lead to a very efficient implementation of such a machine.

Another implementation issue that bears further investigation is the design of a communication network that supports atomic broadcasting. In Chapter 3, techniques for using a broadcast network to implement the message forwarder protocol were presented. The need for a coordinator site is a weak point in this scheme, as failure of the coordinator site stops all atomic broadcasting. It is possible that the function of the coordinator could be implemented in each site's network interface, in such a way that a single site failure would not stop broadcasting. If the network were designed specifically to support atomic broadcasting, it is likely that broadcasting could be made efficient and highly reliable.

Several mechanisms have been developed to maintain several identical copies of a redundant database. As noted in Chapter 4, this problem appears to be somewhat simpler than that of synchronizing transactions in a distributed information system, because each site has a copy of every item. The techniques that have been developed to manage replicated data could be applied to maintaining copies of the process state and message queues of a process, in order to obtain a more robust implementation of a process. Application of techniques for maintaining duplicate data bases to the implementation of processes would be an interesting research problem, and would lead to a more robust implementation of the concurrency control mechanism presented in this thesis.

7.3 Summary

This chapter has presented a summary of the results of this thesis, and presented some of the open questions that this thesis leaves unanswered. Many of the conclusions of this work are not decisive, however I hope that my work has shed some small degree of light on a very murky and poorly understood field.

References

[Akkoyunlu75] Akkoyunlu, E.S., Ekanadham, K., Huber, R.V., "Some Constraints and Tradeoffs in the Design of Network Communications", *Proc. Fifth Symposium on Operating Systems Principles*, November, 1975. (Operating Systems Review, Vol. 9, No. 5).

[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.

[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.

[Cerf74] Cerf, V.G. and Kahn, R.E., "A protocol for Packet Network Interconnection," *IEEE Transactions on Computers*, May, 1974.

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

[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.

[Dijkstra68] Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System," *CACM 11*, 5 pp. 341-346 (May 1968).

[Farber72] Farber, D.J., Larson, K., "The Structure of a Distributed Computer System - Communications", Proceedings of the Symposium on Computer-Communications Networks and Teletraffic, Microwave Research Institute of Polytechnic Institute of Brooklyn, 1972.

[Gray75] Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L., "Granularity of Locks and Degrees of Consistency in a Shared Data Base", IBM Research Report RJ 1654, September, 1975

[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.

[Halstead78] Halstead, R.H., Multiple Processor Implementations of Message-Passing Systems. S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science. January, 1978.

[Hammer78] Hammer, M., Reliability Mechanisms in SDD-1, talk at MIT-LCS. Soon to be written up.

[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.

[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*.

[Liskov77] Liskov, B.H., et al., "Abstraction Mechanisms in CLU," *CACM 20*, 8 (August 1977), pp. 564-576.

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

[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.

[Reed76] Reed, D.P., "Protocols for the LCS Network," LCS-LNN •3, November, 1976.

[Reed78] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Technical Report 205, September, 1978.

[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.

[Stearns76] Stearns, R., et al., "Concurrency control for database systems," *IEEE Symposium of Foundations of Computer Science CH1133-8 C*, October, 1976, pp. 19-32.

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

Appendix A

Proofs of the Protocols

This appendix gives a more formal definition of some of the concepts in the body of this thesis and proofs of some of the results. For simplicity of description, the definitions and proofs in this appendix are for the version of the message forwarder protocol in which each broadcast message is sent immediately to a process which is a common ancestor of all of the receivers and of the sender. In the actual implementation of the protocol described in Chapter 3, each message may travel up the hierarchy in several hops. This difference does not effect the results proven here, so long as only the message receptions which take place in the distribution of a broadcast after that broadcast has reached the common ancestor are used in determining the < ordering. This condition is consistent with the use made of the protocol by the concurrency control mechanism presented in Chapter 4, in which the process steps that take place in the distribution of a message after it has reached the common ancestor are the only ones which have effects that could be observed by process steps related to other messages.

A.1 Formalization of Atomic Broadcasting

Definition: For each process p, there is an ordering $<_p$ on messages sent to p such that $m_1 <_p m_2$ iff m_1 was received at p *before* m_2 was received at p. Each message sent to p is included in $<_p$ when it is received.

Definition: A **broadcast** $B = \{[b_i, p_i] \mid b_i$ is a message which is to be sent to process p_i as a part of $B\}$

Definition: For each message m, let B(m) be the broadcast message from which m was derived. The set $\{m \mid B(m) = B\}$ for some particular broadcast B

then contains both the component messages of B and any other messages that are received in distributing those components.

Definition: For broadcast messages B_1 and B_2, there is an ordering $<$, which is defined as $B_1 < B_2$ if $\exists p, b_1, b_2$ such that $B(b_1) = B_1$, and $B(b_2) = B_2$, and $b_1 <_p b_2$.

Definition: Broadcasting is atomic iff $<$ is cycle free.

Definition: Let $\rightsquigarrow$ be the Synchronization network relationship, which is a relationship among pairs of processes and must satisfy the following constraint. The graph defined by $\rightsquigarrow$ must have no directed or undirected cycles. Thus, there does not exist a set of three or more processes $p_1, \ldots p_n$, all distinct, such that either $p_i \rightsquigarrow p_{i+1}$ or $p_{i+1} \rightsquigarrow p_i$ for all $i<n$, and p_1 and p_n are related by $\rightsquigarrow$.

With these definitions, we can now define the message forwarder protocol by defining the process step specifications of the forwarders.

Definition: The process step specification of a message forwarder f is defined by a function F(B):

$$F(B) = \{[(X(B,p),p] \mid (f \rightsquigarrow p) \wedge \text{The set } X(B,p) \text{ is non empty}\}$$

where B is the message received in a process step, F(B) is the set of pairs, each of which lists one of the output messages produced by that step and its destination process, and $X(B,p)$ is a set describing the contents of one of the output messages of the process step, which is constructed as follows:

$$X(B,p) = \{[b,q] \mid [b,q] \in B \wedge p \rightsquigarrow^* q\}$$

Definition: Communication between message forwarders obeys the constraint of Sequencing, which can be stated as follows. If $b_1 <_f b_2$ for messages b_1 and b_2 and message forwarder f, and if $[b'_1,p] \in F(b_1)$, and $[b'_2, p] \in F(b_2)$, where F is the protocol specification for f, then $b'_1 <_p B'_2$ after both b'_1 and b'_2 have been received by p.

A.2 Proof of Atomic Broadcasting

To show that the protocol defined above distributes the components of a broadcast message atomically, we must show that the < ordering in any system state reachable by an execution of steps specified by the protocol is cycle free. To do so, I will show that for any step permitted by the protocol, if the < ordering is cycle free before that step, then it will be so after also. Because the starting state (before any messages have been received) has an empty < ordering, which is trivially cycle free, by induction in any state reached by following the protocol the < ordering will be cycle free.

Before proceeding with the proof, I would like to make an observation about the protocol that will be useful in the proof. If two processes p and q have each received a message derived from the same broadcast B, then there is a path in the graph defined by the $\sim$ relationship connecting p and q. Each process on that path has also received a message derived from B. This is true because the graph has no cycles, thus there is only one path between p and q, and the protocol allows each broadcast to enter the network at one point, from which it must reach all recipients. It is not possible for two processes to have seen messages from a broadcast B unless all of the processes on the path between those two processes have also seen messages from B.

In each step of the protocol, some message m is received at some process p, possibly adding ordering relationships of the form b(m') < b(m) for messages m' previously received at p. We must show that introducing the relationship b(m') < b(m) for any message m' previously received at p can not introduce a cycle. The proof will be divided into 3 cases, depending on the origin of m and m'.

CASE 1: m was not sent from a process P such that $P \sim p$. In this case, m is the initial

entrance of broadcast message b(m) into the network of message forwarders. Therefore, before the reception of m, there were no ordering relationships in < involving b(m), so that the reception of m could not introduce a cycle in <.

CASE 2: m and m' were both sent by some process P such that $P \sim p$. In this case, the process P must have received a message M such that B(M) = B(m), and a message M' such that B(M') = B(m') in the process steps which produced m and m'. Because of the sequencing of messages between P and p, the messages m and m' must have been sent by P in the same order that they were received at p. Thus the ordering relationship b(m') < b(m) held before the reception of m (because of the receptions of M and M' at P) and therefore, by the assumption that no cycle existed before the reception of m, no cycle is created.

CASE 3: m was sent by some process P for which $P \sim p$, but m' was not sent by P. This is the most difficult case. To show that no cycle is introduced by the reception of m in this case, I will assume that such a cycle is created and show that this assumption leads to a contradiction or a violation of the conditions of the protocol.

Assume that the reception of m creates a cycle in the < ordering. Then prior to the reception of m, it must be the case that there is a sequence of broadcast messages $<B_1, \ldots B_n>$, such that $B_i < B_{i+1}$ for $1 \leq i < n.$, and $B_1 = B(m)$, and $B_n = B(m')$. Consider now the set of processes $p_1, \ldots, p_{n-1}$ at which these broadcasts were ordered. Now by the observation noted above, there exists a path in the network from each of these processes to the next process in the chain. Also, there exists a path between p_1 and P, as both have received messages derived from b(m), and there exists a path between p_{n-1} and p, as both have received messages derived from b(m'). Thus because of this chain of broadcasts, there must exist a path between P and p. If the path implied by the chain of broadcasts does not go through the direct link between P and p, then we have discovered a cycle in the

synchronization network, violating the conditions of the protocol. I will show that if that path does go through the direct link between the two processes, then either the sequencing of messages between P and ϕ has been violated, or a cycle existed in the $<$ relationship before the reception of m at ϕ.

If the path between P and ϕ implied by the sequence of broadcasts includes the direct link, then some broadcast, call it B_j, must have been seen by both P, and ϕ, and furthermore, we know that P must have received a message derived from B_j, and as a result sent a message to ϕ. The broadcasts B_j and $B(m)$ must have been ordered by message receptions at P, and $B(m) < B_j$, as otherwise there would be a cycle in the chain of broadcasts. Now by sequencing, the reception of m at ϕ must precede any reception of the message derived from B_j, which is impossible, as we know that a message derived from B_j must have been received at ϕ. This contradiction demonstrates that it is impossible for a cycle to arise from the reception of m at ϕ if the only path between P and ϕ is the direct link. Thus another distinct path must exist in the synchronization network between P and ϕ, forming a cycle with the direct link.

This completes the proof of the third and last case, thus the synchronization protocol of Chapter 3 correctly coordinates atomic broadcasts.

A.3 Correct Relative Sequencing of Broadcasts

In this section, I demonstrate that the protocol described in Chapter 3 for atomic broadcasting correctly orders atomic broadcasts such that a process never receives some message m before receiving some message that m "should follow". The proof will be for the simplified case in which the $\rightsquigarrow$ relationship is a simple hierarchy.

Recall that the "should follow" relationship among messages was defined as: Each message m sent by a process p in a process step s should follow a message m' whenever:

> a) There is a message m" received by p in process step s or in a step that preceded s, and m' and m" are components of the same broadcast.
>
> OR
>
> b) There is a message m" received by p in step s or in a step that preceded s, and m" should follow m'.

A key factor in this definition is that if m should follow m', then some process must have received a message derived from b(m'). Using the message forwarder protocol of Chapter 3, if any process has received a message derived from a broadcast message B, then for any process p, if p will eventually receive a message derived from B, then that message must be represented in some message awaiting reception at p or at one of the ancestors of p. This is true because each broadcast message enters the hierarchy once, and all components flow downward in the hierarchy from the point of entry. No component can be received before the message is entered in the hierarchy, and once a message is entered, each component is either above or at its ultimate destination.

I will now prove the claim that for any message m, there can be no message m' such that m should follow m', and the message containing the component containing m' is above that for m in the hierarchy. This, combined with the observation about the message forwarder protocol described in the previous paragraph, is sufficient to prove that when a message m is received at a process p, no message m' that should follow m will subsequently be received at p.

The proof of this claim will be by induction. Initially, the claim is true, as there are no messages. We must show that in any state for which the claim is true, the reception of a message m at a process p as specified by the protocol cannot cause the claim to become false. There are two cases: one where m was sent by the parent of p, and one where m was sent by some other process.

CASE 1: m was sent by some process P such that $P \sim p$. When m was sent, all of the messages that m should follow must have been in the hierarchy (or already received) and not above P in the hierarchy. Therefore, because of the sequencing of messages between p and P, messages that m should follow will not be above p in the hierarchy when m is received at p.

CASE 2: m was not sent by the parent of p. In this case, we must consider the messages that m should follow. These are all components of each broadcast message B for which s, the sender of m, had received a component prior to the sending of m. The claim was true when m was sent, so no message that should follow any of these broadcasts could have been above s at the time that m was sent. Therefore, because p must be an ancestor of s, there are no messages that should follow m that are above p when m is received at p.

This completes the proof of the claim, and thus the proof that broadcasts are sequenced correctly by the message forwarder protocol using a hierarchical synchronization network.

While the proof of correct sequencing of messages according to the "should follow" relationship is somewhat involved, the principal of operation of the protocol is simple. Each message pushes the messages that it should follow along paths in the hierarchy as it goes. The protocol works because there is only one path between any two processes in the

hierarchy, so that no message can sneak ahead of its place in the sequence of messages going to some destination process.

Appendix B

An Analysis of the Propogation of Polyvalues

A major area of concern regarding the polyvalue scheme presented in Chapter 5 of this thesis is that failures may cause the number of items having polyvalues to become large. This would waste storage space and cause a great deal of extra computation by the polytransactions acting on the data base. This appendix presents a simple model of the dynamic behavior of a distributed information system using the polyvalue scheme. An analysis is given to show that with reasonable parameters for the expected transactions and failure rates, the number of polyvalues in the data base remains quite small. A simulation of the system agrees well with these predicted results.

B.1 A Model for the Creation and Deletion of Polyvalues

At any point in the execution of a distributed information system, we can calculate the expected rates of creation and deletion of polyvalues, based on some assumptions about the expected transactions, and the failure characteristics of the system. These rates can be expressed as:

Creation Rate = Propagation Rate + New Failure Rate

Deletion Rate = Recovery rate + Propagation Overwrite Rate

Propagation rate is the rate at which polytransactions install polyvalues for their results in items which previously held simple values. New failure rate is the rate at which updates in progress are suspended, causing polyvalues to be installed. Recovery rate is the rate at

which failures which caused polyvalues to be produced are recovered. Finally, propagation overwrite rate is the (probably very low) rate at which an item with a polyvalue is updated by a transaction producing a simple value. This occurs only if a transaction produces an output that is independent of the previous value of the updated item.

With some additional terminology, we can develop more precise expressions for the creation and deletion rates. I will use the following terminology to describe the data base, the transactions, and the failure characteristics of the system:

> U - Update frequency (Updates/Second). This is the rate at which *updates* to the data base (not transactions) are made. U can be calculated from the overall transaction rate, the percentage of transactions which make updates, and the average number of updates per transaction.
>
> W - The probability of an update being delayed by failure. W can be computed from the mean time between failures, the time window in which an update can be delayed by a failure, and the update rate.
>
> I - The number of items in the data base.
>
> R - The recovery rate for failures. This is the reciprocal of the mean time to recover failures (in seconds). The description of failure recovery in this way assumes that the mean time to recover failures is exponentially distributed with mean of 1/r.
>
> Y - Update independence. This parameter is the probability that the new value of an updated item will not depend on its exact previous value. A value of 0 for Y indicates that the new value of an updated item always depends on its previous value.
>
> D - Dependence of updated items on other data items. This parameter specifies on the average the number of data items in the data base on which each update depends.

With these parameters, we can approximate the rates described above. In the expressions given below for the rates, P represents the number of polyvalues in the data base. This is a first order approximation in which the proportion of data items in the data base having polyvalues is assumed to be small, thus terms involving $(P/I)^2$ have been dropped.

Propagation rate = U * D * P / I

New Failure Rate = U * W

Recovery Rate = P * R

Propagation Overwrite rate = U * P * Y / I

These terms can be combined to give the expected rate of change of the number of polyvalues in the data base, yielding:

$$\frac{dP}{dt} = U*W + U*D*P/I - U*P*Y/I - P*R$$

This is a simple linear differential equation for P which indicates that the number of polyvalues would follow an exponential decay from its initial value to the steady state value, given that the parameters accurately describe the behavior of the system. The steady state expected number of polyvales can be obtained by setting the rate of change equal to zero and solving for P. From this we obtain:

$$P = \frac{U*W*I}{I*R+U*Y-U*D}$$

Several non-intuitive properties of this solution should be explained. First, it would seem that the denominator of this expression can go to zero, causing the steady state expected number of polyvalues to be infinite or negative. This situation arises when the propagation rate is equal to or greater than the rate at which polyvalues are removed through failure

recovery or overwritten. If this were the case, we would indeed expect the number of polyvalues in the data base to become large. In fact, the number of polyvalues in the data base would grow so large that this simple first order analysis would no longer be correct, and the number of polyvalues would be limited by second order effects which I have ignored. (I have, for example, ignored the possibility that an item involved in a failed update or the target of propagation already has a polyvalue, and thus does not represent a new polyvalue.)

A second feature of the equation which may seem strange is that it depends in a non-trivial way on I, the number of items in the data base. This is because the creation and deletion of polyvalues directly due to failures is not dependent on the data base size, while the propagation terms depend on the ratio P/I. If I is very large compared to P, then the effect of propagation is small, as the chance that items with polyvalues will be used by transactions is small. If, however, the data base is small, then the chance that items with polyvalues will be involved in transactions is larger, and the propagation terms become more significant.

Another point to notice about these equations is that they are stable, meaning that if the current number of polyvalues is larger than the expected number, the expected change in the number of polyvalues is a decrease. This indicates that if some catastrophe introduces a large number of polyvalues into the data base, the number should soon decrease to the expected number, given that the values of W and R are not effected by the catastrophe.

Table B.1 gives some typical values for P. Several observations can be made about this data. Decreasing R causes an increase in the number of polyvalues, as would be expected. Increasing W causes a proportional increase in the number of polyvalues. Decreasing I causes the number of polyvalues to rise. The parameters Y and D have little effect, unless the values of the parameters are such that the denominator of the equation for

P is near zero.

Notice that even for reasonably pessimistic failure rates and recovery times, the number of polyvalues remains quite small. These results indicate that the polyvalue scheme is feasible in a distributed information system, and that the chances of a combinatorial explosion of the amount of computation are very small. The next section of this appendix contains a brief description of a simulation of the use of polyvalues, which shows that the predictions of this model are reasonably accurate.

Table B.1

Typical Predictions of the Number of Polyvalues in a Database

Parameters						prediction
U	W	I	R	Y	D	P
1	0.0001	1,000,000	0.001	0	1	0.10
1	0.0001	1,000,000	0.001	0	100	0.11
10	0.0001	1,000,000	0.001	0	1	1.01
100	0.0001	1,000,000	0.001	0	1	11.11
10	0.0001	100,000	0.001	0	1	1.11
10	0.0001	100,000	0.001	0	5	2.00
10	0.0001	100,000	0.001	0	7	3.33
10	0.0001	100,000	0.001	1	1	1.00
10	0.0001	20,000	0.001	0	1	2.00
10	0.0001	11,000	0.001	0	1	11.00
10	0.001	1,000,000	0.001	0	1	10.10
10	0.005	1,000,000	0.001	0	1	50.50
10	0.0001	1,000,000	0.0001	0	1	11.00

B.2 Simulation of the Use of Polyvalues

In order to verify that the approximations made in analyzing the above model do not lead to an inaccurate description of the behavior of the polyvalue system, I constructed a simulation of the manipulation of polyvalues in a distributed information system which is based on the above model, but not the approximations made in the analysis.

The simulation assigns unique identifiers to each failure creating a polyvalue, in order to distinguish them. For each item in the data base, the simulation maintains a vector containing the identifiers of the pending transactions on which that item depends, refered to as the state of the item. An item has a polyvalue if its state is non-empty (i.e. if that item depends on a pending transaction).

Updates are simulated at the rate U. Each such update selects a random integer d with mean D, and d random items from the data base. Some random item is selected as the target of the update. The state of the updated item is replaced with a merge of the states of the selected d items. With probability (1-Y), the previous state of the updated item is also merged into its new state.

With a probability W, the update is chosen to fail. A failure is simulated by selecting a new identifier, adding it to the state of the updated item, and selecting a recovery time for the failure. Recovery times are exponentially distributed, with mean 1/R. When the recovery time for a failure is reached, the identifier of that failure is removed from all item states.

The limits of the simulating program prevent the simulation of very large data bases, or very high update rates. However, for the parameters that can easily be simulated, the simulation agrees well with the predictions of the model. Table B.2 contains the results of

the simulation for some sample parameter settings. The numbers of polyvalues obtained through the simulation were in general somewhat smaller than those predicted by the analysis. This difference is primarily due to the fact that the rate at which polyvalues are created is smaller than that predicted by the model, because the target of a failed update or the target of a propagation may already have a polyvalue.

In conclusion, these results show that the polyvalue scheme is feasible for preventing delay due to locking, provided that reasonable measures are taken to minimize the number of failures that introduce polyvalues.

Table B.8

Results of the Simulation of Polyvalues

parameters						predicted	actual
U	W	I	R	Y	D	P	P
2	0.01	10,000	0.01	0	1	2.04	2.00
5	0.01	10,000	0.01	0	1	5.26	2.71
10	0.01	10,000	0.01	0	1	11.11	9.5
10	0.001	10,000	0.01	0	1	1.11	0.74
10	0.01	10,000	0.01	0	5	20	19.8
10	0.01	10,000	0.01	1	5	16.7	15.8

Biographical Note

Warren Montgomery was born on March 29, 1951 in Highland Park, Illinois. He grew up in Deerfield Illinois where he graduated from Deerfield high School in 1969. He received the Rensellaer Science Medal and placed 35th nationally in the M.A.A. high school mathematics competition.

From 1969 to 1973, Warren attended Dartmouth College in Hanover New Hampshire. At Dartmouth, he worked part time on the operating system for the Dartmouth Time-Sharing System. He received the John G. Kemmeny prize in computing for his contributions to this system. He graduated summa cum laude, with a double major in Mathematics and in Engineering. His thesis was entitled "Non-Adiabatic Behavior of Charged Particles in a Magnetic Null Sheet".

From 1973 through 1978 Mr. Montgomery has been a graduate student at the Massachusetts Institute of Technology. He was supported for the first three years as a National Science Foundation Fellow. In 1976, he was awarded simultaneous degrees of Master of Science and Electrical Engineer. His S.M. and E.E. thesis was entitled "A Secure and Flexible Model of Process Initiation for a Computer Utility".

From 1976 to September of 1979 Mr. Montgomery was supported as a research assistant in the Computer Systems Research Group of the M.I.T. Laboratory for Computer Science. In the summer of 1977, he also worked for Prime Computer Company, designing a communication protocol for a high-speed inter-processor communication network.

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

Mr Montgomery is currently working as a member of the technical staff of Bell Telephone Laboratories, investigating the design of software to support new home communication services. He is married to the former Carla P. Westlund.