CRASH RECOVERY IN

A DISTRIBUTED DATABASE SYSTEM

by

M. D. Skeen

Memorandum No. UCB/ERL M82/45

19 May 1982

Crash Recovery in a Distributed Database System

by

Marion Dale Skeen

CRASH RECOVERY IN A DISTRIBUTED DATABASE SYSTEM

by

Marion Dale Skeen

Memorandum No. UCB/ERL M82/45

19 May 1982

ELECTRONICS RESEARCH LABORATORY

Abstract

Consistency in a distributed database system is based upon the notion of a transaction, a distributed atomic action. This dissertation investigates commit protocols for preserving transaction atomicity (and hence consistency) in the presence of failures. Herein, we

(1) Introduce a formal framework for reasoning about the crash recovery problem.

(2) Show fundamental limitations on the fault-tolerance of commit protocols.

(3) Derive sufficient, and in many cases necessary, properties for a protocol to provide maximum fault-tolerance to various classes of failures.

(4) From the above properties, derive families of fault-tolerant protocols.

Two failure classes are studied in detail: site failures and network partitioning.

In designing a commit protocol, the primary and overriding objective is to guarantee atomicity; the secondary objective is to maximize availability of the database. Since availability is limited if pending transactions must block (suspend execution) on failures, our focus is on nonblocking protocols.

The formal model introduced is based on nondeterministic finite state automata with failures viewed as a distinguished type of state transition. The model is used both in determining bounds on fault tolerance and in specifying and verifying the protocols summarized below.

Concerning site failures, we prove the Nonblocking Theorem, yielding necessary and sufficient conditions for a commit protocol to be nonblocking. From this result, we derive a family of protocols, the *three-phase protocols*, that never require an operational site to block on failures by other sites, even if the transaction coordinator fails. Centralized, hierarchical, ring, and decentralized three-phase protocols are illustrated. These protocols are compared with existing commit protocols.

Concerning site recovery, we conclusively prove the nonexistence of nonblocking site recovery protocols. We then study the (potentially catastrophic) special case when all sites inopportunely fail and derive sufficient conditions for safe recovery.

Concerning network partitioning, we again prove nonexistence of nonblocking solutions and then proceed to derive a family of protocols tunable toward maximizing the expected number of nonblocking sites. These protocols are extremely resilient -- resilient when the cause of the failure or even its presence is in doubt.

Michael Stonebraker
Chairman of Dissertation Committee

Acknowledgements

Many individuals have made this thesis a reality.

I would like to thank first and foremost Mike Stonebraker, who introduced me to the area of distributed databases and provided support and direction. I would also like to thank the other members of my reading committee, Larry Rowe and Richard Barlowe.

I would like to thank all of the members of Project INGRES for providing a rich, stimulating, and enjoyable research environment. In particular, I would like to thank Gene Wong and Corcky McCord for many meaningful discussions on subjects varying from query processing to Bangkok.

I am greatly indebted to my friends at Berkeley for providing encouragement and the necessary diversions for maintaining ones perspective. The entire list is too long to include here, but I would like to mention Ken B., Ken K., Norma, and Richard.

Finally, special thanks goes to Tandy, a dear friend without whose help this thesis would never have materialized.

Research sponsored by NESC grant N00039-81-C-0569 and AFOSR grant 78-3596.

Table of Contents

CHAPTER 1

Introduction

Recently considerable research interest has been focused on distributed database systems. The advantages of distributed systems over centralized systems include: increased performance through the exploitation of parallelism, increased cost effectiveness through the exploitation of cheap hardware, the ability to incrementally expand the system, and an increase in fault tolerance and a corresponding increase in availability. However, distributed systems are much harder -- often by an order of magnitude -- to design and implement than their centralized counterparts.

The usefulness of a distributed system is dependent on its ability to maintain a consistent and accessible copy of the database. This is accomplished by providing the abstraction of an *atomic action* which can span several sites [LOME77]. In database terminology, these actions are called *distributed transactions* (henceforth, simply referred to as *transactions*) [ESWA76, GRAY79].

Consider a submitted transaction. If the transaction executes to completion and its effects are installed in the database, then it is said to be *committed*. On the other hand, if it can not execute to completion, then all of its effects are erased from the database and it is said to be *aborted*. Although these are not the only possible outcomes of a transaction, these are the only correct outcomes.

A transaction can be arbitrarily complex. The canonical example of a transaction is a funds transfer from account A at Site 1 to account B at Site 2. This is the simplest type of transaction requiring a single operation at each of two sites. Very complex transactions can also arise in banking applications. At the end of each day, the cash flow for each branch and between branches must be calculated, and tables showing the banks liabilities and assets must be updated. The complete computation of liabilities and assets can be viewed as a single transaction.

The purpose of a *transaction management system* is to provide an implementation of transactions where they appear as atomic operations to the user. The only way to access or modify the database is by using a transaction. The correctness criteria for a transaction management system is that the current database state must be derivable from the initial state by the serial execution of a sequence of transactions [ESWA76]. A database is said to be in a *consistent* state whenever the above criteria holds; otherwise, it is in an *inconsistent* state.

Inconsistency can result from two sources:

(1) concurrently executing transactions read and write the same data items, or

(2) failures can result in a transaction being partially (or incorrectly) executed.

The first problem is a generalization of the readers/writers problem. It has received much attention in the literature where protocols for solving the problem appear under the auspices of "concurrency control." (The interested reader is referred to surveys by Kohler [KOHL81] and Berstein and Goodman [BERN81]. Over two dozen papers appear on the subject [KOHL81].)

The second problem is the subject of this thesis. Herein, we systematically examine the classes of failures that may occur, such as site failures and network failures, and identify the problems that each class introduces. Then for each problem area defined, we examine feasible solutions, and where possible, give necessary and sufficient properties for preserving consistency. We will then apply these results to derive families of resilient protocols -- protocols that ensure consistency for a given class of failures.

A vital element of this approach, but by no means the major result in itself, is the formal model presented in Chapter 3. It is used both in the specification and verification of protocols and in the derivation of the necessary and sufficient properties of resilient protocols.

Whenever possible, we will restrict our attention to a single executing transaction and examine the problem of preserving its atomicity in the presence of failures. Protocols for directing the execution of a single transaction are called commit protocols.

Chapter 1 is divided into four sections. The next section discusses the of failures that are expected to occur in a distributed database system. The third section briefly states the goals and the assumptions of the thesis. The fourth and last section describes the organization of the remaining chapters.

1.1. Failures

The worst possible effect of a failure is that it results in an inconsistent database. For most applications, this is intolerable. Inconsistency is highly undesirable, even if it is detected at a later time and removed, since the transactions executing after the inconsistency is introduced will have to be undone.

A tolerable, but undesirable, effect of a failure is that all transaction processing is blocked until the failure is repaired. This preserves consistency but reduces the availability (for performing useful work) of the distributed system. In fact, the availability of the distributed system is lower than that of its weakest component and, in this sense, it is less available than a centralized system.

The least harmful effect of a failure is that it renders only the failed components unavailable. Hence, it will impact only the transactions which require those components. Since an unfinished transaction can tie up valuable resources and thus block future transactions, it is often considered preferable to abort a transaction rather than allow it to remain pending. Of course, a transaction can be aborted (or committed) only if it can be shown that the action will not be inconsistent with the action taken by another site, in particular, with a failed site. To achieve this, special protocols, called termination protocols, are used. If a transaction can not be safely terminated after a failure, then it is said to be blocked. A protocol that occasionally blocks transactions to ensure data consistency is called a blocking protocol.

In any environment, failures can be classified into two disjoint sets: the expected failures and the unexpected failures. Unexpected failures either can not occur or occur with such a small probability that their presence and consequences can be ignored in designing the system. On the other hand, each expected failure has a nonzero probability of occurring, and must be considered in the design of the system. A hard disk crash is probably considered to be an expected failure in most systems; however, two independent hard disk crashes within a short period of time is an unexpected failure for almost all systems.

The classes of failures that we expect to occur include: site failures, lost messages, and network partitioning. The classes have been listed in increasing order of difficulty. We will discuss them sequentially.

Site failures are the most frequent failures, and there are two major facets to the problem. The first one is the correct termination at operational sites of all outstanding transactions.

The second facet concerns failed sites. A failed site must complete all transaction outstanding at the time of failure. Protocols to accomplish this are called recovery protocols. Compounding this problem is the possibility of a site suffering partial amnesia upon recovering -- that is, it does not remember its exact state at the time of failure. For example, sending a message and logging it are separate operations and failures may occur between them; therefore, a site can not be certain of the last message that it sent. This is true regardless of the order of logging and message sending.

A special case of site failures occurs when all participating sites fail. Standard recovery protocols may not perform satisfactorily in this case.

The second class of failures is lost messages. In many ways, this class of failures can be handled similarly to site failures, since a site's messages are lost while it is inoperative. The major difficulty in designing protocols resilient to lost messages concerns the uncertainty involved. Absence of a message from a particular site could mean either that a message was lost or that the site is down.

Network partitioning, the third class of failures, occurs when the network is divided into at least two groups (partitions) of operational sites and no communication is possible between different groups. A partitioning may result from the failure of one or more physical communication lines interconnecting the sites, or from the failure of one or more sites through which all messages from one group to another must pass.

The partitioning problem does not lend itself to very robust protocols. Normally, the best achievable protocols allow a single group of sites to terminate the transaction, while the remaining groups are forced to block until the failure is repaired. Even this is not achievable when a site failure occurs in conjunction with the network partitioning.

The problems introduced by the above failure are compounded when uncertainty exists in the detection and classification of failures. A site observes a failure through the absence of an expected message from a site. As noted before, the absence could be caused by different types of failures -- in fact all of the above failures first reveal themselves in this way. The sites can then attempt to determine which type of failure actually occurred. In realistic systems, no protocol can reduce this uncertainty to zero (although, for many failures, the probability of misclassifying the failure can be made arbitrarily small). When necessary, uncertainty can be handled by taking a pessimistic approach and having transactions block even though it may be safe to proceed.

The failures that we have mentioned are those that are expected to occur in any system, even if its software is correct. There are also failures (or, perhaps more correctly, errors) which can be caused by aberrant or malicious systems, including spurious messages, fabricated messages, and garbled messages. We include these in the set of unexpected failures and do not attempt to handle them. We also consider a catastrophic site failure, where data is irreparably lost, to be an unexpected failure.

1.2. Scope of Thesis

This thesis systematically examines each major class of failures and attempts to answer the following questions.

The first question considered is always: Does a resilient nonblocking protocol exist for this class of failures? When the answer is no, we must then identify the realizable recovery strategies for resilient protocols. Normally, we will be interested in finding the "best" strategy, where one strategy is considered better than another if, on the average, it allows safe termination of a pending transaction at a greater number of sites.

The next question addressed is: What are the necessary and sufficient conditions for a protocol to be resilient to the given class of failures? We use these results to derive resilient commit protocols and resilient recovery protocols. Given sufficient conditions for resiliency, a family of related protocols can be derived where protocols within such a family differ in their communication structure (e.g. whether communication is centralized or decentralized). Given necessary conditions for resiliency, protocols that minimize message cost or that maximize parallelism can be derived. Many of the derived protocols are formally verified.

We make the following assumptions about a distributed database system. A distributed transaction is executed concurrently at a subset of the sites, which are known as the participating sites. Each participant can unilaterally abort the transaction in the early phases of processing; however, once a site has sent a message to another site indicating its willingness to process the transaction, it forfeits its right to unilaterally abort. A distributed transaction can be committed only if none of the participants decide to unilaterally abort it. Furthermore, a commit or abort decision is _irreversible_.

Unilateral aborts are allowed for two important reasons:

(1) If a site fails during the initial processing of a transaction, then this failure can be viewed as a unilateral abort.

(2) A large class of concurrency control protocols can be be used with a commit protocol allowing unilateral aborts. All concurrency control protocols occasionally detect conflicts that require the abortion of one or more transactions. If all sites performing concurrency control tasks are considered participants in the transaction, then any of them can unilaterally abort the transaction whenever a conflict is detected.

From the underlying network, we require a point-to-point message facility. However, we do _not_ require that messages be delivered in the order sent. We assume that the network experiences no unexpected failures, including the generation of spurious messages and the undetected passing of garbled messages. Messages sent to failed sites are lost.

1.3. Outline of Thesis

This thesis consists of eight chapters, organized as follows.

Chapter 2 is devoted to reviewing the current state of the art in crash recovery for distributed systems. It discusses local crash recovery, identifying the techniques and results that are applicable to distributed systems. The bulk of the chapter is a literature survey of proposed protocols. Several classification schemes for protocols are discussed, in particular, centralized and decentralized classes.

Chapter 3 introduces the theoretical framework used in both the specification and verification of individual protocols. A formal model for commit protocols is proposed using finite state automata. The *correctness* of a protocol is precisely defined. The modeling of failures is also discussed.

Chapter 4 presents several existence proofs for recovery strategies for site failures and for network partitions. These results are important for two reasons: (1) they delineate the feasible solution space for recovery protocols, and (2) they illustrate a proof paradigm that will be used in many subsequent theorems.

Chapter 5 is the first of the two chapters concerned with site failures. This chapter discusses the role of operational sites and, in particular, the design of nonblocking protocols. The Fundamental Nonblocking Theorem (for site failures), one of the major results of this thesis, is proved. From this theorem the *canonical nonblocking protocol* is derived. This protocol is used to design nonblocking commit and termination protocols for a variety of different network environments.

Chapter 6 is the second chapter concerned with site failures. It addresses the problem of recovery of a failed site, using the existence results of the previous chapters. It also discusses the case where all sites fail during the execution of a transaction.

Chapter 7 addresses the problems associated with lost messages and partitioned networks. Since partitioning is the harder of the two problems, it is emphasized. This chapter also addresses a problem that is of similar complexity as the partitioning problem: uncertainty in failure classification. Majority voting schemes, which are resilient to both partitioning and uncertainty, are developed, and two resilient protocols are presented. Primary site protocols, an important limiting case of voting schemes, are also discussed.

Chapter 8 concludes the thesis by restating the major theorems and their implications, and contrasting the significant protocols.

CHAPTER 2

Background

There are two difficult problems in transaction management: (1) concurrent execution of transactions and 2) preserving transaction atomicity when failures occur. Traditionally, issues in fault-tolerant transaction management have been closely coupled to concurrency control. Only part of this coupling is inherent; some of it is due to the popular design methodology of designing concurrency control algorithms first, and adding resiliency in a later design phase.

The approach in this thesis is to separate the issues of fault tolerance from concurrency control whenever possible, and concentrate only on the fault-tolerant issues. We will examine fault tolerance on a "per transaction" basis - i.e. examine what is needed to make a single transaction (executing alone) resilient to failures. The primary rationale for this approach is simple: the mechanisms required for resiliency in the single transaction case must also be present when transactions are executing concurrently. Therefore, a resilient concurrency control algorithm must incorporate these mechanisms in one form or another.

Separation of issues also leads to a clear, good hierarchical approach to distributed a transaction management system. We can design a distributed transaction manager consisting of three layers of abstraction, as illustrated in Figure 2.1.

The innermost layer consists of very resilient single site transaction manager implementing a local notion of a transaction. Concurrency control and crash recovery for single site transactions are well understood problems ([LIND79, EDEL74, GRAY79]). The distributed transaction manager is built on top of this, requiring the notion of a local atomic action that is provided by this layer.

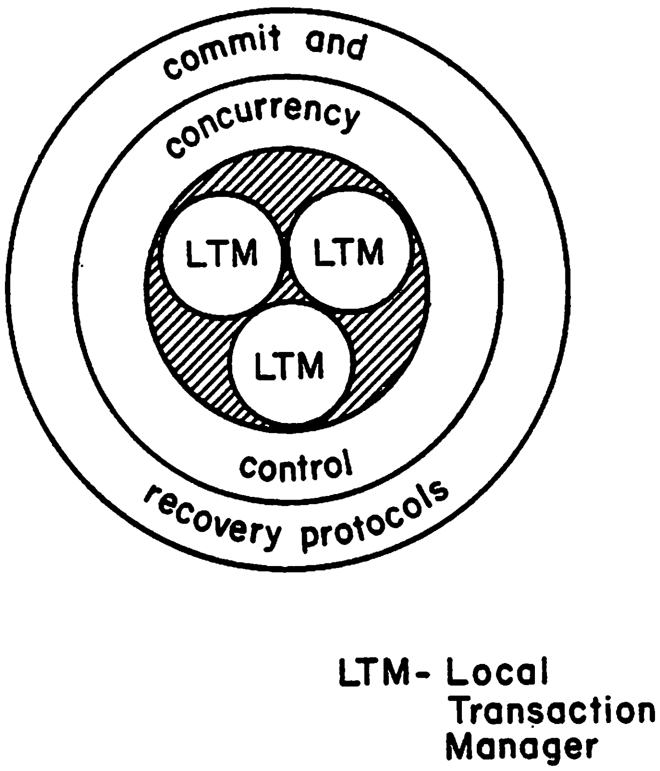


Figure 2.1 The architecture of a distributed transaction manager.

In the layer immediately above the local manager, concurrency control for distributed transactions is provided. This may be through locking, timestamps, or some other means. When a deadlock or a conflict is detected at this level, the concurrency control protocol will unilaterally abort the transaction and communicate this to the layer surrounding it.

In the outermost layer, the notion of a resilient distributed transaction can be implemented. This will consist of a commit protocol and various types of recovery protocols. To facilitate the implementation of a concurrency control scheme, this level allows a transaction to be aborted from a lower level. In addition, the recovery mechanism may decide to abort during recovery.

Since the resiliency issues in transaction management have been divorced from concurrency control issues and since the semantics of a transaction are unspecified, the techniques discussed are applicable to the implementation of "atomic actions" at any level within a distributed system. For example, the results can be applied to updating systems tables (e.g. the name table) in a distributed operating system.

2.1. Local Crash Recovery

Local crash recovery not only provides the basis on which a distributed crash recovery mechanism is implemented, but also contains many concepts which can be applied at the distributed level as well. Within this section, we refer to local crash recovery simply as "crash recovery."

All crash recovery schemes must be _idempotent_ -- performing recovery many times yields the same result as performing it once. Since the system can fail during recovery, it may attempt to recover a transaction several times.

All recovery schemes require some form of auxiliary data, which we will collectively refer to as _recovery data_. The recovery mechanism is no more resilient than the storage medium on which the recovery data resides. The cost of maintaining this recovery data is by far the major cost of any recovery scheme.

Recovery data must be stored in secondary storage if it is to survive soft crashes (processor failures). However, secondary storage is not resilient to hard disk crashes. If resiliency to hard crashes is desired, then _stable storage_ (as proposed by Lampson and Sturgis in [LAMP76]) can be used. Stable storage requires that two copies of each record be stored on disk. A record is modified by using _careful replacement_ -- one copy is updated and verified to be correct, then the second copy is updated. The scheme assures survival from all single instances of hard crashes.

Throughout this thesis, we will use the term "stable storage" in a generic sense. It refers to the most resilient online storage available. (In many systems, it is merely disk storage.)

Local recovery mechanisms present the following view of transaction processing. The sequential execution of a single site transaction consists of two distinct periods separated by a single commit point. If a failure occurs before the commit point is reached, then the transaction is rolled back (all of its effects are undone). If a failure occurs after the commit point, then the transaction is rolled forward (its effects are permanently installed).

Once the commit point is passed the transaction can not be aborted. Hence, deadlock detection and conflict resolution by the concurrency control mechanism must occur during the initial period of transaction processing.

There are two common methods for implementing this abstraction of transaction processing. One uses logs, the other uses a data structure called an intentions list. In both schemes, the commit point corresponds to setting a commit flag in stable storage.

In the first method the recovery data is stored as an incremental log of changes. This allows updating in place -- the affected data record is updated directly. Each modification generates a log record containing the old value (called a before image) of the updated data record. If a crash occurs before the commit point is reached, then the log is read and the before images are re-installed.

Writing a log record and updating a data record two separate operations, and failures may occur in between their executions. The order in which these operations are performed does affect the resiliency of the recovery scheme. A modification is "undoable" only if its log record is stored in stable storage. If an update is made to a data record before the log record is stored, then there is a window of time in which the occurrence of a failure would render the modification undoable. The solution is to use a write-ahead log protocol where the log record is stored in stable storage before the data record is modified ([GRAY79]).

The second method does not update in place; instead, modifications for an entire transaction are collected into a list called the intentions list. The records stored in the intentions list are the after images (updated values) of the modified records. After the commit point is reached, the modified records in the intention list are merged into the database. If a failure occurs before the commit point, then the intention list is simply discarded.

For this method to succeed, the intentions list must be written to stable storage in its entirety before the commit flag is set, and the commit flag must be stored in stable storage before the updates are merged. If any operation is performed out of sequence, then recoverability is compromised.

Conceptually, the intentions list can be thought of as a large log record for the entire transaction (in contrast to having one log record for each data record modified). From this perspective, intentions lists obey the write ahead log protocol: the intentions list is written first before the data is modified. The write ahead log protocol is a simple but important technique. It will also be useful in designing distributed recovery mechanisms.

2.2. Distributed Recovery

A distributed transaction can be viewed as a collection of local subtransactions (with additional data movement between sites) -- one executing at every participating site. The commit protocol uses the local recovery mechanisms to resiliently abort or commit each subtransaction.

The simplest commit protocol allowing unilateral abort is the two phase commit protocol, which is illustrated in Figure 2.2 for two sites ([GRAY79, LAMP76]). It is a master/slave protocol -- a designated site (Site 1 in the figure) coordinates the execution of subtransactions at the other sites. In the first phase of the protocol

SITE 1	SITE 2
(1) Transaction request arrives. Start transaction is sent.	
	Start trans. is received. Site 2 votes: yes to commit, no to abort. The vote is sent to Site 1.
(2) The vote is received. If vote=yes and Site 1 agrees, then commit is sent; else, abort is sent.	
	Either commit or abort is received and processed.

Figure 2.2 The two phase commit protocol (2 sites).

the coordinator distributes the transaction to all sites, and then each site individually votes on whether to commit (yes) or abort (no) it. In the second phase, the coordinator collects all the votes and informs each site of the outcome. In the absence of failures, this protocol preserves atomicity.

Subtransaction processing can be decomposed into three sequential intervals (cf. two intervals for a local transaction). as illustrated in Figure 2.3. The first and last interval are analogous to the two intervals found in local transaction processing. The second interval is a period of uncertainty that has no counterpart in local transactions.

A failure in the first interval causes the subtransaction to be unilaterally aborted, thus precipitating the abortion of the entire

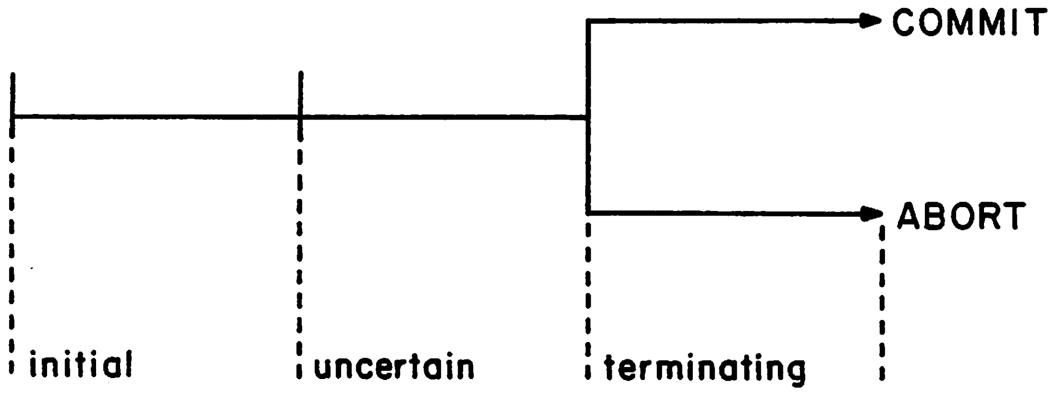


Figure 2.3 The processing of a distributed transaction at a given site. (Note: the "uncertainty" period is skipped for a site that unilaterally aborts.)

transaction. Also, during the interval the site can choose to unilaterally abort the transaction for any of the reasons given previously. The end of this interval is marked by a local commit decision -- after that point the site can no longer unilaterally abort the transaction. Therefore, all local locks must have been granted by that time. This interval corresponds exactly to Phase 1 of the two phase commit protocol.

During the second interval, the period of uncertainty, a site has agreed not to abort the transaction and is waiting to hear from other sites to see if the transaction can be committed. The interval ends when the site receives a message revealing the fate of the transaction. If a failure occurs during this interval, then during recovery the site must wait to hear the outcome of the transaction (i.e. commit or abort) from one of its fellow participants. For a slave executing the two-phase commit protocol, this uncertainty period begins after it sends a "yes" decision to the coordinator and ends when a commit or abort message is received.

On the third interval the fate of the transaction has been determined, and it is the responsibility of the local recovery mechanism to either recoverably roll forward the subtransaction or to roll back the subtransaction. Historically, this is called the "write" interval ([KUNG81]), because this is normally the first time that transaction's updates are accessible to other transactions. The actions performed during this interval are **irreversible.**

The "write" step occurs after a commit decision has been reached; therefore, a concurrency control mechanism cannot choose to restart (abort) a transaction during this step. In particular, all deadlock detection must be performed before transaction processing reaches this stage. In a locking scheme, normally all local locks are acquired during the first step. Local deadlock detection is also performed during this step. Global deadlock detection may be performed during either the first or second step.

2.3. Classes of Protocols

Protocols can be conveniently classified by their communication topology. This describes which sites are allowed to send messages to which other sites in a given protocol and in contradistinction to network topology, which is independent of the protocol and describes which sites are capable of sending messages to which other sites. To a large degree, a protocol's communication topology determines its cost, its difficulty to code, and its inherent symmetry (or asymmetry).

From the myriad of possible topologies there are two important classes: centralized protocols and decentralized protocols.

2.3.1. Centralized Protocols

We have already seen an example of a centralized protocol -- the two-phase commit protocol. The properties of a centralized protocol are:

(1) There is a distinguished site called the coordinator. The remaining sites are called slaves.

(2) There is a master/slave relationship between coordinator and slave.

(3) Communication is solely between coordinator and slave. Slaves do not communicate with one another.

(4) A phase of a centralized protocol consists of the coordinator sending a message to all slaves and the slaves responding.

Centralized protocols tend to be considerably more popular than decentralized protocols because they are easier to conceptualize, to verify, and require fewer messages than decentralized protocols.

This class of protocols has two major drawbacks. Their primary disadvantage is their vulnerability to coordinator failures. When the coordinator fails, slaves normally can not safely proceed until it recovers. Their second disadvantage is the lack of parallelism between the coordinator and the slaves. During a phase, the coordinator sends a message to the slaves and waits for a response; the slave sends a response and waits to hear from the coordinator again. It is normally the case that either the coordinator or the slaves are waiting.

Generalizing the centralized communication structure to a full tree, we obtain the class of hierarchical protocols. Again, there is a single designated site, the coordinator, which is the root of the tree. As before, a phase normally consists of the coordinator broadcasting a message to all participants and then waiting for their replies. Unlike the simpler centralized protocol, the responses are not sent directly to the coordinator, but instead to each site's immediate predecessor in the tree. A site whose position is an internal mode in the communication tree collects messages from all descendants and, based on these messages, will send a single message to its predecessor. This is called a convergecast.

A hierarchical protocol is well-suited for large store-and-forward networks where messages are expensive. A minimal spanning tree rooted at the coordinator and based on point-to-point message costs can be constructed ([PARK81]). If the communication topology of the protocol is defined by such a minimal spanning tree, then the cost of the broadcast and convergecast is minimized.

Another feature of hierarchical protocols is that they can exploit parallelism during unilateral aborts. An internal node learning of an abort can relay it to its predecessor as well as initiating the abort in its subtree.

A subclass of the hierarchical class that merits discussion is linear protocols ([LIND79]). Linear protocols have a communication topology that forms a simple linear chain, each site having one neighbor on the left and one neighbor on the right (except for the leftmost and rightmost sites which send messages to one neighbor). In a linear protocol, by the time a message propagates from the coordinator to the farthest slave, the message has visited every site. We can take advantage of this fact to design even simpler protocols than the hierarchical protocols. A phase now consists of propagating a message from one end to another. The role of coordinator alternates between the sites at either end. (In this environment, the coordinator is the initiator of the next phase.)

As an example, consider the linear two-phase commit protocol (see [LIND79]). Let the sites be numbered 1, 2, ... N with Site i communicating with Sites i-1 and i+1 (except for the end sites 1 and N, which communicate with only one other site). For Phase 1, Site 1 is the coordinator; for Phase 2, Site N is the coordinator. The protocol is the following:

Phase 1. Site 1 receives the transaction and, if it decides not to unilaterally abort the transaction, sends it to Site 2. Site 2 either aborts the transaction or forwards it to Site 3, and so forth. If any site aborts the transaction, then the site sends an abort message to its predecessor which in turn will propagate the abort.

Phase 2. The transaction reaches Site N and is processed and accepted. Site N then sends a commit message to Site N-1 which in turn will send a commit message to its predecessor until the message reaches Site 1.

The linear protocol halves message traffic in each phase at the expenses of decreasing parallelism among the sites. In the original two-phase commit protocol, all of the slaves process and vote on the transaction in parallel; in the linear protocol, the sites process the transaction sequentially.

A linear protocol where messages flow in one direction only and Site N sends messages to Site 1 is called a *ring protocol*. Concerning resiliency issues, ring protocols and linear protocols have the same advantages and disadvantages; however, ring protocols are more popular in the literature ([ELLI77a, LELA80]).[1]

The number of messages required in all four centralized protocols is a linear function of N, the number of sites. The number of end-to-end delays, however, varies greatly among the different classes. For the centralized protocol, it is two delays, while for the linear and ring protocols it is N.

2.3.2. Decentralized Protocols

While the term "decentralized" is descriptive of a large number of classes, we are interested in a very specific class. Decentralized protocols have the following properties.

(1) Every site communicates with every other site,

(2) Every site participates equally, and

(3) A *phase* consists of a message round where all sites exchange messages. A site sends the same message to each of its cohorts during a round.

Decentralized protocols require a lot of messages - approximately N^2 messages are sent during each round. Therefore, they are

[1] Some of the reasons for the greater popularity of ring protocols are related to concurrency control issues. For example, ring protocols release resources in the same order acquired.

primarily suited for network environments where either messages are cheap or a broadcast facility is available. Fortunately, in many local networks (e.g. ETHERNET [METC76]) one or both conditions apply.

The primary advantages of decentralized protocols are (1) the lack of a coordinator, which eliminates one potential bottleneck in any system, and (2) the exploitation of parallelism. Decentralized protocols maximize parallel processing among the sites. These protocols are completely symmetric which tends to make them simpler and easier to implement than centralized protocols.

The two-phase commit can be modified to a decentralized protocol:

Phase 1. The originating site sends the transaction to all cohorts. Each cohort sends its vote to all the other sites.

Phase 2. Each site waits until it receives votes from all cohorts, and then it appropriately commits or aborts.

Contrasting the centralized and the decentralized two-phase commit protocols, we find three striking dissimilarities. The first is the symmetry of the decentralized protocol. Except that initially a single site distributes the transaction, the sites in the above protocol have completely symmetric roles.

The second dissimilarity is along these lines: there is no central decision maker; each site independently makes a decision based on the messages it receives. Of course, in the absence of failures, each site receives an identical set of messages.

The third dissimilarity concerns failure. Since the decision making is decentralized, there are a wider variety of failures that can adversely affect the protocol. This can be contrasted to centralized protocols where only coordinator failures are difficult to handle.

2.4. Existing Literature

We now review several mechanisms that have been proposed for constructing protocols resilient to various classes of failures. We then discuss how these mechanisms are integrated into two proposed systems: distributed INGRES ([STON79]) and SDD-1 ([HAMM80]). A survey of a third proposed system, System-R, can be found in [GRAY81].

The specification and verification of protocols is a closely related area which has been intensely investigated. We review that literature in the next chapter, where we introduce a formal model for commit protocols.

2.4.1. The Two Generals' Problem

This is the classic paradigm on the resiliency of commit protocols. It is one of the earliest results in this area and, unfortunately, it is a negative result.

The problem can be stated as follows ([GRAY80]):

> Two generals are situated on adjacent hills. In the valley between them lies the enemy. The enemy can easily defeat the army of either general, but not the combined armies. Therefore, for this campaign to be successful, either both generals must attack or both must retreat. The generals can only communicate through messengers which are subject to get-

ting lost or being captured.

This problem is a colorful restatement of the problem of designing commit protocols resilient to lost messages. It has been shown that there exists no finitely bounded protocol which insures success for the generals. Like many proofs for this type of result, the proof is by contradiction. We now present it to illustrate its brevity and because it is a good paradigm for many proofs on resiliency results.

Assume that a bounded protocol exists. Let P be such a protocol of minimum length (in the number of messages). Let M be the last message sent by P. Since this message can be lost, we may delete it from the protocol with no loss in resiliency. Let P' be the protocol that results from deleting M. Now, P' is a bounded protocol shorter than P, and P' insures success for the generals. This contradicts our original assumption.

This problem reaffirms our intuition that there exists no (finitely) bounded commit protocol using a network that is less than 100% reliable. Therefore, to build completely reliable commit protocols on top of unreliable components, we must be willing to tolerate unbounded protocols. Alternatively, we can a priori bound the number of failures tolerated by the commit protocol and derive bounded protocols.

In many applications the existence of a finite length protocol is not crucial. Unlike the attacks by the two generals, the commitment of a transaction does not have to occur simultaneously at the participating sites. Instead, it is sufficient that all sites "eventually" commit the transaction.

Various extensions of this problem to N generals have been formulated in [LAMP80] (this is also referred to as the Byzantine General's Problem). The case where generals may be traitors or pass on erroneous or inconsistent information is investigated in [PEAS80].

2.4.2. K-resiliency

A system is K-resilient if it can tolerate arbitrary failures by K distinct sites. The term was introduced in an early paper by Alsberg et al. ([ALSB76]), which examined the problem of achieving K-resiliency in a completely replicated database system. Unilateral aborts were not allowed in their system.

The paper proposed a one-phase protocol for achieving K-resiliency using a scheme where a designated single site holds the primary copy of the database. All transaction requests are forwarded to the site containing the primary copy, which then serves as transaction coordinator. In this capacity, it computes the updates and distributes them to the other sites. After K acknowledgements have been received, a "transaction completion" message is sent to the originating site.

2.4.3. The Two Phase Commit Protocol

We have already discussed this protocol extensively - it is the "archetypical" commit protocol. Historically, the first public description of the protocol is in [LAMP76],[2] although the protocol was known before then. It is also discussed in [GRAY79, and LIND79].

The protocol is popular for several reasons. It is the "simplest" protocol allowing unilateral aborts -- no single phase protocol allows unilateral aborts by arbitrary sites. A variation of this protocol called the _linear two phase commit_ ([LIND79]) is the cheapest (in the number of messages) commit protocol. It can easily tolerate arbitrary failures of slaves.

The protocol is vulnerable to coordinator failures and to network partitions that isolate the coordinator. We now discuss these two problems in turn.

When the coordinator fails in the second phase of the protocol, it is often impossible for the operational slaves to safely proceed with the transaction. This is especially true if another slave fails in conjunction with the coordinator failure. Consider the following scenario.

(1) The coordinator sends a new transaction to two slaves.

(2) Both slaves send their votes to the coordinator. (Assume that the second slave voted "yes.")

(3) The coordinator sends a commit (or abort) message to the first slave and then promptly crashes.

[2] The protocol described in [LAMP76] is actually a three-phase protocol. The extra phase describes writing the commit record to disk. Except for this elaboration, the protocol described is a two-phase protocol.

(4) The first slave receives the message, performs the indicated action, and then crashes.

Now, the sole operational site, the second slave, cannot safely proceed. If either the coordinator or the first slave unilaterally aborted the transaction, then it should abort; otherwise, it should commit. In either case it does not know the appropriate action. During a network partition, only the sites in direct communication with the coordinator can safely proceed in all cases. If the coordinator is completely isolated, then none of the slaves can safely proceed.

Despite its shortcomings, the two-phase commit is very robust considering its cost. Protocols that protect against coordinator failures are much more expensive. Such a protocol is discussed next.

2.4.4. The Four-Phase Commit Protocol

To overcome the deficiencies of the two-phase protocol, a four-phase protocol was introduced in [HAMM80]. It provides K-resiliency against all types of failures including coordinator failure.

The commit protocol uses K backup coordinators, where each backup is capable of assuming the role of coordinator. The protocol extends the two-phase protocol: each phase maps into two phases in the new protocol, where the first new phase communicates with the backups and the second new phase communicates with the slaves as before. The complete four phase protocol is:

Phase 1. The coordinator sends the transaction to each backup and waits for acknowledgements.

Phase 2. The coordinator sends the transaction to each slave, and then waits for the slaves to vote.

Phase 3. The coordinator sends the commit/abort decision to each backup and waits for acknowledgements.

Phase 4. The coordinator sends the commit/abort decision to each slave and waits for acknowledgements.

If the coordinator fails during the protocol, then an election is held among the backups and, assuming at least one backup is operational, a new coordinator is chosen. Several election protocols are known (see [GARC80b] for a survey), and they can be made resilient to failures.

The newly elected coordinator executes a two-phase protocol. The decision to commit or abort is made solely by the new coordinator - it will commit if it received a commit message from the previous coordinator; otherwise, it will abort. Since the new coordinator's commit decision does not require communication with other sites, there is no possibility of the protocol blocking. The backup protocol is essentially the last two phases of the four phase commit. The protocol is:

Phase 1. The new coordinator sends commit/abort to all operational backups and waits for acknowledgements.

Phase 2. The commit/abort decision is sent to all slaves. Again, the backup waits for acknowledgements.

If the new coordinator fails before completing the backup protocol, then a new election is held and the winner re-executes the backup protocol from the beginning.

Notice that two successive coordinators may disagree on whether to commit or abort a transaction. However, consistency among the slaves is still assured because the following invariant holds throughout the execution of both the commit and the backup protocols: _no slave is directed to commit or abort until all operational backup coordinators have acknowledged to commit (or abort)_. Hence, if two backups disagree on the direction of the transaction, then clearly no slave has been directed to commit or abort.

When implementing the protocol care must be taken that backup coordinators are not incorrectly assumed down. If a backup which had been assumed down by a coordinator is elected as a new coordinator, then the invariant given in the previous paragraph may be violated resulting in an inconsistent database. Consider the following example. Let there be a single backup coordinator. Assume that a network partition results in the coordinator and the backup occupying different partitions. Now, if the backup incorrectly diagnoses the problem as a coordinator failure, it will assume the role of a coordinator and independently make a commit decision that may be different from the original coordinator's decision. In particular, the backup may decide to abort whereas the coordinator may decide to commit.

2.4.5. Spooling of Messages

Normally messages destined to a down site are spooled (queued) at the sending site. Therefore, when a site recovers, it will be able to receive its pending messages only if all of the sending sites are operational. If the site was down for any length of time, it is very likely that all sites have spooled messages for it. Hence, if any other site is down when the site recovers, it will lose some of its messages (or, alternatively, block until the other site recovers).

An alternative approach is to appoint a _spooling site_ (or more concisely, a _spooler_) for each failed site. Every message addressed to a failed site is redirected to its spooler. When a site recovers, either all of its pending messages are available, or none, depending on the state of its spooler. However, the probability that all the messages will be available is much greater in this scheme than in the other scheme (assuming that spooling sites are judiciously chosen). Of course, this scheme can be more robust by assigning _K_ spoolers for each failed site. Now, to have even a single message unavailable requires the failure of K sites.

A mechanism very similar to this has been implemented in the SDD-1 Recovery Manager ([HAMM80]). Each site is statically assigned a set of sites for which it will act as a spooler. When a sending site discovers that the intended recipient is down, it resends the message to all of the recipients _K_ spoolers. This redirection is expensive, hence it is not used for all types of messages. In particular, a message that has been redirected to a spooler is not redirected further if the spooler is down.

Powell has suggested dedicated "passive" spoolers for ETHERNET-like networks ([POWE81]). With such a network, a site hears all messages but saves only those addressed to it. K sites could be assigned the task of spooling (or logging) all of the messages on the network. This activity would be completely transparent to the other sites, except during site recovery. Since the bandwidth of such a network approaches memory to disk bandwidth, spoolers would have to be dedicated sites.

Spoolers are used in [LISB81] to achieve a "reliable broadcast." A broadcast message is circulated in a virtual ring[3] of spoolers before it is sent to any of its recipients. If a recipient is down, then the message is available from one of the spoolers when it recovers.

2.4.6. Majority Consensus Approaches

Majority consensus protocols have been proposed to solve two distinct problems:

(1) the synchronization of updates in a completely replicated database ([THOM79, GIFF79]), and

(2) resiliency against failures, especially network partitioning ([STON79]).

[3] A virtual ring is a set of sites that communicate using a ring protocol.

We are concerned primarily with the second problem, although majority consensus solutions to the first problem are also very resilient as a natural consequence of the approach ([THOM79]).

Resilient protocols based on this approach require that a majority of the sites agree to an irreversible action (e.g. a commit) before any site proceeds with the action. The majority consensus requirement ensures that two groups of sites do not act independently. Hence, a majority consensus approach is the obvious solution to network partitioning ([STON79] proposes it to handle partitioning). The approach works even if a partitioning is misdiagnosed as a collection of site failures. The drawback of this approach is that sometimes a majority can not be attained, even when it is safe to proceed.

The simple majority requirement can be generalized to allow the use of quorums. A quorum is the minimum number of sites needed to proceed with an action. It need not be a majority. A quorum size is determined for each possible action (e.g. commit or abort). To ensure that two operations are mutually exclusive, the sum of their quorum sizes must exceed the total number of sites.

An orthogonal generalization is to allow weighted voting. ([GIFF79] describes a quorum-based scheme with weighted voting for concurrency control.)

2.4.7. Logging

Local recovery mechanisms normally log all transactions in some form. Distributed recovery strategies can be defined to utilize these logs and, since the information is available anyway, probably should utilize them. Minimally, only the transaction identification number and the value of the commit flag (if it is set) are required by the distributed recovery strategies and this is all that must be stored.[4] Even in local strategies that do not use a log, the commit flag is still written to disk and this write could just as easily be directed to a log.

In distributed recovery, the logs provide an alternative to spooling commit (abort) messages. A recovering site can query the logs of its cohorts about pending transactions instead of reading messages from spoolers. This scheme is cheaper than spooling and requires no special action of the coordinator when a slave fails. The major weakness of the scheme is that the resiliency to site failures is now a function of the number of participants - at least K sites must participate for the protocol to be K-resilient.

[4] In fact, even less than this needs to be stored. For example, if we assume that most transactions are committed, then only the transaction number and a binary flag need be stored for "in progress" or aborted transactions, and nothing need be stored for commit transactions. (Here the absence of a stored record for the transaction indicates that it was committed.) Normally, much more information is also stored.

2.4.8. Crash Recovery in Distributed INGRES

The INGRES proposal describes two disjoint sets of protocols: (1) the *performance protocols* where resiliency is of secondary importance, and (2) the *reliable protocols* where reliability issues are seriously addressed ([STON79]). Both approaches are based on the two-phase commit protocol described earlier.

The INGRES environment supports partially replicated data. A logical database is partitioned horizontally into fragments and each fragment may be physically replicated any number of times. Although a site can store more than one fragment, we will speak as if each site held exactly one.

In the performance protocols, one copy of each fragment is designated as the primary copy. A site holding a primary copy is a *primary site*; the remaining sites are *copy sites*. Only primary sites participate in the two-phase commit and they participate as slaves. (For this exposition, we will assume that the coordinator does not hold any fragments.) If a primary site fails at any time, then an election is held among the associated copy sites. The chosen site is promoted to the role of primary site.

There are two protocols involved in executing a transaction. First, the standard two-phase commit protocol is executed by the coordinator and the primary sites (i.e. the slaves). After a primary site has committed a transaction, the site executes a second protocol that sends a list of updates to each associated copy site.

The performance advantage results from using only primary sites in the two-phase commit. Fewer participants means that it is less likely that a slow site will delay the commit decision. Furthermore, global deadlock detection involves fewer sites, and hence is cheaper. Fewer messages are sent both during the commit protocol and during deadlock detection. Updates to copy sites can be sent and processed in the background.

Naturally, the protocol exhibits the same deficiencies discussed in the section on the two-phase commit; specifically, a transaction is blocked (or lost) when the coordinator fails, and messages may be lost since they are only spooled at the sending site. There is an additional deficiency: copies may become inconsistent due to the failure of a primary site. This will occur when the primary site commits an update but fails before sending it to the copy sites.

The reliable protocols overcome two of the deficiencies by (1) spooling messages at K sites, and (2) by having every site containing a copy participate in the two-phase commit protocol. Hence, there is no longer any distinction among copies. The reliable protocols are still vulnerable to a coordinator failure.

In both protocols, network partitioning is handled by a majority consensus approach. In the performance protocol, a primary site can be elected only if the majority of the copies are operational and communicating. Therefore, there can never be two primary copies. A similar restriction applies to the reliable protocols.

2.4.9. The SDD-1 Recovery Mechanism

The "System for Distributed Data-1" (SDD-1) is a distributed database system implemented by the Computer Corporation of America. It is a prototype system running on the ARPANET. The SDD-1 recovery manager is discussed in [HAMM80] and other aspects of the system are discussed in [ROTH80 and BERN80].

The recovery manager is the most resilient and the most complex manager in existence today. It is designed to provide K-resiliency at every level. However, the problem of network partitioning is not addressed.

At the innermost layer is RelNet ("reliable network"). It provides a _guaranteed message delivery_ service. A _guaranteed message_ will be delivered regardless of the state of the recipient. For failed sites, K spoolers are used to assure delivery. The spooling and redirection of messages is opaque to the above layers.

In addition to guaranteed delivery, this layer also monitors the status (i.e. up or down) of sites. Whenever a site appears to have failed - normally because of a timeout - a "you have failed" message is sent to the site. Upon receipt of such a message, a site will simulate a failure and initiate a recovery algorithm. Hence, two message losses (the original message and the "you have failed" message) must occur before a failure will be incorrectly diagnosed.

The commit protocol is the four-phase protocol described earlier. Hence, K+1 coordinator failures must occur before a transaction is blocked or lost. Messages from the coordinator to the slaves are sent using the guaranteed delivery facility of RelNet. Once again, K+1 failures must occur before a message loss occurs (i.e. the recipient and all K of its spoolers must fail).

In addition to RelNet and the four-phase commit protocol, the SDD-1 recovery manager also provides guardians. A guardian is a site assigned to detect the failure of its "ward site." It does this by regularly polling its ward. When a guardian detects the failure of (one of) its wards, it immediately verifies the failure - by sending a "you have failed" message - and notifies the other sites.

As expected, the SDD-1 approach is robust but expensive. There are alternative approaches to providing K-resiliency that may be less costly. One approach is to do away with RelNet and require backup coordinators to log each transaction. The backups need only log the transaction number and commit flag if we allow unilateral abort during phase two of the four-phase protocol.

2.5. Related Work

Several recent papers discuss reliability issues in distributed database systems ([GARC80a, LAMP78a SCHA78, SVOB79]). Garcia-Molina ([GARC80a]) considers only a fully replicated database; whereas, the others concentrate on irredundantly stored data.

Reliability issues in operating systems are closely related to many of the issues in database systems. For a survey of such issues in a single site operating system, the reader is referred to

[DENN76]. Saltzer considers some of the same issues for a distributed environment, albeit in less detail, in his survey ([SALT80]). However, many operating system approaches are not sufficient for a database system for the fllowing reasons. In a distributed operating system, a process is the "atomic unit"; whereas, a transaction is the "atomic unit" in a distributed database system. Moreover, the notion of a transaction is a broader concept than that of a process. A process is normally confined to a single site, and synchronization between processes is often explicit. A transaction can span many sites, and synchronization is implicit through common data records.

Except for its fault tolerant aspects, we do not address the problem of concurrency control. Interested readers are referred to surveys by Kohler ([KOHL81]) and by Bernstein and Goodman ([BERN81]). A very general discussion of distributed transactions and the problems of maintaining consistency can be found in [TRAI79]. A discussion of related distributed database problems can also be found in a survey by Rothnie and Goodman ([ROTH77]).

Resiliency issues have been addressed in many concurrency control protocols. Several protocols are resilient to site failures ([ELLI77a, GARC79, MENA80, LECA80, MINO80, ROSE79, THOM79]); whereas, only a few are resilient to (or even address) network partitioning ([MENA80, MINO80, THOM79]). Normally, these protocols include the commit protocol as part of their definition, but they do not attempt to solve the general case regarding resilient commit protocols. In particular, none of the concurrency control protocols allow unilateral aborts, and two of the protocols ([MINO80, THOM79]) consider only completely replicated databases.

CHAPTER 3

A Formal Model

In this chapter we introduce a formal model for distributed transaction processing based on finite state automata (FSA). The model introduced in this chapter will be used to specify and verify protocols presented in subsequent chapters and to answer questions about the existence of resilient protocols. We discuss first the model in the absence of failures, and then discuss simple extensions for modeling failures. This chapter concludes with a small bibliography of formal models for protocols comparing this model to previous models for commit protocols.

3.1. Specifying a Protocol

Let N be the number of sites. For the present we will assume that N is fixed and, for simplicity, we will assume that all sites participate. Sites are uniquely labeled 1, 2, ..., N.

The formal specification of a protocol consists of a collection of nondeterministic finite state automata -- one for each site. The automaton executing at Site i is called the local protocol for Site i. The state of this automaton is the local transaction state (or, more succinctly, the local state) for Site i.

The network is modeled as a completely passive device. It is an unbounded buffer that serves as a common read/write medium for all local protocols. A site can read from the network only those messages addressed to it; it can write onto the network messages addressed to any site (including itself).

An example of a local protocol is illustrated in Figure 3.1. It is the two-phase commit protocol for two sites, where Site 1 is the co-ordinator and Site 2 is the slave. A state transition consists of: the receipt of one or more messages, a change in local state, and the sending of zero or more messages. Messages received during a state transition are shown above the horizontal line; messages sent

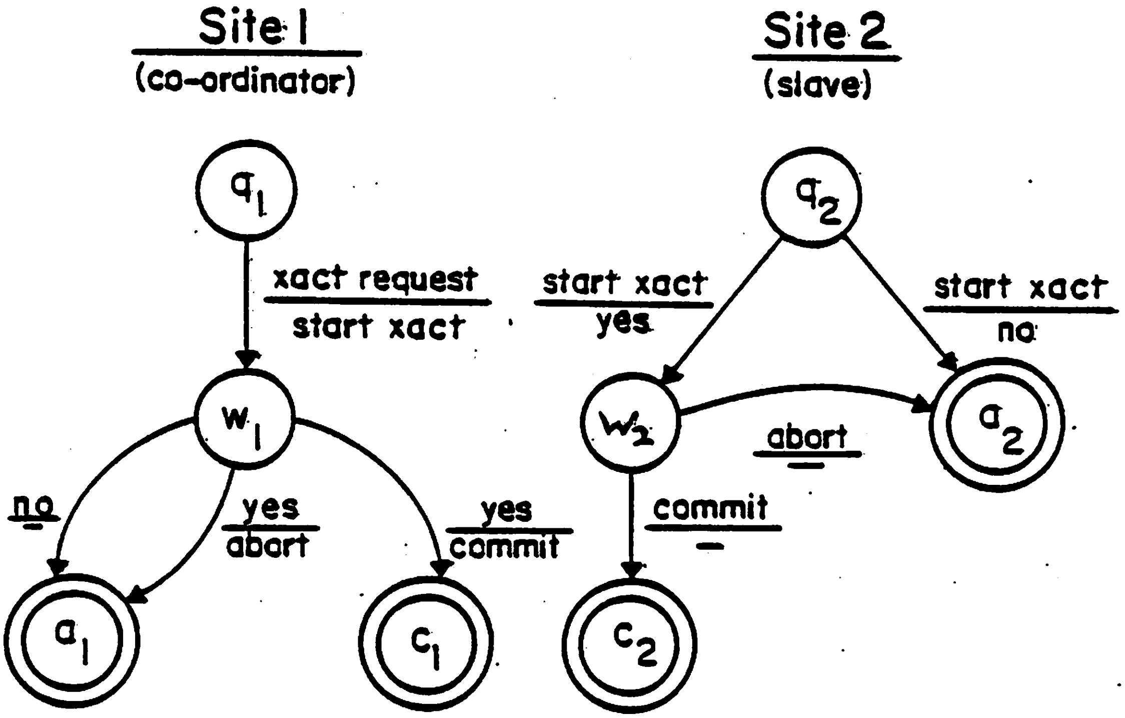


Figure 3.1 The local protocols for the two-phase commit protocol (N=2).

are shown below the line.

Both protocols begin processing in their initial states, (q_1 and q_2). A transaction begins when a request message is received from the application program. The receipt of the request by the coordinator causes a state transition to state w_1 (the wait state) and the sending of the transaction to Site 2. Upon receipt of the transaction, Site 2 nondeterministicly choses either to reply with a yes (to accept the transaction) or a no (to unilaterally abort it) and makes the appropriate state transition. The protocol continues until both sites occupy final states: either commit (c) or abort (a).

The state diagram of Figure 3.1 illustrates the conventions used in the thesis. The states for site i are subscripted by i. Final states are doubly circled.

A state transition from state s_i to state t_i is said to be enabled whenever site i is in state s_i and all the messages required for the transition have been sent. If a site has an enabled transition, then it will eventually make a transition. If more than one transition is enabled simultaneously, then the transition taken is nondeterministically chosen.

State transitions are assumed to be atomic in the absence of failures. It is convenient to consider a transition an instantaneous event and assume that no two transitions occur simultaneously. Transitions among different sites are asynchronous: if transitions are enabled at more than one site, then it is impossible to predict the order in which the transitions will occur.

The model of a local protocol is an extension of the classical nondeterministic finite state automaton since (1) it allows multiple messages to be read and written during a transition, and (2) the ordering of messages on the input medium (the network) is irrelevant. These extensions are more a matter of convenience than necessity since every "extended" automaton is equivalent to a classical automaton.[1]

Formally, a local protocol for site i is defined by a septuple $<Q, \Sigma_I, \Sigma_0, \delta, q, A, C>$ where:

Q - a finite set of states

Σ_I - (the input alphabet) a finite set of "recognizable messages" addressed to the site

Σ_0 - (the output alphabet) the set of allowable messages sent by the site

$\delta : (Q,\Sigma_I^*) \rightarrow (Q,\Sigma_0^*)$ - the state transition function

$q_i \in Q$ - the initial state

[1] This can easily be shown. We assume that the classical automaton has exactly one input tape and one output tape, and the input head can move in either direction. Now we use the following two rules to convert an extended automaton, E, to a classical automaton, C.

1. Every state, e, in E is mapped into k distinct states in C where each state reads and sends exactly one message. If e receives more messages than it sends, then some states in C will send "null" messages. If e sends more than it receives, then some states in C will reread the same message.

2. Every state, c, in C has two state transitions that cycle back into c. One transition moves the read head to the left, the other to the right. This allows C to nondeterministic choose the next message to read.

$\mathbf{A} \subset \mathbf{Q}$ - a set of *abort* states

$\mathbf{C} \subset \mathbf{Q}$ - a set of *commit* states

Collectively, **A** and **C** constitute the *final states*.

3.2. Properties of Local Protocols

3.2.1. Well-formed Protocols

In addition to satisfying the formal requirements, a *well-formed* local protocol has the following properties:

(1) $\mathbf{A} \cap \mathbf{C} = \emptyset$.

(2) $q_i \notin \mathbf{A}$ and $q_i \notin \mathbf{C}$.

(3) There are no transitions from an abort state to nonabort state. Similarly, there are no transitions from a commit state to a noncommit state.

The first restriction precludes a local state from being both an abort state and a commit state. The second restriction precludes a trivial class of protocols. The third restriction corresponds to our notion that commit and abort are irreversible operations.

Notice that either **A** or **C** can be empty. This is reasonable only if the site contains no local data, but still has a significant role in the protocol. The backup co-ordinators of SDD-1 are such sites ([HAMM80]).

Another important property of a protocol is reducibility. A local protocol is said to be *reduced* if all of its state transitions require the receipt of at least one message. Intuitively, this requires that state transition be a reaction to the "global" processing environment rather than a reaction to purely local processing. Reduced protocols can be easily generated from nonreduced protocols.

Throughout this thesis, we assume that a local protocol is well-formed and reduced. For notational convenience, we will also assume that any local state transition sends at most one message to each site. (A protocol containing a transition that sends multiple messages to the same site can be modified such that the messages are encoded as one.)

3.2.2. Equivalence of Local Protocols

The formal definitions of local protocols preclude two sites running identical protocols since Σ_I, Σ_O, and δ are unique for each site (because members of each set contain a subscript identifying the site). However, most protocols consists of a small number (often less than three) of distinct *generic* local protocols, where a generic protocol is derived from a local protocol by stripping away the identifying subscript. We say that two sites execute the same protocol if their local protocols can be derived from the same generic protocol by adding the site identification (and possibly renaming the messages and states[2]).

[2] Any renaming must be one-to-one and onto.

3.2.3. Nondeterminism

Local protocols are nondeterministic: for the same state and the same set of messages, more than one transition may be enabled. This is the case in the slave protocol for the two-phase commit (Figure 3.1) where upon receiving the transaction in the initial state, the site can choose to either accept it or unilaterally abort. This is one source of nondeterminism in the model: it allows sites to make local decisions about the transaction based on criteria external to the model (e.g. the concurrency control mechanism).

There is another source of nondeterminism within local protocols: messages can arrive in any order. When two different sets of outstanding messages enable two different transitions at a site, then either transition may occur. This source of nondeterminism is not explicitly illustrated in the state diagram for local protocols.

Both sources of nondeterminism result from the unpredictability of the external environment (e.g. the concurrency control mechanism or the underlying communication network). Hence, a local protocol must perform correctly for all possible execution paths. Local protocols differ from classical nondeterministic automata in this respect: classical nondeterministic automata require only one execution path to be correct. Nondeterminism in classical automata does not reflect unpredictability in the external environment, but instead it allows multiple alternatives that can be explored in parallel.

The difference between the classical nondeterminism behavior and the nondeterminism behavior of local protocol is a fundamental difference. Fischer [EDEN80] has coined the term, *indeterminism*, for the latter. Within failure prone systems, there is another source of indeterminism: the indeterminism caused by failures. We explore this in a later section.

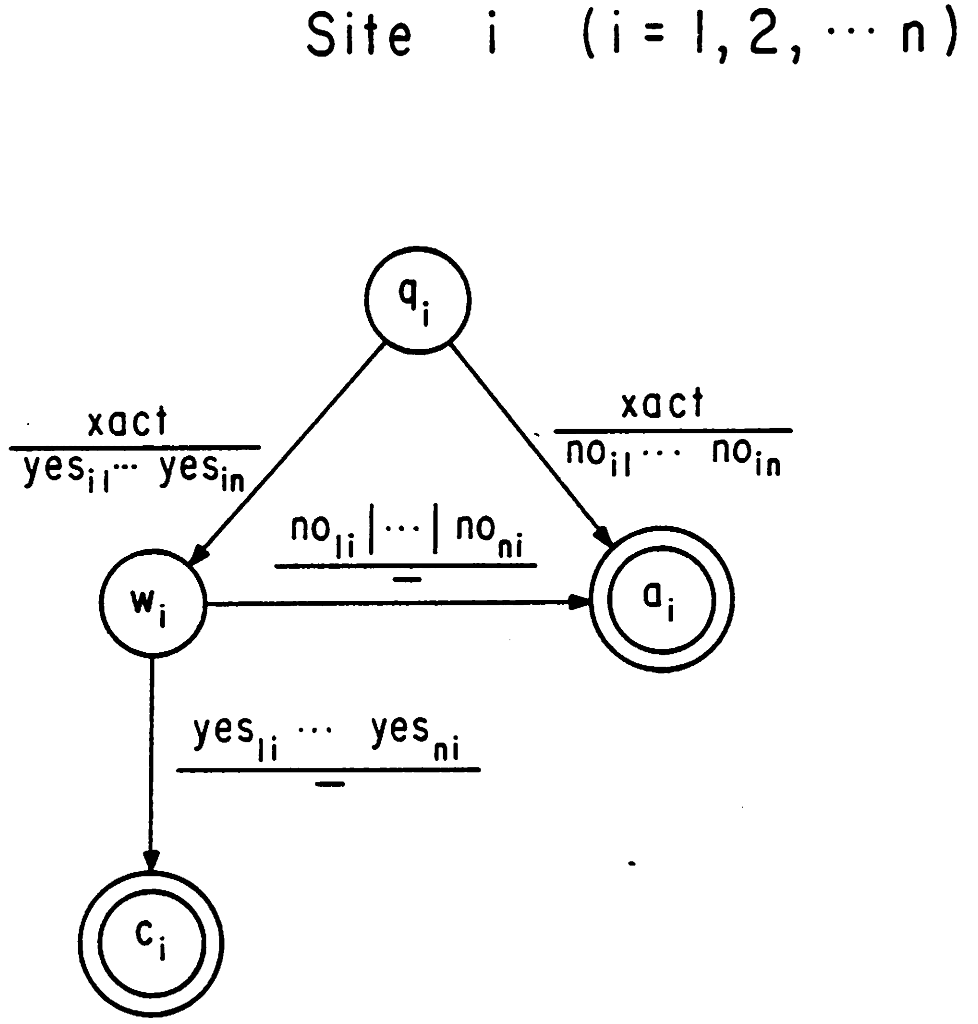

Site i (i = 1, 2, ⋯ n)
q_i
$\frac{\text{xact}}{\text{yes}_{i1} \cdots \text{yes}_{in}}$
$\frac{\text{xact}}{\text{no}_{i1} \cdots \text{no}_{in}}$
$\frac{\text{no}_{1i} | \cdots | \text{no}_{ni}}{-}$
w_i
a_i
$\frac{\text{yes}_{1i} \cdots \text{yes}_{ni}}{-}$
c_i

Figure 3.2 The decentralized two-phase commit protocol.

3.2.4. More Examples of Local Protocols

Figures 3.2 and 3.3 contain two more examples of commit protocols. For the first time multisite (>2 sites) protocols are depicted: the first is illustrated for n sites; the second, for 4 sites. Normally, messages are doubly subscripted to indicate the sender (first subscript) and the receiver (second subscript). However, for centralized protocols, where all messages involve the coordinator, the coordinator's subscript is often omitted for convenience (as was done in Figure 3.3).

The first figure illustrates the local protocol for the decentralized two-phase commit protocol. Except for the number of messages sent and received during each phase (message round), its structure is identical to the (centralized) two-phase commit. This is not coincidental: the two protocols are fundamentally the same except for their communications topology. In both protocols, the votes (i.e. acceptance or rejection) of each site must be communicated to every other site. In the decentralized protocol, the votes are communicated directly; whereas, in the centralized protocol all communication is channeled through the coordinator.

Figure 3.3 contains the local protocols for SDD-1's four phase commit protocol (see chapter 2). This protocol is significantly more complex than the two-phase protocols, primarily because it contains three generic local protocols rather than two. Hence, the coordinator has two distinct groups to direct and this requires four phases. The slave protocol for SDD-1 is identical to the slave protocol in the two-phase commit. The backup coordinator differs from previous local protocols in two ways: it can not unilaterally abort, and it has no final states. Since the backup does not store data, its abort and commit states are not irreversible.

3.3. Modeling Global Processing

3.3.1. The Global Transaction State

The _global state_ of a distributed transaction is defined to consist of:

(1) a global state vector containing the states of the local protocols,

(2) the outstanding messages in the network.

The global state defines the complete processing state of a transaction.

A _global state transition_ occurs whenever a local state transition occurs at a participating site. Barring site failures, this is the only time that global state transitions occur. Since we are assuming that local state transitions among different sites are mutually exclusive in time, exactly one global transition occurs for each local transition.

If there exists a global transition from global state **g** to global state **g'**, then **g'** is said to be _immediately reachable_ from **g**. A global state, together with the definition of the protocol, contains the minimal information necessary to compute all of its immediately

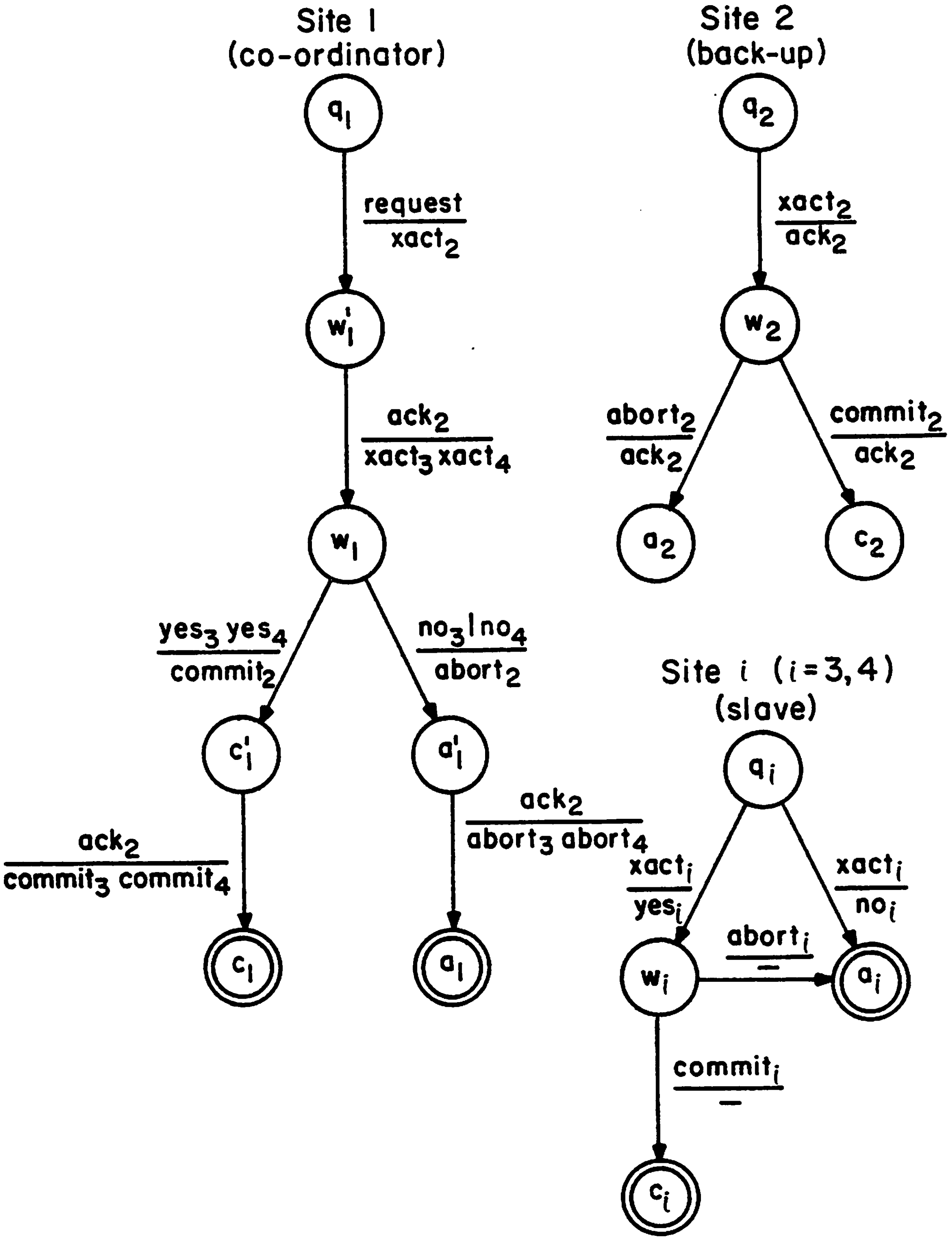


Figure 3.3 The SDD-1 four-phase commit protocol.

reachable states. The transitive closure of the immediately reachable relation yields all reachable states. In Figure 3.4 we illustrate the reachable global state graph for the two-phase protocol discussed earlier.

The reachable global state graph is an invaluable tool in analysis and verification. The graph is easy to generate

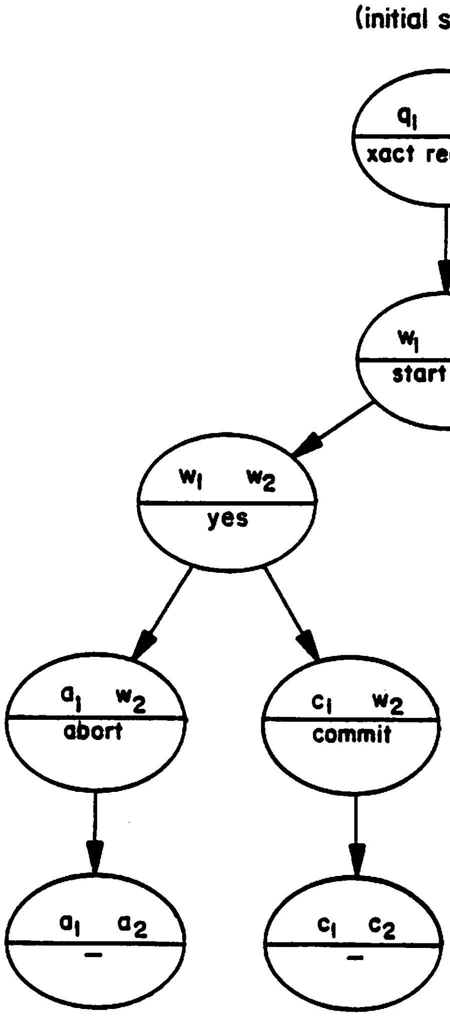


Figure 3.4 Reachable state graph for the two-phase commit protocol.

automatically, but can be quite large (exponential in N, the number of sites). Fortunately, a small N usually serves to illustrate a protocol, and proofs seldom require the generation of the entire graph.

Within a graph, a _terminal_ state is one with no reachable successors. Moreover, a path from the initial global state to a terminal global state in the reachable state graph corresponds to a possible execution sequence of the protocol.

A global state is said to be a _final state_ if all local states contained in the state vector are final states. A global state is said to be _inconsistent_ if its state vector contains both a commit state and an abort state.

A protocol is _functionally correct_ only if its reachable state graph contains no inconsistent states and all terminal states are final states. On the other hand, if the graph for a protocol contains terminal states that are not final states, then it is possible for some sites to never commit or abort the transaction. Figure 3.2 verifies that, in the absence of failures, the two-phase protocol is correct.

3.3.2. The Concurrency and Sender Sets

Two local states are said to be _potentially concurrent_ if there exists a reachable global state that contains both local states.

We now define two sets that will be used extensively in subsequent proofs. Both sets are easily constructed from the global state graph.

Definition. Let s be an arbitrary local state. The concurrency set of a local state s is the set of all local states that are potentially concurrent with it. We denote this set by C(s).

From the reachable state graph for the two-phase protocol given in Figure 3.2, we see that the concurrency set for w_1 consists of $\{q_2, a_2, w_2\}$.

When a site makes a transition from state s to state t, it is convenient to consider the messages received and sent during the transition as being "received" and "sent" by state s. For example, in the two-phase commit we would say that the coordinator's wait state, w_1, sends commit messages that are received by the slaves' wait states, w_i. We are interested in all of the local states that can send messages which are received by a given state, s.

Definition. Let s be an arbitrary local state, and let M be the set of messages that are received by s. The sender set for s, denoted S(s), is $\{t \mid t$ "sends" m and m in M$\}$.

3.2.4. Committable States

A local state is called committable if occupancy of that state by any site implies that all sites have voted yes on committing the transaction (i.e. no site has unilaterally aborted). A state that is not committable is called noncommittable[3]. In the two phase protocol of Figure 3.2, the only committable state is the commit state (c_i); all other states are noncommittable.

For most protocols, the classification of local states into committable and noncommittable states is obvious. When it is not obvious, the classification can be deduced from the global state graph.

3.3.4. Causality

Within the model the notion of one event "occurring before" another event is extremely important. For example, the choice of a recovery protocol may depend on whether failure A occurred before failure B. Fundamentally, we are interested in ordering events insofar as the notion of causality is preserved. Clearly, event A can affect event B only if A occurs before B.

We now formally define the occurs before relation, denoted by "→". The events of interest are local state transitions and failures (which will be discussed in a later section).

The relation "→" on a set of transitions is the smallest relation satisfying the following three conditions (these conditions are taken from [LAMP78b]): (1) if A and B are transitions at the same site, and A comes before B, then $A \rightarrow B$, (2) if the transition A sends a message that is received by transition B, then $A \rightarrow B$, (3) if $A \rightarrow B$

[3] To call noncommittable states abortable would be misleading since a transaction that is not in a final commit state at any site can still be aborted. In fact, sometimes transactions in committable (but not commit) states will be aborted because of failures.

and B → C, then A → C. Two distinct transitions A and B are said to be _concurrent_[4] if neither A → B nor B → A.

Another way of viewing the definition is to say that A → B means that it is possible for event A to causally affect B. It is impossible for two concurrent events to causally affect one another.

In the two-phase commit protocol, slave i unilaterally aborting and slave j unilaterally aborting are concurrent events and hence there is no causal relationship between the two. However, if only one unilaterally aborts, say slave i, then this will cause the coordinator to abort and eventually slave j will abort upon receiving the abort message from the coordinator. Slave i's abortion not only "occurred before" slave j's abortion but in fact caused it.

The relation "occurred before" is a partial ordering and is less restrictive than the ordering defined by a path through the global state graph. The latter ordering is a total ordering, imposing an arbitrary ordering on concurrent transitions. Since concurrent transitions can not causally affect one another, the graph ordering can _not_ be detected by the participating sites.

[4] The term _concurrent_ may not be the most appropriate term, but historically it is the term most commonly used. The notion we are trying to convey is that there is no causal relationship between the events. Here concurrent is being used in a somewhat different context than its use in "concurrent states." The latter term refers to states occupied simultaneously by different sites, and there could be (and often is) a causal relationship between those states.

3.4. Site Failures

In this section we discuss ways to extend the model to include site failures. Models for network partitioning are presented in a subsequent chapter, where the effects of partitioning are examined in more detail.

A failure of any type is normally detected by the absence of an expected message. We assume that each site has at its disposal an interval timer allowing it to bound the time it waits for the receipt of a message. When the timer expires, the site is said to have "timed out" and may take appropriate action. This is modeled by _timeout_ messages that are received like any other message and can cause a state transition.

Site failures are modeled by a _failure transition_, which is a special kind of local state transition ([MERL76]). Such a transition occurs _at the failed site_ the instant that it fails. The resulting local state is the state that the failed site will initially occupy upon recovering. A failure transition can move the site to a new state in the commit protocol or move it to a special recovery protocol. This model assumes that a site can detect when it has failed.

Let us momentarily assume that state transitions are atomic even in the presence of failures. Hence, a failure cannot occur during the middle of a transition and interrupt the sending of messages. In this case a failure transition can be simply defined -- it reads all outstanding messages and sends a _timeout_ message to all participants.

To illustrate the use of failure and timeout transitions, consider enhancing the two-phase commit protocol of Figure 3.1 so that it will be resilient to failures of the coordinator (Site 1). Failure transitions must be added to Site 1, and timeout transitions to slaves. Figure 3.5 illustrates one assignment of these transitions that yields the desired resiliency. (Messages have been elided for clarity.) Since no failure transitions have been added to the slaves, the behavior of the protocol is undefined when one of them fails. The correctness of the protocol can be verified by constructing the global state graph; however, its correctness is dependent on atomic state transitions.

The allowable failure transitions for a given protocol is dependent on the state information that is available during recovery. For example, if all state information is kept in volatile storage, then each failure takes the site back to the initial state. To accurately model such a system, failure transitions must be constrained to terminate at the initial state. On the other hand, if state information is kept on stable storage, then the state occupied at the time of failure is available to the recovery protocol and a failure transition can be a function of that state.

If it desirable for more information to be made available to the recovery protocol, then the implementation can log every message it sends. Assuming a "write ahead" log is used, the log entry for a message is written to stable storage immediately before it is sent. During recovery, only the sending of the latest message is in doubt.

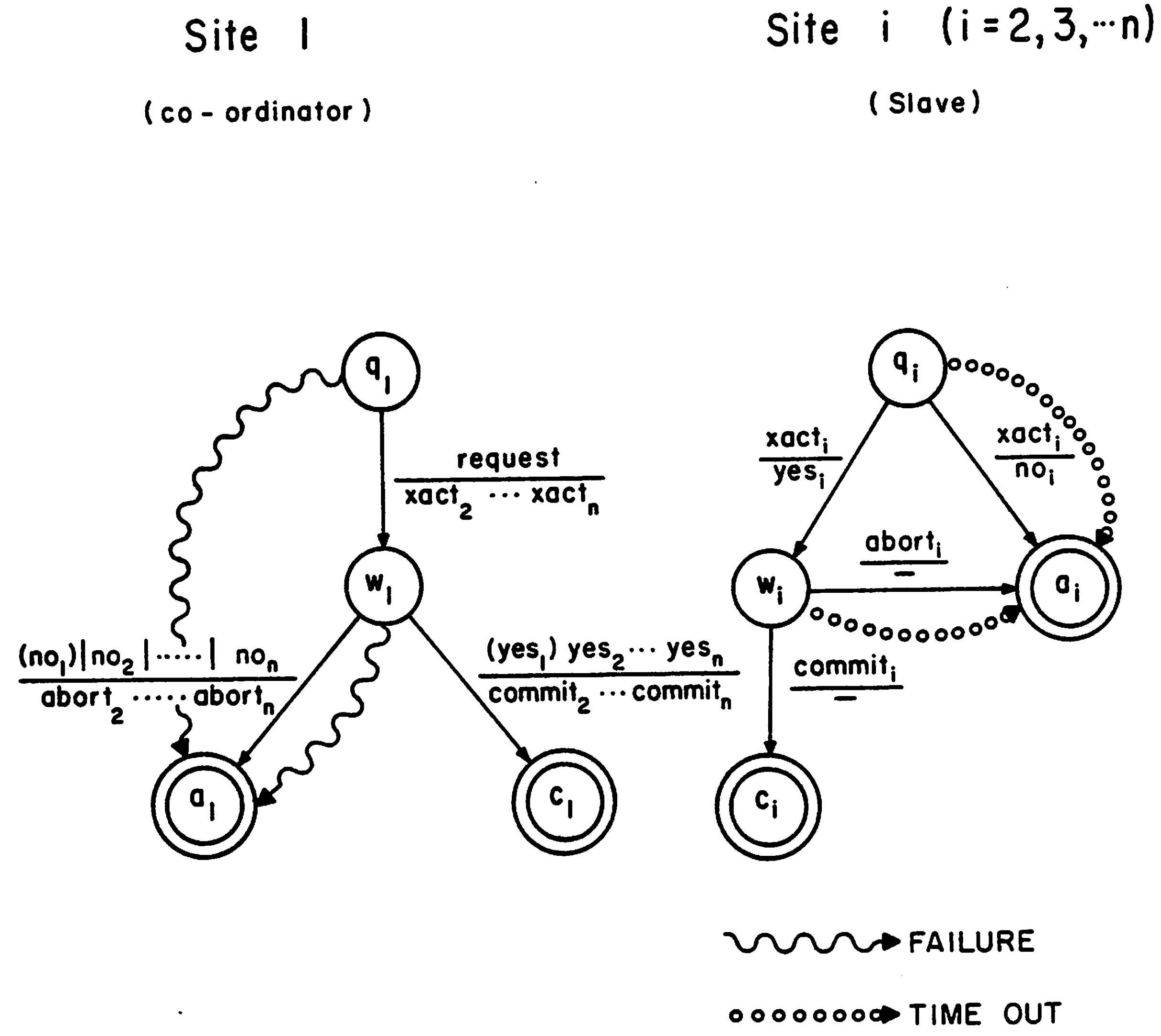
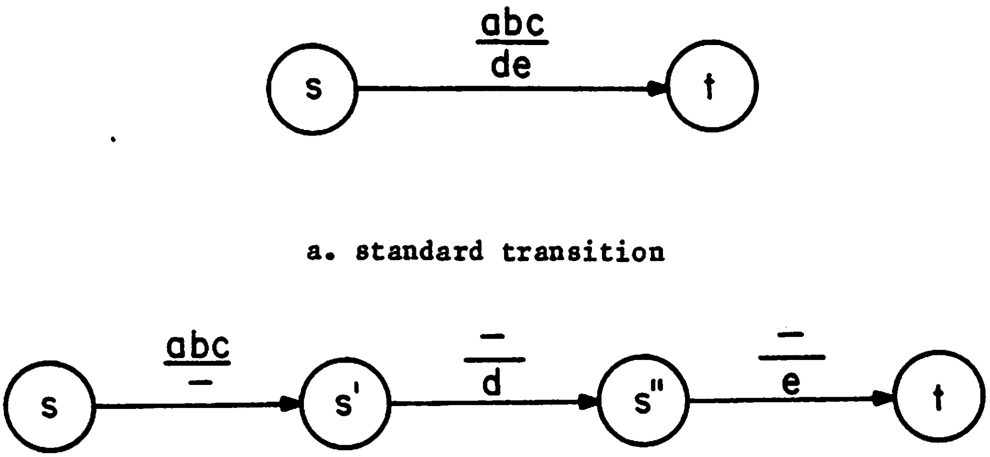


Figure 3.5 The two-phase commit protocol with failure transitions added to the coordinator and timeout transitions added to the slaves.

Since this implementation introduces new and distinct processing states into the protocols -- states that are recognized by the recovery protocol -- the automata should be expanded so that the sending of each message requires a transition into a new state. Figure 3.6 illustrates how this can be done for a single transition sending multiple messages.

In real systems, state transitions are not atomic and sites can fail after sending only a few of the messages associated with a transition. This can modeled simply by allowing a failure transition to send any prefix of the messages normally sent by a valid transition from the same state. These messages are sent in addition to the timeout messages sent by the failure transition. We also allow multiple failure transitions from the same state and these may terminate in different states. This generalization of failure transitions is sufficiently powerful to accurately model the behavior of any implementation of a FSA.



a. standard transition

b. expanded to include the effects of logging

Figure 3.6 Modeling the effects of logging messages before sending them.

3.5. Previous Work

Models for specifying and verifying protocols have been an active research topic for the past several years. The vast majority of the work on modeling protocols has been in the area of network communication protocols, and in particular, verifying the ARPANET protocols. Sunshine presents a good survey of this area ([SUNS79]). Most of the models have been extensions of either Petri nets ([GOST74, MERL76, MERL78]) or finite state machines ([AHO79, BOCH77a, ZAFI79]). Other models include the UCLA graph model ([POST74]), path expressions ([CAMP74]), language models ([LAMP78a, LELAN78]), and a unified model which augments a finite state model with variables and programming constructs ([BOCH77b]). In general, the results are only remotely applicable to our work: from our perspective, these are the underlying communication protocols which we do not explicitly model. However, some of the approaches in error modeling are quite general. Merlin proposed failure transitions in his model using Petri nets ([MERL76]).

Aho, et al. proved some very interesting results concerning the transmission of error-free messages ([AHO79]). They used a deterministic finite state automata model to prove the correctness of a resilient protocol for physically sending bits across a communications line. Their results parallel some of the results presented in this thesis. One of the main differences between the two approaches is our use of indeterminism.

In the area of transaction management protocols, very little formal work has been done. The concurrency control protocols of SDD-1 were formally proved using a graph model ([BERN80]). Issues of reliability were not addressed in that proof and the proof techniques are not appropriate for verifying commit protocols. (In SDD-1, the commit and recovery protocols are entirely disjoint from the concurrency control protocols.) Ellis has a interesting paper using L-systems to prove the correctness of a concurrency control mechanism ([ELLI77b]). He also proves that the protocol is resilient to site failures. However his protocols are for completely replicated data and do not allow unilateral aborts.

Baer et al. ([BAER80]) uses a petri net model to prove the correctness of the two-phase commit protocol. This paper also demonstrates the difficulty of the approach -- the specification of the protocol alone took two pages and over 60 states.

CHAPTER 4

The Existence of Resilient Protocols

In this chapter we investigate existence questions concerning resilient protocols for different classes of failures. One interesting question is: "Can we design commit protocols (with corresponding termination protocols) with enough information kept in each site's local state so that a failed site may recover exclusively on its local state information?" Another question is: "what is the most robust protocol in the presence of network partitions?" Current protocols perform poorly under partitioning, which suggests that this problem is inherently difficult and may not lend itself to robust protocols.

This chapter answers these questions. The existence (nonexistence) proofs presented serve as important paradigms for proofs of similar problems, especially, for network broadcast protocols.

4.1. Independent Recovery

Independent recovery refers to a scheme where a recovering site makes a transition directly to a final state without communicating with other sites. Only local state information is used during the recovery process. Therefore, recovery is independent of any event after the site's failure.

Independent recovery is interesting for several reasons. First, it is easy to implement and leads to simple protocols. One need not be concerned with messages to a failed site being queued in the network or at another site which may be down when the failed site attempts to recover. This recovery strategy is of theoretic interest because it represents the most pessimistic recovery strategy: proving the existence of a class of resilient protocols using independent recovery implies the existence of resilient protocols in all more sophisticated strategies of site failures. Its most important practical aspect is that it qualifies the usefulness of local state information during recovery: if the local state proves to be insufficient for resilient recovery, then operational sites will have to provide a history of the completed transaction. This history will have to be maintained indefinitely until all sites have recovered and completed the transaction. In this regards, independent recovery provides the only true nonblocking recovery strategy -- any strategy requiring a history mechanism will necessarily block when the history becomes temporarily unavailable due to failures.

When discussing independent recovery, we will restrict our attention to the two site case. All of the results are easily extended to the multisite case. The multisite case is not presented here because it requires extra notation and the resulting protocols are of little practical importance.

We will use failure and timeout transitions as discussed in Chapter 3. By definition, failure transitions in an independent recovery scheme terminate in a final transaction state. Since we are dealing exclusively with the two-site case, timeout transitions also will terminate in a final state. (In the multisite case, timeout transitions would "invoke" a termination protocol.)

4.1.1. Failure of a Single Site

Let us first consider the simple case where at most one site fails during a transaction. Our goal is to develop rules for assigning failure and timeout transitions to existing protocols to form protocols resilient to single site failures.

Not all protocols can be made resilient as the next lemma demonstrates.

Lemma 4.1. If a well-formed protocol contains a local state with both abort and commit in its concurrency set, then under independent recovery, it is not resilient to an arbitrary failure of a single site.

Proof. This follows directly from the definition of "concurrency set." Consider a local state, s_i, and its concurrency set, $C(s_i)$. Let $C(s_i)$ contain both an abort state and a commit state. Clearly, s_i cannot have a failure transition to the commit state, since the other site may be in the abort state. Similarly, s_i can not have a failure transition to the abort state, since the other site may be in the commit state. Hence, when Site i is in s_i, it can not safely and independently recover.

If a protocol has no local states violating the necessary condition in the above lemma, then failure transitions can be assigned according to the following rule.

> **Rule 1.** For every intermediate state, s, in the protocol: if C(s), contains a commit, then assign a failure transition from s to a commit state; otherwise, assign a failure transition from s to an abort state.

The (centralized) two-phase commit protocol does not satisfy the condition in the lemma: the concurrency set of the slave's wait state (p_2) contains both c_1 and a_1. Notice that p_2 is the only local state violating this rule, and this occurs because the coordinator moves into the commit state before the slave acknowledges committing the transaction. If instead, the coordinator moves to a prepared state while it is waiting for the acknowledgement from the slave and moves into a commit state only after acknowledgement is received, then it is possible to assign a failure transition to p_2. This "extended" two-phase commit protocol is shown in Figure 4.1 and its (reachable) global state graph is shown in Figure 4.2. It is easy to verify from the global graph that the concurrency set for each state, including the prepared state, contains only one kind of final state. Hence, failure transitions satisfying Rule 1 can be defined for each state. The assignment of failure transitions is depicted in Figure 4.3 (timeout transitions, to be discussed below, are also illustrated).

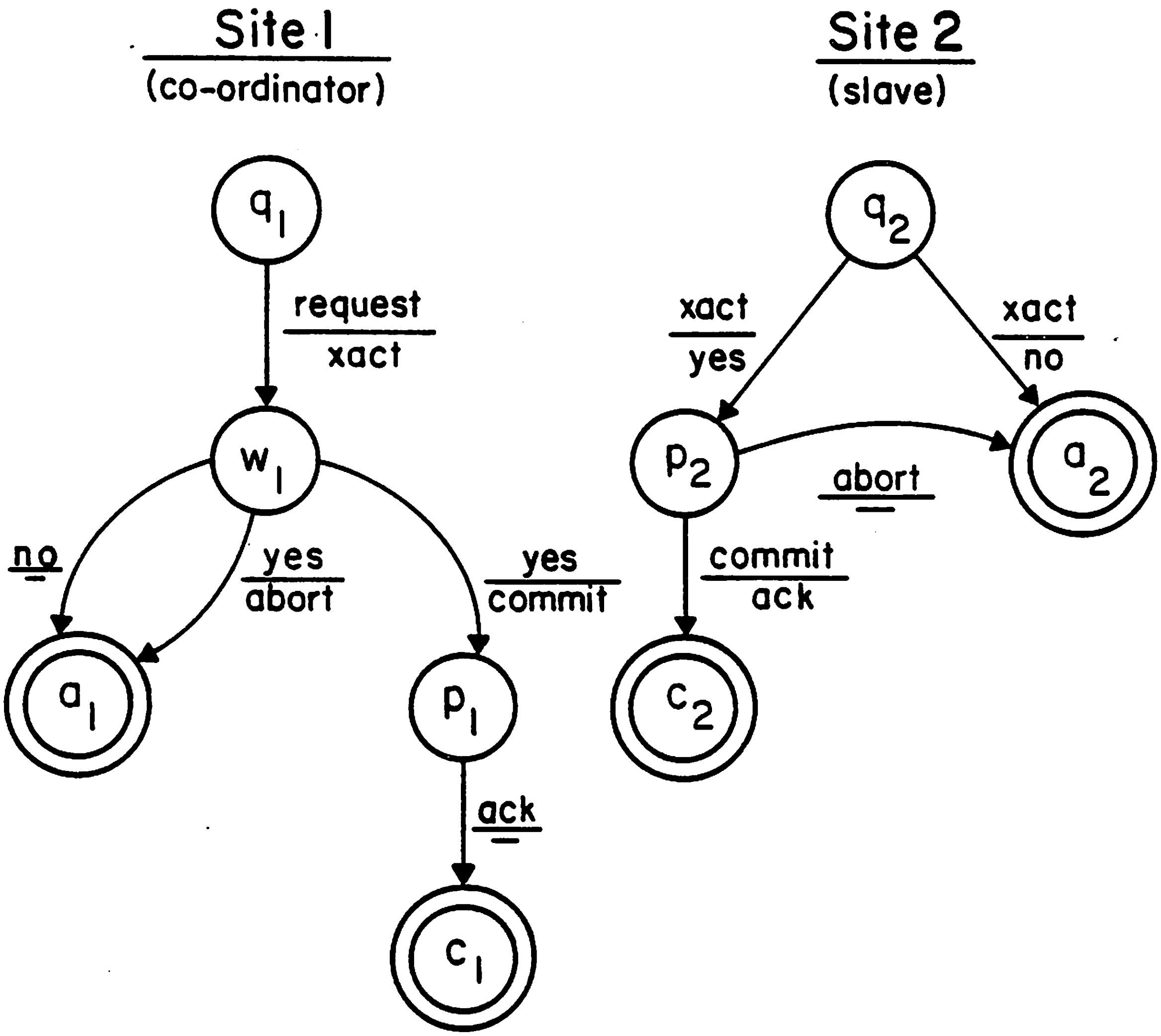


Figure 4.1. The extended two-phase commit protocol.

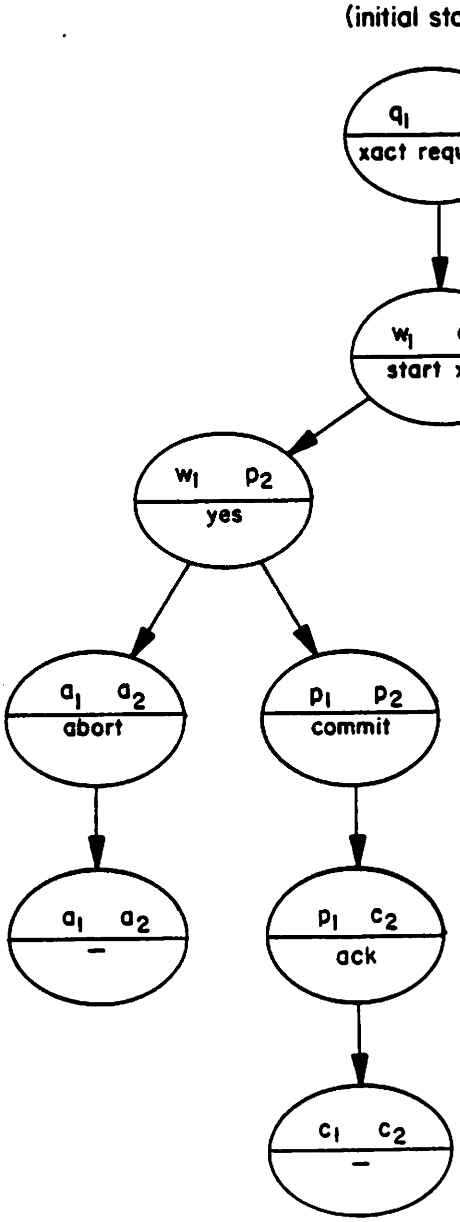


Figure 4.2. The reachable global state graph for the protocol of Figure 4.1.

The second rule deals with timeout transitions.

Rule 2. For each intermediate state s_i: if t_j is in $S(s_i)$ (the sender set for s_i) and t_j has a failure transition to a commit (abort) state, then assign a timeout transition from s_i to a commit (abort) state.

This rule is less obvious than the previous one. A "timeout" can be viewed as a special message sent by a failed site in lieu of a regular message. Like any other message received by state s_i, it must have been "sent" by a state in the senders set for s_i. Moreover, the failed site, using independent recovery, has made a failure transition to a final state. Hence, the receiving state must make a consistent decision.

Timeout transitions for the extended commit protocol are illustrated in Figure 4.3. By assigning failure and timeout transitions according to Rules 1 and 2, we have derived a protocol that is resilient to a single failure by either site. This can be verified by examining its reachable state graph. In fact, these rules always yield a resilient protocol under independent recovery.

Theorem 4.2. Rules 1 and 2 are sufficient for designing protocols resilient to a single site failure.

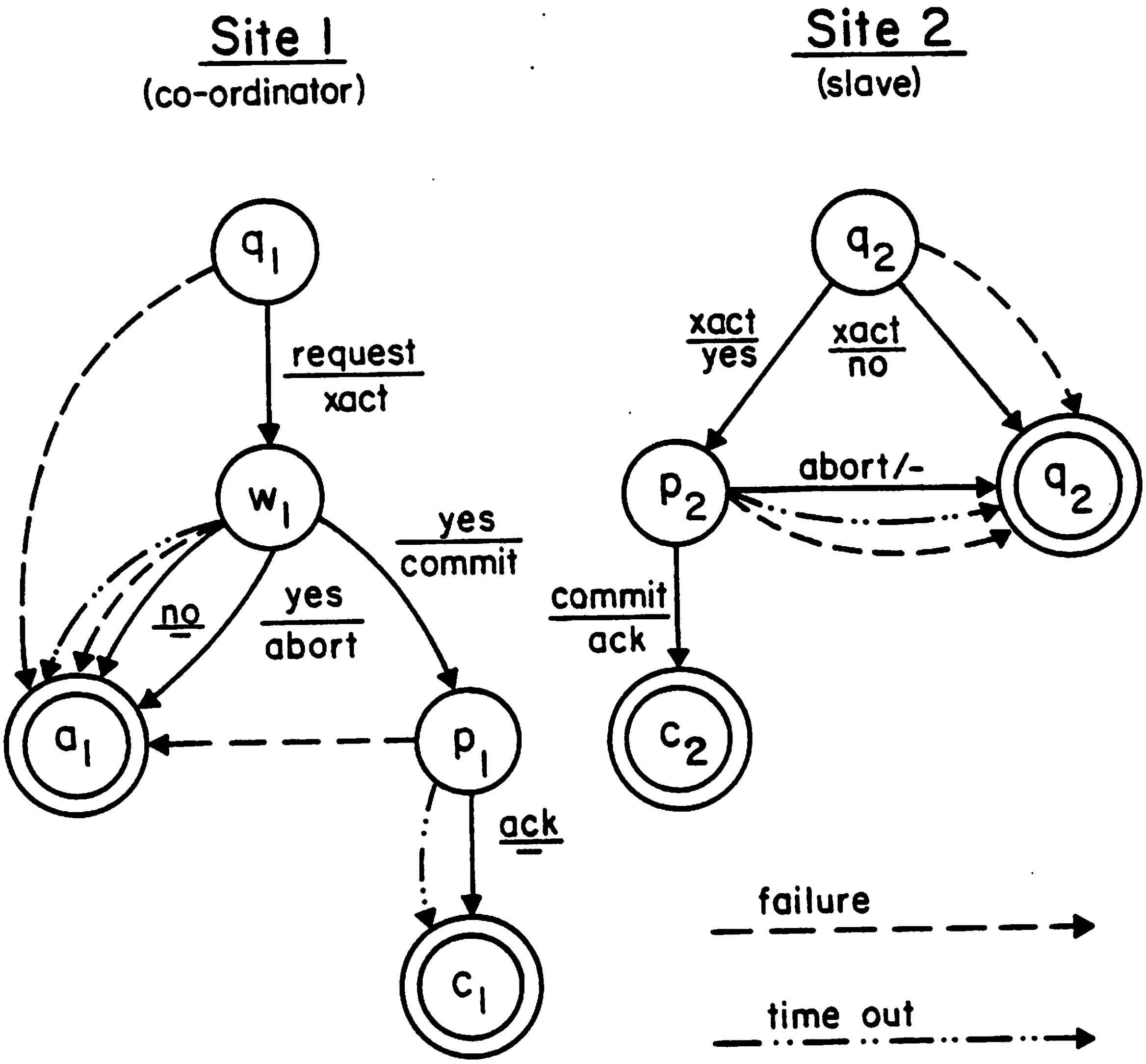


Figure 4.3. The protocol with failure and timeout transitions obeying Rules 1 and 2.

Proof. Let P be a protocol with no local states having a concurrency set containing both a commit state and an abort state. Let P' be the protocol resulting from assigning failure and timeout transitions to P according to the above rules. Now, the proof proceeds by contradiction. We will assume that P' is not resilient to all single site failures. Therefore, there must exist a path from the initial global state to an inconsistent final global state, and this path contains exactly one failure. Without loss of generality assume Site 1 fails, and let it fail in state s_1. Let the inconsistent global state contain the final states: f_1 and f_2. Hence, upon failing, Site 1 made a failure transition from s_1 to f_1. There are two cases depending on whether Site 2 is in a final state or a nonfinal state when Site 1 fails.

Case 1. Site 2 is in the final state f_2. But this implies that f_2 is in $\underline{C}(s_1)$. Therefore, rule 1 is violated.

Case 2. Site 2 is in a nonfinal state. Upon failing Site 1 sends a "timeout" to Site 2 which is received by Site 2 in state s_2. Now, Site 2 makes a timeout transition to f_2 which is inconsistent with f_1. However, observe that, by definition, s_1 is in the sender set of s_2. Therefore, Rule 2 has been violated.

4.1.2. Two Site Failures

The rules given above are sufficient for protocols resilient to a single failure; however, such protocols are not necessarily resilient to the failure of two sites. This can be demonstrated for the protocol of Figure 4.3. If double failures occur when Site 1 is in state p_1 and Site 2 is in state p_2, then an inconsistent final state results. Furthermore, for this protocol, this is the only possible assignment of failure and timeout transitions that satisfy both rules. Hence, this protocol can not be made resilient to two failures.

Can the protocol be extended to deal with double failures? The next theorem yields a negative answer.

Theorem 4.3. There exists no protocol using independent recovery of failed sites that is resilient to two site failures.

Unfortunately, the results of this theorem not only applies to the 2 site case, but applies to the multisite case as well. The proof assumes N sites (N>1).

Proof. Let P be a protocol that always preserves consistency in the absence of failures. Let i and j be two sites. We will show: for every failure-free execution of P that commits the transaction, there exists a point in the execution where a failure of i followed by a failure of j leads to an inconsistency.

A failure-free execution of P corresponds to a path in the global state graph, $G_0, G_1, \ldots, G_m$ where G_0 is the initial global state and G_m is a final global state. We are assuming that G_m is a final commit state. A global state consists of a local state vector and a network state; however, for the remainder of this proof, we will ignore the network state. Hence, we view the global state, G_k, as a vector $<s_{k1}, s_{k2}, \ldots, s_{kN}>$, where s_{ki} is the local state for site i when transaction execution is in state G_k. Let f(s) be the result of making a failure transition while in state s. Now, let F_k be equal to G_k except that that s_{ki} and s_{kj} are replaced by $f(s_{ki})$ and $f(s_{kj})$. Hence, F_k is the global state resulting from a failure of i and j when processing is in G_k. Now, let us examine the sequence $F_0, \ldots, F_m$, and in particular, the local states for i and j in this sequence. The pair $(f(s_{0i}), f(s_{0j}))$ must be equal to (a_i, a_j) since a site will always abort the transaction when it fails in the initial state. Similarly, the pair $(f(s_{mi}), f(s_{mj}))$ must equal (c_i, c_j) since we have assumed that the transaction was committed. Now let k be the smallest k such that either $f(s_{ki})$ or $f(s_{kj})$ yields a commit state. This situation is depicted below:

Commit Sequence (G_1)	**Global State (F_1) Resulting from the Failure of i and j**
$<\ldots, s_{0i}, \ldots, s_{0j}, \ldots>$	$<\ldots, a_i, \ldots, a_j, \ldots>$
$\vdots$	$\vdots$
$<\ldots, s_{k-1,i}, \ldots, s_{k-1,j}, \ldots>$	$<\ldots, a_i, \ldots, a_j, \ldots>$
$<\ldots, s_{ki}, \ldots, s_{kj}, \ldots>$	F_k where either $f(s_{ki})$ or

$f(s_{kj})$ is a commit state.

$\vdots$ $\qquad\qquad$ $\vdots$

$<\ldots,s_{mi},\ldots,s_{mj},\ldots>$ $\qquad$ $<\ldots,c_i,\ldots,c_j,\ldots>$

Since each global transition reflects one local state transition, two adjacent global states differs by exactly one local state. Therefore, either $s_{k-1,i}=s_{ki}$ or $s_{k-1,j}=s_{kj}$ and, thus, either $f(s_{k-1,i})=f(s_{ki})$ or $f(s_{k-1,j})=f(s_{kj})$. Therefore, $f(s_{ki})$ or $f(s_{kj})$ is an abort state. By assumption, the other one is a commit state. Hence, F_k is inconsistent.

4.2. Network Failures

A network failure results in at least two sites which cannot communicate with each other. We model such a partition in two ways. In the first model, all messages are lost at the time partitioning occurs. In the second, no messages are lost at the time partitioning occurs; instead, undeliverable messages are returned to the sender.

We define a simple partition as one where all sites are partitioned into exactly two sets with no communication possible across the boundary. Since all partitions can be viewed as one or more occurrences of a simple partition, we specifically address two classes of failures: a single occurrence of a simple partition, and multiple occurrences of a simple partition (or multiple partition for brevity).

We consider a protocol to be *resilient* to a network partition only if it enforces the *nonblocking* constraint, that is, the protocol must ensure that each isolated group of sites can reach a commit decision consistent with the remaining groups. Since the commit decision within a group is reached in the absence of communication to outside sites, this problem is very similar to the independent recovery paradigm presented in the previous section.

Throughout this section we will restrict our attention to network partitions exclusively and ignore the possibility of site failures.

4.2.1. Partitioning With Lost Messages

As previously, a site detects the occurrence of a partition by a *timeout* and can make a transition on such a message. First, we treat the two site case.

A network partition is modeled as a special type of global state transition. Until now all global state transitions were triggered by one local state transition. However, a network partition is modeled as a global state transition that erases all outstanding messages and "timeouts" are sent to all sites.

After a partition has occurred, each site will make a "timeout" transition. In fact, we have a situation analogous to the double site failure in the independent recovery model of the previous section. The difference is: upon double failures, sites make "failure" transitions; whereas, upon a partitioning, sites make "timeout" transitions. It can be shown that a solution to the double failure problem implies a solution to the simple partitioning problem. An immediate consequence of this result is is the next theorem.

Theorem 4.4. There exists no two site protocol resilient to a network partition when messages are lost.

Sketch of Proof. This can be shown using the proof of Theorem 4.3. as a paradigm. We will restrict our attention to only two sites. Let f(s) represent the result of a timeout transition from state s. Since messages are lost when the partitions occur, we can ignore the message state portion of a global state. Let $G_0, \ldots, G_m$ be a partition-free execution of the protocol that commits the transaction. Define F_i to be the global state resulting from a network partition occurring in state G_i. Using the notation of the previous theorem, $F_i=<f(s_{i1}),f(s_{i2})>$. As in Theorem 4.3, we can find the smallest k such that F_k contains a commit state. (Recall that F_0 contains only abort states.) Now, the difference between F_{k-1} and F_k is one local state. Therefore, F_k must contain an abort state as well, which makes the state inconsistent.

It is straightforward to generalize the theorem to handle the multisite (N>2) case as we did in the proof of the independent recovery theorem.

4.2.2. Partitioning with Return of Messages

In this situation we assume that the network can detect the presence of a partition and return undeliverable messages to their senders. This appears to represent the most optimistic model for partitions, while loss of messages is the most pessimistic one.

In this case a partition causes a global state transition that redirects all undeliverable messages back to their senders and writes timeout messages to the recipients of undeliverable messages. As before, a site can make a transition on a timeout message. Also, a site makes a transition when an undeliverable message is returned to it.

4.2.2.1. Two Site Case

To study this optimistic situation, we now define two design rules that resilient protocols must satisfy.

> **Rule 3.** For a state s_i: if its concurrency set, $\underline{C}(s_i)$, contains a commit (abort) state, then assign a _timeout_ transition from s_i to a commit (abort) state.

Here site i in state s_i was expecting a message when the partition occurred; instead, it received a "timeout". This site will then make a decision to abort or commit the transaction consistent with the state of the site sending the undeliverable message.

The second rule deals with the site sending the undeliverable message. It must make a commit decision consistent with the decision of the intended receiver.

Rule 4. For state s_j: if t_i is in $\underline{S}(s_j)$, the sender set for s_j, and t_i has a timeout transition to a commit (abort) state, then assign an undeliverable message transition from s_i to a commit (abort) state upon the receipt of an undeliverable message.

An observant reader will note that these rules are equivalent to the rules given for independent recovery of failed sites. In fact, the two models are isomorphic. To illustrate the equivalence, consider the information conveyed by a timeout message from a failed site. The following is true when the operational site, i, receives a timeout indicating the failure of the other site.

(1) the last message sent by site i was not received (the other site failed prior to its receipt),

(2) communication with the other site is impossible (it is down),

(3) the other site will decide to commit using independent recovery.

Exactly the same conditions hold when an undeliverable message is returned to site i.

Applying the above design rules to the protocol of Figure 4.3 yields the protocol illustrated in Figure 4.4. As expected, the protocol is identical, to the protocol of Figure 4.3.

In light of this isomorphism, theorem 4.5 is not surprising.

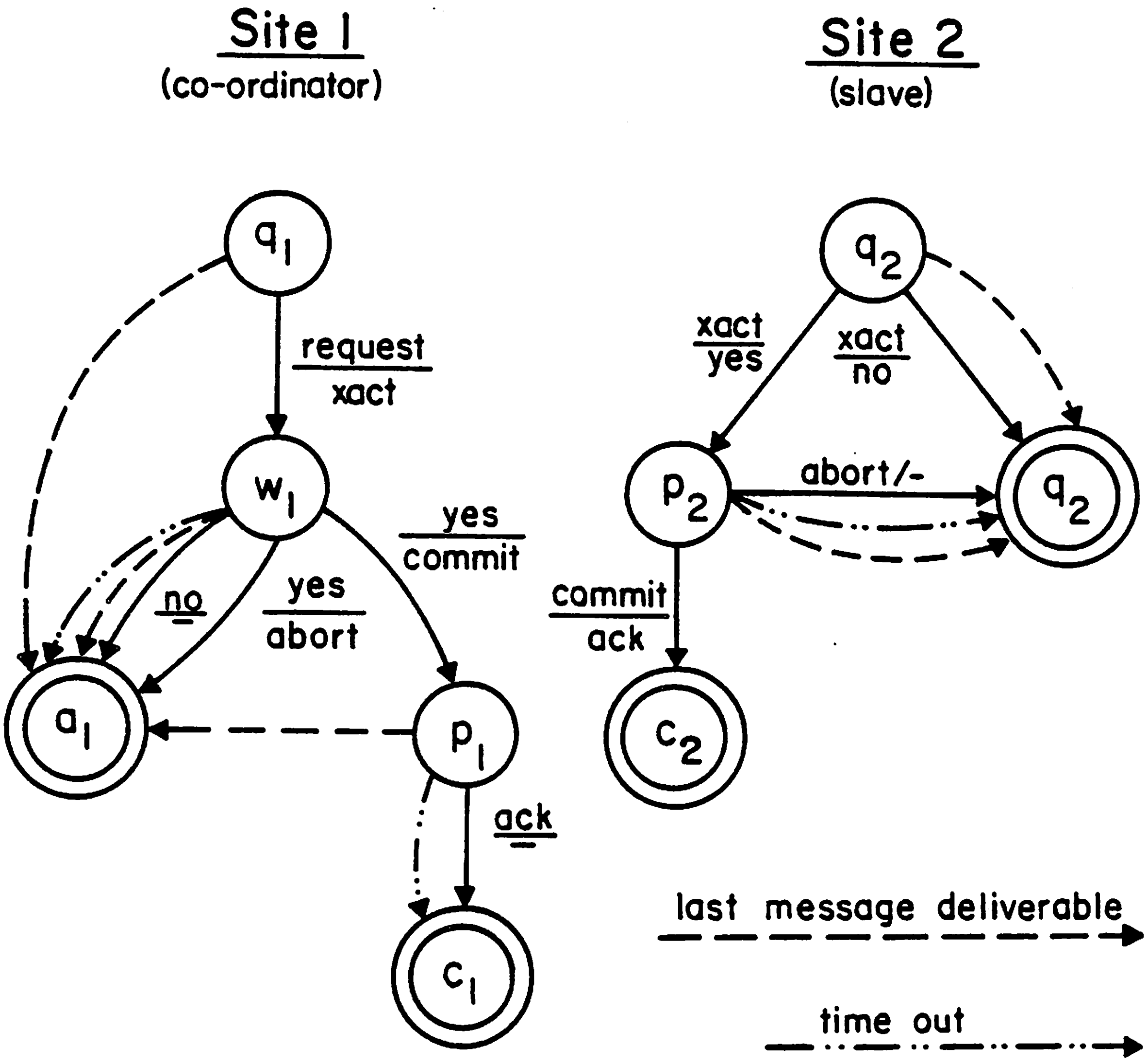


Figure 4.4. The extended two-phase commit protocol (of Figure 4.1b) augmented with timeout transitions and undeliverable message transitions according to Rules 3 and 4.

Theorem 4.5. Design Rules 3 and 4 are necessary and sufficient for making protocols resilient to a partition in a two-site protocol.

Sketch of Proof. In light of the aforementioned isomorphism, we can make use of the proof of Theorem 4.2. In that proof, substitute "undelivered message" for every occurrence of "timeout" and substitute "timeout" for every occurrence of "failure." Finally, substitute "Rule 3" for "Rule 1" and "Rule 4" for "Rule 2." The result is a proof for Theorem 4.5.

4.2.2.2. The Multisite Case

In the absence of site failures, the multisite case is very similar to the two-site case, since preserving consistency within a connected group of operational sites is not difficult. Thus, design rules 3 and 4 can be extended to multisite protocols in a straightforward way. This leads to the following result.

Corollary 4.6. There exist multisite protocols that are resilient to a **simple** partition when undeliverable messages are returned to the sender.

This result is the complement of the results obtained from the pessimistic model discussed earlier. The models differ in their handling of outstanding messages when the network fails: in the pessimistic model, they are lost; whereas in the optimistic model, they are returned to their sender. Since this is the only difference between the two models, the next result is implied.

Corollary 4.7. Knowledge of which messages were undelivered at the time the network fails is necessary and sufficient for recovering from simple partitions.

We now turn to multiple partitions. Since we are dealing with an optimistic situation, we assume that timeouts and undeliverable messages are unaffected by additional partitions. This, in effect, is an assumption that the network is partitioned into all subsets simultaneously, and that the process does not happen sequentially.

Even in this (overly) optimistic model, our results are negative, which implies negative results for all realistic partitioning models.

Theorem 4.8. There exists no protocol resilient to a multiple partition.

In the proof of this theorem we will only consider protocols in which each state transition reads at most one message (however, a transition can still send an arbitrary number of messages). From the previous chapter, we know that for every protocol, P, there exists protocol P' that satisfies this requirement. Furthermore, P' is no less resilient than P, because P' can simulate every timeout or "undeliverable message" transition made by P.[1] This assumption allows a simpler proof. The proof follows the same form as previous nonexistence proofs.

[1] Note that P' will have at least as many local states as P. Therefore, it is straightforward to show that P' can simulate any transition made by P. Again, this is discussed in Chapter 3.

Proof. Let P be a three-site protocol that is correct in the absence of failures. We will assume that P is resilient to multiple partitions. Let $G_0,\ldots,G_m$ be a failure free path in the global state graph for P that commits the transaction. Now, $G_i=(S_i,M_i)$, where $S_i=<s_{i1},s_{i2},s_{i3}>$ is the vector of local states and M_i is the outstanding messages. We will consider M_i to be the union of three sets: M_{i1}, M_{i2}, and M_{i3}, where M_{ij} is the set of messages addressed to Site j.

Let G'_i be the global state resulting from a partitioning occurring during the global state G_i. Without loss of generality, we assume that all outstanding messages are returned to their senders. The recipients will receive only timeouts. Let M'_i be the resulting set of messages. Thus, $G'_i=(S_i,M'_i)$. Now, let f_j denote the transition function that moves Site j to a final state during a partitioning, i.e. $f_j(s_{ij},M'_{ij})$ is Site j's resulting final state when a partitioning occurs during G_i.

Let k be the smallest k such that a multiple partitioning occurring while the transaction is in G_k still results in the transaction being committed. Since we have assumed that the protocol is resilient to such a partitioning, we have $f_j(s_{kj},M'_{kj})$ equals commit for j=1,2,3. Moreover, by our judicious choice of k, we have $f_j(s_{k-1,j},M'_{k-1,j})$ equals abort for j=1,2,3. Now from G_{k-1} to G_k one state transition occurred, and let us assume that Site 1 made that transition. Furthermore, Site 1 read at most one message and, and if so, let this be a message from Site 2.

Notice that we have $s_{k-1,3}=s_{k3}$ and $M'_{k-1,3}=M'_{k3}$ since, between G_{k-1} and G_k, Site 3 did not make a transition and none of its messages were read. Therefore, $f_3(s_{k-1,3},M'_{k-1,3})=f_3(s_{k3},M'_{k3})$. But this is a contradiction.

Therefore, even complete information about message traffic during a partition, and in particular, information about which messages are undeliverable, is insufficient for recovering from multiple partitions.

4.3. Conclusions

The results of this chapter define fundamental limitations on the robustness of protocols with respect to both site failures and network partitions, where the measure of robustness is whether a protocol is nonblocking with respect to the studied class of failures. In general, the results tend to be more illuminating than surprising.

For site failures, the recovery strategy studied was *independent recovery*, where a site recovers using only the local information available at the time of its failure. The alternative to this strategy is to maintain a history, either of the outcome of the transaction at operational sites or of the messages sent to failed sites.

Recovery will be based on this information. History-based schemes are inherently blocking -- the components maintaining the history may fail -- and are more costly to implement.

The results show that independent recovery is resilient to a single failure but not for more than one failure. From the proof of the theorem for the multisite case, we observe that the most difficult failures to recover from are concurrent failures. In general, the database is left in an inconsistent state if independent recovery is attempted on concurrent failures. Since a recovering site can not deduce from its local state whether another site's failure was concurrent with its failure, it is impossible for the site to determine when it is safe to use independent recovery.

While nonblocking recovery protocols have been examined in detail in this chapter, nonblocking commit protocols have not. An important question is whether nonblocking commit protocols exist. This question is treated in the next two chapters.

The results on robust network protocols are more discouraging than the independent recovery results. Even under the idealistic assumption that every site can unmistakenly determine which of its messages were correctly delivered prior to partitioning, resilient protocols exist only for simple partitionings. In more realistic network environments, where a partition can result in lost messages, there exists no nonblocking protocols -- not even for "simple" partitions. At best, we can design protocols which allow one group of sites to continue while the remaining groups block.

CHAPTER 5

Processing in the Presence of Site Failures

When sites fail two different sets of problems must be addressed: (1) those faced by operational sites and (2) those faced by failed sites when they attempt to recover. This chapter deals with the former problems; the next chapter deals with the latter problems.

The first difficulty an operational site faces is the restructuring of the communication topology. For example, when the coordinator fails in a centralized protocol, the whole communication structure of the protocol collapses. The slaves must establish a new communication structure before they can continue processing. One way to do this is to elect a new coordinator; however, this is not the only course available to them.

The second problem faced by the operational sites is that of consistently terminating the transaction. Our major concern is that the transaction can be safely terminated without waiting for the recovery of failed site(s). Of secondary importance is whether the transaction is actually committed or aborted.

When the transaction can not be safely terminated, the operational sites must block until some of the failed sites have recovered.[1] Blocking is undesirable because locks on the database

[1] Actually, there is a second strategy. The operational sites

must be held while the transaction is pending. A good inverse measure of the robustness of a protocol is the number of global states of a protocol where site failures can cause it to block. Protocols that never require an operational site to block are called nonblocking protocols, and in this sense they are maximally robust.

In this chapter we prove a simple but powerful result: the fundamental nonblocking theorem. It describes sufficient properties for designing nonblocking protocols. From this theorem we derive the canonical three phase commit protocol which is the simplest nonblocking protocol. We then give examples of nonblocking protocols, which are derived from the canonical three phase protocol.

Within this chapter, we make the following assumptions:

(1) State transitions are not atomic. Instead, a site may fail in the middle of sending the messages associated with a normal transition. The protocols discussed must deal with this possibility. Failures are modeled formally by failure transitions as described in Section 3.4.

(2) We ignore the possibility of other types of failures (i.e. lost messages and network partitioning). A subsequent chapter discusses those failures.

could terminate the transaction, and if at some later time an inconsistency is discovered at a failed site, steps could be taken to repair the inconsistencies. Since a major premise throughout this thesis is that consistency must be conserved at all times, we do not explore this possibility. Since inconsistencies are, in general, hard if not impossible to repair, this is not an attractive alternative.

5.1. Commit and Termination Protocols

Some site failures are more difficult to handle than others. For example, the failure of the coordinator in the two-phase commit protocol is very hard to handle, since all communication involves the coordinator. Its failure requires the invocation of a special protocol which establishes communication among the slaves and then attempts to terminate the transaction (perhaps polling the sites to reach a commit decision). We reserve the term, termination protocol, for these special purpose protocols, and use the term, commit protocol, for the protocol used in the absence of failures. The term, recovery protocol, (properly, a site recovery protocol) is reserved for the protocol used by a site while recovering.

In contrast to the failure of the co-ordinator, the failure of a slave can easily be handled within the two-phase commit protocol and does not require the invocation of a specialized termination protocol. If the slave fails before voting, then the transaction is aborted; otherwise, the transaction is committed or aborted in the usual way and the slave is told about the decision upon recovering. This strategy works for a slave because a slave communicates with a single site, the co-ordinator. As a rule of thumb, the more sites that a given site communicates with, the harder it is to recover from its failure.

A nonblocking commit protocol is used in conjunction with a nonblocking termination protocol to correctly terminate transactions at operational sites. Given formal properties ensuring nonblocking behavior, a commit protocol and termination protocol can be designed independently. The design of a termination protocol need only depend on the formal properties of nonblocking protocols and not on the characteristics of a particular commit protocol.

Design independence provides for separation of "concerns" between the two types of protocols. Since the commit protocol is invoked for every transaction, it should be designed for speed and to consume a small percentage of the network bandwidth. On the other hand, the termination protocol is invoked only when a failure occurs (and this should be rare); therefore, the its design objectives may be quite different than for those for the commit protocol. For example, ease of implementation may be a significant factor in choosing a termination protocol.

This suggests that the commit protocol and termination protocol can be of different communication topologies. For some network environments, a centralized commit protocol (which has low message overhead) could be used with a decentralized termination protocol (which is easy to implement).

5.2. Properties of Nonblocking Protocols

In the this section we will be interested in defining the properties of nonblocking protocols -- properties applying to both commit and termination protocols. Throughout the discussion, we will ignore recovery protocols for failed sites.

5.2.1. Definition of Nonblocking

Informally, a protocol that is nonblocking on site failures never requires an operational site to wait until a failed site has recovered. As stated in the introduction to the chapter, only sites that have never failed during the processing of the transaction are considered "operational sites."

A protocol is formally defined to be nonblocking (on site failures) if for all possible executions, including all combinations of failures: the protocol correctly terminates the transaction at all operational sites in a bounded number of messages, and it does not require any messages (except for timeouts) to be sent by a site after failing. This is a strong notion of nonblocking.

Generally, we will be interested in a weaker notion of nonblocking where the protocol is not required to terminate the transaction, but only to leave it in a state where it can be terminated in a bounded number of messages from operational sites only. The commit protocols discussed will be nonblocking in this weaker sense: we do not require them to terminate in all cases. In contradistinction, a termination protocol is required to terminate in all cases, hence, by definition, they must be nonblocking in the strong sense.[2]

[2] One could argue that a commit protocol and a termination protocol actually are components of a larger protocol that is nonblocking in the strong sense. However, we are interested in designing commit protocols completely independent of any specific termination protocol; thus, we are interested in what it means to be nonblocking in the weak sense.

A commit protocol is shown to be nonblocking in the weak sense, if for each of its reachable global states there exists a nonblocking termination protocol that can be invoked from that state. Of course, existence is usually demonstrated by giving a protocol, but this is not necessary. On the other hand, showing that a protocol is a blocking protocol in the weak sense is often very hard since the nonexistence of an appropriate nonblocking termination protocol must be shown. A protocol is shown to be nonblocking (or blocking) in the strong sense by examining its global state graph.

5.2.2. Blocking Protocols: An Example

Before discussing nonblocking protocols, it will be instructive to examine why a particular protocol blocks. We will use the (centralized) two-phase commit as an example. Let there be N sites and let the transaction have progressed to the point that all sites have voted on the transaction. A slave is aware only of its own vote (as reflected in its state). Now let the coordinator (Site 1) fail, followed soon afterward by the failure of the first slave (Site 2). The remaining N-2 sites communicate among themselves and discover that all of them voted yes, and none of them received a commit (or abort) message from the coordinator. The operational sites can not safely proceed because from their collective local states they can not distinguish the following two sequences of events:

(1) Site 2 unilaterally aborted the transaction, and

(2) Site 2 accepted the transaction, the coordinator received the "yes" vote and failed after sending a commit message to Site 2. Site 2 received the message, committed and failed.

Notice that the operational sites must block regardless of the state (c_1 or w_1) the coordinator assumes when recovering.

5.2.3. The Fundamental Nonblocking Theorem

In the above scenario, each of the sites was in the wait site (w_i). This state contains insufficient information to allow the transaction to continue, because its concurrency set contains both the abort state and the commit state for the failed slave. Clearly, if the operational sites are to terminate the transaction in a consistent state, then one of their local states must preclude either the abort state or the commit state (i.e. the concurrency set of at least one state must contain only commit or abort but not both). Furthermore, if the abort state is precluded, then at least one of the operational sites must be in a committable state. (Recall that the occupancy of a committable state implies that all sites have accepted the transaction.)

Let us formalize the above observations. First, we must describe the behavior of failed sites. The state of a failed site is defined to be abort or commit if the site had aborted or committed before failing; otherwise, its state is failed. This characterization of failed sites precludes the use of independent recovery; however, from the previous chapter, we know that independent recovery is not resilient to more than one failure. Instead, we assume that a recovery protocol requiring communication with operational sites is used.

We now define the notion of a safe local state. The definition uses concurrency sets (see section 3.3.2) which are defined over all possible states, including the failed state.

Definition. A local state, s, is _safe_ if and only if it satisfies one of the following conditions:

S1: C(s) does not contain a commit state, or

S2: s is a committable state and C(s) does not contain an abort state.

By convention, the "failed" state is safe.[3]

Clearly, for any correct protocol, abort and commit are safe. Intuitively, an operational site in a safe state can safely make a decision to commit or abort the transaction.

A global state is _safe_ if and only if at least one operational site is in a safe local state. A global state is _completely safe_ if and only if all sites are in safe states.[4]

[3] This convention is primarily for convenience. It alleviates the need of saying "either in a safe or the failed state" during our subsequent discussions.

[4] Notice that a transaction is in a (completely) safe state if all sites fail during its execution. This is a consequence of our convention that the failed state is safe.

The term "completely safe" suggests that a transaction can be safely terminated even in the presence of failures. This is indeed a correct interpretation as demonstrated by the protocol in Figure 5.1. This protocol terminates a transaction in a completely safe global state in a bounded number of messages.

Comment. every site, i, executes the following local protocol.

Let s represent the current local state.

Let s' represent the local state of the transaction at the time this protocol is invoked.

```
if s' is committable
  then s := "yes"
  else s := "no"

for 1 to N do
  if s="yes"
     then send "yes" to all sites
     else send "no" to all sites

  wait for messages from all (including failed) sites

  if a "yes" message is received
    then s := "yes"
    else s := "no"
end_for

if s="yes"
  then commit the transaction
  else abort it
```

Figure 5.1. A protocol for terminating transactions in completely safe states.

The protocol is a decentralized protocol consisting of N message rounds (where N is the number of participating sites). During each round, a site sends either a yes (to commit) or a no (to abort) or a timeout (if it has failed) to all sites, including itself. For notational simplicity, failed sites are included in rounds.

Each operational site terminates the transaction according to the last round of messages it received. A consistent decision is reached if the local states of the operational sites have converged by the last round. It is easy to show that N rounds are sufficient to ensure convergence.

Theorem 5.1. The protocol of Figure 5.1 correctly terminates a transaction in a completely safe global state. Moreover, termination is achieved in a bounded number of messages.

Proof. Bounded termination is easy to show. The protocol consists of exactly N rounds. In each round, a site sends N messages. Thus, a round consists of N^2 messages.

Now to show consistency. Note that if all sites fail before the end of the last round, then consistency is trivially preserved. Henceforth, we assume that at least one site is operational. The operational sites will consistently abort or commit if their state variables converge.

Claim 1. If their states during round i converges then they will converge in round i+1 and, by induction, in all subsequent rounds converges.

Justification. The messages sent during round i+1 are dependent solely on the local states set during the previous round. If the local states are identical during round i, then all messages sent and received during round i+1 will be identical, and once again all operational sites will move into identical states.

Claim 2. If no sites fail during round i, then at the end of the round all sites move into the same local state.

Justification. In the absence of failures, the sites receive identical messages, and by symmetry will move into the same state. (Recall, a site sends a message to itself, thus in the absence of failures, an identical set of messages is received by each site).

Since there are N rounds and at most N-1 site failures in the nontrivial case, there must be one failureless round. During that round the local states of operational sites will converge (by note 2) and remain so for the remainder of the protocol (by note 1).

This protocol is the first example of a termination protocol. It is a naive and expensive protocol and of little practical importance, but it can be used in proofs requiring the existence of a termination protocol. For this reason, it is interesting. We will use it when showing that a commit protocol is nonblocking in the weak sense.

Two corollaries of this theorem are:

Corollary 5.2. If the global state graph of a protocol contains only completely safe states, then the protocol is nonblocking.

Proof. The protocol of Figure 5.1 can be invoked at any time to terminate the transaction.

Corollary 5.3. If every local state in a protocol is safe, then the protocol is nonblocking.

Proof. If every local state is safe, then every global state is completely safe. From the previous corollary, the protocol is nonblocking.

In the blocking scenario above, we showed that some local states in the two-phase commit protocol are not safe, and thus, some global states are not completely safe. The scenario describes a blocking situation that occurred when the sites in safe states failed, leaving only sites in unsafe states.

Instead of a protocol being completely nonblocking, we may want to require that it be able to function correctly if only a bounded number, say K, of site failures occur. Let us define K-resiliency to be the property that a protocol can survive K failures without blocking ([ALSB76]). The next corollary gives sufficient conditions for this case.

Corollary 5.4. If every global state in a protocol contains K+1 safe local states, then the protocol is K-resilient.

Proof. A modified version of the above protocol can be invoked at any time and it is guaranteed to terminate the transaction if no more than K sites fail.

In the modified protocol there are only K+1 rounds. Except for the last round only sites in safe states are allowed to participate. These "safe sites" execute the first K rounds of the protocol as before. During the last round they also send messages to "nonsafe sites" who then commit or abort as before. (As before, nonsafe sites do not send messages during this round.) After these K+1 rounds, either a round containing no failures of safe sites has occurred and their states have converged, or K+1 failures have occurred.

The next theorem describes sufficient conditions for a protocol to have only safe local states.

Theorem 5.5. (the Nonblocking Theorem). If a protocol obeys the two rules:

(1) if any site has committed, then all operational sites occupy committable states,

(2) if any site has aborted, then all operational sites occupy noncommittable states,

then it is a nonblocking protocol.

Proof. The rules guarantee that all local states are safe. Consider a local state, s, and its concurrency set, C(s). By the rules given, C(s) can contain an abort state only if it is noncommittable. Similarly, C(s) can contain a commit state only if it is committable. By definition, the set of committable states and the set of noncommittable states are disjoint and include all local states. Therefore, C(s) can not contain both commit and abort, and s is safe.

Each of the two conditions of the theorem can be decomposed into two design rules for nonblocking protocols. For the first condition, these rules are:

(1.1) Before a site can commit, it must verify that all operational sites are in committable states.

(1.2) A site can move from a committable state to a noncommittable state, only if there is no possibility of another site (including a failed site) being in the commit state.

A useful implication of this theorem is the following corollary on K-resiliency:

Corollary 5.6. A protocol is K-resilient if there is a subset of K+1 sites that obey the fundamental nonblocking theorem.

Until now, we have ignored the state of the network in showing termination. Hence, the protocols use only a subset of the information that is available. It is not surprising that the properties in the above theorems are stronger than necessary (if we are considering only site failures). For completeness, we will give an example of a set of weaker properties that utilize messages on the network.

Theorem 5.7. Every nonblocking protocol exhibits the following behavior. For every Site i, when Site i makes a transition to a commit (abort) state, then for every Site j, at least one of the following must hold:

(1) Site j is in a committable (noncommittable) state,

(2) The network contains messages that will unconditionally move Site j to a committable (noncommittable) state.

(3) Site i sends a message that will unconditionally move Site j to a committable (noncommittable) state.

Proof. Constraint (1) taken alone is exactly the nonblocking theorem. Since only one transition to a final state can be made, it is not possible for both "unconditional" abort and "unconditional" commit messages for the same recipient to be present.

We can trivially modify the termination protocol of Figure 5.1 to handle protocols satisfying conditions of the theorem: the protocol simply waits one end-to-end message delay before beginning the first round. This allows any outstanding messages that enforce conditions (2) and (3) to be received. Note that after all outstanding messages have been received, every site must satisfy condition (1), because either it satisfied (1) initially or it moved to a final state. However, in a consistent protocol every final state satisfies (1).

By examining its reachable global state graph, a protocol can be checked for satisfiability of the theorem. However, unlike the previous theorem, this result requires that the state of the network be examined in addition to the local state of the participants.

Protocols designed using properties (2) and (3) exhibit weaknesses which limit the significance of the theorem. They depend on the existence of certain messages in the network, and therefore, are not resilient to the loss of an arbitrary message. Furthermore, they tend to be expensive (requiring $O(N^3)$ messages).

5.3. Nonblocking Commit Protocols

As we have demonstrated before, the canonical two-phase commit protocol is a blocking protocol. In the second phase of this protocol, sites move directly from the wait state, which is a noncommittable state, directly into the commit state. Until the phase is completed, sites will concurrently occupy the wait state and the commit state, and this violates the first rule of the nonblocking theorem. The rule itself implies that the sites must first move into committable intermediate states, before moving into the commit states. This gives rise to the canonical three phase commit protocol:

Phase 1. Each site learns of the transaction, votes, and moves into a noncommittable wait state (or unilaterally aborts).

Phase 2. If any site unilaterally aborted or failed before voting, then all operational sites move directly to an abort state; otherwise, all operational sites move to a committable intermediate

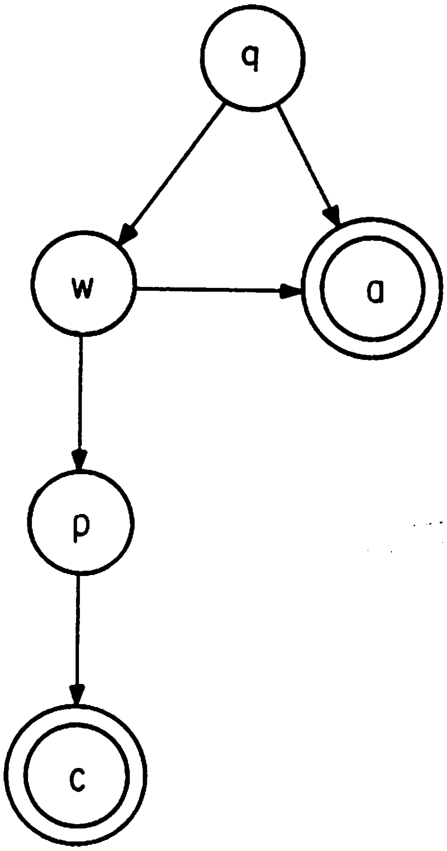


Figure 5.2. The canonical three-phase commit protocol.

state which we shall call the prepared state (p).

Phase 3. The sites commit the transaction.

The state diagram for the canonical three-phase commit protocol is given in Figure 5.2.

Notice that it is more expensive to commit (requires three phases) than to abort (requiring two phases) in this protocol. This is an inherent property of protocols allowing unilateral abort.

From the canonical three-phase commit protocol, we can derive a three-phase protocol for each distinct communication topology, including the central site, the hierarchical, the ring, and the decentralized topologies. These are illustrated in Figures 5.3 through 5.6. Each protocol exhibits only minor perturbations from the canonical protocol. In the coordinator protocols (the central site, hierarchical, and ring), the coordinator moves the other sites into the prepared state. In the decentralized protocol, the addition of the prepared state translates into another message round.

For all classes of protocols, the cost of an additional phase is high. The number of end-to-end message delays increases by 50 percent. The number of messages increases by 50 percent for the ring and decentralized protocols, and a little less (around 40 percent) for the other protocols.

The semantics of the prepared state is very simple: it is a flag to indicate that the transaction is committable. No other action is associated with it. It is important to realize that it is not a com-

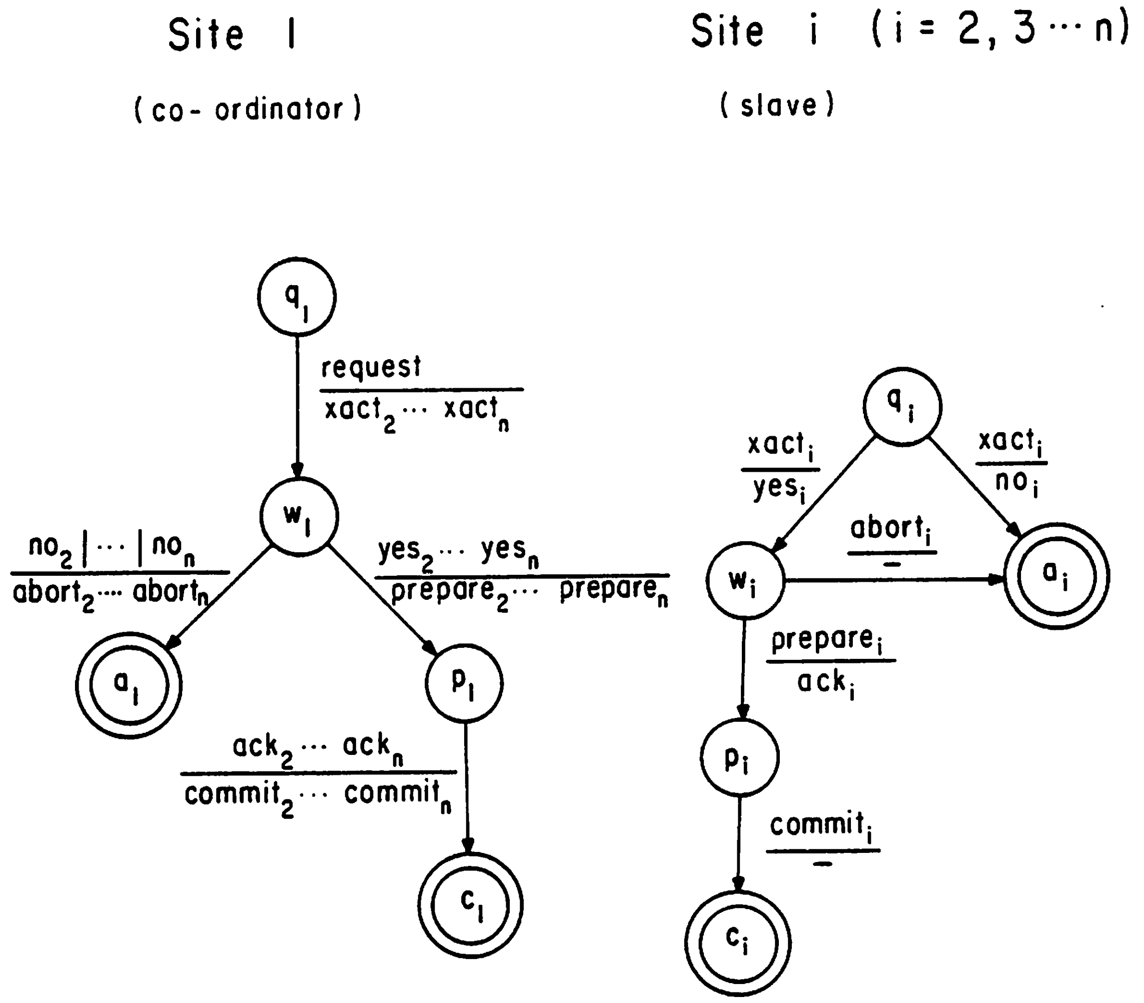
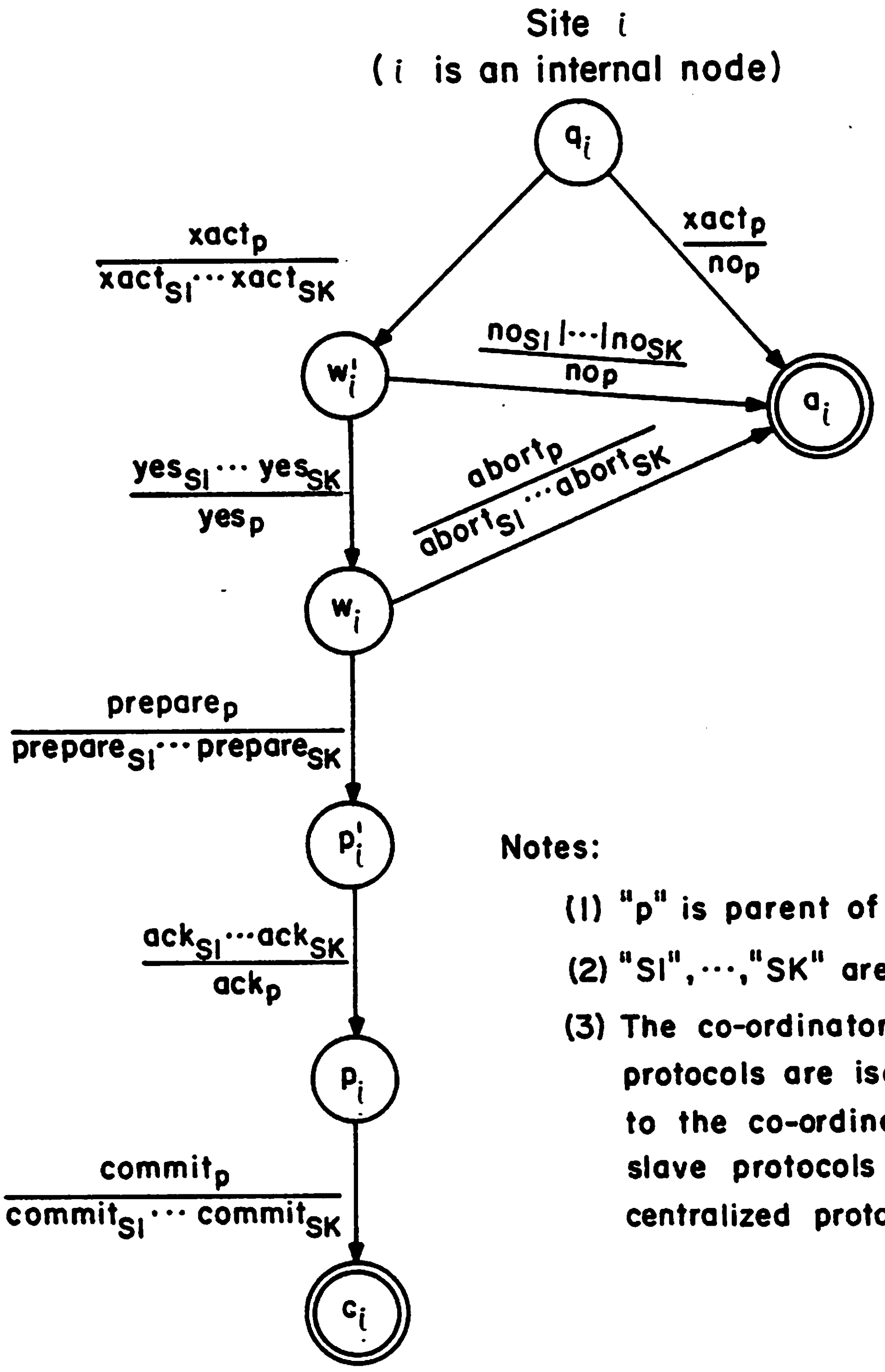

Site 1
(co-ordinator)
Site i (i = 2, 3 ··· n)
(slave)
q1
request / xact2 ··· xactn
w1
no2 | ··· | non / abort2 ··· abortn
yes2 ··· yesn / prepare2 ··· preparen
a1
p1
ack2 ··· ackn / commit2 ··· commitn
c1
qi
xacti / yesi
xacti / noi
wi
aborti / –
ai
preparei / acki
pi
commiti / –
ci

Figure 5.3. The central site three-phase commit protocol.


Site i
(i is an internal node)
q_i
$\frac{xact_p}{xact_{SI} \cdots xact_{SK}}$
$\frac{xact_p}{no_p}$
w_i'
$\frac{no_{SI} | \cdots | no_{SK}}{no_p}$
a_i
$\frac{yes_{SI} \cdots yes_{SK}}{yes_p}$
$\frac{abort_p}{abort_{SI} \cdots abort_{SK}}$
w_i
$\frac{prepare_p}{prepare_{SI} \cdots prepare_{SK}}$
p_i'
$\frac{ack_{SI} \cdots ack_{SK}}{ack_p}$
p_i
$\frac{commit_p}{commit_{SI} \cdots commit_{SK}}$
c_i

Notes:

(1) "p" is parent of i

(2) "SI",···,"SK" are sons of i

(3) The co-ordinator and "leaf" protocols are isomorphic to the co-ordinator and slave protocols in the centralized protocol.

Figure 5.4. The hierarchical three-phase commit protocol.

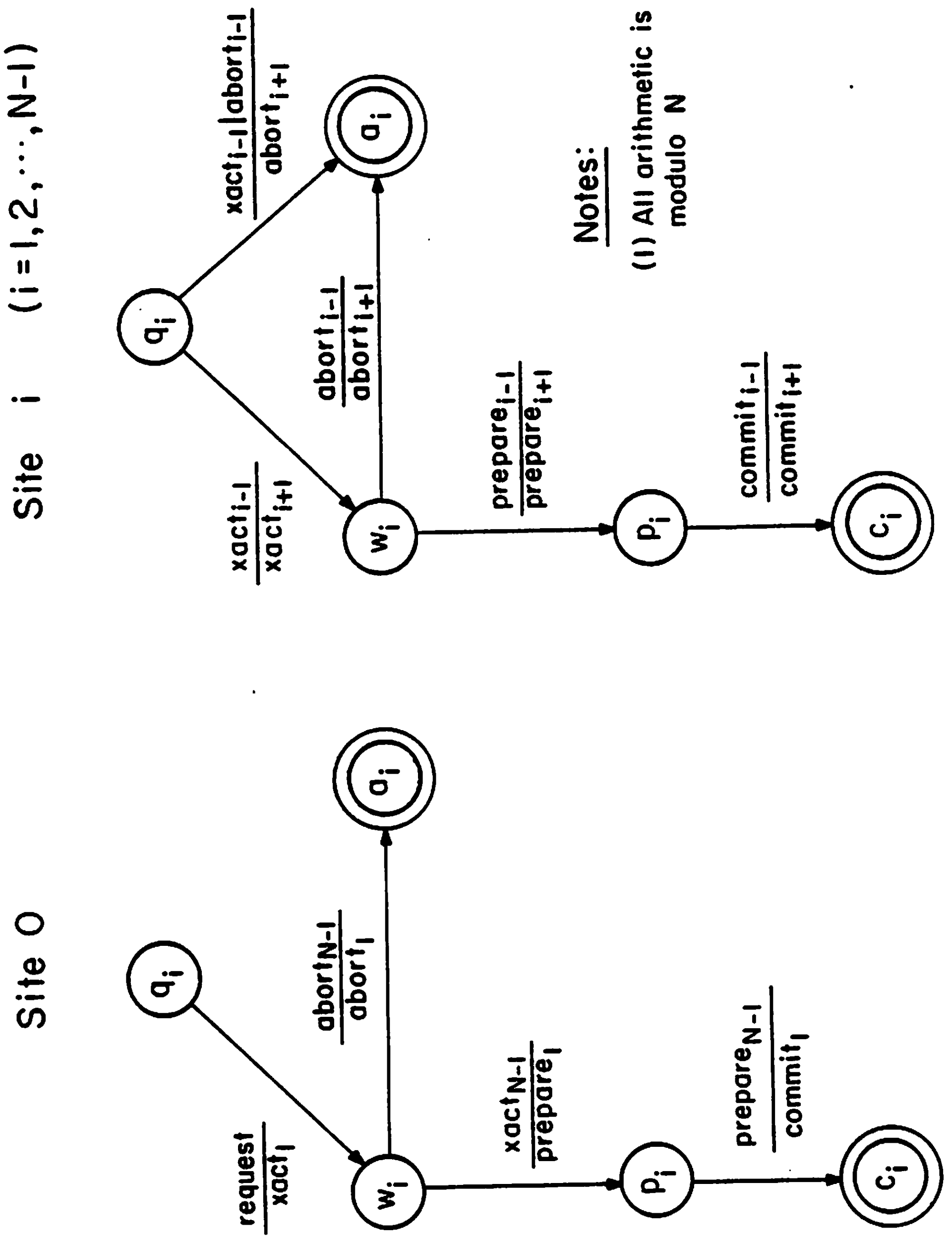


Figure 5.5. The ring three-phase commit protocol.

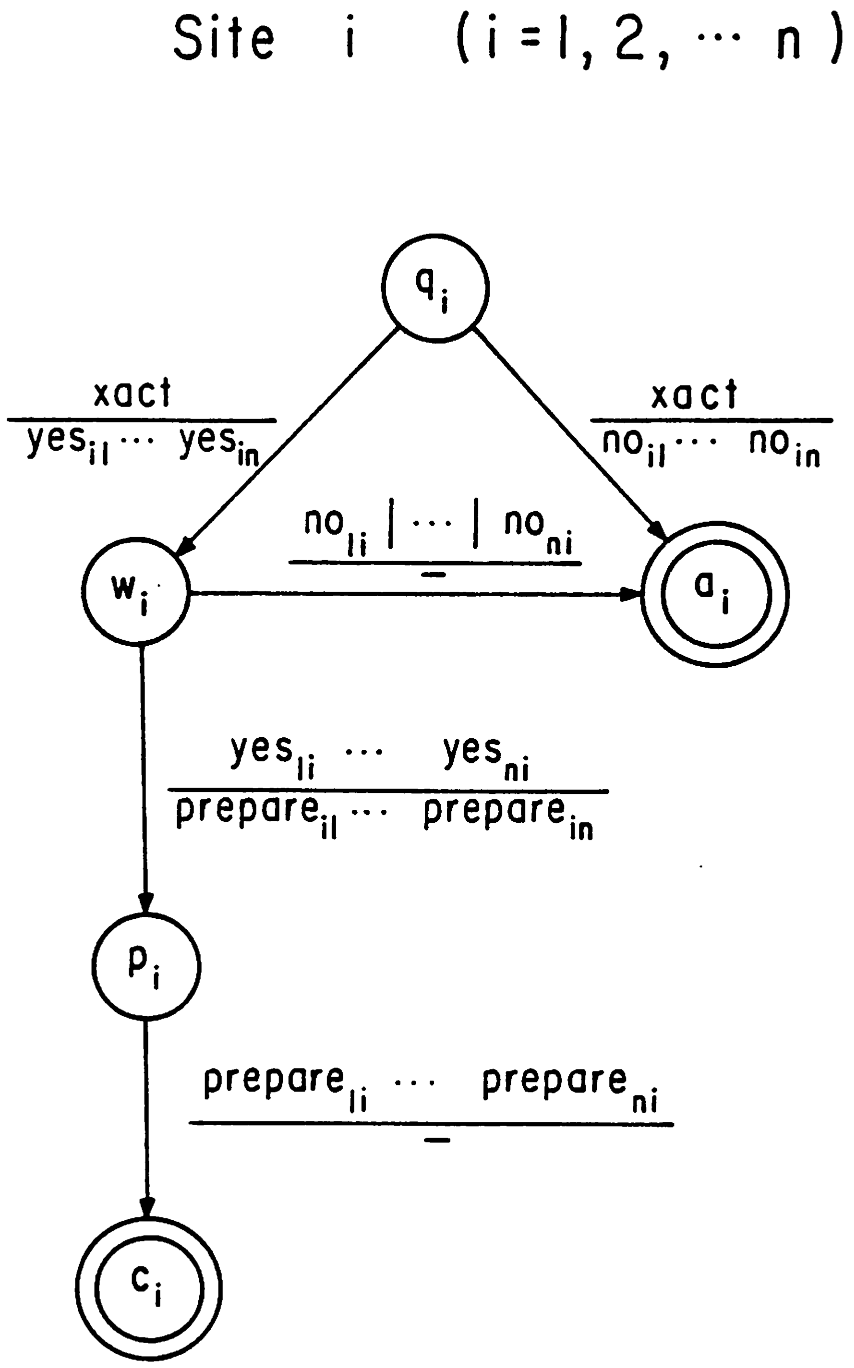

Site i (i = 1, 2, ⋯ n)
q_i
$\frac{xact}{yes_{i1} \cdots yes_{in}}$
$\frac{xact}{no_{i1} \cdots no_{in}}$
w_i
$\frac{no_{1i} \mid \cdots \mid no_{ni}}{-}$
a_i
$\frac{yes_{1i} \cdots yes_{ni}}{prepare_{i1} \cdots prepare_{in}}$
p_i
$\frac{prepare_{1i} \cdots prepare_{ni}}{-}$
c_i

Figure 5.7. The decentralized three-phase commit protocol.

mit state. Until the completion of the second phase and all sites are in committable states, a transaction may still be aborted by a termination protocol. Therefore, locks must be held throughout this state. Moreover, a site failing in the prepared state can not independently recover to a commit state.

5.3.1. A Historical Perspective

The centralized three-phase commit protocol was concurrently discovered by several researchers ([GARC79, SVOB79, SKEE81b] and others). The decentralized three-phase commit and an early version of the nonblocking theorem appeared in [SKEE81b]. Even though the three-phase commit is the cheapest nonblocking commit protocol, it has yet to be implemented in a database system.

SDD-1's four-phase protocol claims to be K-resilient where K is a constant that can be adjusted to the desired level of resiliency ([HAMM78]). The protocol was described in Chapter 2 and illustrated in Chapter 3.

A distinguishing feature of the protocol is the K back-up coordinators. We observe that the original coordinator and its backups move to a committable state during the third phase of the protocol while the slaves do not commit until the fourth phase (the back-ups themselves contain no data and hence never commit). A similar statement is true for aborted states. Therefore, the protocol satisfies the nonblocking theorem (for K sites), and is K-resilient.

5.4. Termination Protocols

The primary design criterion for a termination protocol is that it consistently terminate a transaction that is in a completely safe state. If we are interested in only K-resiliency, then it need only terminate when no more than K failures have occurred. Other design goals include: minimizing the number of messages, minimizing expected time of termination, or committing the transaction whenever it is safe to do so. These are conflicting goals; normally, only one can be achieved.

When discussing termination protocols, the following property of nonblocking protocols is helpful:

> If any operational site is in a noncommittable state, then the transaction can be safely aborted. Similarly, if any operational site is in a committable state, then the transaction can be safely committed.

This property follows directly from the requirement that a site may commit (abort) if and only if all currently operational sites are in committable (noncommittable) states.

To a limited extent a termination protocol can choose whether to abort or commit a transaction. A protocol that always commits a transaction when it is safe to do so is described as a progressive protocol.

5.4.1. Central Site Termination

The basic idea of the protocol is to choose a back-up coordinator from the set of operational sites. The backup coordinator assumes the role of coordinator and completes the transaction. Since the backup can fail before terminating the transaction, the protocol must be reentrant.

The scheme described is similar to the scheme used in SDD-1. The major difference is that the candidates for becoming backup coordinator are not distinguished sites, rather any site may be elected (the mechanics of election are discussed later). The protocol requires only that the invoking protocol satisfy the nonblocking theorem. Any of the three-phase protocols (including the decentralized version) can invoke it.

Once a backup has been chosen, it will base the commit decision only on its local state. It executes the following two-phase protocol:

Phase 1. Given that the backup is in a committable (noncommittable) state, it issues a message to all sites to move to a committable (noncommittable) state.

Phase 2. The backup issues a commit (abort) message to all sites.

(An optimization is to omit Phase 1 when the backup is initially in a commit or abort state.)

Lemma 5.8. Given that the initial global state of the transaction satisfies the Nonblocking Theorem, then the

Central Site Termination Protocol preserves the conditions of the nonblocking theorem.

Proof. Phase I preserves the conditions since it does not commit or abort but merely moves sites to a state already occupied by an operational site. At the end of Phase I all operational sites are in a committable (noncommittable) state; therefore, Phase II, which commits or aborts the transaction, satisfies the properties of the nonblocking theorem. If Phase II completes, then the transaction has been consistently terminated at every operational site.

Theorem 5.9. Given that at least one site remains operational, the central site termination protocol correctly terminates a transaction.

Proof. We will assume that a unique backup coordinator can be elected at each instantiation of the protocol. (This has been shown by [GARC80b].)

Given that the conditions of the nonblocking theorem hold when the commit protocol invokes the first instantiation of the termination protocol, by induction and using Lemma 5.8 the conditions hold for each subsequent invocation. The recursion terminates either when the backup does not fail and the current instantiation executes to completion, or when all sites fail.

5.4.2. Progressive Termination

One potential disadvantage of the central site termination protocol is that it is a nonprogressive protocol. We can construct a progressive termination protocol by adding an initial phase and by slightly modifying the other phases. In the new initial phase the backup polls the other participants as to their state. If any site reports a committable state, then the backup moves all sites to a committable state in the second phase (alias the old Phase I); otherwise, the backup moves all sites directly to the abort state (recall that all sites currently occupy a noncommittable state). The third phase (alias the old Phase II) is only executed for committed transactions. It moves all sites to the commit state.

The resulting termination protocol is very similar to the three-phase commit protocol. The major difference between them is in the second phase: the commit protocol requires unanimous acceptance while the termination protocol requires a single positive response (i.e. a site in the committable state). However, a single committable state implies that all sites have previously accepted the transaction.

5.4.3. Electing a New Coordinator

The central site termination protocol (and its proof) require the election of a new and unique coordinator. Garcia-Molina ([GARC80b]) surveys several election strategies under various failure assumptions (including arbitrary site failures) and presents informal proofs of resiliency. We refer the interested reader to this work for a detailed discussion of election protocols.

We will briefly describe an election protocol that is well suited to a centralized environment. Since both the central site commit protocol and a single instantiation of the central site termination protocol exhibit a linear cost (i.e. linear in the number of sites) message traffic, it is reasonable to require that the election protocol exhibit linear cost as well. This rules out any brute force approach.

The protocol we will discuss is implemented in SDD-1 ([HAMM80]). Sites are assigned a linear order in which they will assume the role of coordinator. For convenience, we assume that the order is equal to the site number. Hence, Site 1 is the coordinator, Site 2 is the first backup, etc.

The protocol makes the operational site with the highest ranking the new coordinator. The protocol proceeds as follows: Let Site i be the current coordinator (this occurs only if Sites 1, 2,..., i-1 have failed). When Site j discovers the failure of the current coordinator, it sends an "are you up" message to Site j-1. If this request times out, then Site j sends the same message to Site j-2. This continues until Site j receives a response, or until Site i+1 is polled and there is no response.

In the first case, at least one higher ranking site is operational; therefore Site j is not the new coordinator. In the latter case, Site j is the highest ranking operational site (recall that sites 1 through i-1 failed earlier, and Site i has just recently failed). Hence Site j becomes the new coordinator, and initiates a new instantiation of the termination protocol. If Site j fails before initiating the termination protocol, then the other sites will eventually time out and hold another election.

5.5. Decentralized Termination Protocol

Our primary goal in designing a decentralized termination protocol is to minimize the number of message rounds. Normally, more than one round is required, and in the worst case a round is required for each failure. As a secondary goal, we are interested in finding a progressive protocol.

During each message round, a site will send the status of its current state (i.e. committable or noncommittable) to every other site. We can now apply the nonblocking theorem to derive the protocol. From the theorem, we know that if all the sites send committable (noncommittable) messages during a message round, then it is safe to commit (abort) the transaction. We still have two cases to consider concerning the result of a message round: what to do when both committable and noncommittable messages appear during a round, and what to do about subsequent site failures.

Let us consider the case of conflicting states first. The transaction can not safely be terminated, for then it would violate the nonblocking rules. In concordance with our progressive criteria, the state transition rule should be: any site receiving a committable message moves into a committable state. Since the possibility of a failure occurring during the round exists, another message round is initiated (see below).

If no additional failures occur during the protocol, then two message rounds are sufficient to terminate a transaction. The transaction is terminated in the first round if all sites are in committable states or all sites are in noncommittable states; otherwise, a second round is needed. In the latter case, where the round contains mixed messages, all sites move to committable states during the first round. In the second round, all sites will receive "committable" messages, and subsequently commit.

A site failure during a round is more difficult to handle, since the sites no longer receive the same information. If a site fails while sending messages, some of the sites will receive a message from the site (and not detect that it has failed), while others will not receive a message but will detect its failure. In the worst case, each site failure can precipitate another message round. This is illustrated in Figure 5.4 for five sites.

This figure illustrates that a given site can not safely abort the transaction when it detects a (new) site failure during a round of "uncommittable" messages. It is unsafe because the failed site may have sent messages to some of its cohorts, changing their states to committable. The given site must participate in another round of messages. If two consecutive rounds of "uncommittable" messages reveal no additional site failures, the transaction can be safely

MESSAGES RECEIVED

	SITE 1	**SITE 2**	**SITE 3**	**SITE 4**	**SITE 5**
initial state	committable	non	non	non	non
round 1	(1)	CNNNN	-NNNN	-NNNN	-NNNN
round 2	FAILED	(1)	-CNNN	--NNN	--NNN
round 3	FAILED	FAILED	(1)	--CNN	---NN
round 4	FAILED	FAILED	FAILED	(1)	---CN
round 5	FAILED	FAILED	FAILED	FAILED	----C

NOTE: (1) site fails after sending a single message.

Figure 5.4. Worst case execution of the resilient termination protocol.

aborted.

On the other hand, it is always safe to commit the transaction during a round of "committable" messages, even when additional site failures are detected, because in a progressive protocol an operational site in a committable state never moves to a noncommittable state. Therefore, a failed site can never influence its committable cohorts.

The protocol is summarized in Figure 5.5. To commit a transaction may take only a single message round; however, to abort a transaction normally requires at least two message rounds. The first message round is required to establish the operational sites because generally a site is not certain of which sites are currently up. In

First message round:

type of transaction state	**message sent**
final abort state	abort
committable state	committable
all other states	noncommittable

Second and subsequent rounds:

message received from previous round	**message sent**
one or more abort messages	abort
one or more committable messages	committable
all noncommittable messages	noncommittable

a. Summary of rules for sending messages.

The transaction is terminated if:

condition	**final state**
receipt of a single abort message	abort
receipt of **all** committable messages	commit
2 successive rounds of messages where all messages are noncommittable and no site failures occur	abort

b. Summary of commit and termination rules.

Figure 5.5 Summary of the resilient decentralized termination protocol.

the first round, the site will detect the failed sites (by the absence of a message from that site), but generally it will not know if a site failed before or during the first round. If it failed during the round, then the protocol requires another round. Because of this uncertainty, two message rounds are usually required.

We now argue that the protocol is correct.

Theorem 5.10. The decentralized termination protocol of Figure 5.5. consistently terminates a transaction given that the properties of the nonblocking theorem holds upon its invocation and at least one site remains optional.

Instead of proving the theorem directly, we will present a series of lemmas concerning the protocol. Collectively, the lemmas imply the correctness of the protocol.

Lemma 5.11. A transaction that has been committed by at least one site at invocation time will be committed by the termination protocol.

Proof. Since the commit protocol is nonblocking, a commit at one site implies that all sites are in a "committable" state. Therefore, all messages sent during round one of the termination protocol must be committable messages. Hence, all sites will commit in round one.

Lemma 5.12. A transaction is committed only if at least one site occupies a committable state at the time the termination protocol is invoked.

Proof. Recall from the nonblocking theorem that if any site occupies a committable state, then the transaction can be safely committed. We will argue the contrapositive of this lemma: if no site is in a committable state then the transaction is not commited. If no sites occupy a committable state initially, then no committable messages will be sent. Hence, no sites will move into a committable state. By induction, no subsequent rounds can include committable messages. Therefore, the transaction cannot be commited.

Lemmas 5.11 and 5.12 prove that the termination protocol correctly commits in all cases. This implies that the protocol correctly aborts in all cases _if_ it always terminates. We will now show termination.

Let n be the number of participants at the beginning of the protocol. Let $N_i(r)$ be the set of sites sending _noncommittable_ messages to site i during round r.

We have:

Lemma 5.13. $N_i(r+1) \subseteq N_i(r)$

Proof. This follows directly from Lemma 1: for a site to send a noncommittable message in round r+1, it must have sent a noncommittable message in round r.

Lemma 5.14. If $N_i(r+1) = N_i(r) \neq \emptyset$, then all messages received by site i during both rounds r and r+1 were noncommittable messages.

Proof. Without loss of generality assume that Site i is operational. The argument proceeds by contradiction. Let $N_i(r+1) = N_i(r)$ and let round r contain messages other than noncommittable messages. We will only discuss the case where committable messages appear. There are two subcases depending on the message sent by i during round r:

Case 1. Site i sends a noncommittable message during round r. In round r+1, it will send a committable message because it received a committable message during round r (by assumption). This contradicts the claim $N_i(r+1) = N_i(r)$.

Case 2. Site i sends a committable message during round r. Since Site i did not fail in round r, all sites received a committable message (from i) during that round. Therefore, in round r+1 all sites will send committable messages. Again this is a contradiction.

Lemmas 5.13 and 5.14 show that the number of sites sending noncommittable messages either monotonically decreases toward zero with each round, or two rounds will occur with the same number. In the former case, the transaction will be terminated by the time the number reaches zero (and this requires at most N rounds). In the latter case, the transaction will be aborted because of the termination rule.

5.5.1. Enhancements

The above protocol is slow to abort the transaction - even when there are no sites in committable states and abortion is inevitable. In the absence of additional site failures, abortion requires two message rounds, and with site failures, additional rounds are required.

Abortion can be expedited by the addition of a new message type to the above protocol. The new message is abort and is sent whenever a site is in a final abort state (or in the initial state). Upon receiving an abort message, a site can immediately abort the transaction. Notice that the nonblocking theorem guarantees that both an abort message and a committable message can not be sent during a message round. This protocol was carefully described and its correctness proved in [SKEE81c].

5.6. Cost of Termination

Let us compare the cost of the termination protocols for the two classes we have discussed. let N represent the total number of sites, and let μ represent the number of sites currently up (operational).

A single invocation of the centralized protocol entails electing a new coordinator and executing the two-phase termination protocol. An election requires approximately $2(\mu-1)$ messages and two end-to-end delays. Assuming that all sites remain operational, the two-phase protocol requires $3(\mu-1)$ messages and three end-to-end delays. In total we have approximately $5(\mu-1)$ messages and 5 end-to-end delays per invocation.

The decentralized termination protocol requires μ^2 messages per round and experiences one end-to-end delay. For a single failure, either one or two messages are needed, with two rounds being the most probable. For each additional failure, at most one additional round is required.

Thus, for the failure of a single site, which for most systems is by far the major type of failure, the central site protocol requires approximately 5N messages and 5 end-to-end delays, as compared with approximately $5N^2$ messages and one end-to-end delay. For N-1 site failures (the greatest number of failures any termination protocol can tolerate) occurring at the worst possible times, the central site protocol would send $5N^2/2$ messages and incurs 5N delays, while the decentralized protocol would send $5N^3/3$ messages and incurs N delays.[5] It is interesting to note that in the most pessimistic case, the decentralized protocol sends fewer messages for N<8.

The major advantages of the centralized scheme are:

(1) Fewer messages are sent per invocation (round), and

[5] The calculations assume that operational sites do not send messages to known failed sites. If f(u) is the number of messages sent when u sites are operational, we have computed $\sum_{u=1}^{N-1} f(u)$.

(2) Another invocation is required only if the current coordinator fails - slave failures can be handled appropriately.

Since the decentralized protocol can require an additional round if any site fails, the central site protocol is expected to require fewer invocations than the decentralized protocol requires rounds.

The major disadvantage of the central site protocol is that it is fairly complex to implement since it requires both an election protocol and the two or three-phase termination protocol. Another disadvantage is the number of end-to-end delays incurred per invocation.

The weaknesses of the central site protocol are the strengths of the decentralized protocol. The decentralized protocol is very simple and straightforward to implement, and this is perhaps its strongest asset. Therefore, the decentralized protocol is a viable choice even in network environments where messages are expensive as long as N is relatively small and failures are infrequent. In this case, the cost of implementing the central site protocol could outweigh the savings in execution.

Decentralized protocols are better suited for network environments where broadcasts are cheap. A round in the decentralized protocol requires μ broadcast messages where an invocation of the central site protocol requires 2 broadcast messages (by the coordinator) and 3 μ point-to-point messages (by the slaves and during the election protocol).

5.7. Minimum Phase Termination Protocol

Notice that both protocols require O(N) phases in the worst case. An interesting question is: does there exist a protocol that terminates in less than N phases in the worst case?

For a central site protocol, it is easy to argue that the number of elections required in the worst case will always be equal to the number of failures because of the protocol's vulnerability to coordinator failure. Regardless of which site chosen during an election, that site may be the next one to fail, and thus precipitating a new election.

For a decentralized protocol, it is not obvious that N phases are required. Perhaps changing the order in which messages are sent or adding new types of messages would help. We have tacitly assumed in previous sections that during a message round, messages are physically sent in the order of the recipient's number (i.e. the message to Site 1 first, to Site 2 second, etc.). Upon failing, a site may send out any valid prefix of its intended messages. Therefore, if any site gets a message from the failing site, it would be Site 1.

One possible alternative to this scheme is to let Site i first send to Site i+1 modulo N. Does this or perhaps another order for sending messages speed up the termination protocol? In the worst case, the answer is "no."

Observation. For every message sending order, there exists a worst case execution of the decentralized termination protocol that requires N phases.

For every message order, a worst case scenario like the one displayed in Figure 5.4 can be constructed. In the figure, Site 1 sends a committable message to Site 2 then crashes; in round 2, Site 2 sends a committable message to Site 3 and then crashes, and so on. for an arbitrary message ordering, the worst case example is similar: let Site 1 send one message and crash. In round two the receiver of Site 1's message will send a message and then crash. The receiver of that message will send one message and crash in the next round, etc. Changing the ordering of messages does not speed up the protocol in the worst case. However, if for each site the probability distribution function for failing is known, then a protocol minimizing the expected number of rounds can be designed.

The above argument suggests, and this will be proved in the next theorem, that each site failure can sometimes cause an additional message round in any decentralized termination protocol. Thus, for any such protocol, N rounds are required in the worst case. This implies that the decentralized protocols presented earlier are minimum cost protocols (in the worst case).

Before stating the theorem, we need the following result. Let us define a message round as being redundant if elimination of the round never changes the outcome of the protocol.

Lemma 5.15. A message round where a single but arbitrary site can fail and not influence the outcome of the round (except at the failed site) is redundant.

Notice that only one site may fail, but the failing site is arbitrary. The lemma tells us that participation of each site in a nontrivial message round is significant. We can not tolerate the loss of an arbitrary site and still expect the result of the round to always be the same. This can be contrasted with error correcting codes in some memories where the loss of a single bit can be tolerated. No such "code" exists here.

Before proving the lemma we need to introduce some terms. Let t_{ji} indicate that a timeout was received by Site i from Site j. Define $M_i(s_i,j)$ to be the set of messages from j that are recognized by Site i while in state s_i. A message m_{ji} is in $M_i(s_i,j)$ if and only if it occurs in a state transition from s_i.

Proof of Lemma 5.15. Let i be an arbitrary site. Let δ_i be the transition function for i, s_i the state of Site i at the beginning of the round, and m_{ji} the message sent from Site j to Site i during the round. We assume that the operational sites numbered 1 through μ. One of these sites will fail during the round.

Now $\delta_i(s_i,m_{1i}m_{2i} \cdot\cdot\cdot m_{\mu i})$ defines the new state after the round. The lemma assumes that for all j, j≠i, we have:

$$\delta_i(s_i,m_{1i} \cdot\cdot\cdot m_{ji} \cdot\cdot\cdot m_{\mu i}) =$$

$$\delta_i(s_i,m_{1i} \cdot\cdot\cdot t_{ji} \cdot\cdot\cdot m_{\mu i})$$

That is, the failure of any one site does not change the outcome of the message round. By transitivity of the above equality, we have:

$$\hat{\delta}_i(s_i, m_{1i} \cdot\cdot\cdot m'_{ji} \cdot\cdot\cdot m_{\mu i}) =$$
$$\hat{\delta}_i(s_i, m_{1i} \cdot\cdot\cdot m''_{ji} \cdot\cdot\cdot m_{\mu i})$$

where m'_{ji} and m''_{ji} are arbitrary messages in $M_i(s_i, j)$.

Now we can show that for any two arbitrary sets of (recognized) messages, $m'_{1i}m'_{2i} \cdot\cdot\cdot m'_{\mu i}$ and $m''_{1i}m''_{2i} \cdot\cdot\cdot m''_{\mu i}$, Site i will make the same state transition. Again, using only transitivity, we have:

$$\hat{\delta}_i(s_i, m'_{1i}m'_{2i} \cdot\cdot\cdot m'_{\mu i})$$
$$= \hat{\delta}(s_i, m''_{1i}m'_{2i} \cdot\cdot\cdot m'_{\mu i})$$
$$= \hat{\delta}(s_i, m''_{1i}m''_{2i} \cdot\cdot\cdot m'_{\mu i})$$
$$\vdots$$
$$= \hat{\delta}(s_i, m''_{1i}m''_{2i} \cdot\cdot\cdot m''_{\mu i})$$

Since this state transition always moves i to the same state, it is redundant. Since Site i is an arbitrary site, all sites (except those that fail) will behave similarly and make redundant transitions. Therefore, the entire round is redundant.

Using this lemma, we can derive

Theorem 5.16. Any decentralized termination protocol requires at least f rounds (in the worst case) where f is the number of failed sites.

Proof. The proof is by contradiction. Let P be a protocol that can always terminate the transaction in f' rounds where $f' < f$. Furthermore, let P be a minimum round protocol in the sense that no protocol can terminate the transaction in less than f' rounds. Clearly, P can have no redundant rounds; otherwise, we could delete that round and obtain a protocol with fewer rounds. Since there are less rounds than there are failures, the last failure could occur during the last round, and any site could fail. Since some sites may detect the failure through timeouts and others not (instead, receiving a message sent before the site failed), this protocol must be able to tolerate an arbitrary site failure and yet give the same outcome. Lemma 5.15 tells us that this round is "redundant"; thereby contradicting our assumption.

One final question remains concerning both termination protocols. Note that they generally require two-phases (rounds) to terminate a transaction even if there are no further site failures. Is it possible to define a termination protocol requiring only one phase (round) in such cases? If failures are rare and independent events, such a protocol could halve the expected cost of termination.

The answer is an unequivocal "no." Commit protocols can fail at any point during the execution, and hence leave the sites in a mixed transaction state, including both committable and noncommittable local states. The nonblocking theorem dictates two phases for these cases: one phases to move all operational sites to the same state, and a second phase to commit or abort. Furthermore, in the decentralized protocol, the first phase (round) identifies the operational sites so that further site failures can be detected during subsequent rounds. Often an arbitrary site will not have exact knowledge at the beginning of the first round of the currently operational sites.

CHAPTER 6

Site Recovery

When a failed site becomes operational again, it must terminate all transactions outstanding at the time of failure. For each pending transaction, the recovering site will execute a _recovery protocol_.[1]

In the first part of the chapter, we consider recovery protocols based on a history mechanism. These protocols ensure consistency if the "history" is consistent, and they are nonblocking only if the history is available. Mechanisms for maintaining history are also discussed.

In the second part of the chapter, we consider recovery from _total site failure_ -- where all sites fail during the processing of the transaction. With nonblocking protocols the database is left in a consistent but hard to recover state. A correct strategy is to block transaction completion until all sites have recovered, but this is generally unacceptable since the time of recovery for all sites depends on the recovery time of the least reliable site. Alternatively, we are interested in finding sufficient conditions for a group of sites to safely recover based on their collective state information and independent of the remaining sites. These conditions

[1] It is convenient to discuss recovery protocols assuming one invocation per transaction. Of course, for performance considerations, one invocation may handle all outstanding transactions.

can then be used to derive recovery protocols from total failure. At the crux of all of this lies a subtle and intriguing subproblem: identifying the last site to fail during the execution of the protocol.

6.1. Recovery and Logging

The policy implemented by the commit/termination protocols in the previous chapter always allows operational sites to continue with the execution of the transaction. As a consequence of using nonblocking commit/termination protocols, recovery protocols can not base recovery on local state information. Therefore, those sites terminating the transaction must maintain a history of the outcome of the transaction for the recovering sites.

The accessibility of the history of the distributed transaction determines the performance of the recovery protocol. For this reason it should be maintained at more than one site. Only three values of the state of the transaction need to be recorded: _in progress_, _aborted_, _committed_. (Recall that the local state of any one particular site is insufficient for recovery; therefore, we need not be more discriminating about nonfinal states than _in progress_.)

There are three popular history mechanisms:

(1) A (virtual) distributed log as a collection of individual sites' logs ([STON78]). In this scheme, each individual site maintains a log of transactions executed at that site. The log consists of tuples of the form: <transaction id, partial local state>,

where the partial state is one of "in progress," "aborted," or "committed." The distributed log is a union of all of the individual logs.

(2) Dedicated logging machines ([LAMP81]). In the simplest case, a single machine (process) logs the progress of every transaction. Each participating site sends the outcome of each transaction to this machine. For resiliency to failures, more than one machine is normally used. (Logging machines may also serve as arbiters in conflicts among concurrent transactions.)

(3) Persistent messages ([HAMM80,LISB81]). A persistent message is guaranteed to be delivered to a failed site. This is implemented by spooling the message at several operational sites and sending it when the failed site recovers.

There is little real difference between (2) and (3) -- a logging machine maintains a log of <transaction ids, outcome> pairs; whereas, a spooler for persistent messages maintains a log of messages. However, (3) is a "read once" log.

Since sites must maintain logs for local crash recovery, the first scheme is simple to implement and requires no additional mechanism. It has the disadvantage that the availability of the "distributed log" is proportional to the number of participating sites. There is a way to circumvent this by requiring a minimum number of participants in every transaction. Some participants may not actively process the transaction, but they do log the state changes of the transaction. This has the additional advantage of reducing the probability that all sites fail during the processing of the transaction, since this probability is inversely proportional to the number of participating sites.

The second and third schemes do not suffer from this disadvantage since the number of spoolers or loggers is normally fixed. However, they require a significant effort to implement. The logs in these schemes can be viewed as specialized distributed databases that must be consistently and resiliently maintained. In essence, we have embedded a smaller distributed transaction management problem within the general problem. Observe that in the first scheme, consistency among the logs is a natural consequence of maintaining consistency in the database.

In its simplest form, a recovery protocol is a simple loop -- it repetitively queries the history mechanism until it gets a response which moves it to a final state. If it gets an _in progress_ response, it blocks for awhile, and then repeats its query. Eventually, it will get a _commit_ or _abort_ response, unless all sites have failed before terminating the transaction.

There is a special case to recovery which can be handled expeditiously. If a site fails in the initial state (during the first phase of the commit protocol and before it votes on committing the transaction), it can unilaterally abort the transaction. The commit protocols require that **all** sites vote yes before a transaction is committed. They treat a nonvote like a no vote, and abort the transaction. If the recovering site is unsure of its state at the time of failing, then it must execute the normal recovery protocol.

A more complicated recovery strategy allows a recovering site to rejoin the operational sites executing the commit/termination protocol. If a central site protocol is used, then the recovering site must find the current coordinator and send its current local state. The coordinator will respond with a new state for it to occupy. Until the coordinator responds, the site has not fully recovered and can not be considered an active partner in the protocol.

If the commit/termination protocol is a decentralized protocol, then the recovering site must broadcast its state information to **all** sites. This will be interpreted as a request to join the protocol by the operational sites and they will include the recovering site in the next message round. After hearing from all operational sites in the next round, recovery is complete and the site may actively participate in the following rounds.

There is little to be gained by allowing sites to rejoin the commit/termination protocol. It does reduce the probability of all sites failing before any terminates the transaction, but the reduction is usually insignificant. Since phases (rounds) tend to be brief, except for the first, and since the number of phases is expected to be small, there is little opportunity for a failed site to successfully rejoin.

6.2. When All Sites Fail

The recovery protocols discussed in the previous section require either that a log of distributed transactions be maintained, or messages be spooled for down sites. In this section we want to consider recovery strategies when all sites involved in a transaction (including sites spooling messages and logging sites) fail. The term _total failure_ is used to describe this. Neither the above strategies nor independent recovery work in this situation. Instead, the sites must be capable of detecting that a total failure has occurred and terminate the transaction as part of the recovery process.

An overriding constraint in defining a strategy for recovering from total failure is that the strategy impose a negligible overhead on the commit protocol. Total failures are (or should be) rare events, hence crash resistance to them must be cheap in order to be cost effective. We propose a strategy requiring no modifications to the commit protocol or the termination protocols. It uses knowledge about the order of failures among the sites. For each site, this knowledge is maintained in its _alive set_, which is defined in the next section. Unlike previous strategies, this one requires exact knowledge of the local state of the site at the time it failed. Hence, to use the strategy, a site must record its local state on stable storage.

To simplify the discussion, we will ignore complex logging strategies and strategies using persistent messages. Instead, we will assume that the only history of the transaction is maintained in local logs at participating sites. Moreover, we assume that the log entry for the new state is recorded on stable storage before a state transition is made. Therefore, a site failing during a state transition will assume the new state upon recovering. Note that some of the messages normally sent during a transition may not be sent if the site failed in the middle of the transition.

6.2.1. Necessary Conditions for Safe Recovery

Termination protocols allow a single operational site to terminate a transaction. When all sites fail during the transaction, it is clear that the last site to fail (or the last group of sites if several failed concurrently) must recover before the transaction can be safely terminated.[2] Unfortunately, the last site to fail is not necessarily aware of the status of the transaction at other sites. Thus, at first glance, it appears that all sites must recover before it can be discovered that no site terminated the transaction.

Consider the following scenario using the centralized three-phase commit protocol and the centralized termination protocol. The coordinator sends a single commit message to the first slave and then promptly fails. The slave receives this message, commits, and then promptly fails. Let us suppose that the newly elected coordinator fails before telling any sites to commit, and the next elected coordinator fails before any more commits, and so on. Hence, a single

[2] By last failure (or failures) we mean that this failure did not occur before any other failure. See Section 3.3.4 for a discussion on the partial ordering of events. Note that last in this sense may not be the last site to fail according to Greenwich Mean Time.

site has committed early in the protocol while all the other sites, including those that failed at a later time, have not. For each of the three-phase protocols, a similar scenario can be constructed where a single site commits (or aborts) early in the protocol.

Nonetheless, the local state of the last failing site is important. While the last site can not infer from its local state whether any site terminated the transaction, it can infer which final state (if any) the other sites must occupy since the concurrency set of its local state will contain at most one type of final state. Hence, the last site failing in a committable state (noncommittable state) implies that no site aborted (committed) the transaction. This is not necessarily true for a site other than the last failing site. For example, a slave can fail in the wait state, which is a noncommittable state, early in the protocol, but the coordinator may still commit the transaction.

Before proceeding, we need to develop the concept of _concurrent failures_. These failures are a type of "concurrent transitions" as described in Section 3.3.4. Intuitively, the failures of Sites i and j are concurrent if Site i fails before observing the failure of j and j fails before observing the failure of i.[3]

Considering only the case where all sites fail, we now define the _last group of sites to concurrently fail_. A site is a member of

[3] A site "observes" a failure by receiving a timeout from the failed site or by receiving a message from a site that has observed the failure.

this set if and only if its failure was observed by no other site. Hence, all members of this set must have concurrently failed.

The above observation about the last site to fail can be extended to all members of the last group.

Theorem 6.1. For a nonblocking protocol, if all members of the last group of sites to concurrently fail occupied committable states (noncommittable states), then the transaction was not aborted (committed) by any site. If members of the last group occupied both committable and noncommittable states, then the transaction was terminated at no site.

Proof: The proof follows directly from these two properties of nonblocking protocols:

(1) all operational sites are in committable states (noncommittable states) before any site commits (aborts).

(2) once any site has committed (aborted), transitions to noncommittable (committable) states are prohibited.

Both of these properties are either stated or directly inferred from the Nonblocking Theorem.

Let Site i be a site that commits (aborts) and then fails. Let Site j be a site that does not fail before Site i commits (aborts). Clearly, if Site j fails while in a noncommittable (committable) state, then either (1) is violated (in the case where Site j failed concurrently with Site i), or (2) is violated (if Site j fails after Site i).

The theorem states that if any member of the last group recovers, then its local state can be used to terminate the transaction safely. Furthermore, the transaction can be safely terminated only if a member of the last group recovers. Hence, recovering sites must be blocked until they are certain that such a member has recovered.

6.2.2. Determining the Last Group to Fail

The major obstacle to recovery is determining the last group of sites to fail. This is a nontrivial task: since a site can not detect concurrent failures, it can not distinguish between concurrent failures and failures occurring strictly after its own. Hence, a site can not independently determine whether it is a member of the last group.

To simplify the discussion we will initially assume that a site fails at most once during the observation period. Throughout this section the phrase "Site i failed before Site j" replaces the technically precise phrase "the failure of Site i occurred before the failure of Site j." Again, "occurred before" is used in the formal sense defined in Section 3.2.4.

6.2.2.1. Using Complete Information

Let us temporarily assume that a site can maintain a record of all sites failing strictly before itself and that this record (actually its complement) is maintained in its Alive Set. We will show later that although the complete Alive Set is not maintainable, a close approximation is.

> **Definition.** The Alive Set for Site i, denoted $\mathbf{A}_i$, is the set of all sites whose failure occurred concurrently with or strictly after i's failure.

Note that a site's Alive Set always contains itself.

From basic definitions we have:

> **Observation.** Assuming that all sites fail during the transaction, the intersection of the Alive Sets for all participating sites is the last group of sites to concurrently fail.

Clearly this last group is a subset of every Alive Set. If a failure is observed by any site, then it is removed from that site's Alive Set, and subsequently would not appear in the intersection.

The observation suggests the following scheme for determining this last group. Let F_{LAST} be the last group of sites concurrently failing. Let $R = \{ r_1, r_2, \cdot \cdot \cdot , r_k \}$ be the set of sites which have recovered and currently are attempting to terminate the transaction. All other participants are down. Let $\mathbf{A}_R$ be the intersection of the Alive Sets for all the recovering sites, i.e. $\mathbf{A}_R = \bigcap_{r \in R} \mathbf{A}_r$. Of course, $F_{LAST} \subseteq \mathbf{A}_R$. We have:

> **Theorem 6.2.** If $\mathbf{A}_R \subseteq R$, then $\mathbf{A}_R = F_{LAST}$.

Proof. First we assert that for any Site i, either $i \in F_{LAST}$ or i failed before some member of F_{LAST}. Consider the case where i is not in F_{LAST}. Therefore, i failed before some site i_1. Now, either $i_1 \in F_{LAST}$ or i_1 failed before some site i_2. In this way we obtain a chain where i failed before i_1, i_1 failed before i_2, and so on. This chain must terminate because each failure involves a unique site and there are a finite number of sites. Now, the last site in this chain, let this be i_k, must be a member of F_{LAST} since it failed before no other site. By transitivity, all sites preceding i_k in the chain, including Site i, failed before i_k.

The remainder of the proof proceeds by contradiction. Assume $\mathbf{A}_R \subseteq R$ and $\mathbf{A}_R \neq F_{LAST}$. Since F_{LAST} is in all Alive Sets, we know that $F_{LAST} \subset \mathbf{A}_R$. Therefore, there must exist a site s such that $s \in \mathbf{A}_R$ and s is not in F_{LAST}. However, from the previous paragraphs, we know that s failed before some member of F_{LAST}. Now, this site would not have s in its Alive Set, and therefore s would not be in the intersection of the Alive Sets, $\mathbf{A}_R$. This contradicts our assumptions.

Let us now consider the following "obvious" implementation of Alive Sets. Whenever a site receives a timeout from another site, say j, it immediately deletes j from its Alive Set and then appends the notice "j has failed" to the next message to each site. The receiver of such a piggybacked notice first deletes j from its Alive

Set and then processes the message. The purpose for piggybacking the failure notice, rather than sending it first, is to ensure that routing delays does not postpone its arrival until after the arrival of the main message.

This scheme has a singular deficiency: it does not work when a site fails after detecting a failure but before forwarding that information to all of its cohorts. A very simple example illustrates. Site 1 fails causing Site 2 to fail. Now, Site 3 detects the failure of 2 but not of 1 (perhaps 3 rarely communicates with 1), and then itself fails. Clearly, Site 3's Alive Set should not contain Site 1, but it does.

Although such a simple scenario is unlikely to present problems to a recovery protocol, more complex failure sequences can. An interesting example is given in Figure 6.1. At the end of the scenario, all sites are down. Observe that the Alive Set for the third slave is incomplete since no notice of the first slave's failure was included in the timeout from the coordinator. The situation is particularly precarious since the coordinator managed to commit some of the slaves before failing.

Consider now the consequences of Slaves 1 and 3 recovering before the others. Intersecting their Alive Sets yields {1,3}, which satisfies the premise of Theorem 6.2. Of course, basing recovery on Slave 1's local state would lead to inconsistency. Unfortunately, the information available to the sites does not indicate which of the sites can be safely used for recovery.

Setting. A coordinator (Site 0) and three slaves (Sites 1, 2, and 3) are executing the three-phase commit protocol. In the first phase all sites agreed to process the transaction. The protocol is now entering its second phase.

(1) The coordinator sends "prepare to commit" to all slaves.

(2) Slave 1 crashes.

(3) Slaves 2 and 3 receive the message and acknowledge it.

(4) After receiving two acknowledgements and a timeout (the latter from Slave 1), the coordinator crashes while in the middle of sending commit messages. A singular commit message is sent (to Slave 2) and appended to it is the notification of Slave 1's failure.

(5) Slave 2 receives the message, commits, and then crashes.

(6) Slave 3 "times out" and records the coordinator's failure. It then initiates a termination protocol and discovers that Slave 2 is down. Finally, Slave 3 fails before discovering the status of Slave 1.

At this point, the Alive Sets are:

$$A_0 = \{0,2,3\} \qquad A_2 = \{0,2,3\}$$
$$A_1 = \{0,1,2,3\} \qquad A_3 = \{1,3\}$$

The transaction states, s_i, are:

$$s_0 = ?? \qquad s_2 = \text{commit}$$
$$s_1 = \text{wait} \qquad s_3 = \text{prepared}$$

Figure 6.1. A pathological sequence of failures resulting in incomplete Alive Sets.

The fundamental problem with implementing Alive Sets is that failure notices can not be piggybacked on top of timeout messages. Therefore, cascaded failures can not propagate failure information, necessarily leaving the Alive Sets incomplete. Furthermore, deciding whether a site's Alive Set is complete is inherently as difficult as determining F_{LAST}. Similar problems arise with recovery strategies based on on logical clocks and timestamps: timeouts do not have timestamps.

With incomplete Alive Sets, all sites may need to recover before F_{LAST} can be determined, even if the sites in F_{LAST} recovered very early in the protocol. An alternative strategy to determining the entire set is to find one member of F_{LAST}. Recall that a single member is sufficient for recovery protocols. With incomplete failure information, this is easier than determining the entire set. In contrast, given complete Alive Sets, testing membership is equivalent to determining the entire set.

6.2.2.2. Using Incomplete Information

Let a_j, called the (little) alive set for j, be an implementable subset of A_j. Minimally, we require that a_j not contain any site whose failure was directly observed (through a timeout) by j. Since a failure of a nonmember of F_{LAST} must have been directly observed by at least one other site, a nonmember will be missing from at least one alive set. Consequently, the above minimum requirement suffices to ensure $\bigcap_{i \in \mathbf{N}} a_i = F_{LAST}$, where $\mathbf{N}$ is the set of sites.

Site j's alive set yields a superset of the sites concurrently failing with or failing after j. By examining the alive sets of other sites, j may be able to infer additional sites failing before it, thus reducing its alive set.

Definition. The *reduction of* a_j *with respect to a set of sites S*, denoted a_j^S, is the set a_j minus the sites that can be shown to have failed before j by examining only the alive sets of the sites in S.

We always assume that $j \in S$. a_j^S can be calculated by the algorithm given in Figure 6.2. We will primarily be interested in a_j^R where R

Comment. Compute a_j^S given j, S, and a_i for all $i \in S$.

Declarations.
TRIED - set of sites already used in the reduction.

Algorithm.

```
$a_j^S = a_j$;
TRIED = {j};
while $(\sim a_j^S - \text{TRIED}) \cap S \neq \emptyset$ do
    choose x from $(\sim a_j^S - \text{TRIED}) \cap S$;
    $a_j^S = a_j^S \cap a_x$;
    TRIED = TRIED $\cup$ {x};
    end;
```

(Notation: $\sim a_j^S$ is the complement of a_j^S)

Figure 6.2. Algorithm for calculating the reduction of a_j with respect to set S.

is the set of recovered sites.

Finally, define a_R analogous to A_R, that is $a_R = \bigcap_{j \in R} a_j$. a_R contains the possible candidates for membership in F_{LAST}.

Theorem 6.3. If $r \in a_R$ and $a_r^R \subseteq R$, then $r \in F_{LAST}$.

Let us return to the previous scenario and apply Theorem 6.3, setting the four sites' (little) alive sets equal to the incomplete Alive Sets displayed in the figure. Assume once again that Slaves 1 and 3 recover before the rest. Now, $a_1^R = \{0,1,2,3\}$ and $a_3^R = \{1,3\}$. Since $a_3^R \subseteq R$, Slave 3 is a member of F_{LAST}. Now, a_1^R is not contained in R, hence we can conclude nothing about it. In particular, from the available information we can **not** conclude that Slave 1 is not a member of F_{LAST}. Notice also that to conclusively determine F_{LAST} would require the recovery of either the coordinator or the remaining slave.

The proof of this theorem is remarkably simple. Recall that r must be a member of R; otherwise, a_r^R is undefined.

Proof. (By contradiction.) Assume both $r \in a_R$ and $a_r^R \subseteq R$. Now, suppose that r is not a member of F_{LAST}. Therefore, there exists a j such that the failure of r was directly observed by j, hence, r is not a member of a_j. Clearly, j is not a member of R; otherwise, we would have r is not a member of a_R, violating our assumption. Since j failed after r, j must be a member of a_j^R. Consequently, a_j^R is not a subset of R (because j is not a member of R). We have contradicted our assumption.

Since F_{LAST} is always contained in a_r^R, all members of F_{LAST} must recover in order for the membership test to succeed. Recall that with complete information, this was sufficient to determine the entire set F_{LAST}.

There are several ways to implement alive sets. As stated before, the minimum requirement is that a site merely record every timeout it receives by deleting its sender. Although this is sufficient, it is probably not desirable since the less complete the alive sets the larger the expected size of the set of sites required for recovery. Of course, the implementation for Alive Sets suggested above is a correct implementation for (little) alive sets. Moreover, since it provides maximum propagation of failure information, its fault tolerance is strictly superior to that in all other implementations.

6.2.3. Recovery Protocols

Once a subset of F_{LAST} has been determined by applying Theorem 6.3, recovery can proceed safely. The recovery protocol is similar to a termination protocol, with one important exception. Since the states of nonmembers of F_{LAST} may be incongruous with the states of members of F_{LAST} and, more importantly, with sites that are still down, the initial phase of the recovery protocol must only use local state information from sites known to be in F_{LAST}. If a centralized termination protocol is used, then the coordinator must be chosen from among these sites. If a decentralized protocol is used, then only these sites can send messages during the first round. However, in both cases the messages in the first round should be addressed to all sites. In the second and subsequent rounds, all operational sites will have received at least one message from a member of F_{LAST} and may now fully participate.

Until now, we have ignored resetting the alive sets during recovery. If the sites are to be able to recover from another total failure, then they must be re-evaluated. However, this can not occur until the operational nonmembers of F_{LAST} have moved into states that are known to be consistent with the states of the members of F_{LAST}.

Let us consider the case where the sites are executing the centralized termination protocol. During the first phase, the newly elected coordinator will move all the operational sites into the same local state. At the end of the phase, the coordinator will receive acknowledgements from all operational sites, and at that time it can update its alive set. At the beginning of the second phase, the coordinator can send its copy of the alive set (which is the only updated set) to all slaves. The set can be piggybacked to the message that is normally sent in this phase.

The decentralized protocol can be treated similarly: the operational sites will update their alive sets at the end of the second round. However, in the decentralized protocol, updating their sets does not generate any more message traffic (not even piggybacked messages) because each site can update its set from the messages it normally receives during a phase (round) of the protocol.

The reader should recall that both termination protocols require only two phases (rounds) to complete in the absence of additional failures. Also, updating the alive sets requires two phases (rounds). Thus, there is little to be gained by updating the sets since the probability that a total site failure will occur between the time that the sets are updated and the transaction is terminated is very low. This is especially true for the centralized protocol where additional messages are required to perform the update. If the coordinator remains operational during the second phase, then the transaction will be terminated anyway; on the other hand, if it fails, then not all of the alive sets will have been updated.

We conclude this section with an interesting observation. When there are few communication paths between sites, the probability that a failure will go unobserved increases, and concurrent failures may span a relatively long interval of time. Hence, the expected size of F_{LAST} is influenced by the density of the communication graph for the commit/termination protocol: the more sparse the graph, the larger the expected size of F_{LAST}. Of course, the larger F_{LAST}, the longer the expected wait before the recovery protocol described in this section can be invoked. Central-site protocols have a sparse graph; decentralized, a dense graph.

The major weakness of this approach is its vulnerability to other types of failures. The approach requires that site failures are correctly observed, and even more importantly, that other types of failures are not mistaken for site failures. It is possible for a single mistake of the latter type to lead to an inconsistency (although this is highly improbable).

6.3. Alternative Recovery Strategies

Until now, our assumption has been that operational sites should not be penalized (by blocking) for the failures of their less reliable cohorts. Hence in the protocols of the last two chapters, operational sites are always allowed to independently proceed. This is a policy decision reflecting our perception of the needs and demands of the user community.

As a consequence of this policy, a single arbitrary site may remain operational while all others fail, terminate the transaction, and then fail. All recovering sites will be forced to block until this single site has recovered. Furthermore, even if this lone operational site crashes before terminating the transaction, the remaining sites will be forced to wait since they will not be able to detect whether the transaction had been terminated at the lone site. At the other extreme, when using a central site two-phase commit protocol, sites are vulnerable to failure of the coordinator. In such a strategy, the coordinator is omnipotent - forcing the slaves to block on its failure. The protocol is sensitive only to the failure of this one site; a slave failure can never force another site to block.

A compromise policy is to allow a majority of sites to continue regardless of their failure histories. In the next chapter, we discuss generalized majority voting schemes.

CHAPTER 7

Network Partitioning

This chapter is primarily concerned with recovery from network partitions -- a problem much harder than previous problems. In the first section, we discuss some possible recovery strategies and the tradeoffs involved. We then develop commit and recovery protocols for the most promising strategy: the "quorum-based" protocols. These protocols are extremely resilient, resilient not only to network partitioning but also to arbitrary site failures and undetected failures.

The last section of this chapter is concerned with a problem arising in the implementation of quorum-based protocols and other recovery protocols using local state information. This problem, called _partial amnesia_, arises because the last action performed by a transaction is inherently vulnerable to being "forgotten" during a site failure. The possibility of partial amnesia is unavoidable on all current systems. A simple strategy that safely handles this is examined.

7.1. Strategies for Recovering from Partitions

From the results of Chapter 4, we know that there exists no resilient nonblocking protocols for this problem. Hence, we must be satisfied either with a blocking protocol that always maintains consistency or with a nonblocking protocol that allows the database to become inconsistent.[1] As we have done in previous chapters, we will focus protocols that ensure consistency. We refer the reader to several recent papers that discuss the latter class of protocols (see [PARK81, PARK82, DAVI81]).

It is always safe to allow a single partition to terminate all outstanding transactions and continue processing. The schemes that we shall discuss differ primarily in how they choose this partition.

The simplest scheme allows only the partition containing a designated site, called the primary site, to continue. This partition is called the primary partition. The concept of a primary site is similar to, but not equivalent to, the concept of a coordinator in a centralized commit protocol. A coordinator can be chosen independently for each transaction; whereas, the choice for a primary site must remain constant for all concurrently executing transactions. This is because a partition can continue processing only if it can terminate all outstanding transactions. If each transaction had a different primary site, then the probability that all primary sites would occupy the same partition after a network failure would be very low. Normally, it is safe to reassign the designation of a primary site only if there are no outstanding transactions.

If the coordinator for a transaction is also the primary site, then the two-phase commit protocol can be used (recall the two-phase

[1] The usual definition of nonblocking applies here -- all operational sites (including those in different partitions) are allowed to terminate an outstanding transaction.

protocol blocks only when the coordinator fails or is partitioned from the remaining sites). On the other hand, if they are different, then a more complex protocol must be used -- one that allows the primary site to terminate the transaction during a partitioning. Any of the three phase protocols suffices.

The choice of a primary site is based on two major criteria: it should be reliable and it should maximize the expected size of the primary partition. The last criterion assumes that it is desirable to have the largest number of sites continue processing after a partitioning occurs. This precludes choosing a spur, a site that is connected to the network through a single link, as a primary site since it is likely to be in a partition consisting only of itself.

The major weakness of this scheme is its vulnerability to a primary site failure. If it can be determined with absolute certainty that the primary site has failed, a new primary site can be elected.[1] However, the assumption that failures can be identified with one hundred percent accuracy is unrealistic in almost all environments. Hence, in a realistic network, when the primary site fails, outstanding transactions will be blocked. In many networks the probability of a primary site failure exceeds the probability of a network partitioning.

[1] If the protocol is to survive more than a single failure, then this requires a commit protocol more sophisticated than the two-phase commit (e.g. the three-phase commit).

An ideal scheme for handling partitions is to always allow the "largest" partition to proceed and require the others to block. The largest partition could be interpreted as the one containing the most sites or, more generally, the one where the sum of the weights of the contained sites is maximal. However, there is a fundamental problem with the scheme: how does a partition decide if it is the largest? Since this can not be effectively decided, the ideal is not achievable. However, a partition can easily decide if it contains a majority of the sites, and clearly it is safe to allow the partition to proceed in this case. Moreover, if a majority exists, then the "largest partition" criterion has been satisfied.

Majority-based schemes -- those that allow a partition to continue if it contains a majority of the sites -- are popular alternatives to primary site schemes. Such a scheme does not suffer the inherent weakness of a primary site scheme: a majority of the sites must fail before the protocol blocks (compared to a single site for the primary site scheme). Also, it is unconcerned with the type of failures that have occurred, it needs only to test whether a majority of the sites are still in communication. However, it is clearly a suboptimal solution when there are partitions containing "close" to a majority, but no partition contains a majority.

In the next section, we develop quorum-based protocols, which are a generalization of protocols based on a simple majority.

7.2. The Use of Quorums

The quintessence of a majority consensus approach is: _before terminating a transaction, a majority of the sites must agree on the direction of the transaction_. Note that _agreeing_ to a final state is not the same as a moving to that final state - and this difference is crucial. When a majority of the sites reaches a consensus, then the sites in the minority must conform to the majority.

Voting schemes have been widely proposed for concurrency control for replicated data. Thomas introduced a majority consensus scheme in [THOM78] as a concurrency control mechanism for replicated databases. Gifford extends it in [GIFF79] using quorums rather than a simple majority. The scheme developed herein differs from previous schemes in the following ways:

(1) It does not require replicated data. The data can be redundantly or irredundantly stored.

(2) It is not a concurrency control mechanism. It preserves transaction atomicity on a _per transaction basis_ in the presence of site failures. (However, it can form a basis for a concurrency control mechanism on replicated data.)

(3) It allows unilateral aborts.

Before proceeding, we need to stress the following "obvious" but sometimes overlooked point.

Observation. While we can require that a majority of sites be operational and agree to commit (abort) the transaction before any sites commit (abort), we can **not** require that a majority of the sites remain operational and in communication long enough to actually commit (abort) it.

Between the time a site agrees to commit and then actually commits, it can always fail. Since any number of sites can fail during this waiting period, there is no way to enforce a commit rule requiring a minimum number of communicating sites.

The remainder of this section discusses the integrated design of commit, termination, and recovery protocols using quorums. While the design of these protocols is interdependent, we will treat them sequentially.

In all of these protocols, sites are allowed to unilaterally abort during the first phase of the protocol, which is an important feature of the protocol. Traditionally, we have called this the voting phase, and have referred to the replies sent during this phase as "votes." In order to avoid confusing this phase with later phases requiring a majority consensus of the sites, we will simply refer to it as the "first phase." It is only during this phase that sites are allowed to unilaterally abort.

7.2.1. Definition of a Quorum

Let V be the total number of votes assigned to the participating sites. Each site is allocated an integral nonnegative number of votes. (This can be zero, in which case the site is a passive participant.)

A transaction must collect a *commit quorum* of V_C votes before it is committed by any site. It can be unilaterally aborted during the first phase of processing; otherwise, it must collect an *abort quorum* of V_A sites before it is aborted. Of course, $0 < V_{A,} V_C \leq V$. To prevent two groups of sites from independently deciding to terminate the transaction in opposing states, we must have $V_A + V_C > V$. To guarantee eventual termination, we should choose the quorum sizes such that $V_A + V_C = V + 1$.

As sites fail and recover, the currently operational sites will try to form either a commit quorum or an abort quorum. An individual site cannot arbitrarily attempt to form a given type of quorum (e.g. an abort quorum); instead, a site's actions must depend on its current state (i.e. committable or noncommittable), its previous participation in groups attempting to form a quorum, and communication with its cohorts. We now state sufficient properties for a quorum-based protocol to be completely resilient.

Theorem 7.1. (The Quorum Requirement). Let V, V_C, and V_A be as previously defined. A quorum-based protocol is resilient to arbitrary sites failures if:

(1) $V_C+V_A>V$, where $0\leq V_C,V_A\leq V$

(2) When any site is in the commit state, then at least a *commit quorum* of sites are in *committable* states.

(3) When any site is in the abort state, then at least an *abort quorum* of sites are in *noncommittable* states.

Proof. Inconsistency can arise only if both a commit and an abort quorum is formed. But, from (1), this requires that at least one site participates in both quorums. From (2) and (3), this site must occupy a state that is both committable and noncommittable. By definition, no such state exists.

The requirements in the Quorum Theorem are very similar to those for _K-resiliency_ (see Corollary 5.6 and the Nonblocking Theorem). Both requirements state sufficient conditions for terminating a transaction, and in both cases these conditions require that a minimum number of sites agree _a priori_ before an _irreversible_ decision is made by any site. Hence, it is not surprising then that a theorem similar to the Nonblocking Theorem exists for quorums.

The most significant difference between the theorems is that the conditions of the Nonblocking Theorem apply only to operational sites; whereas, the Quorum Requirement applies to all sites. However, this is a crucial difference, since it means that the protocol does not have to distinguish between failed sites and operational sites that are partitioned from the remainder of the network.

Let us now consider the requirements of Theorem 7.1 in sequence. Requirement (1) is obvious, and has been mentioned before.

Requirement (2) can be viewed as two subrequirements: (2.1) Before the first site commits, a commit quorum of sites in committable states must be obtained, and (2.2) after any site has committed, a commit quorum must be maintained. As a consequence of (2.2), a site can safely move from a committable state to a noncommittable state if and only if it can be shown that no site has committed the transaction, or it can be shown that this will not destroy a commit quorum.

Requirement (3), which concerns abort quorums, is analogous to (2). Hence, there exists (3.1) and (3.2) which are the analogs of (2.1) and (2.2).

7.2.2. Commit Protocols with Quorums

Our first observation is that two-phase commit protocols can not satisfy the requirements of the above theorem: they allow sites to commit before a quorum of sites are in committable states. (This, of course, is similar to the reason why two-phase protocols could not satisfy the Nonblocking Theorem.)

The canonical three-phase commit protocol can be modified to include quorums (which is not surprising due to the similarity between the Quorum Theorem and the Nonblocking Theorem). The modification is straightforward: in the third phase, the protocol blocks until a commit quorum of sites acknowledge occupying committable states. The protocol is presented in Figure 7.1.

The quorum-based commit protocol contains the same states as the regular three-phase commit. All sites move from the initial state to the wait state, and then either move to the prepared-to-commit state followed by the commit state, or move directly to the abort state. Like the three-phase commit protocol, it is a <u>nonretreating</u> protocol:

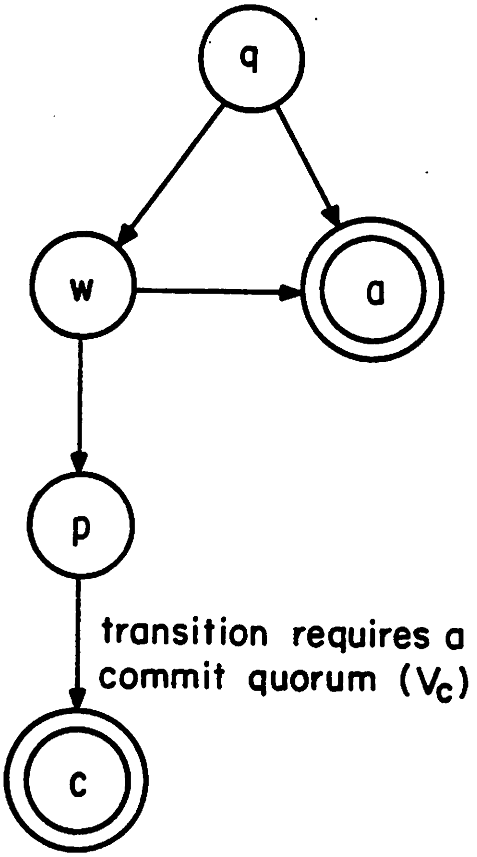


Figure 7.1 The canonical quorum-based commit protocol.

once it has been determined that the transaction is committable, then the commit protocol will not abort the transaction (however, it may block).

Notice that the protocol never verifies that an abort quorum of sites are in noncommittable states. This is not necessary since, at the start of phase two, all sites are either in the wait state, a noncommittable state, or in the abort state. Hence, the current transaction state trivially satisfies Quorum Requirement (3). This property will be used by the termination protocols presented in the next subsection.

As a concrete example of a quorum-based three-phase commit protocol, we present the centralized protocol in Figure 7.2 Notice that the slave protocol is unchanged from the centralized three-phase commit protocol (c.f. Figure 5.3a).

```
COORDINATOR                                              SLAVE'S RESPONSE

(1) Transaction is received.
    Subtransactions are
      sent to each slave.

                                                             Yes to commit
                                                              No to abort

(2) If all sites respond yes
      then
        prepare to commit is sent;
        continue to phase (3)
      else
        abort is sent;
        stop.

                                                                 Ack

(3) If the sum of the weights
    of the responding sites equals
    or exceeds $V_C$
      then
        send commit to all
      else
        block until the partitioning
          is resolved.

                                                                 --
```

Figure 7.2. The quorum based commit protocol.

In addition to the protocol blocking in phase three, it is also possible that the coordinator fails or the network partitions before the protocol terminates. With either type of failure, there will exist at least one partition without a coordinator. Each leaderless partition must invoke one of the termination protocols described in the next section.

7.2.3. Termination Protocols with Quorums

In our previous discussion on termination protocols, we completely ignored the issues of site recovery. This was possible because (by the very nature of nonblocking protocols) only operational sites actively participated in determining the outcome of the transaction.[3] This no longer holds for quorum-based protocols; instead, recovering sites participate in forming quorums.[4] Hence, when a site recovers, it will run a simple recovery protocol to reestablish communication with its cohorts and then rejoin the termination protocol. Therefore, the termination protocol must tolerate sites that repeatedly fail.

Within this section, we will discuss the common characteristics and requirements of all termination protocols. The requirements are inferred from the Quorum Requirements and properties of the canonical three-phase protocol discussed in the last section. In addition, we also discuss a few design guidelines -- some are stronger than

[3] Except in the special case where all sites fail before any site terminates the transaction.

[4] Indeed, a quorum may not be possible without their participation.

necessary but greatly simplify the construction of termination protocols. Alternatives to these guidelines are presented in a subsection labeled "Alternatives and Enhancements." From these requirements and guidelines, we derive in detail two specific termination protocols: a centralized protocol and a decentralized protocol.

The correctness of a quorum-based scheme is derived from the restriction that a site can participate in only one kind of quorum (i.e. either a commit or an abort quorum). Our first observation is that a site may not know if an attempt at a quorum was successful. This uncertainty can exist for two reasons:

(1) the site may fail (or become partitioned from the other sites) after agreeing to participate in a quorum but before it receives confirmation that a quorum was successfully formed, and

(2) the attainment of a quorum may be recognized by a single site (e.g. the co-ordinator) that fails before informing other sites that a quorum had been established.

Hence, even if a quorum is formed, it may be the case that less than a quorum terminate the transactions. When uncertainty exists about the success of a quorum in which a given site has participated, then it is unsafe for that site to attempt to form the opposite kind of quorum. Since situations involving uncertainty are complex, we will require all protocols to take a conservative approach: after a site has agreed to participate in the formation of one type of quorum, it can no longer participate in the formation of the opposite type of quorum.

To indicate that a site has "taken sides," two local states are required. We will call them prepared-to-commit (denoted pc in diagrams) and prepared-to-abort (denoted pa in diagrams). The prepared-to-commit state, as its name implies, is similar to the committable, intermediate state present in the nonblocking commit protocols and present also in quorum-based commit protocol. The prepared-to-abort state is a new state; it has not appeared in previous protocols.

The prepared-to-abort state is the noncommittable analog of the prepared-to-commit state. A site in this state can participate only in the formation of an abort quorum. In this respect, this state differs from the wait state, which is also a noncommittable intermediate state -- the wait state in an "undecided" state which enables a site to join either type of quorum. An attempt to form a quorum moves a site from the wait state to the appropriate "prepared" state.

Until now, we have discussed the formation of a quorum in vague terms. We can now make this more formal.

> **Definition.** A commit quorum exists whenever the sum of the weights of sites occupying either prepared-to-commit or commit states is at least V_C. A abort quorum exists whenever the sums of the weights of sites occupying either prepared-to-abort or abort states is at least V_A.

Since sites in a prepared-to-commit state cannot move to a prepared-to-abort state, and vice versa, an attempt to form a quorum is aimed at moving sites from the wait state to the appropriate *prepared*.[5]

We have discussed five distinct local transaction states in a quorum-based protocol: initial, wait, prepared-to-commit, prepared-to-abort, commit, and abort. Figure 7.3 gives the valid local state transitions. These are the only valid state transitions that can occur in both commit and termination protocols.[6] The solid lines (→) indicate the sequence of transitions taken when a site participates in a quorum. The squiggly lines (⇝) indicates a unilateral abort. The dashed line (⇢) indicates a transition taken when a site is informed that: (1) a quorum in which it did not participate has terminated the transaction, or (2) the transaction was unilaterally aborted by another site (in latter case, only the transition from wait to abort is relevant).

A very conspicuous property of the diagram is its lack of cycles. Every transition moves the site closer to a final state. If there is no state which is blocked forever, then the protocol will eventually terminate. If one assumes that there is a nonzero chance of recovering from every failure in a finite time, then there will be no infinite blocking.

We conclude this subsection by describing the rules associated with state transitions in a "canonical" termination protocol. These

[5] We have ignored sites in the *initial* state. Such sites can be handled by allowing them to complete the first phase of processing which will take them into either a wait state or an abort state.

[6] The above commit protocol bypassed the prepared-to-abort state.

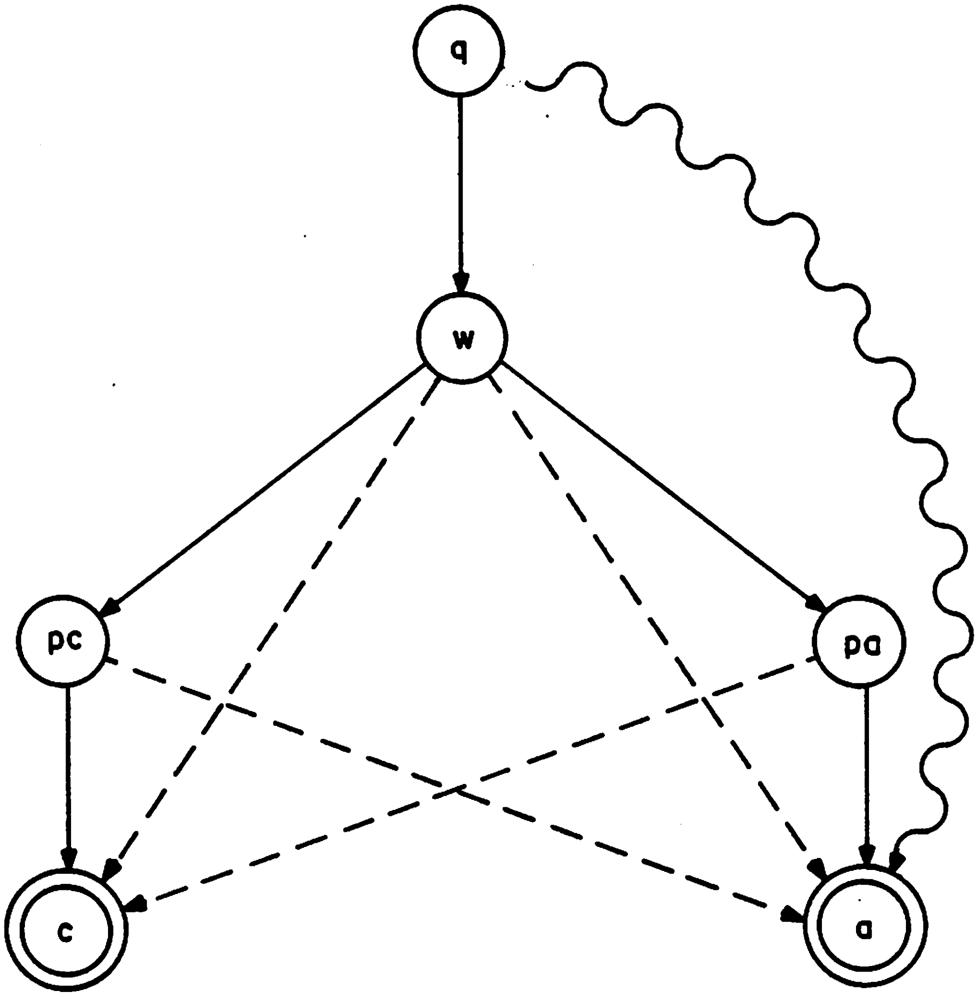


Figure 7.3 Valid state transitions in a quorum-based protocol.

rules apply only to state transitions illustrated in Figure 7.3, no other transitions are allowed. Given that the termination protocol is invoked by a commit protocol which obeys the Quorum Requirements, these rules preserve those requirements. Let **G** be a partition of operational sites, and let Site i be an arbitrary site in **G**. The rules are:

(1) Site i can make a transition to the prepared-to-commit state only if all sites replied "yes" in the first phase or there is a site in $\mathbf{G}$ in a committable state.

(2) Site i can make a transition to the commit state only if a commit quorum exists within $\mathbf{G}$ or there is a site in $\mathbf{G}$ in the commit state.

(3) Site i can make a transition to the abort state only if a abort quorum exists within $\mathbf{G}$ or there is a site in $\mathbf{G}$ in the abort state.

(4) Site i can make a transition from the wait state to the prepared-to-abort state at any time.

The first rule enforces the constraint that all sites must agree to commit in the first phase. The second and third rules enforce the Quorum Requirements.

Notice that the first three rules are disjunctions, and furthermore, the first term in each disjunction is a restatement of a necessary precondition for occupying the specified state. Whenever one of these preconditions becomes true, it remains true for the remainder of the protocol. Therefore, the truth of the precondition needs to be established only once -- by the first site to enter the specified state. The remaining sites need only find a site in that state to establish the truth of the precondition.

We now discuss the details of two termination protocols that obey the above rules.

7.2.3.1. Centralized Termination Protocol

The centralized termination protocol is invoked whenever a partition of sites, G, can no longer communicate with the coordinator (either it has failed or it is partitioned from G). There may be several termination protocols executing concurrently -- each in different partition.

The termination protocol, which is illustrated in Figure 7.4, consists of two parts. The first part elects a new coordinator (any of the protocols discussed in Section 5.4.3 can be used). The second part consists of a three-phase protocol which enforces the state transition rules in the previous section.

In the first phase, the coordinator queries the sites as to their local transaction state. This phase can be omitted if the coordinator already occupies a final state. The second and third phases implement the state transition rules. If any site occupies a final state or if a quorum already exists, then during the second phase, the coordinator sends the appropriate message, "commit" or "abort", to all sites. In this case, no third phase is required.

If a quorum does not exist and no site occupies a final state, then the second phase will check whether there is a sufficient number of sites to form a quorum. If so, an appropriate quorum will be attempted by sending messages to sites in the wait state. These messages instruct the slaves to move to the appropriate "prepared" state. If a sufficient number of acknowledgements to these messages are received, then the coordinator sends the appropriate messages (i.e. commit or abort) during phase three. The coordinator will block in phase two if there is an insufficient number of available sites to form a quorum or in phase three if an insufficient number of acknowledgements are received.

When the protocol blocks, it remains blocked until a failure is repaired.[7] The recovery process can be viewed as a merger of two or more partitions forming a new partition. To recover, the newly formed partition can invoke the termination protocol given above. However, in this case, the election process can be streamlined, for example, the new coordinator can be elected from the previous coordinators.

Site recovery is a special case of a merger involving a partition containing exactly one site.

The centralized termination protocol in this section ensures consistency even if several coordinators are elected within the same partition. Since each coordinator executes the same protocol, they behave identically in the absence of failures. Thus, it is clear that consistency would not be compromised in this case. Even if failures occur causing the coordinators to behave differently, consistency is still guaranteed by the state transition rules given

[7] There is an underlying assumption that the repair of a failure is detectable. Clearly, this is a reasonable assumption for site failures. For communication link failures resulting in a partitioned network, we assume that an underlying protocol periodically tests the link.

PART I. A new coordinator is elected.

PART II. The coordinator executes the following 3-phase protocol (for the protocol executed by the slaves, see note following the protocol).

(1) Request local state.

(2) The slaves' state information is received and the coordinator responds according to the following table.

slave responses	coordinator's actions
≥ 1 "commit"	send "commit"; commit the transaction
≥ 1 "abort"	send "abort"; abort the transaction
≥ 1 "prepared-to-commit" and weights of slaves sending "waits" and "prepared-to-commit" $\geq V_C$	send "prepared-to-commit"; continue with (3a)
weights of slaves sending "wait" and "prepared-to-abort" $\geq V_A$	send "prepared-to-abort"; continue with (3b)
OTHERWISE	block until a merge occurs

```
(3a) if $\geq V_C$ ack's
       then send "commit";
            commit the transaction
       else block until a merge occurs

(3b) if $\geq V_A$ ack's
       then send "abort";
            abort the transaction
       else block until a merge occurs
```

(NOTE: slaves respond with their local state in Phase 1 and with an acknowledgement in Phase 2).

Figure 7.4 The centralized quorum-based termination protocol.

above. In this case performance may suffer (certainly more message traffic would be generated), but consistency is preserved.

7.2.3.2. Decentralized Termination Protocol

The decentralized protocol is essentially the centralized protocol where the election has been eliminated and each site executes the coordinator protocol. Of course minor performance enhancements to the protocol are possible. For example, a site can broadcast its local state in the first phase instead of waiting for every site to request it. However, the protocol is fundamentally unchanged.

The decentralized version is given in Figure 7.5. Like its centralized counterpart, it occasionally blocks. Once it blocks, it will remain blocked until a failure is repaired, whereupon the protocol is restarted from Phase 1.

7.2.3.3. Alternatives and Enhancements

The largest improvement in the performance of the termination protocol occurs if each site remembers the state information of all of its cohorts. In the proposed protocol, quorums consist solely of sites within the same partition. If it were known that a site in a different partition had moved to a "prepared" state beforehand, then that site could safely be counted toward a quorum in this partition. This observation depends on the property that these protocols are nonretreating -- once in a "prepared" state, a site will not retreat to a wait (state or to a different "prepared").

Local Protocol for Site i

(1) Broadcast local state.

(2) Wait until state info is received from all sites, then respond according to the following table.

<u>messages received</u>	<u>Site i's actions</u>
≥1 "commit"	commit the transaction
≥1 "abort"	abort the transaction

≥1 "prepared-to-commit" and weights of slaves sending "waits" and "prepared-to-commit" $\geq V_C$	prepare to commit;
	broadcast "prepared-to-commit";
	continue with (3a)

weights of slaves sending "wait" and "prepared-to-abort" $\geq V_A$	prepare to abort;
	broadcast "prepared-to-abort";
	continue with (3b)

OTHERWISE	block until a <u>merge</u> occurs

```
(3a) if ≥V_C "prepared-to-commit"
       then commit the transaction
       else block until a merge occurs

(3b) if ≥V_A "prepared-to-abort"
       then abort the transaction
       else block until a merge occurs
```

Figure 7.5 The decentralized quorum-based termination protocol.

For better performance, each site could keep a list of the sites it knows to be in the prepared-to-commit state, and another list for sites in the prepared-to-abort state. Each list must be a conservative estimate -- sites which have recently moved to a "prepared" state or moved after a network partitioning, may not be on the list. For the scheme to work, no false entries can appear in either list. Now, when a list at any site satisfies an appropriate quorum, the transaction can be terminated.

In a centralized scheme, maintaining the lists is primarily the responsibility of the coordinator. During every phase the coordinator sends its most current version of both lists to the operational sites. Whenever a newly-elected coordinator queries the slaves concerning their states, it can include its lists as part of the same message. A slave's reply can include additions to the lists if they differ.

In the decentralized protocol, the lists can be maintained without additional communication since all operational sites broadcast messages to one another. However, lists between sites will tend to diverge slowly over time due to sites failing in the middle of sending or receiving messages. This divergence does not affect the resiliency of the protocol, nor is it likely to significantly affect its performance. (Of course, an alternative is to include the lists with each interaction.)

The design philosophy of the original protocol was to delay the movement of a site into a "prepared" state until it was apparent that a quorum had responded. This philosophy was influenced by: (1) only sites within the same partition could participate in the formation of a quorum, and (2) sites in the wait state were free to participate in either type of quorum. Since a failed site is completely passive, it was not advantageous for it to be in a "prepared" state.

On the other hand, if each site maintains state information about all of its cohorts (as suggested above), then a failed site in a "prepared" state can indeed contribute to the formation of a quorum. Clearly, it is advantageous for a site to occupy a "prepared" state before failing; therefore, unreliable sites should move into a "prepared" state early. However, sites that rarely fail should still defer moving into the "prepared" state until the formation of a quorum appears likely.

7.2.4. Performance

Every quorum-based protocol will occasionally block until a single site recovers. This occurs in the case of a tie, where every site, except one failed site, is in a prepared state, but neither type of quorum is established. However, all known protocols that allow network partitioning and site failures possess this weakness. Even the protocol for catastrophic site failures occasionally blocks for a single failed site.

All of the quorum-based protocols, including commit and termination protocols, are three-phase protocols. Each phase requires approximately 2N messages for centralized protocols and N^2 messages for decentralized protocols. In the absence of failures, three phases is always sufficient to terminate the transaction (sometimes two suffices).

To estimate the cost of the protocols in the presence of failures, we must first estimate the number of invocations of the termination protocol. This will be proportional to the number of failures and "mergers" occurring during the execution of the protocol. Let us assume that every failure is a simple partitioning (i.e. a single group of sites is split into two noncommunicating groups). Let us also assume that each merger merges exactly two partitions. Hence every failure requires one merger to repair it.

With the above assumptions, consider the worst case cost of terminating a transaction when f failures occur. Clearly, this occurs when all f failures must be repaired before the transaction is terminated. In the centralized protocols, every failure will cause an invocation of the termination protocol (the new partition that is coordinatorless will invoke it) and every merger will cause another invocation. Therefore, the worst case cost is (2f)(6N) plus the cost of 2f invocations of the election protocol.[2] The 12*f*N term dominates the cost.

In the decentralized protocols, a failure may causing blocking, but it never triggers a new invocation. However, a merger requires a new invocation. Thus, $f(3N^2)$ is the worst case bound for the number of messages.

It is very difficult to analyze the expected performance of quorum-based protocols, even if very simple, independent probability

[2] f of the 2f total invocations of the election protocol occur during merging when the election can be very cheaply performed.

distribution functions are used to describe site failures. For nonzero failure probabilities, it is clear that the worst case performance is unbounded, which is expected in light of the Two Generals Problem. However, we have argued that under realistic failure assumptions the protocol does eventually terminate -- there are no infinite cycles.

There are two sets of parameters that determine the performance of the protocol in the presence of failures: the weights assigned to individual sites, and the values for V_C and V_A.

The assignment of weight is often influenced by policy considerations external to the implementation of the system. However, some factors that are relevant to the implementation issues are percentage downtime, failure rate, and percentage of data stored at the site. Perhaps the most intuitive rule is to assign weights inversely proportional to the percentage downtime.[9] Perhaps a better policy is to use the rate of failure rather than the actual downtime. The crashing of a site during a transaction is harder to handle than the site being down at the onset.

The rationale for assigning weights according to percentage of stored data can be justified on the principle of "conservation of

[9] Assigning weights inversely proportional to the percentage downtime is **not** equivalent to assigning them proportional to percentage uptime. Consider two sites, A and B, whose downtimes are 0.2% and 0.1% respectively. Using this to determine weights, a consistent assignment would give 1 to A and 2 to B. Using uptime, we find A and B are up, respectively, 99.8% and 99.9% of the time. A consistent weighting assigns 998 to A and 999 to B.

labor" -- work should not be discarded needlessly. If the work performed by the site is proportional to the amount of data it holds, then the site should have a larger voice in the outcome of transactions.

In choosing quorum sizes, it is not necessary that V_C equal V_A. In fact, there are several strong arguments for choosing $V_C > V_A$. First, commit protocols allow unilateral aborts, and if a significant number of transactions are unilaterally aborted, then this suggests using a smaller V_A. A second and stronger reason is that most site failures are expected to occur during Phase 1 of the protocol since most of the transaction execution time is spent in Phase 1. All of the data processing takes place during this phase; whereas, Phase 2 and Phase 3 synchronize state information among the sites and require very little local processing. If sites fail during Phase 1, then the transaction must be aborted. Hence, it should be easy to abort.

An interesting heuristic for choosing V_A is based on a rough estimate of the failure distribution of the sites. Let $P(V_A)$ be the probability that at least an abort quorum is operational. $P(V_A)$ is a decreasing function in V_A. The point is to choose the maximum V_A such that $V_A \leq V_C$ and $P(V_A)$ exceeds a minimum level of desired availability.

As mentioned before, the weight of a site can be zero, in which case the site contributes nothing toward forming a quorum. (However, such a site can still unilaterally abort the transaction.) A zero-weighted site can be eliminated from all phases whose purpose is to form a quorum. Hence, in the commit protocol, zero-weighted sites can be omitted from Phase 2. In the extreme case, where only a single site has a nonzero weight, the quorum based commit protocol degenerates into the standard two-phase protocol with all of its disadvantages. Specifically, the slaves normally must block on the failure of the only nonzero weighted site (i.e. the coordinator).

7.3. Partial Amnesia

An unfortunate truism of current distributed systems is that sending a message and recording that fact in stable storage can not be implemented as an atomic action. Consequently, for every message sent, there exists a window of time in which a failure leaves the status of the message in doubt: the site can not know whether the message was sent or not. We call this partial loss of state information *partial amnesia*.

Since a state transition which sends one or more messages can not be implemented atomically, an interesting question arises: For maximum resiliency, how should the individual operations constituting a transition be ordered? On the one hand if a site sends messages first and fails before changing state, then it has incorrectly informed other sites about the transition. (Upon recovering, it will not have changed states.) On the other hand, if the change in local state occurs first, then the site may fail before informing other sites of the change.

When site recovery is based on a distributed log or a history mechanism, the ordering of operations within a transition is immaterial since the local state is not used in making the commit decision. Therefore, the recovery mechanism used with nonblocking protocols is resilient in either ordering of operations.

Partial amnesia becomes a crucial issue only when recovering sites take an active part in determining the fate of the transaction, as in quorum-based protocols. The consistency of quorum-based schemes depends on the rule that once in a "prepared" state, a site cannot retreat to a wait state. The rule is to insure that a site participates in only one kind of quorum.

Let's explore what may happen when messages are sent before a change of state occurs. Consider the situation where a site sends messages to other sites informing them of its transition from the wait to a "prepared" state, but the site fails before changing its local state. Upon recovery, the site will still occupy the wait state. In essence, it has behaved as though it made a transition to the "prepared" state and then (silently) retreated back to the wait state, and thus, violating the above rule.

Figure 7.6 illustrates a scenario where Site 3 fails and recovers in such a manner, enabling the site to participate in both kinds of quorums. This leads to inconsistency. Notice that the only site informed by Site 3 of its willingness to participate in a commit quorum (i.e. Site 1) is down when Site 3 recovers. Therefore, no site observes Site 3's traitorous behavior.

Sites 1, 2, and 3 are processing a transaction using the centralized quorum-based commit protocol. Site 1 is the coordinator. Both an abort and a commit quorum requires two sites.

The sequence of events are:

(1) Site 1 sends the transaction to 2 and 3.

(2) Site 2 and 3 both reply "accept"; Site 2 fails.

(3) Site 1 also "accepts" and sends out "prepare-to-commit" messages.

(4) Site 3 sends back an "ack", but fails before moving into the prepared-to-commit state. (Site 3 will not remember sending the "ack".)

(5) Site 1 receives the "ack" from 3. Sites 1 and 3 constitute a commit quorum; therefore, Site 1 commits and sends a message to 3 (which is not received).

(6) Site 1 fails.

(7) Site 2 and 3 recovers. Both sites recover to the *wait* state. (NOTE: Site 3 has suffered partial amnesia.)

(8) Site 2 is elected the new coordinator. Since both sites are in the wait state, an abort quorum is formed and both sites eventually abort.

Figure 7.6 A scenario where partial amnesia experienced by recovering site (Site 3) results in an inconsistent database.

If a change of state occurs before messages are sent, then the nonretreating properties of the protocol are preserved, and hence resiliency is guaranteed. In general, protocols where recovering sites assume an active role in terminating the transaction, require that the state change be recorded first. This is similar to the *write ahead log rule* used in local logs. Gray was the first to observe that it is desirable to apply this rule to distributed logs as well ([GRAY79]).

CHAPTER 8

Conclusions

The contributions of this thesis consist of complementary theoretic and applied results. The theoretic results of Chapters 3 and 4 include a formal model and existence (or more accurately, nonexistence) results for many classes of resilient protocols. The applied results include a collection of practical commit and recovery protocols for various failure-prone environments.

8.1. The Formal Model and Existence Proofs

Chapter 3 introduced a powerful, finite state model. The model was sufficiently abstract to develop the existence results of Chapter 4, yet sufficiently concrete to specify and verify the resilient protocols presented in the later chapters.

Models based on finite state automata have previously been used in network applications, for example, modeling error-free transmissions at the bit level ([AHO79]). They are particularly useful for modeling concurrent processes that: (1) can be characterized by a small number of states and (2) where the significant events consists of sending and receiving messages. Commit and recovery protocols are examples of such processes.

Perhaps the aspect of distributed processing that is most difficult to model is the physical network and the mechanism of message passing. For commit protocols we proposed a simple yet novel approach, the network is an unbounded buffer that is nondeterministically read by the automata which are processing at each site.

The existence results in Chapter 4, concerning independent recovery and network partitioning, were more illuminating than surprising. In a nutshell, the partitioning results conclusively showed that any realistic protocol resilient to partitioning must be a blocking protocol. The independent recovery results, which we will now summarize, were less pessimistic.

Independent recovery from site failures is attractive for a variety of reasons; two of the most important ones being: (1) a distributed history of each transaction does not have to be maintained and (2) a site can always recover even if the network or companion sites fail. Alternative recovery schemes require some type of distributed history, and necessarily block recovering sites when that history becomes unavailable due to failures.

The results were positive in the special case where resiliency to only a single failure is required, but were negative in all other cases. Furthermore, it is impossible for a recovering site to determine when it is safe to independently recover.

The results in themselves are not as interesting as their implications, particularly in the design of commit protocols. This is most evident in the design of the nonblocking protocols in Chapter 5. These protocols were derived directly from the conditions stated in the Nonblocking Theorem and completely independent of the properties of any recovery protocol. The resulting nonblocking protocols precluded the use of independent, and hence nonblocking, recovery protocols. The existence results not only justify this approach, moreover, they prove that even a more complex commit protocol can not simplify the required recovery protocols.

The independent recovery results together with the Nonblocking Theorem demonstrate that a commit/recovery protocol pair can ensure that one and only one group of communicating sites independently and consistently terminate the transaction. Although this observation appears fairly intuitive, its implications may not be so obvious. For example, we know that it is possible to design protocols where the coordinator (but no other site) can independently recover -- the two-phase commit is an example of this. Also, we know that it is possible to design centralized protocols where operational sites can always continue processing a transaction even in the presence of a coordinator failure. The three-phase commit in Chapter 5 is such an example. However, in that protocol the coordinator can not independently recover. It would be desirable to design a protocol that allowed both: operational sites can always terminate and the coordinator can always independently recovery. However, from the above observation, we know that no such protocol exists.

The existence results also reveal the limited usefulness of local state information during recovery. Unless a site is in a final state, the local state information can not be used, instead, a distributed history of the transaction must be accessed. Therefore, nonfinal local state information need not survive a processor failure and can be stored in volatile memory. (The only exceptions are recovery from catastrophic failures and from total failures.)

Finally, as we have previously mentioned, a distributed history of the transaction is necessary (and sufficient) for recovery protocols that are resilient to arbitrary failures.

8.2. The Design of Resilient Protocols

Chapters 5, 6, and 7 dealt with the design of resilient protocols, first for site failures and later for network partitioning. Two major paradigms were presented: the three-phase nonblocking protocols, and the quorum-based protocols. In both paradigms, protocol design was decomposed into two parts: the design of a protocol that executes in the absence of a major failure, and the design of a protocol that is invoked after a major failure has occurred. The former protocol is called a commit protocol, and the latter, a termination protocol.

For each class of failures we developed two families of protocols: a family of commit protocols and a family of termination protocols. (Minimally, each family contained a centralized and a decentralized protocol.) The families were compatible in the sense that any termination protocol could be used in conjunction with any commit protocol. For example, a centralized commit protocol (which has low message overhead) could be used with a decentralized termination protocol (which is easy to implement).

This approach provides for separation of "concerns" between the two types of protocols. Since the commit protocol is invoked for every transaction, it should be designed for speed and to consume a small percentage of the network bandwidth. On the other hand, the termination protocol is invoked only when a failure occurs (and this should be rare); therefore, the objectives for its design may be quite different than for those for commit protocol. For example, ease of implementation may be a significant factor in choosing a termination protocol.

We now review nonblocking protocols and quorum-based protocols.

8.2.1. Nonblocking Protocols

Nonblocking protocols were presented in Chapter 5. All such protocols are based on the following rule:

> **Nonblocking Rule.** Before any site commits, all operational sites must occupy committable states; and similarly, before any site aborts, all operational sites must occupy noncommittable states.

This rule guarantees that a transaction can be safely terminated by any operational site (see Theorem 5.14). Hence, if at least one site remains operational during the execution of the protocol, the transaction will be correctly terminated. If a designated group of K sites obey the above rule, then only one of these needs to remain operational to guarantee correct termination (this property is called K-resiliency). Despite the rule's simplicity, it is a powerful and very useful result.

The popular two-phase commit protocol does not satisfy the nonblocking rule; therefore, it occasionally blocks on site failures. Specifically, it blocks on a coordinator failure. Other two-phase commit protocols, including the decentralized two-phase commit protocol discussed in Chapter 2, also fail to satisfy this rule. In fact, it is easy to argue that no two-phase commit protocol satisfies the rule.

By adding an extra phase to the two-phase commit, a three-phase protocol satisfying the nonblocking rule is derived. In its most general form, the three-phase protocol is:

Phase 1. The transaction is distributed to all sites. All sites vote on it.

Phase 2. If **all** sites vote "yes," then all **operational** sites move into a committable state (the prepared to commit state); otherwise, all **operational** sites abort, and the protocol terminates.

Phase 3. All operational sites commit.

The distinguishing feature of the protocol is the addition of the prepared to commit state and the requirement that all of the operational sites must enter this state before any site commits. Aborting a transaction still requires only two phases.

The three-phase commit protocol is substantially more expensive than the two-phase commit protocol, incurring a fifty percent increase in end-to-end message delays and from a forty to a fifty percent increase in message traffic. While this is a significant cost increase, its impact on the performance of a distributed transaction management system is unclear since a commit protocol may not be a significant consumer of resources. One would expect this to be true in a system where large distributed transactions predominate. However, for a small transaction, the elapsed execution time may be dominated by the number of end-to-end message delays, in which case the elapsed time will increase by almost fifty percent.

Finally, we note that nonblocking protocols are susceptible to a special kind of "catastrophic" failure -- the event that all sites fail during the processing of the transaction. Such a failure does not compromise consistency, but it must be detected and recovering sites must execute a special recovery protocol. In Chapter 6 we proposed a detection mechanism based on alive sets. These sets can be cheaply maintained without additional message traffic. In the same chapter, the special recovery protocol is described.

8.2.2. Quorum-Based Protocols

The quorum-based protocols discussed in Chapter 7 are resilient to all failures that leave a site's local state information intact. They satisfy the following rules.

Quorum Requirements.

(1) If any site occupies the commit state, then a commit quorum of sites are in committable states.

(2) If any site occupies the abort state, then an abort quorum of sites are in noncommittable states.

(3) The sum of the votes required for a commit quorum and those required for an abort quorum must exceed the total number votes.

Again, no two-phase protocol satisfies these rules; however, there are three-phase protocol that do. The quorum-based, three-phase protocol is essentially the three-phase protocol of the previous section, except all state transitions require the consensus of an appropriate quorum of sites.

The similarity between the quorum-based protocol and the nonblocking protocol is not surprising due the similarity between the above rules and the Nonblocking Rule. The major difference between the two sets of rules is that the Nonblocking Rule is concerned only with operational sites; whereas, the Quorum Requirements are concerned with all sites (even failed ones). In fact, the reason that quorum-based protocols are more resilient is that they do not distinguish between operational and nonoperational sites. However, they are intrinsicly blocking protocols, but this is a necessary property of any protocol that is resilient to network partitioning.

Another difference between the two types of protocols is that a separate recovery protocol is not required. A failed site is merely a passive participant in the commit or termination protocol, and it becomes an active participant after it recovers. However, unlike nonblocking protocols, local state information must be preserved across site failures since a failed site will eventually rejoin the protocol as an active participant. Thus, a site's local state must be stored in stable storage.

Within this simple recovery scheme, there are no catastrophic failures[1] and therefore, no special recovery protocols.

8.2.3. A Comparison

Since both the nonblocking protocol and the quorum-based protocol are three-phase protocols, they have almost identical costs. In some environments, the quorum-based protocols might be slightly faster, since they require only a quorum of responding sites before proceeding to the next phase.

The major factor in choosing between the two types of protocols is whether network partitioning can occur and, if so, whether they are detectable. If the probability of partitioning is nonnegligible and it is difficult to distinguish between several site failures and a partitioning, then a quorum-based protocol should be used. In most other environments, and especially environments where the probability of a site failure exceeds that of a partitioning by orders of magnitude, the nonblocking protocol should be used. However, whenever there is a possibility that a partitioning has occurred, a nonblocking protocol must block in order to preserve consistency.[2]

[1] Except for failures which destroy local state information.

[2] Recall that a "nonblocking protocol" is nonblocking only with respect to site failures.

Hybrid strategies are possible: a transaction management system could run a nonblocking protocol that obeys the Quorum Requirements whenever a partitioning is suspected. To ensure consistency, the size of a commit quorum and an abort quorum should be statically defined and not depend on the number of sites that is conjectured to be operational. The protocol for site recovery is complicated since a recovering site must now determine which rule was used to terminate the transaction.

8.3. The Design Methodology

In this thesis we adopted a systematic design approach that is conceptually similar to programming methodologies based on abstract data types ([LISK77]). Broadly speaking, the approach first derives the formal properties for commit (or termination) protocols resilient to a given class of failures, and then develops implementable protocols from these properties. In the following paragraphs, we review the approach in more detail.

For each class of failures, we postulated sufficient conditions for resiliency. For sites failures, these were the conditions in the Nonblocking Theorem; for partitioning, these were the Quorum Requirements. From the sufficient conditions, we then derived a canonical protocol. A canonical protocol differs from a conventional one in that none of the details of message passing, including the communications topology of the protocol, are specified. The canonical protocol is an "abstract protocol" where only its formal properties are developed. From these properties, a state transition diagram is derived. This diagram together with the formal properties is a formal specification of the canonical protocol.

The state diagram defines all of the local transaction states and all possible transitions between those states. The formal properties define the necessary conditions for a local state transition to occur. These properties are normally expressed as predicates on the global state of the protocol (see, for example, the Quorum Requirements which are the formal properties of the canonical quorum-based commit protocol of Chapter 7).

A "concrete" protocol is derived from a canonical protocol by specifying, for each transition, an exchange of messages which verify the truth of the associated condition described in the canonical protocol. In this way a canonical protocol serves as a template for the derivation of a concrete protocol. By varying, for example, the specification of the communications topology, many concrete protocols can be derived from the same canonical protocol. In the thesis, we developed a centralized and a decentralized protocol from each canonical protocol. Hierarchical and ring protocols can be derived likewise.

This design approach has several advantages. The canonical protocol can be formally verified from the postulated sufficient conditions, using the same methods used in the formal verification of concurrent processes ([HOAR69,GRIE76]). To verify a concrete protocol, one need only show that it obeys the properties of a canonical protocol. This is much easier than verifying a concrete protocol from scratch. Moreover, many concrete protocols can be derived and verified from the same canonical protocol.

8.4. Further Research

The most important remaining questions are: (1) what impact does a resilient protocol have on the performance of a real system, and (2) does the increased resiliency offered by such a protocol justify its cost? Of course, the answer to the first question requires an implementation and, with one exception, none of these protocols have been implemented. The exception is SDD-1's four-phase commit protocol; however, due to the manner in which the system was implemented, a meaningful evaluation of performance is difficult.

The answer to the second question depends not only on the results of the first, but also on the reliability of future systems and their use. However, it is probably safe to assume that, regardless of the cost of a given level of resiliency, there will be some applications that require at least that level.

Another open problem is the complexity of these protocols. In particular, lower bounds on message passing and the number of end-to-end delays are needed. We gave bounds only for the decentralized nonblocking protocols. In that case, the worst-case and best-case bounds coincided with the bounds of our proposed protocol. In general, it would be nice to have lower bounds that are independent of the protocol's communication topology.

REFERENCES

[AHO79] A.V. Aho, J.D. Ullman, and M. Yannakakis, "Modeling Communications Protocols by Automata," 20th Annual Symposium on Foundations of Computer Science, Oct. 29-31, 1979, pp. 267-273.

[ALSB76] P.A. Alsberg, G.G. Belford, J.D. Day, and E. Grapa, "Multi-Copy Resiliency Techniques," Center for Advanced Computation, CAC 202, University of Illinois, Urbana, May 1976.

[BAER80] J.-L. Baer, et al., "The Two-Step Commitment Protocol: Modeling Specification and Proof Methodology," Univ. of Washington, 1980.

[BAYE80] R. Bayer, H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," Trans. on Database Systems, 5, 2, June 1980, pp. 139-156.

[BERN80] P.A. Bernstein, and D.W. Shipman, "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)," Trans. on Database Systems, 5, 1, March 1980, pp. 52-68.

[BERN81] P.A. Bernstein, and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, 13, 2, June 1981, pp. 185-222.

[BOCH77a] G.V. Bochmann, "Finite State Description of Communication Protocols," Computer Networks, 2, 4, October 1977, pp. 361-372.

[BOCH77b] G.V. Bochmann, and J. Gecsei, "A Unified Method for the Specification and Verification of Protocols," Proc. IFIP Congress, 1977, pp. 229-234.

[CAMP74] R.H. Campbell, and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," Proc. Int. Symp. Held at Rocquencourt on Operating Systems, 1974.

[DAVI81] Susan Davidson, and Hector Garcia-Molina, "Protocols for Partitioned Distributed Database Systems," TR No. 283, EECS Dept., Princeton University, March 1981.

[DENN76] Peter Denning, "Fault-Tolerant Operating Systems," Computing Surveys, 8, 4, December 1976, pp. 359-389.

[EDEL74] M. Edelberg, "Data Base Contamination and Recovery," Proceedings of the ACM SIGMOD Workshop on Data Description, Access, and Control, New York, 1974, pp. 97-147.

[EDEN80] "Eden Project Proposal: Research in Intergrated Distributed Computing," Department of Computer Science, University of Washington, Technical Report 80-10-1, October 1980.

[ELLI77a] C.A. Ellis, "A Robust Algorithm for Updating Duplicate Databases," Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May

1977, pp. 146-158.

[ELLI77b] C.A. Ellis, "Consistency and Correctness of Duplicate Database Systems," Xerox Palo Alto Research Center, Palo Alto, California, May 1977.

[GARC79] Hector Garcia-Molina, Performance of Update Algorithms for Replicated Data in a Distributed Database, Ph.D. Thesis, Computer Science Dept., Stanford University, June 1979.

[GARC80a] Hector Garcia-Molina, "Reliability Issues for Completely Replicated Distributed Databases," TR No. 266, EECS Dept., Princeton University, April 1980.

[GARC80b] Hector Garcia-Molina, "Elections in a Distributed Computing System," TR No. 280, EECS Dept., Princeton University, December 1980.

[GIFF79] David Gifford, "Weighted Voting for Replicated Data," Operating Systems Review, 13, 5, December 1979, pp. 150-9.

[GRAY79] J.N. Gray, "Notes on Database Operating Systems," Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[GRAY81] J.N. Gray, et al., "The Recovery Manager of the System R Database Manager," ACM Computing Surveys, 13, 2, June 1981, pp. 223-242.

[GRIE76] D. Gries, and S. Owicki, "Verifying Properties of Parallel Programs: An Axiomatic Approach," CACM, 19, 5, May 1976.

[HAMM80] Michael Hammer, and David Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," Trans. on Database Systems, 5, 4, December 1980, pp. 431-466.

[HOAR69] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," CACM, 12, 10, October 1969.

[KOHL81] W.H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," ACM Computing Surveys, 13, 2, June 1981, pp. 149-184.

[KUNG81] H.T. Kung, and J.T. Robinson, "An Optimistic Methods for Concurrency Control," Transactions on Database Systems, 6, 2, June 1981, pp. 213-226.

[LAMP78a] Leslie Lamport, "The Implementation of Reliable Distributed Multiprocess Systems," Computer Networks, 2, 2, May 1978.

[LAMP78b] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," pp. 558-565, Communications of the ACM, 21, 7, July 1978.

[LAMP80] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," Technical Report 54, Computer Science Laboratory, SRI International, March 1980.

[LAMP76] B. Lampson, and H. Sturgis, "Crash Recovery in a Distributed Storage System Tech. Report," Computer Science Laboratory, Xerox Parc, Palo Alto, California, 1976.

[LAMP81] B. Lampson, "Replicated Commit," Computer Science Laboratory, Xerox Parc, Palo Alto, California, January 7, 1981.

[LELA78] G. LeLann, and H. LeGoff, "Verification and Evaluation of Communication Protocols," Computer Networks, 2, 1, 1978.

[LELA80] G. Lelann, "Consistency Synchronization and Concurrency Control," Distributed Data Bases, I. W. Draffan, and F. Poole (editors), Cambridge University Press, 1980.

[LIND79] B.G. Lindsay, et al., "Notes on Distributed Databases," IBM Research Report, RJ2571, July 1979.

[LISB81] E.T. Lisboa, and P. Penny, "The Communication Manager: Providing Reliable Communication Mechanisms for Distributed Systems," Univ. of Southwestern Lousiana Technical Report, 1981.

[LISK77] B. Liskov, and S. Zilles, "An Introduction to the Formal Specification of Data Abstractions," Current Trends in Programming Methodology, Volume 1, R.T. Yeh (editor), 1977.

[LOME77] D.B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions," Sigplan Notices, 12, 3, March 1977, pp. 128-137.

[LORI77] R. Lorie, "Physical Integrity in a Large Segmented Data Base," ACM Transactions on Data Base Systems, 2, 1, March 1977.

[MENA80] D.A. Menasce, G.J. Popek, and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," *Trans. on Database Systems*, 5, 2, June 1980, pp. 103-138.

[MERL76] P.M. Merlin, "A Methodology for the Design and Implementation of Communication Protocols," *IEEE Transactions of Communications*, COM-24, pp. 614-621, 1976.

[MERL79] P.M. Merlin, "Specification and Validation of Protocols," *IEEE Transactions on Communications*, COM-27, 1979, pp. 1671-1680.

[MINO80] Toshimi Minoura, "A New Concurrency Control Algorithm for Distributed Database Systems," *Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks*, 1980, pp. 221-234.

[PARK81] D.S. Parker, et al., "Detection of Mutual Inconsistency in Distributed Systems," *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1981, pp. 172-186.

[PARK82] D.S. Parker, and R.A. Ramos, "A Distributed File System Supporting High Availability," *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks* (*to appear*), February 1982.

[PEAS80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, 27, 2, April 1980, pp. 228-234.

[POWE81] Michael Powell, private communication, February 1981.

[POST74] Jon Postel, "A Graph Model Analysis of Computer Communications Protocols," UCLA-ENG-7410, Univ. of California, Los Angeles, Ph.D. thesis, 1974.

[RAND78] B. Randell, P.A. Lee, and P.C. Treleaven, "Reliability Issues in Computing System Design," ACM Computing Surveys, 10, 2, June 1978, pp. 123-166.

[ROSE78] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems, 3, 2, June 1978, pp. 178-198.

[ROTH77] J.B. Rothnie, Jr., and N. Goodman, "A Survey of Research and Development in Distributed Database Management," Proc. Third Int. Conf. on Very Large Databases, IEEE, 1977.

[SALT78] J.H. Saltzer, "Research Problems of Decentralized Systems with Largely Autonomous Nodes," Operating System Reviews, 12, 1, January 1978, pp. 43-52.

[SCHA78] R. Schapiro, and R. Millstein, "Failure Recovery in a Distributed Database System," Proc. 1978 COMPCON Conference, September 1978.

[STON79] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," IEEE Transactions on Software Engineering, May 1979.

[SUNS79] C.A. Sunshine, "Formal Techniques for Protocol Specification and Verification," IEEE Computer, September 1979, pp. 20-27.

[SVOB79] L. Svobodova, "Reliability Issues in Distributed Information Processing Systems," Proc. 9th IEEE Fault Tolerant Computing Conference, Madison, Wisc., June 1979.

[THOM79] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control," Transactions on Database Systems, 4, 2, June 1979.

[TRAI79] I.L. Traiger, et al., "Transactions and Consistency in Distributed Database Systems," IBM Research Division, Report RJ2555(33155), San Jose, California, June 1979.

[VERH78] J.S.M. Verhofstad, "Recovery Techniques for Database Systems," ACM Computing Surveys, 10, 2, June 1978, pp. 167-195.

[ZAFI79] P. Zafiropulo, et al., "Towards Analyzing and Synthesizing Protocols," IBM, RZ 963 (No. 33588), July 1979.