CONCURRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS

Philip A. Bernstein
Wang Institute of Graduate Studies

Vassos Hadzilacos
University of Toronto

Nathan Goodman
Kendall Square Research Corporation

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts ■ Menlo Park, California
Don Mills, Ontario ■ Wokingham, England ■ Amsterdam ■ Sydney
Singapore ■ Tokyo ■ Madrid ■ Bogotá ■ Santiago ■ San Juan

This book is in the Addison-Wesley Series in Computer Science
Michael A. Harrison, Consulting Editor

Library of Congress Cataloging-in-Publication Data

Bernstein, Philip A.
Concurrency control and recovery in data-
base systems.

Includes index.
1. Data base management. 2. Parallel
processing (Electronic computers)
I. Hadzilacos, Vassos. II. Goodman, Nathan.
III. Title.
QA76.9.D3B48 1987 004.3 86-14127
ISBN 0-201-10715-5

PREFACE

The Subject

For over 20 years, businesses have been moving their data processing activities on-line. Many businesses, such as airlines and banks, are no longer able to function when their on-line computer systems are down. Their on-line databases must be up-to-date and correct at all times.

In part, the requirement for correctness and reliability is the burden of the application programming staff. They write the application programs that perform the business's basic functions: make a deposit or withdrawal, reserve a seat or purchase a ticket, buy or sell a security, etc. Each of these programs is designed and tested to perform its function correctly. However, even the most carefully implemented application program is vulnerable to certain errors that are beyond its control. These potential errors arise from two sources: concurrency and failures.

Multiprogramming is essential for attaining high performance. Its effect is to allow many programs to interleave their executions. That is, they execute *concurrently.* When such programs interleave their accesses to the database, they can interfere. Avoiding this interference is called the *concurrency control problem.*

Computer systems are subject to many types of failures. Operating systems fail, as does the hardware on which they run. When a failure occurs, one or more application programs may be interrupted in midstream. Since the program was written to be correct only under the assumption that it executed in its entirety, an interrupted execution can lead to incorrect results. For example, a money transfer application may be interrupted by a failure after debiting one account but before crediting the other. Avoiding such incorrect results due to failures is called the *recovery problem*.

Systems that solve the concurrency control and recovery problems allow their users to assume that each of their programs executes atomically — as if no other programs were executing concurrently — and reliably — as if there were no failures. This abstraction of an atomic and reliable execution of a program is called a *transaction*.

A *concurrency control algorithm* ensures that transactions execute atomically. It does this by controlling the interleaving of concurrent transactions, to give the illusion that transactions execute serially, one after the next, with no interleaving at all. Interleaved executions whose effects are the same as serial executions are called *serializable*. Serializable executions are correct, because they support this illusion of transaction atomicity.

A *recovery algorithm* monitors and controls the execution of programs so that the database includes only the results of transactions that run to a normal completion. If a failure occurs while a transaction is executing, and the transaction is unable to finish executing, then the recovery algorithm must wipe out the effects of the partially completed transaction. That is, it must ensure that the database does not reflect the results of such transactions. Moreover, it must ensure that the results of transactions that do execute are never lost.

This book is about techniques for concurrency control and recovery. It covers techniques for centralized and distributed computer systems, and for single copy, multiversion, and replicated databases. These techniques were developed by researchers and system designers principally interested in transaction processing systems and database systems. Such systems must process a relatively high volume of short transactions for data processing. Example applications include electronic funds transfer, airline reservation, and order processing. The techniques are useful for other types of applications too, such as electronic switching and computer-aided design — indeed any application that requires atomicity and reliability of concurrently executing programs that access shared data.

The book is a blend of conceptual principles and practical details. The principles give a basic understanding of the essence of each problem and why each technique solves it. This understanding is essential for applying the techniques in a commercial setting, since every product and computing environment has its own restrictions and idiosyncrasies that affect the implementation. It is also important for applying the techniques outside the realm of database systems. For those techniques that we consider of most practical value, we explain what's needed to turn the conceptual principles into a workable database system product. We concentrate on those practical approaches that are most often used in today's commercial systems.

Serializability Theory

Whether by its native capabilities or the way we educate it, the human mind seems better suited for reasoning about sequential activities than concurrent ones. This is indeed unfortunate for the study of concurrency control algorithms. Inherent to the study of such algorithms is the need to reason about concurrent executions.

Over the years, researchers have developed an abstract model that simplifies this sort of reasoning. The model, called *serializability theory*, provides two important tools. First, it provides a notation for writing down concurrent executions in a clear and precise format, making it easy to talk and write about them. Second, it gives a straightforward way to determine when a concurrent execution of transactions is serializable. Since the goal of a concurrency control algorithm is to produce serializable executions, this theory helps us determine when such an algorithm is correct.

To understand serializability theory, one only needs a basic knowledge of directed graphs and partial orders. A comprehensive presentation of this material appears in most undergraduate textbooks on discrete mathematics. We briefly review the material in the Appendix.

We mainly use serializability theory to express example executions and to reason abstractly about the behavior of concurrency control and recovery algorithms. However, we also use the theory to produce formal correctness proofs of some of the algorithms. Although we feel strongly about the importance of understanding such proofs, we recognize that not every reader will want to take the time to study them. We have therefore isolated the more complex proofs in separate sections, which you can skip without loss of continuity. Such sections are marked by an asterisk (*). Less than 10 percent of the book is so marked.

Chapter Organization

Chapter 1 motivates concurrency control and recovery problems. It defines correct transaction behavior from the user's point of view, and presents a model for the internal structure of the database system that implements this behavior — the model we will use throughout the book. Chapter 2 covers serializability theory.

The remaining six chapters are split into two parts: Chapters 3–5 on concurrency control and Chapters 6–8 on recovery.

In Chapter 3 we cover two phase locking. Since locking is so popular in commercial systems, we cover many of the variations and implementation details used in practice. The performance of locking algorithms is discussed in a section written for us by Dr. Y.C. Tay. We also discuss non–two-phase locking protocols used in tree structures.

In Chapter 4 we cover concurrency control techniques that do not use locking: timestamp ordering, serialization graph testing, and certifiers (i.e., optimistic methods). These techniques are not widely used in practice, so the chapter is somewhat more conceptual and less implementation oriented than Chapter 3. We show how locking and non-locking techniques can be integrated into hundreds of variations.

In Chapter 5 we describe concurrency control for multiversion databases, where the history of values of each data object is maintained as part of the database. As is discussed later in Chapter 6, old versions are often retained for recovery purposes. In this chapter we show that they have value for concurrency control too. We show how each of the major concurrency control and recovery techniques of Chapters 3 and 4 can be used to manage multiversion data.

In Chapter 6 we present recovery algorithms for centralized systems. We emphasize undo-redo logging because it demonstrates most of the recovery problems that all techniques must handle, and because it is especially popular in commercial systems. We cover other approaches at a more conceptual level: deferred updating, shadowing, checkpointing, and archiving.

In Chapter 7 we describe recovery algorithms for distributed systems where a transaction may update data at two or more sites that only communicate via messages. The critical problem here is *atomic commitment*: ensuring that a transaction's results are installed either at all sites at which it executed or at none of them. We describe the two phase and three phase commit protocols, and explain how each of them handles site and communications failures.

In Chapter 8 we treat the concurrency control and recovery problem for replicated distributed data, where copies of a piece of data may be stored at multiple sites. Here the concurrency control and recovery problems become closely intertwined. We describe several approaches to these problems: quorum consensus, missing writes, virtual partitions, and available copies. In this chapter we go beyond the state-of-the-art. No database systems that we know of support general purpose access to replicated distributed data.

Chapter Prerequisites

This book is designed to meet the needs of both professional and academic audiences. It assumes background in operating systems at the level of a one semester undergraduate course. In particular, we assume some knowledge of the following concepts: concurrency, processes, mutual exclusion, semaphores, and deadlocks.

We designed the chapters so that you can select whatever ones you wish with few constraints on prerequisites. Chapters 1 and 2 and Sections 3.1, 3.2, 3.4, and 3.5 of Chapter 3 are all that is required for later chapters. The subsequent material on concurrency control (the rest of Chapter 3 and Chapters 4–5) is largely independent of the material on recovery (Chapters 6–8). You can go as far into each chapter sequence as you like.

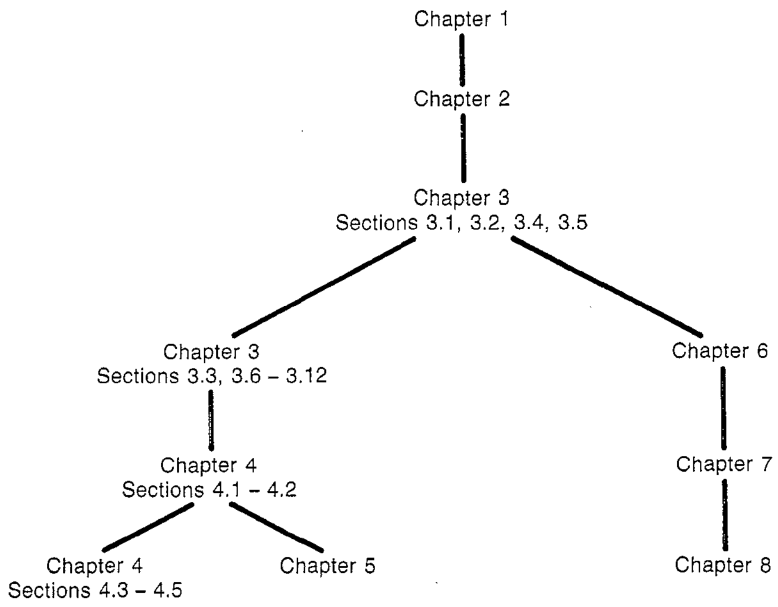

FIGURE 1
Dependencies between Chapters

A minimal survey of centralized concurrency control and recovery would include Sections 3.1–3.7, 3.12, and 3.13 of Chapter 3 and Sections 6.1–6.4 and 6.8 of Chapter 6. This material covers the main techniques used in commercial database systems, namely, locking and logging. In length, it's about a quarter of the book.

You can extend your survey to distributed (nonreplicated) data by adding Sections 3.10 and 3.11 (distributed locking) and Chapter 7 (distributed recovery). You can extend it to give a more complete treatment of centralized systems by adding the remaining sections of Chapters 3 and 6, on locking and recovery, and Chapter 5, on multiversion techniques (Section 5.3 requires Section 4.2 as a prerequisite). As we mentioned earlier, Chapter 4 covers non-locking concurrency control methods, which are conceptually important, but are not used in many commercial products.

Chapter 8, on replicated data, requires Chapters 3, 6, and 7 as prerequisites; we also recommend Section 5.2, which presents an analogous theory for multiversion data. Figure 1 summarizes these prerequisite dependencies.

We have included a substantial set of problems at the end of each chapter. Many problems explore dark corners of techniques that we didn't have the space to cover in the chapters themselves. We think you'll find them interesting reading, even if you choose not to work them out.

For Instructors

We designed the book to be useful as a principal or supplementary textbook in a graduate course on database systems, operating systems, or distributed systems. The book can be covered in as little as four weeks, or could consume an entire course, depending on the breadth and depth of coverage and on the backgrounds of the students.

You can augment the book in several ways depending on the theme of the course:

- Distributed Databases — distributed query processing, distributed database design.
- Transaction Processing — communications architecture, applications architecture, fault-tolerant computers.
- Distributed Computing — Byzantine agreement, network topology maintenance and message routing, distributed operating systems.
- Fault Tolerance — error detecting codes, Byzantine agreement, fault-tolerant computers.
- Theory of Distributed Computing — parallel program verification, analysis of parallel algorithms.

In a theoretical course, you can augment the book with the extensive mathematical material that exists on concurrency control and recovery.

The exercises supply problems for many assignments. In addition, you may want to consider assigning a project. We have successfully used two styles of project.

The first is an implementation project to program a concurrency control method and measure its performance on a synthetic workload. For this to be workable, you need a concurrent programming environment in which processing delays can be measured with reasonable accuracy. Shared memory between processes is also very helpful. We have successfully used Concurrent Euclid for such a project [Holt 83].

The second type of project is to take a concurrency control or recovery algorithm described in a research paper, formalize its behavior in serializability theory, and prove it correct. The bibliography is full of candidate examples. Also, some of the referenced papers are abstracts that do not contain proofs. Filling in the proofs is a stimulating exercise for students, especially those with a theoretical inclination.

Acknowledgments

In a sense, work on this book began with the SDD-1 project at Computer Corporation of America (CCA). Under the guidance and support of Jim Rothnie, two of us (Bernstein and Goodman) began our study of concurrency

control in database systems. He gave us an opportunity that turned into a career. We thank him greatly.

We wrote this book in part to show that serializability theory is an effective way to think about practical concurrency control and recovery problems. This goal required much research, pursued with the help of graduate students, funding agencies, and colleagues. We owe them all a great debt of gratitude. Without their help, this book would not have been written.

Our research began at Computer Corporation of America, funded by Rome Air Development Center, monitored by Tom Lawrence. We thank Tom, and John and Diane Smith at CCA, for their support of this work, continuing well beyond those critical first years. We also thank Bob Grafton, at the Office for Naval Research, whose early funding helped us establish an independent research group to pursue this work. We appreciate the steady and substantial support we received throughout the project from the National Science Foundation, and more recently from the Natural Sciences and Engineering Research Council of Canada, Digital Equipment Corporation, and the Wang Institute of Graduate Studies. We thank them all for their help.

Many colleagues helped us with portions of the research that led to this book. We thank Rony Attar, Catriel Beeri, Marco Casanova, Ming-Yee Lai, Christos Papadimitriou, Dennis Shasha, Dave Shipman, Dale Skeen, and Wing Wong.

We are very grateful to Dr. Y.C. Tay of the University of Singapore for writing an important section of Chapter 3 on the performance of two phase locking. He helped us fill an important gap in the presentation that would otherwise have been left open.

We gained much from the comments of readers of early versions of the chapters, including Catriel Beeri, Amr El Abbadi, Jim Gray, Rivka Ladin, Dan Rosenkrantz, Oded Shmueli, Jack Stiffler, Mike Stonebraker, and Y.C. Tay. We especially thank Gordon McLean and Irv Traiger, whose very careful reading of the manuscript caught many errors and led to many improvements. We also thank Ming-Yee Lai and Dave Lomet for their detailed reading of the final draft.

We are especially grateful to Jenny Rozakis for her expert preparation of the manuscript. Her speed and accuracy saved us months. We give her our utmost thanks.

We also thank our editor, Keith Wollman, and the entire staff at Addison-Wesley for their prompt and professional attention to all aspects of this book.

We gratefully acknowledge the Association for Computing Machinery for permission to use material from "Multiversion Concurrency Control — Theory and Algorithms," *ACM Transaction on Database Systems* 8, 4 (Dec. 1983), pp. 465–483 (© 1983, Association for Computing Machinery, Inc.) in Chapter 5; and "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems* 9, 4 (Dec. 1984), pp. 596–615 (© 1984, Association for Computing Machin-

ery, Inc.) in Chapter 8. We also acknowledge Academic Press for allowing us to use material from "Serializability Theory for Replicated Databases," *Journal of Computer and System Sciences* 31, 3 (1986) (© 1986, Academic Press) in Chapter 8; and Springer-Verlag for allowing us to use material from "A Proof Technique for Concurrency Control and Recovery Algorithms for Replicated Databases," *Distributed Computing* 2, 1 (1986) in Chapter 8.

Finally, we thank our families, friends, and colleagues for indulging our bad humor as a two-year project stretched out to six. Better days are ahead.

Tyngsboro, Mass. P.A.B.
Toronto, Canada V.H.
Cambridge, Mass. N.G.

7 DISTRIBUTED RECOVERY

7.1 INTRODUCTION

In this chapter we discuss the reliability issues that arise when transactions are processed in a distributed database system. We are assuming that data items are not replicated, that is, each data item is stored at a single site. This means that there is a unique scheduler and data manager in charge of controlling access to any given data item. Data replication is addressed in the next chapter.

A distributed transaction *T* has a "home site" — the site where it originated. *T* submits its operations to the TM at its home site, and the TM subsequently forwards the operations to the appropriate sites. A Read(*x*) or Write(*x*) operation is forwarded to the site where *x* is stored and is processed by the scheduler and DM of that site as if it were an operation submitted by a local transaction. The result of the operation is then returned to the TM of *T*'s home site. Thus, aside from the straightforward matter of routing requests and responses between sites, the processing of Read and Write operations in a distributed DBS is no different than in a centralized one.

Consider now the Commit operation of *T*. To which sites should this operation be forwarded? Unlike Read(*x*) or Write(*x*), which concern only the site where *x* is stored, a Commit operation concerns all sites involved in the processing of *T*. Consequently, the TM of *T*'s home site should pass the Commit operation of *T* to all sites where *T* accessed data items. The same is true for Abort. Thus, the processing of a logically single operation (Commit or Abort) must take place in multiple places in a distributed DBS. This is a substantial difference between centralized and distributed transaction processing.

The problem is more subtle than it may appear at first. Merely having the TM of a distributed transaction's home site send Commit operations to all other sites is not enough. This is because a transaction is not committed by virtue of the TM's sending a Commit, but rather by virtue of the DM's executing the Commit. It is possible that the TM sends Commit to the scheduler but the scheduler rejects it and aborts the transaction. In this case, if the transaction is distributed, it should abort at all other sites where it accessed data items.

Another significant difference between transaction processing in a centralized and a distributed DBS concerns the nature of failures. In a centralized system, a failure is an all-or-nothing affair. Either the system is working and transactions are processed routinely, or the system has failed and no transaction can be processed at all. In a distributed system, however, we can have partial failures. Some sites may be working while others have failed.

The fact that failures in distributed systems do not necessarily have the crippling effect they do in centralized ones creates opportunities for greater reliability. This is one of the most widely advertised features of distribution. Less widely advertised is the fact that the partial failures that make this possible are often the source of non-trivial problems that must be solved before the "opportunities for greater reliability" can be realized.

In transaction processing (and in the absence of data replication), the only such non-trivial problem is that of consistent termination. As we saw, the Commit or Abort operation of a distributed transaction must be processed at all sites where the transaction accessed data items. Ensuring that a single logical action (Commit or Abort) is consistently carried out at multiple sites is complicated considerably by the prospect of partial failures.

An algorithm that ensures this consistency is called an *atomic commitment protocol* (*ACP*). Our main goal in this chapter is to present ACPs that are as resilient to failures as possible. Before we do so, we must examine in more detail the nature of failures that the protocol must worry about.

7.2 FAILURES IN A DISTRIBUTED SYSTEM

A distributed system consists of two kinds of components: sites, which process information, and communication links, which transmit information from site to site. A distributed system is commonly depicted as a graph where nodes are sites and undirected edges are bidirectional communication links (see Fig. 7–1).

We assume that this graph is connected, meaning that there is a path from every site to every other. Thus, every two sites can communicate either directly via a link joining them, or indirectly via a chain of links. The combination of hardware and software that is responsible for moving messages between sites is called a *computer network*. We won't worry about how to route messages

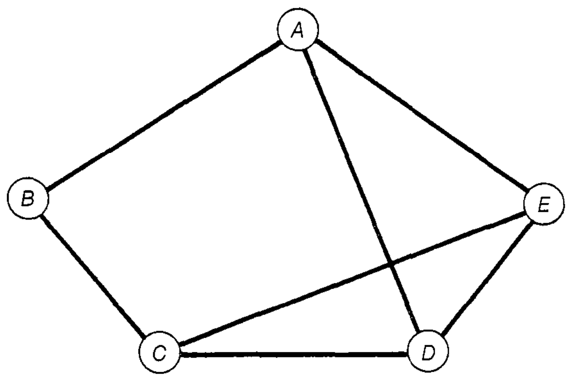

FIGURE 7-1
A Computer Network

from one site to another, since routing is a computer network service normally available to the distributed database system.

Site Failures

When a site experiences a system failure, processing stops abruptly and the contents of volatile storage are destroyed. In this case, we'll say the site has *failed*. When the site recovers from a failure it first executes a *recovery procedure* (called *Restart* in Chapter 6), which brings the site to a consistent state so it can resume normal processing.

In this model of failure, a site is always either working correctly (is *operational*) or not working at all (is *down*). It never performs incorrect actions. This type of behavior is called *fail-stop*, because sites fail only by stopping.

Surely this is an idealization of a site's possible faulty behavior. Computers can occasionally act incorrectly due to software or hardware bugs. By using extensive testing during implementation and manufacturing, and built-in redundancy in hardware and software, one can build systems that approximate fail-stop behavior. But we will not discuss these techniques in this book. We'll simply assume that sites are fail-stop. The correctness of the protocols we'll discuss in this chapter depends on this assumption.

Even though each site either is functioning properly or has failed, different sites may be in different states. A *partial failure* is a situation where some sites are operational while others are down. A *total failure* occurs when all sites are down.

Partial failures are tricky to deal with. Fundamentally, this is because operational sites may be uncertain about the state of failed ones. As we'll see, operational sites may become blocked, unable to commit or abort a transaction, until such uncertainty is resolved. An important design goal of atomic commitment protocols is to minimize the effect of one site's failure on other sites' ability to continue processing.

Communication Failures

Communication links are also subject to failures. Such failures may prevent processes at different sites from communicating. A variety of communication failures are possible: A message may be corrupted due to noise in a link; a link may malfunction temporarily, causing a message to be completely lost; or a link may be broken for a while, causing all messages sent through it to be lost.

Message corruption can be effectively handled by using error detecting codes, and by retransmitting a message in which the receiver detects an error. Loss of messages due to transient link failures can be handled by retransmitting lost messages. Also, the probability of losing messages due to broken links can be reduced by rerouting. If a message is sent from site A to site B, but the network is unable to deliver the message due to a broken link, it may attempt to find another path from A to B whose intermediate links and sites are functioning properly. Error correcting codes, message retransmission, and rerouting are usually provided by computer network protocols. We'll take them for granted.

Unfortunately, even with automatic rerouting, a combination of site and link failures can disable the communication between sites. This will happen if *all* paths between two sites A and B contain a failed site or a broken link. This phenomenon is called a *network partition*. In general, a network partition divides up the operational sites into two or more *components*, where every two sites within a component can communicate with each other, but sites in different components cannot. For example, Fig. 7–2 shows a partition of the system of Fig. 7–1. The partition consists of two components, $\{B, C\}$ and $\{D, E\}$, and is caused by the failure of site A and links (C, D) and (C, E).

As sites recover and broken links are repaired, communication is reestablished between sites that could not previously exchange messages, thereby merging components. For example, in Fig. 7–2, if site A recovers or if either link (C, D) or (C, E) is repaired, the two components merge and every pair of operational sites can communicate.

We can reduce the probability of a network partition by designing a highly connected network, that is, a network where the failure of a few sites and links will not disrupt all paths between any pair of sites. However, making a network highly connected requires the use of more components and therefore entails more expense. Moreover, the network's topology is often constrained by other factors, such as geography or the communication medium. Thus, our ability to avoid partitions is limited.

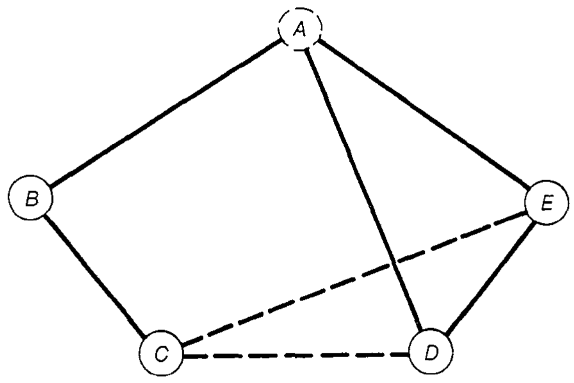

FIGURE 7–2
A Network Partition
Components shown in broken lines are faulty.

To sum up, a *communication failure* occurs when a site A is unable to communicate with site B, even though neither site is down. Network partitions are one cause of communication failures. (We'll see another one shortly.) If two sites can communicate, messages are delivered correctly (uncorrupted).

Undeliverable Messages

Site and communication failures require us to deal with undeliverable messages. A message may be undeliverable because its recipient is down when the message arrives, or because its sender and recipient are in different components of a network partition. There are two options:

1. The message *persists*. The computer network stores the message, and delivers it to its destination when that becomes possible.
2. The message is *dropped*. The computer network makes no further attempt to deliver it.

We'll adopt option (2) — as, alas, many postal services do. Option (1) is *not* routinely supported by computer networks. It requires fairly elaborate protocols, quite similar to ACPs, and therefore merely moves the atomic commitment problem to a different part of the system (see Exercise 7.1).

Some computer networks that adopt option (2) attempt to notify the sender of an undeliverable message that the message was dropped. But this is inherently unreliable. If a site fails to acknowledge the receipt of a message, the network cannot tell whether the site did not receive the message or it received the message but failed before acknowledging it. Even if it could make this distinction, notifying the sender of nondelivery may lead to unbounded recursion. If a notification message itself cannot be delivered, its sender (the notifier) must be notified and so on. Thus, such notifications of nondelivery can*not* be relied upon. We will therefore assume they don't exist. For our purposes, undeliverable messages simply vanish.

Detecting Failures by Timeouts

Both site failures and communication failures manifest themselves as the inability of one site to exchange messages with another. That is, if site A cannot communicate with site B, it is either because B has failed or because A and B belong to different components of a partition. In general, A cannot distinguish these two cases. It just knows that it can't communicate with B.

How can A find out that it can't communicate with B? Usually this is done by using *timeouts*. A sends a message to B and waits for a reply within a predetermined period of time δ called the *timeout period*. If a reply arrives, clearly A and B can communicate, as evidenced by the pair of messages just exchanged. If the period δ elapses and A has not yet received a reply, A concludes that it cannot communicate with B. δ must be chosen to be the maximum possible time it can take for the message to travel from A to B, for B to process the message and generate the reply, and for the reply to travel back to A. Computing a value for the timeout period is not a simple matter. It depends on many hard-to-quantify variables: the physical characteristics of the sites and communication lines, the system load, message routing algorithms, and accuracy of clocks, among others. Pragmatically, it is usually possible to select a timeout period that works well most of the time. When we use the timeout mechanism, we assume that an appropriate value for δ has been determined.

If the timeout period is underestimated, a site may think it cannot communicate with another when, in fact, it can. This can also happen because the clock that's measuring the timeout period is too fast. Such errors are called *timeout failures* or *performance failures*. Timeout failures are, in effect, communication failures. However, unlike network partitions, they can give rise to very peculiar situations, such as A thinking that it can communicate with B but B thinking it can't communicate with A; or A thinking it can communicate with B and B with C, but A thinking it cannot communicate with C.

In Chapters 7 and 8, we'll be careful to indicate whether each algorithm under consideration can tolerate site failures only, or both site and communication failures.

7.3 ATOMIC COMMITMENT

Consider a distributed transaction T whose execution involves sites S_1, S_2, ..., S_n. Suppose the TM at site S_1 supervises T's execution. Before the TM at S_1 can send Commit operations for T to S_1, S_2, ..., S_n, it must make sure that the scheduler and DM at each of these sites is ready and willing to process that Commit. Otherwise, T might wind up committing at some sites and aborting at others, thereby terminating inconsistently. Let's look at the conditions that a scheduler or DM must satisfy to be "ready and willing" to commit a transaction.

The scheduler at a site may agree to process Commit(T) as long as T satisfies the recoverability condition at that site. That is, every value read by T at that site was written by a transaction that has committed. Note that if the scheduler produces executions that avoid cascading aborts (or *a fortiori*, are strict), then this is true at all times. In this case, since the scheduler is always able to process Commit(T), S_i's TM need not get the scheduler's approval to send a Commit.

The DM at a site may agree to process Commit(T) as long as T satisfies the Redo Rule at that site. That is, all values written by T at that site are in stable storage — the stable database or the log, depending on the DM's recovery algorithm. If T has submitted only Reads to some site, it need *not* request the consent of that site's DM.

The TM at S_1 can issue Commit(T) to the schedulers and DMs of S_1, S_2, ..., S_n only after having received the schedulers' and DMs' consent from *all* these sites. In essence, this is the *two phase commit* (2PC) protocol that we'll study in detail in the next section. Why must we devote a separate section to what seems like such a simple idea? The reason is that the preceding discussion does not address site or communication failures. What if one or more sites fail during this process? What if one or more messages are lost? The real difficulty of atomic commitment is to design protocols that provide maximum resistance to such failures.

As an aside, we have already encountered a protocol analogous to 2PC in our discussion of distributed certifiers (cf. Section 4.4). To certify a distributed transaction T, the local certifiers of all sites where T executed had to agree. If even one site did not certify T, the transaction had to be aborted at *all* sites. In our discussion of distributed certification, we evaded the issue of site and communication failures. Our discussion of how to handle failures in 2PC will apply to the distributed certification protocol as well. In practice, the two protocols would be combined.

To simplify the discussion and concentrate on the essentials of atomic commitment, it is convenient to deviate from the TM-scheduler-DM model. To separate atomic commitment from the other aspects of transaction processing, we'll assume that for each distributed transaction T, there is a process at every site where T executed. These processes carry out the atomic commitment protocol for T. The process at T's home site is called T's *coordinator*. The remaining processes are T's *participants*. The coordinator knows the names of all the participants, so it can send them messages. The participants know the name of the coordinator, but they don't necessarily know each other.

We emphasize that the coordinator and participants are abstractions that we adopt only for pedagogical convenience. Do not imagine that implementing an atomic commitment protocol necessarily requires one process per transaction at each site involved in the transaction's execution. In most cases, such an implementation would be inefficient because managing so many processes would be too expensive. The coordinator and participant processes are abstractions whose functionality can be provided at each site by one or more processes, possibly shared by many transactions.

We'll also assume that each site contains a *distributed transaction log* (*DT log*) where coordinators and participants at that site can record information about distributed transactions. The DT log must be kept in stable storage, because its contents must survive site failures. In practice, the DT log may be part of the site's DM log. Again, for expository convenience we'll assume it's a separate entity.

Roughly speaking, an *atomic commitment protocol* (*ACP*) is an algorithm for the coordinator and participants such that either the coordinator and all participants commit the transaction or they all abort it. We can state this more precisely as follows. Each process may cast exactly one of two *votes*: *Yes* or *No*, and can reach exactly one of two *decisions*: *Commit* or *Abort*. An ACP is an algorithm for processes to reach decisions such that:

AC1: All processes that reach a decision reach the same one.

AC2: A process cannot reverse its decision after it has reached one.

AC3: The Commit decision can only be reached if *all* processes voted Yes.

AC4: If there are no failures and all processes voted Yes, then the decision will be to Commit.

AC5: Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

This abstract formulation of the problem relates to transaction processing in the TM-scheduler-DM model in the following way. The process at site *A* votes Yes only if *A*'s scheduler and DM are "ready and willing" to commit the transaction (as we explained earlier in this section). If the process decides Commit (or Abort), then *A*'s DM will execute the Commit (or Abort) operation. In executing that operation, site *A* acts exactly like an autonomous centralized DBS, using one of the algorithms from Chapter 6. Indeed, different sites processing a transaction could be using different DM algorithms.

Some discussion of these conditions is now in order. Condition AC1 says that the transaction terminates consistently. Note that we do not require that all processes reach a decision. This would be an unattainable goal, since a process may fail and never recover. (We say that a process fails when the site on which it runs fails.) We do not even require that all processes that remain operational reach a decision. This is also unattainable, though the reason why is less obvious (see Proposition 7.1 at the end of this section). However, we do require that all processes be able to reach a decision once failures are repaired (AC5). This requirement excludes from consideration uninteresting protocols that allow processes to remain forever undecided in any execution where some failure has taken place.

Condition AC2 says that the termination of a transaction at a site is an irrevocable decision. If a transaction commits (or aborts), it cannot be later aborted (or committed).

Condition AC3 says that a transaction cannot commit unless all sites involved in its execution agree to do so. AC4 is a weak version of the converse of AC3. Among other things, it assures that there are circumstances in which Commit must be decided and thus excludes from consideration trivial (and useless) protocols in which processes always decide Abort! However, we do not require the converse of AC3, in its full generality. It is possible (if failures occur) for all processes to have voted Yes, and yet the decision to be Abort (see Exercise 7.2). A very important consequence of AC3 is that each process can unilaterally decide Abort at any time, if it has not yet voted Yes. On the other hand, after voting Yes a process cannot take unilateral action. The period between the moment a process votes Yes and the moment it has received sufficient information to know what the decision will be is called the *uncertainty period* for that process. A process is called *uncertain* while it is in its uncertainty period. During this period the process does not know whether it will eventually decide Commit or Abort, nor can it unilaterally decide Abort.

Scenario I: A failure disables communication between a process *p* and all other processes, while *p* is uncertain. By the definition of uncertainty period, *p* cannot reach a decision until after the communication failure has been repaired.

When a process must await the repair of failures before proceeding, we say that it is *blocked*. Blocking is undesirable, because it can cause processes to wait for an arbitrarily long period of time. A transaction may remain unterminated, uselessly consuming resources (such as holding locks), for arbitrarily long at the blocked process's site. Scenario I shows how communication failures can cause a process to become blocked.

Scenario II: *p* fails while in its uncertainty period. When *p* recovers, it cannot reach a decision on its own. It must communicate with other processes to find out what the decision was.

The ability of a recovering process to reach a decision without communicating with other processes is called *independent recovery*. This ability is very attractive, because it makes recovery cheaper and simpler. Moreover, lack of independent recovery in conjunction with *total failures* (when all processes fail) gives rise to blocking. To see this, suppose that p in Scenario II is the first process to recover from a total failure. Since p is uncertain, it must communicate with other processes before it can reach a decision. But it can't communicate with them, since they all are down. Thus p is blocked.

These two scenarios show that failures while a process is uncertain can cause serious problems. Can we design an ACP that eliminates uncertainty periods? Unfortunately, not. Doing so would essentially require that a process cast its vote *and* learn the votes of *all* other processes all at once. In general, this is impossible. Thus, we have the following important observations.

Proposition 7.1: If communication failures or total failures are possible, then every ACP may cause processes to become blocked. □

Proposition 7.2: No ACP can guarantee independent recovery of failed processes. □

Proposition 7.1 does not preclude the existence of a non-blocking ACP if only site failures, but not *total* site failures, can occur. In fact, such a protocol exists, as we'll see in Section 7.5.

Propositions 7.1 and 7.2 can be formulated as theorems. Unfortunately, we lack a precise enough model of distributed computation to carry out rigorous proofs, the keys to which are Scenarios I and II. Developing such a model would lead us astray. The Bibliographic Notes cite proofs of these propositions.

7.4 THE TWO PHASE COMMIT PROTOCOL

The simplest and most popular ACP is the *two phase commit* (*2PC*) protocol. Assuming no failures, it goes roughly as follows:

1. The coordinator sends a VOTE-REQ[1] (i.e., vote request) message to all participants.
2. When a participant receives a VOTE-REQ, it responds by sending to the coordinator a message containing that participant's vote: YES or NO. If the participant votes No, it decides Abort and stops.
3. The coordinator collects the vote messages from all participants. If all of them were YES and the coordinator's vote is also Yes, then the coordinator decides Commit and sends COMMIT messages to all participants. Otherwise, the coordinator decides Abort and sends ABORT messages to

[1]We use all small capital letters to indicate messages.

all participants that voted Yes (those that voted No already decided Abort in step (2)). In either case, the coordinator then stops.

4. Each participant that voted Yes waits for a COMMIT or ABORT message from the coordinator. When it receives the message, it decides accordingly and stops.

The two phases of 2PC are the voting phase (steps (1) and (2)) and the decision phase (steps (3) and (4)). A participant's uncertainty period starts when it sends a YES to the coordinator (step (2)) and ends when it receives a COMMIT or ABORT (step (4)). The coordinator has no uncertainty period since it decides as soon as it votes — with the knowledge, of course, of the participants' votes (step (3)).

It is easy to see that 2PC satisfies conditions AC1 – AC4. Unfortunately, as presented so far, it does not satisfy AC5 for two reasons. First, at various points of the protocol, processes must wait for messages before proceeding. However, such messages may not arrive due to failures. Thus, processes may be waiting forever. To avoid this, timeouts are used. When a process' waiting is interrupted by a timeout, the process must take special action, called a *timeout action*. Thus, to satisfy AC5, we must supply suitable timeout actions for each protocol step in which a process is waiting for a message.

Second, when a process recovers from a failure, AC5 requires that the process attempt to reach a decision consistent with the decision other processes may have reached in the meanwhile. (It *may* be that such a decision can't be made until after some other failures have been repaired as well.) Therefore, a process must keep some information in stable storage, specifically in the DT log. To satisfy AC5 we must indicate what information to keep in the DT log and how to use it upon recovery.

We consider these two issues in turn.

Timeout Actions

There are three places in 2PC where a process is waiting for a message: in the beginning of steps (2), (3) and (4). In step (2), a participant waits for a VOTE-REQ from the coordinator. This happens before the participant has voted. Since any process can unilaterally decide Abort before it votes Yes, if a participant times out waiting for a VOTE-REQ, it can simply decide Abort and stop.

In step (3) the coordinator is waiting for YES or NO messages from all the participants. At this stage, the coordinator has not yet reached any decision. In addition, no participant can have decided Commit. Therefore, the coordinator can decide Abort but must send ABORT to every participant from which it received a YES.

In step (4), a participant p that voted Yes is waiting for a COMMIT or ABORT from the coordinator. At this point p is uncertain. Therefore, unlike the previous two cases where a process can unilaterally decide, in this case the participant must consult with other processes to find out what to decide. This consultation is carried out in a *termination protocol* (for 2PC).[2]

The simplest termination protocol is the following: *p* remains blocked until it can re-establish communication with the coordinator. Then, the coordinator can tell *p* the appropriate decision. The coordinator can surely do so, since it has no uncertainty period. This termination protocol satisfies condition AC5, because if all failures are repaired, *p* will be able to communicate with the coordinator and thereby reach a decision.

The drawback of this simple termination protocol is that *p* may be blocked unnecessarily. For example, suppose there are two participants *p* and *q*. The coordinator might send a COMMIT or ABORT to *q* but fail just before sending it to *p*. Thus, even though *p* is uncertain, *q* is not. If *p* can communicate with *q*, it can find out the decision from *q*. It need not block waiting for the coordinator's recovery.

This suggests the need for participants to know each other, so they can exchange messages directly (without the mediation of the coordinator). Recall that our description of the atomic commitment problem states that the coordinator knows the participants and the participants know the coordinator, but that the participants do not initially know each other. This does not present any great difficulty. We can assume that the coordinator attaches the list of the participants' identities to the VOTE-REQ message it sends to each of them. Thus, participants get to know each other when they receive that message. The fact that they do not know each other earlier is of no consequence for our purposes, since a participant that times out before receiving VOTE-REQ will unilaterally decide Abort.

This example leads us to the *cooperative termination protocol*: A participant *p* that times out while in its uncertainty period sends a DECISION-REQ message to every other process, *q*, to inquire whether *q* either knows the decision or can unilaterally reach one. In this scenario, *p* is the *initiator* and *q* a *responder* in the termination protocol. There are three cases:

1. *q* has already decided Commit (or Abort): *q* simply sends a COMMIT (or ABORT) to *p*, and *p* decides accordingly.
2. *q* has not voted yet: *q* can unilaterally decide Abort. It then sends an ABORT to *p*, and *p* therefore decides Abort.
3. *q* has voted Yes but has not yet reached a decision: *q* is also uncertain and therefore cannot help *p* reach a decision.

With this protocol, if *p* can communicate with *some q* for which either (1) or (2) holds, then *p* can reach a decision without blocking. On the other hand,

[2]In general, a termination protocol is invoked by a process when it fails to receive an anticipated message while in its uncertainty period. Different ACPs have different termination protocols associated with them.

if (3) holds for all processes with which p can communicate, then p is blocked. This predicament will persist until enough failures are repaired to enable p to communicate with a process q for which either (1) or (2) applies. At least one such process exists, namely, the coordinator. Thus this termination protocol satisfies AC5.

In summary, even though the cooperative termination protocol reduces the probability of blocking, it does not eliminate it. In view of Proposition 7.1, this is hardly surprising. However, even with the cooperative termination protocol, 2PC is subject to blocking *even if only site failures occur* (see Exercise 7.3).

Recovery

Consider a process p recovering from a failure. To satisfy AC5, p must reach a decision consistent with that reached by the other processes — if not immediately upon recovery, then some time after all other failures are also repaired.

Suppose that when p recovers it remembers its state at the time it failed — we'll discuss later how this is done. If p failed before having sent YES to the coordinator (step (2) of 2PC), then p can unilaterally decide Abort. Also, if p failed after having received a COMMIT or ABORT from the coordinator or after having unilaterally decided Abort, then it has already decided. In these cases, p can recover independently.

However, if p failed while in its uncertainty period, then it cannot decide on its own when it recovers. Since it had voted Yes, it is possible that all other processes did too, and they decided Commit while p is down. But it is also possible that some processes either voted No or didn't vote at all and Abort was decided. p can't distinguish these two possibilities based on information available locally and must therefore consult with other processes to make a decision. This is a reflection of the inability to have independent recovery (Proposition 7.2).

In this case, p is in exactly the same state as if it had timed out waiting for a COMMIT or ABORT from the coordinator. (Think of p as having used an extraordinarily long timeout period, lasting for the duration of its failure.) Thus, p can reach a decision by using the termination protocol. Note that p may be blocked, since it may be able to communicate only with processes that are themselves uncertain.

To remember its state at the time it failed, each process must keep some information in its site's DT log, which survives failures. Of course, each process has access only to its local DT log. Assuming that the cooperative termination protocol is used, here is how the DT log is managed.

1. When the coordinator sends VOTE-REQs, it writes a **start-2PC** record in the DT log. This record contains the identities of the participants, and may be written before or after sending the messages.

2. If a participant votes Yes, it writes a yes record in the DT log, *before* sending YES to the coordinator. This record contains the name of the coordinator and a list of the other participants (which is provided by the coordinator in VOTE-REQ). If the participant votes No, it writes an **abort** record either before or after the participant sends NO to the coordinator.
3. *Before* the coordinator sends COMMIT to the participants, it writes a **commit** record in the DT log.
4. When the coordinator sends ABORT to the participants, it writes an **abort** record in the DT log. The record may be written before or after sending the messages.
5. After receiving COMMIT (or ABORT), a participant writes a **commit** (or **abort**) record in the DT log.

In this discussion, writing a **commit** or **abort** record in the DT log is the act by which a process decides Commit or Abort.

At this point it is appropriate to comment briefly on the interaction between the commitment process and the rest of the transaction processing activity. Once the **commit** (or **abort**) record has been written in the DT log, the DM can execute the Commit (or Abort) operation. There are a number of details regarding how writing **commit** or **abort** records to the DT log relates to the processing of the commit or abort operations by the DM. For example, if the DT log is implemented as part of the DM log, the writing of the **commit** or **abort** record in the DT log may be carried out via a call to the Commit or Abort procedure of the local DM. In general, such details depend on which of the algorithms we discussed in Chapter 6 is used by the local DM (see Exercise 7.4).

When a site S recovers from a failure, the fate of a distributed transaction executing at S can be determined by examining its DT log:

- If the DT log contains a **start-2PC** record, then S was the host of the coordinator. If it also contains a **commit** or **abort** record, then the coordinator had decided before the failure. If neither record is found, the coordinator can now unilaterally decide Abort by inserting an **abort** record in the DT log. For this to work, it is crucial that the coordinator *first* insert the **commit** record in the DT log and *then* send COMMITS (point (3) in the preceding list).
- If the DT log doesn't contain a **start-2PC** record, then S was the host of a participant. There are three cases to consider:
  1. The DT log contains a **commit** or **abort** record. Then the participant had reached its decision before the failure.
  2. The DT log does not contain a **yes** record. Then either the participant failed before voting or voted No (but did not write an **abort** record before failing). (This is why the yes record must be written

before YES is sent; see point (2) in the preceding list.) It can therefore unilaterally abort by inserting an **abort** record in the DT log.

3. The DT log contains a yes but no **commit** or **abort** record. Then the participant failed while in its uncertainty period. It can try to reach a decision using the termination protocol. Recall that a yes record includes the name of the coordinator and participants, which are needed for the termination protocol.

Figures 7–3 and 7–4 give the 2PC protocol and the cooperative termination protocol, incorporating the preceding discussion on timeout actions and DT logging activity. The algorithms employed by each process are expressed in an *ad hoc*, but hopefully straightforward, language. We use **send** and **wait for** statements for inter-process communication. The statement "**send** *m* **to** *p*," where *m* is a message and *p* is one or more processes, causes the executing process to send *m* to all processes in *p*. The statement "**wait for** *m* **from** *p*," where *m* is one or more messages and *p* is a process, causes the executing process to suspend until *m* is received from *p*. If messages from multiple destinations are expected, the statement takes one of two forms: "**wait for** *m* **from all** *p*," in which case the waiting persists until messages *m* have been received from all processes in *p*, and "**wait for** *m* **from any** *p*," in which case waiting ends when *m* is received from *some* process in *p*. To avoid indefinite waiting, a **wait for** statement can be followed by a clause of the form "**on timeout** *S*," where *S* is some statement. This means that if the messages expected in the preceding **wait for** statement do not arrive within a predetermined timeout period, waiting is discontinued, statement *S* is executed, and control flows normally to the statement after the interrupted **wait for**, unless otherwise specified by *S*. If the expected messages arrive within the timeout period, *S* is ignored. We assume the timeout period is magically set to an appropriate value.

Although we have been presenting ACPs for a single transaction's termination, it is clear that the DT log will contain records describing the status of different transactions relative to atomic commitment. Thus to avoid confusing records of different transactions, the **start-2PC**, **yes**, **commit**, and **abort** records must contain the name of the transaction to which they refer. In addition, it is important to garbage collect DT log space taken up by outdated information. There are two basic principles regarding this garbage collection:

GC1: A site cannot delete log records of a transaction *T* from its DT log until at least after its RM has processed RM-Commit(*T*) or RM-Abort(*T*).

GC2: At least one site must not delete the records of transaction *T* from its DT log until that site has received messages indicating that RM-Commit(*T*) or RM-Abort(*T*) has been processed at all other sites where *T* executed.

Coordinator's algorithm

```
send VOTE-REQ to all participants;
write start-2PC record in DT log;
wait for vote messages (YES or NO) from all participants
    on timeout begin
        let $P_Y$ be the processes from which YES was received;
        write abort record in DT log;
        send ABORT to all processes in $P_Y$;
        return
    end;
if all votes were YES and coordinator votes Yes then begin
    write commit record in DT log;
    send COMMIT to all participants
end
else begin
    let $P_Y$ be the processes from which YES was received;
    write abort record in DT log;
    send ABORT to all processes in $P_Y$
end;
return
```

Participant's algorithm

```
wait for VOTE-REQ from coordinator
    on timeout begin
        write abort record in DT log;
        return
    end;
if participant votes Yes then begin
    write a yes record in DT log;
    send YES to coordinator;
    wait for decision message (COMMIT or ABORT) from coordinator
        on timeout initiate termination protocol /* cf. Fig. 7-4 */
    if decision message is COMMIT then write commit record in DT log
    else write abort record in DT log
end
else /* participant's vote is No */ begin
    write abort record in DT log;
    send NO to coordinator
end;
return
```

FIGURE 7–3
Two Phase Commit Protocol

Initiator's algorithm

```
start: send DECISION-REQ to all processes;
    wait for decision message from any process
         on timeout goto start; /* blocked! */
    if decision message is COMMMIT then
         write commit record in DT log
    else /* decision message is ABORT */
         write abort record in DT log;
    return
```

Responder's algorithm

```
    wait for DECISION-REQ from any process p;
    if responder has not voted Yes or has decided to Abort (i.e., has an
        abort record in DT log) then send ABORT to p
    else if responder has decided to Commit (i.e., has a commit
        record in DT log) then send COMMIT to p
    else /* responder is in its uncertainty period */ skip;[3]
    return
```

FIGURE 7–4
Cooperative Termination Protocol for 2PC

GC1 states that a site involved in *T*'s execution can't forget about *T* until after *T*'s effects at that site have been carried out. GC2 says that *some* site involved in *T*'s execution must remember *T*'s fate until that site knows that *T*'s effects have been carried out at *all* sites. If this were not true and a site recovered from a failure and found itself uncertain about *T*'s fate, it would never be able to find out what to decide about *T*, thus violating AC5.

GC1 can be enforced using information available locally at each site. However, GC2 calls for communication between sites. In particular, site *A* involved in *T*'s execution must acknowledge the processing of RM-Commit(*T*) or RM-Abort(*T*) at site *A* to each site for which GC2 must hold. There are two extremes in the spectrum of possible strategies for achieving GC2: GC2 is true for only one site, typically *T*'s coordinator, or GC2 is true for all sites involved in *T*'s execution (see Exercise 7.5).

Our study of ACPs from the viewpoint of a single transaction has also hidden the issue of site recovery. When a site recovers, it must complete the ACP for all transactions that might not have committed or aborted before the

[3]"Skip" is the "do-nothing" statement (no-op).

failure. At what point can the site resume normal transaction processing? After the recovery of a centralized DBS, transactions cannot be processed until Restart has terminated, thereby restoring the committed database state. A similar strategy for the recovery of a site in a distributed DBS is unattractive, in view of the possibility that some transactions are blocked. In this case, the DBS at the recovered site would remain inaccessible until *all* transactions blocked at that site were committed or aborted.

Methods to avoid this problem depend on the type of scheduler being used. Consider Strict 2PL. After a site's recovery procedure has made decisions for all unblocked transactions, it should ask its scheduler to reacquire the locks that blocked transactions owned before the failure. In fact, a blocked transaction T need only reacquire its write locks. Read locks may be left released because doing so does not violate either the two phase rule (T must have obtained all of its locks, or else it would have been aborted, instead of blocked) or the strictness condition (which requires that *write* locks be held until after Commit is processed). The problem is that lock tables are usually stored in main memory and are therefore lost in a system failure. To avoid losing this information, the process that manages T's atomic commitment at a site must record T's write locks at that site in the yes record it writes in the DT log. This is unnecessary if that information can be determined from the log maintained by the RM at that site. This is the case, for example, in the undo/redo algorithm described in Chapter 6, since before T votes Yes at a site all of its update records will be in the log (and therefore in stable storage). These records can be used to determine the set of T's write locks, which can then be set.

Evaluation of 2PC

One can evaluate an ACP according to many criteria:

- *Resiliency*: What failures can it tolerate?
- *Blocking*: Can processes be blocked? If so, under what conditions?
- *Time Complexity*: How long does it take to reach the decision?
- *Message Complexity*: How many messages are exchanged to reach a decision?

The first two criteria measure the protocol's *reliability*, and the other two its *efficiency*. Reliability and efficiency are conflicting goals; each can be achieved at the expense of the other. The choice of protocol depends largely on which goal is more important to a specific application. However, whatever protocol is chosen, we should usually optimize for the case of no failures — hopefully the system's normal operating state.

We'll measure an ACP's time complexity by counting the number of message exchange rounds needed for unblocked sites to reach a decision, in the worst case. A *round* is the maximum time for a message to reach its destination. The use of timeouts to detect failures is founded on the assumption that such a maximum message delay is known. Note that many messages can be sent in a single round — as many as there are sender-destination pairs. Two messages must belong to different rounds if and only if one cannot be sent until the other is received. For example, a COMMIT and a YES in 2PC belong to different rounds because the former cannot be sent until after the latter is received. On the other hand, all VOTE-REQ messages are assigned to the same round, because they can all be in transit to their destination concurrently. The same goes for all COMMIT messages. An easy way to count the number of rounds is to pretend that all messages in a round are sent at the same time and experience the same delay to all sites. Thus each round begins the moment the messages are sent, and ends at the moment the messages arrive.

Using rounds to measure time complexity neglects the time needed to *process* messages. This is a reasonable abstraction in the common case where message delays far exceed processing delays. However, two other factors might be taken into account to arrive at a more precise measure of time complexity.

First, as we have seen, a process must record the sending or receipt of certain messages in the DT log. In some cases, such as a file server in a local area network, accessing stable storage incurs a delay comparable to that of sending a message. The number of accesses to stable storage may then be a significant factor in the protocol's time complexity.

Second, in some rounds a process sends the same message to all other processes. For instance, in the first round the coordinator sends VOTE-REQs to all participants. This activity is called *broadcasting*. To broadcast a message, a process must place n copies of the message in the network,[4] where n is the number of receivers. Usually, the time to place a message in the network is negligible compared with the time needed to deliver that message. However, if n is sufficiently large, the time to prepare a broadcast may be significant and should be accounted for.

Thus, a more accurate measure of time complexity might be a weighted sum of the number of rounds, accesses to stable storage, and broadcast messages. However, we'll ignore the latter two factors and concern ourselves only with the number of rounds.

We'll measure message complexity by the number of messages used by the protocol. This is reasonable if individual messages are not too long. If they are, we should count the length of the messages, not merely their number. In all the protocols of this chapter messages are short, so we'll be content with counting the number of messages.

Let us now examine how 2PC fares with respect to resiliency, blocking, and time and message complexity.

[4]We are assuming, of course, that the communication medium is *not* a multiple access channel. In that case only one message needs to be placed in the channel; the receivers are all tapping the common channel and can "hear" the broadcast message.

Resiliency: 2PC is resilient to both site failures and communication failures, be they network partitions or timeout failures. To see this, observe that our justification for the timeout actions in the previous subsection did *not* depend on the timeout's cause. The timeout could be due to a site failure, a partition, or merely a false timeout.

Blocking: 2PC is subject to blocking. A process will become blocked if it times out while in its uncertainty period and can only communicate with processes that are also uncertain. In fact, 2PC may block even in the presence of only site failures. To calculate the probability of blocking precisely, one must know the probability of failures. This type of analysis is beyond the scope of this book (see the Bibliographic Notes).

Time Complexity: In the absence of failures, 2PC requires three rounds: (1) the coordinator broadcasts VOTE-REQs; (2) the participants reply with their votes; and (3) the coordinator broadcasts the decision. If failures happen, then the termination protocol may need two additional rounds: one for a participant that timed out to send a DECISION-REQ, and the second for a process that receives that message and is outside its uncertainty period to reply. Several participants may independently invoke the termination protocol. However, the two rounds of different invocations can overlap, so the combined effect of all invocations of the termination protocol is only two rounds.

Thus, in the presence of failures it will take up to five rounds for all processes that aren't blocked or failed to reach a decision. This is independent of the number of failures! The catch is that some processes may be blocked. By definition, a blocked process may remain blocked for an unbounded period of time. Therefore, to get meaningful results, we must exclude blocked processes from consideration in measuring time complexity.

Message Complexity: Let n be the number of participants (so the total number of processes is $n + 1$). In each round of 2PC, n messages are sent. Thus, in the absence of failures, the protocol uses $3n$ messages.

The cooperative termination protocol is invoked by all participants that voted Yes but didn't receive COMMIT or ABORT from the coordinator. Let there be m such participants, $\leq m \leq n$. Thus m processes will initiate the termination protocol, each sending n DECISION-REQ messages. At most $n-m+1$ processes (the maximum that might not be in their uncertainty period) will respond to the first DECISION-REQ message. As a result of these responses, one more process may move outside its uncertainty period and thus respond to the DECISION-REQ message of another initiator of the termination protocol. Thus, in the worst case, the number of messages sent by the termination protocol (with m initiators) will be

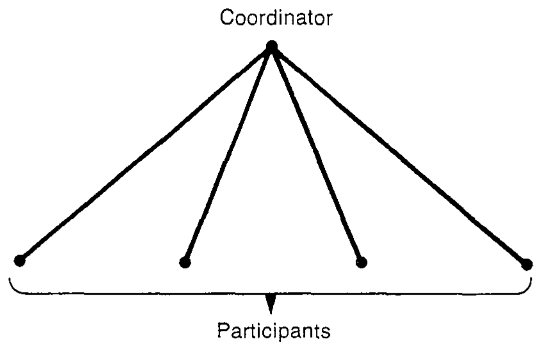

FIGURE 7–5
Communication Topology for 2PC with Five Processes

$$nm + \sum_{i=1}^{m} (n-m+i) = 2nm - m^2/2 + m/2.$$

Elementary calculus shows that this quantity is maximized when $n = m$ (recall that $0 \leq m \leq n$), that is, when all participants time out during their uncertainty period. Thus, the termination protocol contributes up to $n(3n+1)/2$ messages, for a total of $n(3n+7)/2$ for the entire 2PC protocol.

Alternative Communication Topologies for 2PC

The *communication topology* of a protocol is the specification of who sends messages to whom. For example, 2PC without the termination protocol has a communication topology in which the coordinator sends messages to the participants and *vice versa*. Participants do not send messages directly to each other. This communication topology is represented by a tree of height 1, with the coordinator as the root and the participants as the leaves (see Fig. 7–5.)

In an attempt to reduce the time and message complexity of 2PC, two other protocols have been proposed, the *decentralized 2PC protocol* and the *linear* (or *nested*) *2PC protocol*. Both have the same fundamental properties as *centralized 2PC*, the 2PC protocol we have studied so far. But they use different communication topologies than centralized 2PC.

Decentralized 2PC is designed to improve time complexity. Instead of funneling messages through the coordinator, processes may communicate directly with one another. Thus, the communication topology is represented as a *complete graph*, one that has an edge between every pair of nodes (see Fig. 7–6.)

Decentralized 2PC works as follows. Depending on its vote, the coordinator sends YES or NO to the participants. This message has a dual role: It

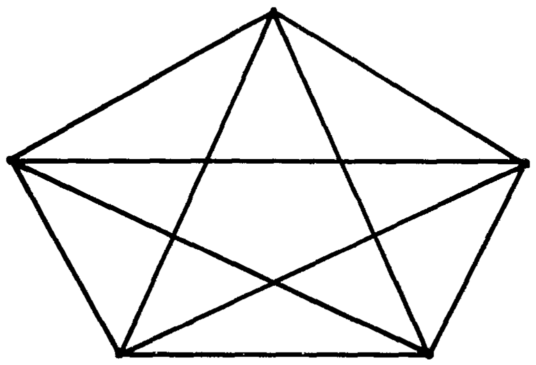

FIGURE 7–6
Communication Topology for Decentralized 2PC with Five Processes

informs the participants that it is time to vote (recall VOTE-REQ in centralized 2PC) and also tells them the coordinator's vote. If the message is NO, each participant simply decides Abort and stops. Otherwise, it responds by sending its own vote to *all* other processes. After receiving all the votes, each process makes a decision: If all were Yes and its own vote was Yes, the process decides Commit; otherwise it decides Abort. Timeout actions can be supplied just as in centralized 2PC (see Exercise 7.6).

In the absence of failures, decentralized 2PC requires only two rounds: one for the coordinator's YES or NO and another in which the participants broadcast their own votes. By not funneling the votes through the coordinator, we reduce the round complexity. Unfortunately, we also increase the message complexity. Let n be the number of participants. We now need n messages for the coordinator's vote and n^2 messages for the participants' votes, because each participant must send its vote to every other process, for a total of $n^2 + n$ messages. We need these messages even in the absence of failures, a case that centralized 2PC handles with only $3n$ messages.

Linear 2PC, on the other hand, is designed to reduce the number of messages. The processes are linearly ordered as shown in Fig. 7–7. Each process can communicate with its left and right neighbors. The protocol is initiated by the coordinator, which is the leftmost process in the linear order (numbered 1 in Fig. 7–7). The coordinator sends a message to its right neighbor (process 2) containing its vote, Yes or No. This message informs process 2 of the coordinator's vote and tells *it* to vote too. In general, a process p waits for a message from its left neighbor. If p receives a YES and its own vote is Yes, it forwards a YES to its right neighbor. If p receives a YES and its own vote is No, or if it receives a NO, then p forwards a NO to its right neighbor. If these rules are observed, then the rightmost process will have all the information it

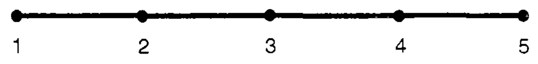

FIGURE 7–7
Communication Topology for Linear 2PC with Five Processes

needs to make a decision: If it receives a YES and its own vote is Yes, then the decision is Commit; otherwise the decision is Abort. Having made the decision, the rightmost process sends a COMMIT or ABORT to its left neighbor informing it of the decision. Each process that receives the decision message decides accordingly and then forwards that message to its left neighbor. Eventually the message reaches the leftmost process, at which time the protocol ends. It is possible to define timeout actions in this protocol similar to those we discussed for centralized 2PC (see Exercise 7.7). The timeout period for each process should depend on its position in the linear order because this influences the amount of time it will take for a message to arrive (see Exercise 7.8).

This protocol requires $2n$ messages (where n is the number of participants): n traveling left to right conveying votes, and n traveling right to left conveying the decision. Unfortunately, the economizing in messages is achieved at the expense of rounds. Linear 2PC requires as many rounds as messages because no two messages are sent concurrently. Thus linear 2PC uses $2n$ rounds, as compared with 3 for centralized 2PC and 2 for decentralized 2PC.

There are various ways to improve linear 2PC. The decision messages could all be sent concurrently by the rightmost process. That is, we could have a hybrid protocol that uses the communication topology of Fig. 7–7 for the voting phase of 2PC and that of Fig. 7–5 for the decision distribution phase (where the rightmost process is the tree's root). The resulting protocol still has message complexity $2n$ but uses $n+1$, rather than $2n$ rounds. Another improvement is suggested in Exercise 7.9.

Figure 7–8 summarizes the message and time complexities of the three variants of 2PC in the absence of failures for $n+1$ processes (n participants and the coordinator). If messages are too expensive, linear 2PC may be most appropriate. If end-to-end delays must be minimized, decentralized 2PC could be used. Centralized 2PC is a good middle ground. The choice of protocol might also be influenced by the physical characteristics of the network. A linear bus or ring network may be well suited for linear 2PC while a (nearly) completely connected network may be appropriate for decentralized 2PC.

	Messages	Rounds
Centralized 2PC	$3n$	3
Decentralized 2PC	$n^2 + n$	2
Linear 2PC	$2n$	$2n$

FIGURE 7–8
Comparison of 2PC Variants (No-Failures Case)

7.5 THE THREE PHASE COMMIT PROTOCOL

By Proposition 7.1, we cannot hope to devise a non-blocking protocol that tolerates communication failures or total site failures. Unfortunately, 2PC may cause blocking even if only non-total site failures[5] take place. In this section we study the *three phase commit* (*3PC*) protocol. In particular, we'll examine two variations of 3PC. The first, to which we devote most of the section, is designed to tolerate *only* site failures. The protocol is non-blocking in the absence of total site failures. In the event of a total failure, blocking may occur but correctness is still assured. Unfortunately, this protocol may cause inconsistent decisions to be reached in the event of communication failures.

The second variation of 3PC, discussed in the last subsection, can tolerate both communication and site failures. However, it is prone to blocking. Indeed, blocking may occur even if only processes have failed. This is unavoidable, in view of Proposition 7.1 and the fact that communication and site failures manifest themselves in the same manner, namely, as the inability to exchange messages within a pre-determined timeout period.

Both variations are more involved and have greater message and round complexity than 2PC. For systems built to tolerate only site failures, the first variation of 3PC has the advantage over 2PC that it completely eliminates blocking (except, unavoidably, in the event of total failures).

The second variation of 3PC can be used in systems designed to tolerate both site and communication failures. It does not completely eliminate blocking but causes blocking less frequently than 2PC. For instance, in 2PC processes may be blocked even if just one process — the coordinator — fails; in the second variation of 3PC no process will be blocked (in the absence of communication failures), as long as a majority of the processes are still operational.

[5]Henceforth, in this section, "site failures" means "non-total site failures" unless otherwise specified.

In most practical applications, the circumstances under which 2PC causes blocking are sufficiently rare that blocking is usually not considered a big problem. Consequently, almost all systems we know of that employ atomic commitment protocols use some version of 2PC.[6] Even though 3PC is not used in practice, it is an interesting protocol both in its own right and also because it illustrates a number of important techniques used in the design of fault-tolerant communication protocols. For these reasons, we feel that its study is a worthwhile endeavor. However, the rest of this section can be skipped without loss of continuity.

Basic Structure of 3PC

The first version of 3PC we'll describe is designed to handle site but not communication failures. Consequently, *we assume that only site failures occur.* Communication failures do not happen. There is a subtlety in this assumption. Depending on the network topology, site failures may cause communication failures as a side effect. For instance, in Fig. 7–1 if sites *A* and *C* fail, then sites *B* and *E* cannot communicate even though both are operational. Such site failures count as communication failures, too.

There are two major implications of assuming that only site failures happen. First, all operational processes can communicate with each other. Second, a process that times out waiting for a message from process *q* knows that *q* is down and therefore that no processing can be taking place there. In particular, no other process can be communicating with *q*. Neither of these statements is true if communication failures are possible.

In 2PC, if all operational processes are uncertain, they are blocked. They can't decide Abort, even if they know that processes they cannot communicate with have failed, because some failed process could have decided Commit before failing.

Suppose we've managed to design an ACP with the following "non-blocking property":

NB: If any operational process is uncertain then no process (whether operational or failed) can have decided to Commit.

If the operational sites discover they are all uncertain, they can decide Abort, safe in their knowledge that the failed processes had *not* decided Commit. When the failed processes recover they can be told to decide Abort too. This way blocking is prevented.

3PC is a protocol that satisfies NB. The idea is simple. Consider why 2PC violates NB. The coordinator sends COMMITs to the participants while the

[6] A notable exception is SDD-1, which uses a peculiar protocol that resembles 2PC in some respects and 3PC in others (see Exercise 7.13).

latter are uncertain. Thus if participant p receives a COMMIT before participant q, the former will decide Commit while the latter is still uncertain. 3PC avoids this as follows: After the coordinator has found that all votes were Yes, it sends PRE-COMMIT messages to the participants. When a participant p receives that message, it knows that all processes voted Yes and is thereby moved outside its uncertainty period. p does *not*, however, decide Commit yet. At this point p knows that it *will* decide Commit *provided it does not fail.*

Each participant acknowledges the receipt of PRE-COMMIT. When the coordinator has received all the acknowledgments to PRE-COMMITs, it knows that no participant is uncertain anymore. It then sends COMMITs to all participants. When a participant receives a COMMIT, it can decide Commit. This decision does not violate NB since no process is uncertain any longer.

If a process votes No, then 3PC behaves just like 2PC. The coordinator sends ABORTs to all processes. We now give the full description of the 3PC protocol.

1. The coordinator sends a VOTE-REQ to all participants.
2. When a participant receives a VOTE-REQ, it responds with a YES or NO message, depending on its vote. If a participant sends NO, it decides Abort and stops.
3. The coordinator collects the vote messages from all participants. If any of them was NO or if the coordinator voted No, then the coordinator decides Abort, sends ABORT to all participants that voted Yes, and stops. Otherwise, the coordinator sends PRE-COMMIT messages to all participants.
4. A participant that voted Yes waits for a PRE-COMMIT or ABORT message from the coordinator. If it receives an ABORT, the participant decides Abort and stops. If it receives a PRE-COMMIT, it responds with an ACK (i.e., acknowledgment) message to the coordinator.
5. The coordinator collects the ACKs. When they have all been received, it decides Commit, sends COMMITs to all participants, and stops.
6. A participant waits for a COMMIT from the coordinator. When it receives that message, it decides Commit and stops.

Messages received at steps (5) and (6) have the peculiar property of being known to their recipients even *before* they are received! In step (5) the coordinator knows it may only receive ACKs, and in step (6) a participant knows it can only receive a COMMIT. This casts some doubt on the utility of such messages. Their importance is that they inform their recipients of the occurrence of certain events. The receipt of ACK from participant p tells the coordinator that p is no longer uncertain. And since a COMMIT is sent only after all ACKs have been received, a participant that receives the COMMIT knows that *no* participant is uncertain. Thus it can decide Commit without violating NB.

This protocol works fine as long as there are no failures. To handle failures we must supply timeout actions describing what a process must do if an expected message does not arrive and must explain how a process can reach a decision after recovering from a failure. We address these issues in turn.

Timeout Actions

What a process should do when it times out depends on the message it was waiting for. There are five places in which a process waits for some message in 3PC:

1. In step (2) participants wait for VOTE-REQ.
2. In step (3) the coordinator waits for the votes.
3. In step (4) participants wait for a PRE-COMMIT or ABORT.
4. In step (5) the coordinator waits for ACKS.
5. In Step (6) participants wait for COMMIT.

Cases (1) and (2) are handled exactly as in 2PC. In both cases the process that times out knows that no process can have decided Commit. Thus, it can unilaterally decide Abort. In case (1) the participant can simply stop once it has decided Abort; in case (2) the coordinator should also send ABORTS to all participants that had voted Yes.

In case (4) the coordinator times out because one or more participants failed before sending an ACK. The coordinator does not know whether these participants failed before or after receiving a PRE-COMMIT. But it does know that these participants had voted Yes and were therefore prepared to decide Commit. Thus it ignores the failures and proceeds to send COMMIT to the operational participants as if it had received all ACKS. The failed participants, when they recover, are responsible for finding out that the decision was to Commit. With this timeout action, processes might decide Commit while some *failed* participant is uncertain. This will happen if some participant that did not send an ACK had actually failed before even receiving the PRE-COMMIT from the coordinator (cf. step (4) of 3PC). This does *not* violate NB, which requires only that no operational or failed process has decided Commit while some *operational* process is uncertain.

Cases (3) and (5) are more problematic. Here processes cannot act autonomously in response to the timeout. They must communicate with other processes to reach a consistent decision. It is clear why such communication is necessary in case (3); the timeout of an uncertain participant can't be handled unilaterally. But why does a participant p that times out in case (5) need to communicate with others? p has already received a PRE-COMMIT and is therefore not uncertain. After all, p knows that only COMMIT could possibly arrive from the coordinator. Why can't p ignore the timeout and simply decide Commit?

The reason is that by deciding Commit, *p* could be violating condition NB. To see this, suppose that the coordinator failed after having sent the PRE-COMMIT to *p* but before sending it to some other participant *q*. Thus *p* will time out in case (5) — outside its uncertainty period — while *q* will time out in case (3) — inside its uncertainty period. If *p*, on timeout, were to decide Commit while *q* (which is operational) is still uncertain, it would violate NB. This suggests that *before* deciding Commit, *p* should make sure that all operational participants have received a PRE-COMMIT, and have therefore moved outside their uncertainty periods. The termination protocol that a participant invokes if it times out in cases (3) and (5) does just that.

To explain how the termination protocol works, it is convenient to define the *state* of a process relative to the messages it has sent or received. There are four possible states:[7]

- *Aborted:* the process has not voted, has voted No, or has received an ABORT (i.e., it has either decided Abort or can unilaterally decide so).
- *Uncertain:* the process is in its uncertainty period.
- *Committable:* the process has received a PRE-COMMIT but has not yet received a COMMIT.
- *Committed:* the process has received a COMMIT (i.e., it has decided Commit).

First, note that any process is in precisely one state at any time. Second, some pairs of states cannot coexist, that is, cannot be occupied at the same time by two operational processes. Figure 7–9 shows which states can coexist and which cannot: A "*Y*" entry means that the states in the corresponding row and column can coexist, while an "*N*" means they cannot (see Exercise 7.14).

The termination protocol works as follows. when a participant times out in case (3) or (5), it initiates an *election protocol*. This protocol involves all processes that are operational and results in the "election" of a new coordinator. (The old one must have failed; otherwise no participant would have timed out!) We'll describe an election protocol later. For now let's just assume we have one. Once the new coordinator has been elected, the termination protocol proceeds as follows: The coordinator sends a STATE-REQ message to all processes that participated in the election. (By our assumption about failures, all operational sites will participate.) A participant in the termination protocol (i.e., any operational process other than the new coordinator) responds to this message by sending its state to the coordinator. The coordinator collects these states and proceeds according to the following *termination rule*:

TR1: If some process is Aborted, the coordinator decides Abort, sends ABORT messages to all participants, and stops.

[7]We capitalize the first letter of these four process states in the rest of the chapter.

	Aborted	Uncertain	Committable	Committed
Aborted	*Y*	*Y*	*N*	*N*
Uncertain	*Y*	*Y*	*Y*	*N*
Committable	*N*	*Y*	*Y*	*Y*
Committed	*N*	*N*	*Y*	*Y*

FIGURE 7–9
Coexistence of States

TR2: If some process is Committed, the coordinator decides Commit, sends COMMIT messages to all participants, and stops.

TR3: If all processes that reported their state are Uncertain, the coordinator decides Abort, sends ABORT messages to all participants, and stops.

TR4: If some process is Committable but none is Committed, the coordinator first sends PRE-COMMIT messages to all processes that reported Uncertain, and waits for acknowledgments from these processes. After having received these acknowledgments the coordinator decides Commit, sends COMMIT messages to all processes, and stops.

A participant that receives a COMMIT (or ABORT) message, decides Commit (or Abort), and stops.

Of course, processes can fail during the termination protocol. Thus we must supply timeout actions for all places in which a process is waiting in *this* protocol! It seems as though we are chasing our tail, but that's not so. Once elected, the coordinator will ignore any participants that fail before reporting their state. It will base its decision on the states of the participants that do report their state, in accordance with the termination rule. So, participant failures during the termination protocol are easily handled.

If the coodinator fails during the termination protocol, one of the participants waiting for the decision will time out. That participant will initiate a new election protocol, resulting in the election of a new coordinator. The termination protocol will then be started all over again. All processes will report their states to the new coordinator, which will proceed according to TR1–TR4. Before all operational processes reach a decision, several invocations of the termination protocol may be required, one for each coordinator failure. Eventually either some coordinator will finish the protocol or all processes will fail, resulting in total failure. We'll discuss total failures shortly.

It should be emphasized that processes that fail and then recover while the termination protocol is in progress are not allowed to participate in that protocol. Such processes will reach a decision using the recovery procedure that will be presented soon.

Correctness of 3PC and Its Termination Protocol

We now show that in the presence of only site failures, 3PC (with its termination protocol) is non-blocking and correct. Correctness amounts to proving that the five conditions of ACPs are satisfied. The only one that's non-trivial to show is AC1, i.e., that all processes that reach a decision reach the same one. AC2, AC3, and AC4 can be easily verified and we won't discuss them. To show that AC5 is satisfied we must consider failure recovery, which is the topic of the next subsection. Here we'll concentrate on proving the consistency of the decision reached by the processes. Although the argument is lengthy, it's worth studying carefully, because it elucidates how 3PC works in the case of failures.

Lemma 7.3: For any set of states received by the coordinator in the termination protocol, *exactly one* of the four cases of the termination rule (TR1-TR4) applies.

Proof: The reported states must coexist. Using Fig. 7–9 it is easy to verify that for any set of states that pairwise coexist, one and only one of TR1–TR4 must apply. □

Theorem 7.4: In the absence of total failures, 3PC and its termination protocol never cause processes to block.

Proof: If the coordinator does not fail then all operational processes will reach a decision, so in this case they do not block. If the coordinator fails, any processes that have not reached a decision will initiate the termination protocol and elect a new coordinator. By Lemma 7.3 at least one of the termination rule cases will apply. Moreover, in each case of the termination rule a decision is reached. Thus if the coordinator of the termination protocol does not fail, all remaining processes will reach a decision; again there is no blocking. If the coordinator fails, a new invocation of the termination protocol will be initiated. This will be repeated until either all remaining processes reach a decision or all processes fail (a total failure). Therefore, in the absence of total failures, all operational processes will reach a decision without blocking. □

Lemma 7.5: All processes that reach a decision on the same invocation of the termination protocol reach the same one.

Proof: By Lemma 7.3, at most one case of the termination rule applies and therefore the rule is unambiguous. This means that the coordinator sends the same decision to all processes (to which it sends anything at all). □

Lemma 7.6: If NB holds before the termination protocol starts, it will hold after *even a partial execution of that protocol.*

Proof: A (possibly partial) execution of the termination protocol can violate NB only if some process decides Commit while an operational process is Uncertain. A Commit decision is reached only in cases TR2 or TR4. By Fig. 7–9, in case TR2 all operational processes are Committable or Committed and thus are not Uncertain. In case TR4 all processes that are Uncertain and do not, in the meanwhile, fail are explicitly moved out of their uncertainty periods by receiving PRE-COMMITs and acknowledging them. Thus in all cases NB is preserved. □

Lemma 7.7: Consider the i-th invocation of the termination protocol, that is, the invocation after i coordinators have failed. If a process p that is operational during at least part of this invocation is Committable, then some process q that was operational in (at least part of) the $(i-1)$st invocation was Committable then.[8]

Proof: If p itself was Committable in the $(i-1)$st invocation we are done. If p became Committable on the i-th invocation (see case TR4 of the termination rule), some process q must have reported its state as being Committable. But q must have been Committable in the previous invocation, because no process changes its state in an invocation until after the coordinator has received all the state reports. □

Theorem 7.8: Under 3PC and its termination protocol, all operational processes reach the same decision.

Proof: We'll prove that all processes that have reached a decision by the i-th invocation of the termination protocol have reached consistent decisions. The proof is by induction on i.

BASIS: $i=0$. Consider the "0-th invocation" of the termination protocol, i.e., the execution of the basic 3PC protocol. In this case a process decides according to the COMMIT or ABORT it receives from the coordinator. Since the coordinator sends the same message to all processes (to which it sends anything at all), all processes that reach a decision during this invocation reach a consistent one.

[8]The "0-th invocation" of the termination protocol is taken to be the execution of the basic 3PC protocol (as described at the beginning of this section).

INDUCTION STEP: Suppose all processes that have reached a decision through the $(i-1)$st invocation of the termination protocol $(i>0)$ have reached a consistent one. Consider now the processes that reach a decision during the i-th invocation of the termination protocol. By Lemma 7.5 all these processes must reach the same decision (among themselves). Thus it remains to show that this decision is consistent with earlier ones. We do this by considering each case of the termination rule, according to which the decision was reached in the i-th invocation of the termination protocol.

TR1: If this case applies, all processes that decide in this invocation will decide Abort. So we must show that no processes that had reached a decision earlier could have decided Commit. For TR1 to apply, some operational process p must have been Aborted, meaning that (1) p has not voted; or (2) p has voted No; or (3) p had received an ABORT before this invocation of the termination protocol. In cases (1) or (2) no process could have decided Commit in a previous invocation. In case (3), p had already decided Abort in an earlier invocation and, by induction hypothesis, all processes that had decided in earlier invocations must have decided Abort.

TR2: In this case there must be an operational process p in the Committed state. Thus, p had received a COMMIT and had therefore decided Commit in some previous invocation. By induction hypothesis, all processes that had decided in previous invocations must have decided Commit, which is the decision reached by processes when TR2 applies.

TR3: In this case all processes that decide in the i-th invocation will decide Abort. So we must show that no process could have previously decided Commit. For TR3 to apply, all operational processes must be Uncertain. By Lemma 7.6 (and induction) NB is satisfied through all invocations of the termination protocol. Since some operational process is Uncertain (indeed all of them are!), NB implies that no process could have previously decided Commit.

TR4: In this case, all processes that reach a decision in this invocation decide Commit. Suppose, by way of contradiction, that some process q had decided Abort in the j-th invocation of the termination protocol, for some $j < i$. At that time q was in the Aborted state. By Fig. 7–9, no operational process could have been Committable in the j-th invocation. By Lemma 7.7 (and induction) then, no process that's operational during the i-th invocation can be Committable, contradicting the fact that for TR4 to apply some operational process must be Committable.

This completes our proof that under 3PC and its termination protocol all processes reach a consistent decision. □

Recovering from Failures

We now explore how processes that recover from failures can reach a decision consistent with that reached by the operational sites. Let us assume that when a process recovers, it knows its state at the time it failed. A process extracts this knowledge from information it has kept in the DT log. We'll discuss the participants' recovery actions. Those for the coordinator are similar.

If a recovering participant had failed before having sent a YES to the coordinator, it can unilaterally decide Abort. Also, if *p* failed after having received a COMMIT or ABORT from the coordinator, then it has already decided. As in 2PC, these are the two cases in which a process can independently recover.

The remaining case is that *p* had failed after voting Yes but before receiving a COMMIT or ABORT. Thus, *p* needs help from other processes. It sends messages asking them what the decision was. Under our failure assumptions 3PC is non-blocking, so a decision either has already been made or is in the process of being made by the operational sites.[9] So *p* will eventually receive a message with the decision and adopt it.

Note that *p* must ask other processes about the decision even if, before failing, it had received a PRE-COMMIT and thereby left its uncertainty period. In this case, *p* cannot unilaterally decide Commit, because process failures could have occurred so that the termination protocol caused the operational processes to decide Abort (see Exercise 7.17).

To make recovery possible, each process must record in the DT log its progress towards commitment. By analyzing the DT log, the recovery procedure can determine how the recovering process can reach its decision. As in 2PC, each process writes records in the DT log that mark the sending or receipt of various messages. In fact, exactly the same messages are logged in 3PC as in 2PC. As we have seen, knowing that a participant has received a PRE-COMMIT doesn't help recovery, so such messages need not be logged. The same is true for ACKs. Since there is nothing new about logging in 3PC we won't discuss the issue any further. (The fine points of which messages are logged, and when, are described in Fig. 7–10.)

Total Failures

Suppose that a total failure has taken place and consider the first process *p* to recover. If *p* had failed after voting Yes but before having decided Commit or Abort, it cannot make a decision autonomously. Unless *p* was the last process to have failed it cannot proceed to terminate the transaction on its own. This is because the processes that failed after *p* could have decided either Commit or Abort, and *p* doesn't know the decision that was reached (see Exercise 7.18). Thus, after a total failure, the recovering processes must remain blocked until

[9]Unless, of course, *p* is recovering from a total failure, a situation we'll address momentarily.

a process q recovers such that either q can recover independently (i.e., can reach a decision without communicating with other processes) or q was the last process to have failed. In the former case, q simply communicates its decision to the other processes. In the latter case, q invokes the termination protocol.[10] All processes that have recovered will therefore reach a decision according to that protocol. We emphasize that processes recovering from a total site failure cannot use the termination protocol to reach a decision *unless they include the last process to have failed.* Otherwise, inconsistent decisions may be reached by these processes and the last process to have failed.

Coordinator's algorithm

```
send VOTE-REQ to all participants;
write start-3PC record in DT log;
wait for vote messages (YES or NO) from all participants
    on timeout begin
        let $P_Y$ be the processes from which YES was received;
        write abort record in DT log;
        send ABORT to all processes in $P_Y$;
        return
    end;
if all messages were YES and coordinator voted Yes then begin
    send PRE-COMMIT to all participants;
    wait for ACK from all participants
        on timeout skip; /* ignore participant failures */
    write commit record in DT log;
    send COMMIT to all participants
end
else /* some process voted No */ begin
    let $P_Y$ be the processes from which YES was received;
    write abort record in DT log;
    send ABORT to all processes in $P_Y$;
end;
return
```

FIGURE 7–10
The Three Phase Commit Protocol

[10]In our discussion of the termination protocol we said that processes that have failed do not participate in that protocol when they recover. This is only true if there has not been a total failure.

FIGURE 7–10
The Three Phase Commit Protocol (*continued*)

Participant's algorithm

```
wait for VOTE-REQ from coordinator
    on timeout begin
        write abort record in DT log;
        return
    end;
if participant's own vote is Yes then begin
    write yes record in DT log;
    send YES to coordinator;
    wait for message (PRE-COMMIT or ABORT) from coordinator
        on timeout begin
            initiate election protocol; /* See next subsection. */
            if elected then invoke coordinator's algorithm of
                termination protocol /* See Fig. 7-11. */
            else invoke participant's algorithm of termination protocol;
            return
        end;
    if message received is PRE-COMMIT then begin
        send ACK to coordinator;
        wait for COMMIT from coordinator
            on timeout begin
                initiate election protocol;
                if elected
                then invoke coordinator's algorithm of termination protocol
                else invoke participant's algorithm of termination protocol;
                return
            end;
        write commit record in DT log
    end
    else /* ABORT was received from coordinator */ write abort record in DT log;
end
else /* participant's vote is No */ begin
    send NO to coordinator;
    write abort record in DT log
end;
return
```

Apparently, correctly recovering from a total site failure requires that a set of processes be able to determine whether it includes the last process to have failed. They could wait until *all* processes have recovered. Certainly this set includes the last process to have failed! In the next subsection we'll describe a method in which processes don't wait that long.

The 3PC protocol and its termination protocol are shown in Figs. 7–10 and 7–11. In the pseudo-code, it is the writing of a **commit** (or **abort**) record in the DT log that corresponds to a process' deciding to Commit (or Abort).

(New) **Coordinator's algorithm**

```
send STATE-REQ to all participants (of the election protocol);
wait for state report messages from all participants
        on timeout skip; /* ignore participant failures */
if any state report message was Aborted or
    the coordinator is in the Aborted state then begin /* case TR1 */
    if the coordinator's DT log does not contain an abort record then
        write abort record in DT log;
    send ABORT to all participants
end
else if any state report message was Committed or
    the coordinator is in the Committed state then begin /* case TR2 */
    if the coordinator's DT log does not contain a commit record then
        write commit record in DT log;
    send COMMIT to all participants
end
else if all state report messages were Uncertain and
    the coordinator is also Uncertain then begin /* case TR3 */
    write abort record in DT log;
    send ABORT to all participants
end
else /* some processes are Committable - Case TR4 */ begin
    send PRE-COMMIT to all participants that reported Uncertain state;
    wait for ACK from all processes that reported Uncertain state
        on timeout skip; /* ignore participant failures */
    write commit record in DT log;
    send COMMIT to all participants
end;
return
```

FIGURE 7–11
The 3PC Termination Protocol

FIGURE 7–11
The 3PC Termination Protocol (*continued*)

Participant's algorithm

```
start: wait for STATE-REQ from coordinator
          on timeout begin
             initiate election protocol;
             if elected then begin execute coordinator's algorithm (above); return end
             else goto start
          end;
if this process had not voted in 3PC protocol or had voted No or
    has received an ABORT then state := Aborted
else if this process has received a COMMIT then state := Committed
else if this process has received a PRE-COMMIT then state := Committable
else state := Uncertain;
send state to coordinator;
wait for response from coordinator
    on timeout begin
        initiate election protocol;
        if elected then begin execute coordinator's algorithm (above); return end
        else goto start
    end;
if response was ABORT then begin
    if DT log does not contain an abort record for this process then
        write abort record in DT log
end
else if response was COMMIT then begin
    if DT log does not contain a commit record for this process then
        write commit record in DT log
end
else /* response was PRE-COMMIT */ begin
    send ACK to coordinator;
    wait for COMMIT from coordinator
        on timeout begin
            initiate election protocol;
            if elected then begin execute coordinator's algorithm (above); return end
            else goto start
        end;
    write commit record in DT log
end;
return
```

Election Protocol and Detecting the Last Process to Fail

For the purposes of the election protocol, the processes agree on a linear ordering among themselves using, for example, their unique identifiers. If p precedes (or follows) q in this ordering, we write $p < q$ (or $p > q$). The election rule is that if the present coordinator fails, the smallest of the operational processes will become the new coordinator.

Under our assumption that only site failures happen, this idea can be implemented simply and efficiently. Each process p maintains a set, UP_p, of all the processes it believes are operational. Initially, UP_p contains all processes.[11] Suppose a participant p detects the failure of the present coordinator c (by timing out while waiting for a message from c). Then it removes c from UP_p, and selects the smallest process q in UP_p. If $q = p$, then p considers itself elected and becomes the coordinator of the termination protocol; otherwise it sends a UR-ELECTED (you-are-elected) message to q and becomes a participant of the termination protocol. When q receives UR-ELECTED, it considers itself elected and becomes the termination protocol coordinator.

When the new coordinator q has been elected, some process p' may not yet have discovered the failure of c. When p' receives a STATE-REQ from q, it deduces that c has failed. p' will therefore remove c (and any other processes $q' < q$) from UP_p' and will become a termination protocol participant with q as its coordinator.

Message delays can cause a process to receive a STATE-REQ from an old coordinator that failed before a new coordinator was elected. Thus, a termination protocol participant p that considers q to be the coordinator ignores STATE-REQ messages from any $q' < q$. On the other hand, if p receives a STATE-REQ from $q' > q$, then it deduces that q has failed and q' has been elected as the new coordinator. As before, p removes from UP_p all processes preceding q' and becomes a participant in a new invocation of the termination protocol, taking q' as its coordinator.

Maintaining the UP_p sets is also useful in recovering from total failures. It helps a set of processes R determine whether they contain, among them, the last process that failed.

Suppose that, when a process p recovers, it can retrieve the value of UP_p at the time it had failed. Thus, p knows that the last process to have failed must be in UP_p. Suppose a set of processes R have recovered from a total failure. The last process to have failed must be a process that each process in R believed to have been operational when it failed, that is, a process in the set $\bigcap_{p \in R} UP_p$. Thus, the processes in R know that the last process to have

[11]The coordinator, which by assumption knows all the processes, can send the set of all processes to each participant along with the VOTE-REQ messages, so that participant p can properly initialize UP_p. The fact that UP_p is not initialized until after the receipt of the VOTE-REQ is not a problem, since the election protocol is relevant only to a process after it has voted.

failed is in R if and only if they contain *all* potential candidates; that is, if $R \supseteq \bigcap_{p \in R} UP_p$ (*).

Therefore, when enough processes R recover after a total failure so that property (*) is satisfied, the processes in R can initiate the termination protocol to reach a decision as we discussed in the previous subsection.

For p to retrieve the value of UP_p when it recovers, it must have saved UP_p in stable storage — for example, in the DT log. Note that it is not necessary for p to save UP_p in stable storage every time UP_p's value changes. If p only periodically changes UP_p in stable storage, the value retrieved for UP_p when p recovers will be a superset of the actual value of UP_p at the time p failed. This is safe in that a set R satisfying (*) still contains the last process to have failed. However, a larger set R of processes may have to recover from a total failure before (*) is satisfied.

Evaluation of 3PC

Let's summarize the properties of 3PC with respect to resiliency, blocking, and time and message complexity.

Resiliency and Blocking: 3PC is resilient to site failures only. It is non-blocking except for a total site failure. By Propositions 7.1 and 7.2 this is the maximal level of fault tolerance a non-blocking ACP can attain.

Time Complexity: In the absence of failures, 3PC uses at most five rounds of messages: (1) to distribute VOTE-REQs; (2) to deliver votes; (3) to distribute PRE-COMMITs; (4) to acknowledge the PRE-COMMITs; and (5) to distribute COMMITs. (If the decision is Abort, only three rounds are needed.) Each invocation of the termination protocol contributes at most five more rounds, plus the election protocol, which requires only one round to send UR-ELECTEDs. Thus if f processes fail, at most $6f + 5$ rounds are needed. This may appear deceptively worse than the five rounds required, in the worst case, for 2PC (independent of the number of failures!). But recall that 2PC may cause blocking and the five round bound concerned only *non-blocked* processes.

Message Complexity: In each round of 3PC at most n messages are sent, where n is the number of participants. Thus, in the absence of failures, 3PC requires up to $5n$ messages. In each round of an invocation of the termination protocol the number of messages is the number of remaining participants. In the i-th invocation, there are at most $(n-i)$ operational participants left, so the number of messages is at most $6(n-i)$. Therefore, if there are f process failures, the number of messages is at most $5n + \Sigma_{i=1}^{f} 6(n-i) = 3(f+1)(2n-1)-n$.

3PC and Communication Failures

Up to this point in this section we have assumed that no communication failures occur. Let us now remove this assumption. Unfortunately, the termination protocol for 3PC we've presented may result in processes reaching inconsistent decisions. To see this, suppose that the processes are partitioned into two components A and B. It is possible that, at the time the partition occurred, all processes in A are Uncertain while all those in B are Committable (see Fig. 7–9). According to the termination protocol, the processes in A will decide Abort (case TR3), while those in B will decide Commit (case TR4).

In this subsection we'll describe a new termination protocol that avoids such inconsistencies and guarantees correctness even in the presence of communication failures. The new termination protocol may introduce blocking, but this is unavoidable in view of Proposition 7.1.

For now it is best to think of process failures as permanent. That is, a process that fails never recovers. Later we'll show how recoveries can be handled in a very simple manner.

The overall structure of the termination protocol is as before. That is, a coordinator is elected that collects the states of participants and decides how to proceed on the basis of the states it has received. The problem, illustrated in the example outlined previously, is that communication failures may result in the election of multiple coordinators (unable to communicate among themselves), each deciding on how to terminate the protocol on the basis of the states of disjoint sets of participants.

Seen in this light, the following remedy suggests itself. We will allow a coordinator to reach a decision only if it can communicate with a majority of processes. This ensures that decisions reached by two different coordinators will be based on the state of at least one process in common. The termination protocol will be designed in such a manner that the common process will prevent inconsistent decisions.

Here is how this is achieved. When a coordinator receives the states of (a majority of) processes, it determines its *intention*. The intention is (to decide) Commit, if at least one process has reported a Committable state; it is Abort, if all states the coordinator received were Uncertain.[12] *Intention* to decide Commit (or Abort), however, is not the same as *deciding* Commit (or Abort). In particular, before converting its intention to an actual decision, the coordinator must make sure that a majority of processes know its intention. This will prevent inconsistent decisions because a process that knows that some coordinator ever intended Commit (or Abort) will not allow another coordinator to decide Abort (or Commit).

[12]We are ignoring the situations where a Committed or Aborted state is received, since these can be handled in a straightforward manner; that is, the coordinator adopts the corresponding decision and relays it to every process whose state indicates it has not reached a decision.

To inform others of its intention to decide Commit, a coordinator sends PRE-COMMIT messages to all processes. A process that receives a PRE-COMMIT from its coordinator responds with a PRE-COMMIT-ACK. The coordinator can convert its intention to decide Commit to a Commit decision only when it has received PRE-COMMIT-ACKS from at least a majority of processes. After deciding Commit in this manner, the coordinator sends COMMIT messages to all processes. On receipt of such a message a process simply adopts the Commit decision.

The procedure for informing other processes of the intention to decide Abort is analogous. The coordinator sends PRE-ABORT messages and waits for PRE-ABORT-ACKS. It converts its intention to decide Abort to a decision upon receipt of such messages from at least a majority of processes. At that time it also sends ABORT messages to all processes.

Note that the PRE-ABORT message type is new; no such messages were used in the 3PC termination protocol we saw previously. By analogy to the Committable state, we'll say that a process that has received a PRE-ABORT, but not an ABORT, message is in the *Abortable* state (or simply is Abortable). Thus when a coordinator receives the states of other processes, it may receive Committable, Uncertain, or Abortable states.

If a coordinator receives an Abortable state it must not decide Commit. This is because an Abortable state is a signal that a coordinator had the intention to decide Abort. For all we know, it may have succeeded. Similarly, if a coordinator receives a Committable state it must not decide Abort.

Unfortunately, it is possible for one process to be Abortable and another to be Committable (see Exercise 7.24). Thus, unless there is a safe rule for convincing one to "convert" to the other state (at least when all failures are repaired), we risk perpetual indecision, which is a violation of the requirements for ACPs.

Fortunately, such a safe rule exists. If a coordinator has received a majority of non-Abortable states, including at least one Committable, then it can convince any Abortable processes to become Committable. The fact that there exists a majority of non-Abortable processes means that the coordinator that sent the PRE-ABORT messages to the Abortable ones did not, after all, succeed in forming a majority of Abortable processes. Consequently, it couldn't have decided to Abort. And the fact that there is a Committable state means that it is legitimate to decide Commit (i.e., all processes had originally voted Yes). By the same argument, it is easy to see that if there is a majority of non-Committable processes, any Committable ones can be "converted" to Abortable.

In summary, then, the termination protocol is this: After being elected (by a protocol to be described shortly), a coordinator sends STATE-REQ messages to all processes. A process that receives this message from its coordinator responds by sending its present state. The coordinator waits for a period of time, collecting responses. It then proceeds by using the following *Majority Termination Rule*:

> *MTR1:* If a Committed state is received, then the coordinator decides Commit, sends COMMIT messages to all processes, and stops.
>
> *MTR2:* If an Aborted state is received, then the coordinator decides Abort, sends ABORT messages to all processes, and stops.
>
> *MTR3:* If at least one Committable and a *majority* of non-Abortable (but no Committed) states are received, then the coordinator sends PRE-COMMIT messages to all processes that did not report a Committable state. A process that receives a PRE-COMMIT changes its state to Committable and responds with a PRE-COMMIT-ACK. The coordinator waits for PRE-COMMIT-ACKs until the set of processes that had reported Committable and of processes that have sent a PRE-COMMIT-ACK constitutes a majority (i.e., until the coordinator knows that a majority of processes are Committable). It then decides Commit, sends COMMIT messages to all processes, and stops. If the required number of PRE-COMMIT-ACKs is not received, the coordinator and all processes that elected it become blocked.
>
> *MTR4:* If a majority of non-Committable (but no Aborted) states are received, the coordinator sends PRE-ABORT messages to all processes that did not report an Abortable state. A process that receives a PRE-ABORT message changes its state to Abortable and responds with a PRE-ABORT-ACK. The coordinator waits until the set of processes that had reported Abortable or have sent a PRE-ABORT-ACK constitutes a majority. It then decides Abort, sends ABORT messages to all processes, and stops. As in MTR3, if the majority for which the coordinator is waiting does not materialize, it and all the processes that elected it become blocked.
>
> *MTR5:* In all other cases, the coordinator and all processes that elected it become blocked.

A process that receives a COMMIT (or ABORT) message decides to Commit (or Abort) and stops.

We can state the critical property of the communication-failure-tolerant version of 3PC and its termination protocol as follows: Before a process can decide Commit, a majority of processes must be Committable or Committed; and before a process can decide Abort, either no process is Committable or Committed, or a majority must be Abortable or Aborted. A precise proof of correctness for this protocol can be developed on the basis of this property (see Exercise 7.25).

To complete our discussion of this protocol we must address the issue of elections and what to do with blocked processes.

The election protocol works as follows. Each process p maintains a set UP_p consisting of all the processes p currently believes it can communicate with. Initially, UP_p is the set of all processes.[13] The rule of the election algorithm is that a process will become coordinator only if it cannot communicate with a smaller process ("smaller" in some predetermined linear ordering known to all processes). Thus, the coordinator of p at any time is the smallest process in UP_p. To avoid making a special case for the original coordinator, we assume that it is the smallest process.

Suppose now that p loses the ability to communicate with its present coordinator. It removes that coordinator from UP_p and selects the smallest process left in UP_p, say q. If $p=q$ (i.e., p can't communicate with any other smaller process), p elects itself coordinator. Otherwise p sends a UR-ELECTED message to q. When q receives such a message, it checks if it can communicate with any smaller process. If it cannot, q removes any smaller processes from UP_p and elects itself coordinator. If, however, q can communicate with some $q' < q$, it ignores p's UR-ELECTED message.

In the meanwhile, p has sent the UR-ELECTED message to q and is waiting for a STATE-REQ from it. If this message does not arrive (in the timeout period set by p), p concludes that communication between it and q is impossible. This could be either for physical reasons (failure of q or a communication failure) or for logical ones (q ignored the UR-ELECTED message from p because it can still communicate with some $q' < q$ while p, presumably, cannot communicate with q'). Whatever the reason, p will remove q from UP_p and will repeat the same protocol with the smallest process now left in UP_p.

It is possible that, during the termination protocol, a process receives STATE-REQ, PRE-COMMIT, or PRE-ABORT messages from a process other than its own coordinator. Such messages are ignored (see Exercise 7.26).

A blocked process p periodically runs the election protocol. If all failures are repaired, all processes will elect the smallest process as their coordinator. The elected coordinator can communicate with all processes (and, *a fortiori*, with a majority). Therefore, the majority termination rule guarantees that a decision will be reached. Note that it is not necessary for *all* failures to be repaired. In general, if there is a set of processes A that forms a majority and such that every process in A can communicate with the smallest process in A, then all processes in A will reach a decision (i.e., will become unblocked).

When a process that has failed recovers, it can act as if it had been disconnected from the others during the period of its failure. That is, it runs the election protocol and proceeds accordingly.

As we have seen, certain actions can be taken by the coordinator only if it can communicate with a majority of processes. The crucial property of majori-

[13] We are assuming that the original coordinator attaches the set of all participants to the VOTE-REQ messages, so that participant p can properly initialize UP_p upon receipt of that message.

ties that is needed here is that any two sets of processes that are majorities must intersect.

A quorum is a generalization of the concept of majority that also satisfies this intersection property. Suppose that each process p is assigned a non-negative weight $w(p)$. A set of processes A constitutes a *quorum* if the processes in A have a majority of the weight, that is, if $\Sigma_{p \in A}\, w(p) > (\Sigma_{p \in P}\, w(p))/2$, where P is the set of all processes. A majority is a special case of a quorum where all processes have equal weight. Since quorums have the intersection property (i.e., if A and B are quorums then $A \cap B \neq \{\}$), we can replace "majority" by "quorum" in the Majority Termination Rule.

It is sometimes useful to use this more general concept, because it allows us to assign weights to processes in a way that reflects their "importance." For instance, we may wish to assign more weight to the original coordinator than to other processes, to increase the chances that the group of processes that maintains communication with that coordinator will form a quorum. Or, we may assign greater weight to processes that run in sites that fail very infrequently, thereby maximizing the chance that a quorum will be formed and minimizing the chance that *all* processes will become blocked.

BIBLIOGRAPHIC NOTES

The two phase commit protocol is due to [Gray 78] and [Lampson, Sturgis 76]. Linear 2PC was devised independently by [Gray 78] and [Rosenkrantz, Stearns, Lewis 78]. Decentralized 2PC is from [Skeen 82a]. Variants of 2PC optimized to reduce the number of records written in the DT log for various types of transactions are described in [Mohan, Lindsay 83]. The cooperative 2PC termination protocol has been used in the Sirius-Delta system [LeLann 81], and in Prime's distributed database system [Dubourdieu 82].

The 3PC protocol was devised and analyzed by Skeen [Skeen 82a], [Skeen 82b], and [Skeen 82c]. The performance of several commitment protocols with respect to blocking is studied in [Cooper 82]. [Dwork, Skeen 83] and [Ramarao 85] derive upper and lower bounds for the complexity of non-blocking commitment protocols. Communication-failure-tolerant termination protocols for 3PC are proposed in [Cheung, Kameda 85], [Chin, Ramarao 83], and [Skeen 82c].

Election protocols are discussed in [Garcia-Molina 82]. [Skeen 85] addresses the issue of determining the last process to fail.

Answers to Exercises 7.15 and 7.16 can be found in [Skeen 82a]. An atomic commitment protocol that satisfies the properties stated in Exercise 7.13 is described in [Hammer, Shipman 80].

EXERCISES

7.1 Design protocols for implementing persistent messages, that is, protocols that ensure that any message sent by a process p to process q will eventually be received by q, even if failures make the delivery of the message to q impossible at the time it is sent. Your protocol should guarantee that messages are received by q in the order they were sent by p. It should also guarantee eventual delivery of all "pending" messages if all failures are repaired and no new failures take place for sufficiently long. Under your protocol a recovering process may have to wait for some failures to be repaired before it can be assured that it has received all the messages sent to it while it was down. This is a form of blocking. Investigate ways of reducing the chance that such blocking may occur and explore the trade-off between the probability of blocking and the overhead of the protocol.

7.2 Give an ACP that also satisfies the converse of condition AC3. That is, if all processes vote Yes, then the decision must be Commit. Why is it not a good idea to enforce this condition?

7.3 Consider 2PC with the cooperative termination protocol. Describe a scenario (a particular execution) involving site failures only, which causes operational sites to become blocked.

7.4 For each of the DM implementations described in Chapter 6, describe, in some detail, how the actions of the 2PC protocol relate to the actions of the DM in processing distributed transactions.

7.5 Explore and compare various techniques whereby sites can garbage collect DT records that will never be needed.

7.6 Write down the decentralized 2PC protocol in detail. For each step in which a process is waiting to receive a message, specify suitable timeout actions.

7.7 Write down the linear 2PC protocol in detail. For each step in which a process is waiting to receive a message, specify suitable timeout actions.

7.8 Consider linear 2PC. Assuming that the maximum delay for a message to go from a process to its left (or right) neighbor is the same for all processes, indicate the timeout period that each process should set when waiting for a message (as a function of the maximum delay, the position of the process in the linear chain and the type of message expected).

7.9 In the description of the linear 2PC protocol given in Section 7.4, we have a chain of left-to-right traveling messages (votes) and then a chain of right-to-left traveling messages (the decision). It is possible to speed the protocol up in the event a process' vote is No — namely, the process passes a NO message to the right and, at the same time, passes an ABORT message to the left. Develop this variation of the linear 2PC protocol.

7.10 Is there any process in linear 2PC which is never in an uncertainty period? If so, which one(s)? If not, describe the uncertainty period of each process.

7.11 In the 2PC protocol the coordinator *first* decides Commit (by writing a **commit** record to its site's DT log) and *then* sends COMMIT messages to the participants. Suppose the order of these two steps is reversed. Show that this variation of 2PC is non-blocking if

a. there are no communication failures and at most one process can fail, or

b. there are no communication failures and processes have the ability to broadcast messages atomically (i.e., so that either all recipients get it or no recipient does).

How does this variation of 2PC affect the recovery procedures? Give recovery procedures that work for the modified 2PC protocol.

7.12 In the 2PC termination protocol each process independently sends a DECISION-REQ message, and each process that knows the decision responds to all the DECISION-REQs it receives. An alternative approach would be to elect a new coordinator that will send the decision to all (if a decision can be reached, i.e., if not all operational processes are uncertain). Develop a termination protocol that uses this approach. The protocol should handle both site and communication failures. Analyze the worst case round and message complexity of the resulting protocol. How do these compare to the round and message complexity of the protocol given in Section 7.4? (Be sure to include the rounds and messages needed for election in your analysis.)

7.13 Design an ACP that guarantees that no process gets blocked if there are no communication failures and only up to k processes fail (for some specified number k). Describe the protocol in some detail, giving suitable timeout actions for each step in which a process is waiting for a message. First design the protocol that guarantees consistency only under site failures. Then modify the protocol (using ideas in the last subsection of the chapter) to obtain a protocol that works even in the event of communication failures (but is still non-blocking if there are no communication failures and no more than k processes fail).

7.14 Prove that the "co-existence table" for 3PC (Fig. 7–9) is correct. That is, two operational sites can occupy two states s and s' at the same time if and only if the entry whose row corresponds to s and whose column corresponds to s' contains a "Y."

7.15 In the 3PC termination protocol of Section 7.5 that guarantees consistency if only site failures happen, the elected coordinator first collects the votes of the remaining participants and on the basis of these it decides how to proceed (using termination rules TR1-TR4). It is actually possible to have the new coordinator proceed to terminate the transaction on the basis of its own state, without polling the participants. Develop such a termination protocol and prove that it ensures consistency. What are the advantages and disadvantages of your protocol relative to the one discussed in the text?

7.16 It is possible to do away with the need for elections in 3PC, by using a *decentralized termination protocol*. The basic idea is this: Participants, when they discover that the original coordinator has failed, send to each other their present state. Each participant collects the others' states and on that basis decides how to proceed. Design such a protocol and prove that it ensures that consistent decisions are reached. (Note that a participant may fail after having sent its state to one process but before sending it to another. It is this type of behavior that makes this protocol non-trivial.) First develop a non-blocking decentralized termination protocol that guarantees consistency under the assumption that only process failures happen. Then modify this (using ideas in the last subsection of the chapter) to obtain a decentralized termination protocol that may cause blocking, but that guarantees consistency under both process and communication failures.

7.17 Consider the 3PC version that guarantees consistent decisions when only site failures take place. Describe a scenario (involving site failures only) in which a process that failed was Committable at the time it failed, yet the other processes wound up deciding Abort.

7.18 Consider the 3PC version that guarantees consistent decisions when only site failures take place. Give two scenarios involving total site failures in both of which p fails (and therefore recovers) in the same state s but such that in one some process has decided Commit and in the other some process has decided Abort. (This implies that if p, on recovery, finds itself in state s, it cannot reach a decision *on its own* for, no matter what it decides, some other process may have decided the opposite!) For which state(s) s are such scenarios possible? Is it possible that p is the last process to have failed?

7.19 Complete Figures 7–10 and 7–11, by giving the pseudo-code for the election protocol.

7.20 Suppose we are using the 3PC protocol in conjunction with (distributed) strict two phase locking (i.e., locks are supposed to the held until the "end of transaction"). Can the locks held by a transaction be released at a site when the 3PC process supervising the termination of the transaction at that site receives the PRE-COMMIT message, or should locks be held until the COMMIT is actually received?

7.21 Investigate the possibility of alternative communication topologies for 3PC. In particular, develop a decentralized and a linear commitment protocol based on 3PC. Analyze the complexity of these protocols and compare them to the complexity of the 3PC protocol given in Section 7.5.

7.22 In 3PC, does the relative order of sending COMMIT messages and writing a **commit** record in the DT log matter? What about the relative order of sending ABORT messages and writing an **abort** record?

7.23 Write a pseudo-code specification for the 3PC version described in the last subsection of the chapter.

7.24 Consider the 3PC protocol described in the last subsection of this chapter. Describe a scenario in which some process is in the Committable and another in the Abortable state.

7.25 Give a careful proof of the fact that the 3PC version described in the last subsection of the chapter guarantees the consistency of the decisions reached, even in the presence of communication failures. (Hint: Your proof could be based on the following property of the protocol: If Commit is decided, then a majority of processes are either Committable or Committed; if Abort is decided, then either no process is Committable or Committed, or a majority of processes are Abortable or Aborted.)

7.26 In the 3PC version given in the last subsection of the chapter, it is possible (due to communication failures) for a process *p* to receive a STATE-REQ, PRE-COMMIT or PRE-ABORT message from a process that is not *p*'s present coordinator. It is said in the text that *p* must ignore such messages. Give three scenarios (one for each message type) that illustrate what could go wrong if *p* did not ignore these messages. Also show that if *p* receives a STATE-REQ and *p* has decided, it is safe for it to respond as if the STATE-REQ had been received from its coordinator.