COMMIT PROCESSING IN DISTRIBUTED DATABASE SYSTEMS AND IN HETEROGENEOUS MULTIDATABASE SYSTEMS

by

Yousef J. Al-Houmaily

B.S. in Computer and Information Sciences,
King Saud University, Saudi Arabia
M.S. in Computer Science, George Washington University, U.S.A.

Submitted to the Graduate Faculty
of the Graduate School of Engineering
in partial fulfillment of
the requirement for the degree of
Doctor
of
Philosophy

University of Pittsburgh
1997

COMMITTEE SIGNATURE PAGE

This dissertation was presented by
Yousef J. Al-Houmaily

It was defended on
April 16, 1997

and approved by

Panos K. Chrysanthis, Assistant Professor
Committee Co-Chairman

Steven P. Levitan, Associate Professor
Committee Co-Chairman

Sujata Banerjee, Assistant Professor
Committee Member

J. Thomas Cain, Associate Professor
Committee Member

Richard Hall, Associate Professor
Committee Member

Ronald G. Hoelzeman, Associate Professor
Committee Member

ACKNOWLEDGMENTS

I would like to thank the esteemed members of my Ph.D. committee, especially my two Co-chairmen and academic advisors, Professor Panos K. Chrysanthis and Professor Steven P. Levitan. I have known both of my advisors for almost six years now. Despite their busy schedules, both of them have always been available for me with open doors at all times. Their suggestions, comments and guidance are clearly reflected in the contents of this document. I greatly thank both of them for sharing with me their time and knowledge. I would also like to express my deep gratitude to Professor Chrysanthis for teaching me the basics of databases, persuading me to seek a career in this ever evolving area of research, and promoting excellence in my research.

I would like also to thank the other four members of my dissertation committee: Professor Sujata Banerjee, Professor J. Thomas Cain, Professor Richard Hall and Professor Ronald G. Hoelzeman, for their interest in my Ph.D. work and their constructive comments and suggestions.

I thank my colleagues Robert Conticello, Fahad Hoymany, Jae Oh and Gary Walborn for the good times and inspiring discussions. I would also like to thank Rob for his great help in my simulation studies.

I gratefully acknowledge the government of my home country, The Kingdom of Saudi Arabia, for its support through a Graduate Student Scholarship as well as the National Science Foundation for a research assistantship under grants IRI-9210588 and IRI-95020091.

ABSTRACT

Signature________________________
Panos K. Chrysanthis

Signature________________________
Steven P. Levitan

COMMIT PROCESSING IN DISTRIBUTED DATABASE SYSTEMS AND IN HETEROGENEOUS MULTIDATABASE SYSTEMS

Yousef J. Al-Houmaily, Ph.D.
University of Pittsburgh

The focus of this dissertation is on *performance* of atomic commit protocols in distributed database systems (DDBSs) and on *compatibility* of atomic commit protocols in heterogeneous, multidatabase systems (MDBSs). This dissertation offers four major contributions, three of which are in the context of DDBSs while the fourth is in the context of MDBSs. Specifically, its first contribution are two highly efficient atomic commit protocols in DDBSs, called the *implicit yes-vote* (IYV) and *implicit yes-vote with a commit coordinator* (IYV-WCC) which exploit the characteristics of future gigabit-networked database systems, assuming different degrees of site reliability.

The second contribution is the performance evaluation of IYV and four other well known two-phase commit variants based on a simulation system. This includes the performance gains when read-only optimizations are used. The simulation study, explicitly models (1) the propagation latency of the communication network, (2) the overhead of the management of the database buffer and of flushing the transaction and protocol execution log records and (3) the overhead of recovery from site failures.

The third contribution in DDBSs are two new presumed commit two-phase commit variants for the multi-level transaction execution model and a new read-only optimization called the *unsolicited update-vote* (UUV). The two new presumed commit variants significantly reduce the overhead of the original presumed commit protocol and, when combined with UUV optimization, they nullify the classical argument that solely favors the presumed abort protocol.

The last contribution is the characterization of the concept of a *safe* state with respect to the commitment of transactions and its application in the context of MDBSs through the development of a new atomic commit protocol called the *presumed any* (PrAny). PrAny interoperates the basic *two-phase commit* protocol and its most commonly known two variants, namely, the *presumed abort* and the *presumed commit* protocols, despite the conflicting presumptions about the outcome of terminated transactions and without violating the autonomy of the constituent sites.

DESCRIPTORS

Atomic Commit Protocols
Database Systems
Distributed Database Systems
Multidatabase Systems
Transaction Management
Two-phase Commit Protocol

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION AND MOTIVATION

Current advances in computer and network technologies enable an entirely new class of advanced applications which manage vast amounts of data. These applications will be distributed over different computing systems that are interconnected via high-speed communication networks and will access data objects stored at different database sites. Collaborative computer-aided design applications, biomedical databases, world trading and banking, multimedia information systems, are few examples from a long list of such advanced database applications. These applications will typically use *transactional* support due to the abstract correctness properties of *transactions*. Transactions provide application programmers with a high level system interface that hides both the effects of concurrency among the different activities in the system and the presence of failures. At the same time, transactions guarantee data consistency and database integrity despite concurrent data sharing and system failures.

In a traditional transactional database system, a *database management system* (DBMS) is employed to provide uniform access to the data objects stored in the database and to ensure their consistency by means of transactions. That is, application programs and users access the data (in the database) by submitting transactions to the DBMS. In a *distributed database system* (DDBS), the data objects are stored at multiple sites that are interconnected via a communication network. Access to the data objects in a DDBS is governed by a *distributed database management system* (DDBMS) which executes each transaction in a distributed fashion at different sites based on the location of the data objects that the transaction requires to access.

Since site and communication failures are possible, transactions might end up partially executed, producing unpredictable results that violate data consistency. Hence, for a distributed transaction that executes across multiple sites, the sites need to agree about *when* and *how* the transaction should terminate to avoid any inconsistencies. That is, all the sites participating in a transaction execution need to (1) *eventually* reach an agreement; and (2) all agree on a *binary* value that is either to *commit* the transaction making all its effects persistent, or to *abort* the transaction obliterating all its effects as if the transaction has never executed. A protocol that achieves this kind of agreement is called an *atomic commit protocol* (ACP) and is used to realize *atomicity* which is one of the abstract properties of transactions. By using an ACP, a distributed transaction always produces results that preserve data consistency despite site and communication failures.

The *two-phase commit* protocol (2PC) (1, 2)* is the simplest and most used ACP. The performance of 2PC as well as other ACPs is measured using three metrics (3, 4, 5). The first metric is *message complexity* which deals with the number of messages that are needed to be exchanged between the systems participating in the execution of a transaction to reach a consistent decision regarding the final status of the transaction. The second metric is *log complexity* which accounts for the amount of information that needs to be recorded at each participant site in order to achieve resiliency to failures. The third metric is *time complexity* which corresponds to the number of rounds of messages that are required in order to reach a decision. Using today's technology, a disk access requires 20 milliseconds whereas the propagation latency of a message from one site to another is typically 50 milliseconds in high-speed wide-area networks. These costs do not take into consideration the queuing delays over the CPUs and disks, which are much higher than these basic costs especially in high-volume transactional systems and, generally, is on the order of hundreds of milliseconds. Hence, eliminating the need for a disk access or a message from the commit processing of transactions, greatly reduces queuing delays and congestion over the system resources including contention over the data objects that are stored in a database. In fact, it has been the goal of all 2PC variants proposed in the literature (6, 7, 8, 9), to reduce the cost of 2PC which consumes a substantial amount of a transaction's execution time during normal processing (10). For example, the most notable 2PC variants, namely, *presumed abort* (PrA) and *presumed commit* (PrC), reduce the message complexity of 2PC by eliminating a single message from each participant site for aborting and committing transactions, respectively.

*Parenthetical references placed superior to the line of text refer to the bibliography.

Existing ACPs are not expected to scale well in future distributed database environments and applications for three reasons. First, although some ACPs are designed to support different communication topologies (1, 11, 12), in general they do not exploit the semantics of the underlying database management mechanisms and the characteristics of the distributed environment to improve their performance. Semantics-based techniques have been successfully applied to enhance the performance of database systems in the context of concurrency control and recovery protocols, e.g., (13, 14, 15, 16, 17). Second, existing protocols are designed for either highly reliable or failure-prone environments which make them non-adaptive to the changes in the behavior of transactions and the distributed environment. Finally, in the presence of a failure, all ACPs are designed to abort a transaction if the sites participating in the execution of the transaction have not reached an agreement to commit the transaction by the time of the failure. Since transactions in advanced applications are expected to be long-executing, aborting a transaction due to a partial failure and re-submitting it again after the failure is fixed leads to unnecessary waste of computing resources.

Existing protocols are also not compatible with each other and therefore cannot be integrated in a straight forward manner in *(heterogeneous) multidatabase systems* (MDBSs). A MDBS interoperates pre-existing database sites that are supporting their own applications and users, preserving their autonomy. That is, joining in a MDBS (ideally) does not require any changes to existing database management systems and applications. The incompatibility of ACPs arises due to the differences in the semantics of the coordination messages and actions that are taken during the course of the execution of the different protocols.

This dissertation addresses two issues making ACPs scalable for future database systems. The first issue deals with the *performance* of ACPs[2] in DDBSs while the other issue deals with *compatibility* of ACPs in *(heterogeneous) multidatabase systems* (MDBSs). It offers four major contributions, three of which are in the context of DDBSs while the fourth is in the context of MDBSs.

[2]In other words, the ACPs implications on system performance.

The first contribution in DDBSs is the development of two highly efficient ACPs called the *implicit yes-vote* (IYV) and the *implicit yes-vote with a commit coordinator* (IYV-WCC) protocols with different assumptions about the reliability of the database sites. Both protocols enhance performance over the current ACPs by exploiting (1) the characteristics of high speed networks and (2) the underling database management mechanisms to achieve low message, log and time complexities. The two protocols also supports our notion of *forward recovery* in which we allow partially executed transactions that are still executing in the system, after a participant site failure has occurred and has been fixed, to continue their execution on the failed participant, without having to abort them.

The second contribution in DDBSs is the performance evaluation of IYV and four other atomic commit protocols based on a simulation system under the assumption that they always succeed in committing a transaction. In our study, we also evaluate the performance gain when read-only optimizations are used. In contrast to other recent comparative performance evaluations of two-phase commit variants in local area networks (18, 19), in our simulation study, we explicitly model (1) the propagation latency of the communication network, (2) the overhead of the management of the database buffer and of flushing the transaction and protocol execution log records and (3) the overhead of recovery from site failures. By factoring in the overhead associated with these aspects, our simulation results capture more accurately the relative performance of the evaluated protocols as well as the magnitude in their performance differences.

The third contribution in DDBSs is the introduction of two new presumed commit 2PC protocol variants, called the *rooted presumed commit* and *re-structured presumed commit*, and a new read-only optimization, called the *unsolicited update-vote*. The two new presumed commit variants are proposed in the context of the multi-level transaction execution model, which is the model adopted by the distributed transaction processing standards and implemented in commercial database systems, which significantly reduce the log complexities of the original PrC and consequently the cost of commit processing in this model. The unsolicited update-vote reduces the cost of commit processing associated with read-only participants when compared to the traditional read-only optimization (3, 4). In conjunction with the unsolicited update-vote, the two new variants nullify the argument that solely favors the *presumed abort 2PC protocol*, the current choice of distributed transaction processing standards (20, 21), and argues in favor of presumed commit to become part of current database standardization efforts.

In terms of compatibility of ACPs, the fourth contribution is the characterization of the concept of a *safe* state with respect to committing transactions. This concept defines when it is possible to interoperate database sites that employ different and incompatible ACPs in a practical manner. This is especially important in the context of MDBSs. Based on the safe state concept, we develop a MDBS commit protocol called the *presumed any* (PrAny). PrAny successfully interoperates the 2PC, PrA and PrC protocols in a practical manner without violating the autonomy of the constituent database site, a necessary requirement in MDBSs. This is in spite of the conflicting presumptions about the outcome of terminated transactions in any of the three two-phase commit variants.

Road Map

This dissertation consists of three parts. Part I provides some background material for database management systems and includes Chapters 2 and 3. In Chapter 2, we discuss database system fundamentals to set the stage for the discussion of the contributions of this dissertation. Specifically, we define the transaction model that will be used in this dissertation and the properties of transactions. Then, we discuss some of the traditional techniques that are commonly used to ensure the consistency of data. In Chapter 3, we first present a taxonomy of distributed database systems. After classifying the different database systems, we discuss the protocols that have been proposed in the literature for each of the two distributed environments under consideration.

Part II focuses on the performance of atomic commit protocols in DDBSs and consists of Chapters 4, 5 and 6. In Chapter 4, we present IYV and IYV-WCC. We also thoroughly discuss the motivation behind their design as well as their behavior during normal processing and in the case of failures. Furthermore, we evaluate the performance of our two protocols and compare it with the performance of some of the protocols that we discuss in Chapter 3 based on the traditional analytical method. In Chapter 5, we evaluate the performance of IYV and four other ACPs with respect to transaction throughput using simulation. We also evaluate the performance gain when read-only optimizations are used. In Chapter 6, we present the two new presumed commit variants and the unsolicited update-vote optimization. Based on these results, we argue in favor of presumed commit to become part of the current distributed database standards.

Finally, Part III focuses on interoperability and consists of Chapter 7 in which we present our safety criterion and the *presumed any* protocol.

We conclude this dissertation with Chapter 8 in which we summarize the contributions of this dissertation and discuss some of our expected future work in the context of ACPs.

2.0 DATABASE SYSTEM FUNDAMENTALS

In this chapter, we review fundamental concepts and techniques of database technology to set the stage for the rest of this dissertation. Specifically, we will discuss traditional concurrency control and recovery techniques that are used by *database management systems* (DBMSs) to ensure the consistency of the databases that they manage.

In the next section, we define the concept of transactions and in section 2.2, we discuss the traditional properties of transactions. To guarantee the properties of transactions, a database management system combines two protocols, a concurrency control protocol and a recovery protocol which in the case of distributed transactions also includes an atomic commit protocol. Hence, in section 2.2.1, we review some concurrency control protocols, emphasizing the most commonly implemented one in the industry while in Section 2.2.2, we review recovery protocols, also emphasizing the most commonly used one. The atomic commit protocols will be examined in the related work chapter (Chapter 3).

2.1 Transactions

A transaction is a program segment that is written in a high level language to manipulate a shared database. Regardless of the type of the language used to express transactions, each transaction is translated into a set of data operations and transaction management primitives. The data objects stored in the database are manipulated by data operations which are referred to as *operation events.* Transaction management primitives, on the other hand, are termed *significant events* and used for the management of the database.

In the traditional transaction model, transactions are *atomic*. That is, each transaction is executed as a single logical unit where all its effects are either reflected in the state of the database or not at all. In this model, a transaction is a partial order set of *events* that starts by invoking a *Begin* significant event to indicate the start of the transaction. The Begin event is followed by a sequence of operation events. For example, operation events can be a *Read* or a *Write* operation on a data item. When a transaction invokes a Read operation on an object it retrieves the state (i.e., the value) of the data object whereas, when the transaction invokes a Write operation on the object, it updates the state of the object.

When a transaction finishes its execution, it invokes a significant event that is either a *Commit* or an *Abort* to indicate its intention to install the changes that it has made in the state of the database. If the event is a Commit, it means that the transaction wants to install all its effects in the state of the database, whereas, if the event is an Abort, it means that the transaction requests to cancel (i.e., rollback) it effects from the state of the database.

2.2 Database Consistency and Concurrent Transactions

To maximize transaction throughput and resource utilization in database systems, transactions are executed concurrently allowing them to interleave their operations on the database. Since the concurrent execution of transactions may cause them to interfere with each other over the data objects, database consistency might be violated. Database consistency might be also violated in the case of failures because transactions might end up partially executed, producing unpredictable results. As an example, consider a fund transfer transaction that fails after it debits one account and before it credits another one. In this example, the total credit in the two accounts becomes invalid after the transaction failure. Failures could be due to software, such as an operating system failure, or hardware, such as a power outage.

In traditional centralized database systems, database inconsistencies that are due to failures and concurrency are prevented by satisfying the, commonly known, ACID properties [1, 22, 23], which are associated with transactions. The ACID properties are: (1) *Atomicity*, (2) *Consistency*, (3) *Isolation* and (4) *Durability*. Atomicity ensures that, regardless of failures, all the operations of a transaction are treated as a single, indivisible, atomic unit that is either is performed when the transaction *commits* (i.e., finishes its execution successfully) or not at all when the transaction *aborts* (i.e., fails). Consistency is a requirement that is placed on transactions in the sense that each transaction is a computation that maintains the database consistency constraints. Isolation allows transactions to execute concurrently in the system without violating the consistency of the database. Durability ensures that the effects of committed transactions are made permanent on the state of the database, surviving any subsequent failures.

The ACID properties are usually ensured by combining two sets of algorithms. The first set ensures the isolation property and are referred to as *concurrency control protocols*, whereas the second set ensures the atomicity and durability properties and are referred to as *recovery protocols*. The consistency property is, commonly, ensured by designing transactions such that each one of them preserves the consistency of the database within its boundaries.

Now, let us review some concurrency control protocols and the governing criterion that captures their correctness.

2.2.1 Concurrency Control Protocols

Serializability [5] is the traditional correctness criterion that is used to reason about the correctness of concurrency control protocols. Since each transaction preserves the consistency of the database across its boundaries, serializability states that the concurrent execution of a set of transactions is correct if the execution is *equivalent* to some serial execution. An execution is serial if a set of transactions are executed, to their completion, one after the other. Hence, such a concurrent execution of transactions is called *serializable* and satisfies the isolation property.

Conflict serializability (5) is the criterion that is used in all commercial database systems for practical reasons. In this criterion, two operations that are to be performed on the same data object are said to be in conflict if they belong to two different transactions and one of the operations is a Write operation. This is because a Write operation affects the execution order of the two operations (24, 25, 26). Otherwise, the two operations commute and can be executed in any order. Testing whether an execution is conflict serializable or not is achieved by checking the cyclicity of a precedence graph called *serialization graph.*

There are many concurrency control protocols proposed in the literature that realize conflict serializability, such as *timestamp ordering* and *serialization graph testing* (5). In general, depending on how a protocol is implemented, it can be regarded as *pessimistic* or *optimistic.* In a pessimistic implementation, a transaction does not access a data item unless it is guaranteed to be serializable. In an optimistic implementation, transactions access to data objects is not coordinated during their execution, but at commit time, they are validated with respect to their serializability.

The most widely used concurrency control protocol that realizes conflict serializability is a pessimistic implementation of a protocol called *two-phase locking* protocol (2PL) (25). 2PL is characterized by its simplicity and ease of implementation. In 2PL, each transaction goes through a *growing phase* and a *shrinking phase.* During the growing phase, a transaction has to acquire a *lock* on each data object it wishes to access prior to accessing it. During the shrinking phase, a transaction starts releasing the locks that it has acquired during its growing phase and it is not allowed to acquire any more locks. If a transaction requests a lock on a data object and the lock cannot be granted because another transaction is holding a conflicting lock on the same data object, the requesting transaction is forced to wait until the lock is released. Since 2PL is susceptible to deadlocks (27), a deadlock detection or prevention algorithm is usually coupled with any 2PL implementation (5).

In commercial database management systems, a variant of 2PL protocol called *strict two-phase locking* (S2PL) is actually implemented. In S2PL, the locks held by a transaction T_i are not released until the transaction is either committed or aborted.

Hence, a transaction T_j can not observe the partial effects of T_i and, therefore, T_j cannot be aborted as a side effect if T_i aborts. This situation is commonly referred to as *avoiding cascading aborts* (5). Furthermore, S2PL prevents the updates of T_i from being overwritten by T_j until T_i is committed or aborted. Thus, S2PL does not only ensure serializability, from the concurrency point of view, but it also simplifies the implementation of recovery protocols that we discuss the next section.

2.2.2 Recovery Protocols

For performance reasons, portions of the stable database, that is kept on a stable (i.e., non-volatile) storage which is usually a disk, are brought into the *database buffer* in main memory based on the data requirements of transactions. After a site failure, these portions are lost and the state of the stable database might not reflect the state of entire database which also includes the state of the database buffer as it was prior to the failure. The state of the database might be left in an inconsistent state because not all the effects of committed transactions are guaranteed to have been propagated to the stable database. Similarly, there is no guarantee that none of the effects of aborted transactions were propagated to the stable database. Hence, this violates both the atomicity and durability properties of transactions.

The goal of a recovery protocol is to ensure that (1) all the effects of committed transactions persist on the stable state of the database and (2) not any of the effects of aborted transactions are reflected on the state of the stable database, despite failures. Recovery protocols are usually implemented by supporting two basic actions that are performed on the state of the stable database, namely, the *undo action* and the *redo action*. The undo action, which is required for atomicity, undoes the effects of aborted transactions from the state of the database. The redo action which is required for durability, redoes the effects of committed transactions on the state of the database. The information needed for the undo and redo actions is kept in a *log* (which is also called a *journal*).

A log can be *physical* or *logical* and is is stored as a *sequential* file in main memory. The log is *forced-written* (i.e., flushed) into stable storage when it runs out of main memory space, periodically or when an event occurs that has to be remembered in the case of a system failure. The commitment of a transaction is such an event. Each log record can be identified by its *log sequence number* (LSN) that is assigned to the records in an increasing order. In the case of a physical log, for each operation that modifies the state of a data object, the old state as well as the new state of the object are recorded in the log. The old state of an object, called the *before image*, is used by the undo action while, the new state of the object, called the *after image*, is used by the redo action. In case of a logical log, on the other hand, the log contains a high level description of the executed operation and its parameters.

To ensure that the log contains all necessary information for recovery, the updates of a transaction are not applied on the state of the database until after the log records corresponding to the updates are in stable storage, and the commitment of the transaction is not acknowledged until after all its log records are in stable storage. The former is known as the *undo rule*, whereas the latter is known as the *redo rule.*

If the propagation of the effects of transactions to the stable database is restricted, there is no need to perform either the undo or the redo actions. Thus, the need for either of the two actions depends on the management of the page replacement of the database buffer. If the updates of uncommitted transactions stored in the database buffer are not allowed to be moved to the stable database before transactions are committed, the undo action is not required. On the other hand, if the updates of committed transactions are propagated to the stable database before acknowledging the transactions' commitments, the redo action is not required. Depending on whether a recovery protocol requires the undo action, redo action or both actions, there are four types of recovery protocols, namely, Undo/Redo, Undo/No-Redo, No-Undo/Redo, No-Undo/No-Redo.

The *Undo/Redo* recovery protocol, commonly known as *write-ahead logging* (WAL), requires both undo and redo actions. In WAL, swapping into (from) the database buffer from (into) the stable database is not constrained by the commit point of transactions. This flexibility in the management of the database buffer maximizes efficiency during normal operation by increasing the degree of multi-programming at the expense of a less efficient recovery processing when compared to the other recovery protocols that do not require undo or redo. For this reason, WAL is the one used by commercial systems.

At this point, let us sketch how recovery is performed after a system failure assuming WAL and S2PL. After a system crash, the entire log is scanned in an *analysis phase* to determine which transactions have committed without having all their effects reflected in the state of the stable database and which transactions have aborted with some of their effects already reflected in the state of the stable database. After analysis, the required undo and redo actions are performed in two sequential phases, namely the *undo phase* and the *redo phase*. Depending on which phase precedes the other, there are two types of crash recovery protocols: The first one is Undo-Redo in which the undo phase precedes the redo phase (5), and the second one is Redo-Undo in which the redo phase precedes the undo phase (28). According to the LSNs recorded in the log, during the redo phase, the redo actions are performed in an increasing order while, during the undo phase, the undo actions are performed in a decreasing order. This recovery procedure guarantees that the effects of committed transactions are installed on the data objects in the same sequence as they were executed prior to the failure. It also ensures that the final state of each data object reflects the effect of the last committed transaction that has accessed the data object but not any of the aborted transactions.

Over time, the log that is used for recovery might grow substantially large which adversely affects the cost of recovery after a failure. In addition, the log might grow to a point where it might be impossible to store it in the stable storage due to the latter's limited space. Hence, a number of methods called *checkpointing* (5) have been proposed to alleviate both problems. The theme behind checkpointing methods is to periodically and their effects have been reflected in the state of the stable database, and which transactions have been aborted and their effects have been obliterated from the state of the stable database. In this way, from checkpoint marks, a recovery protocol infers from which point it should start the recovery process and which transactions it should consider. Thus, reducing the cost of recovery after a failure and, at the same time, limiting the size of the log that needs to be kept in stable storage.

2.3 Summary

In this chapter, we briefly reviewed fundamental database techniques to set the stage for the rest of this dissertation. We have defined the traditional notion of transactions that we will be using in this dissertation and their ACID (i.e., *Atomicity, Consistency, Isolation and Durability*) properties. To guarantee the ACID properties of transactions, a database management system combines a concurrency control protocol and a recovery protocol. The concurrency control protocol ensures the isolation property of transactions whereas the recovery protocol ensures the atomicity and durability properties. In the case of distributed transactions, the atomicity property is ensured using an atomic commit protocol which is the focus of this dissertation that we discuss in the next chapter.

3.0 RELATED WORK

Ensuring the atomicity property of transactions is harder in a distributed database environment than in a centralized database environment because it has to be guaranteed across multiple sites. In this chapter, we review the most commonly known *atomic commit protocols* that can be used to realize the atomicity of transactions. Before we review the different atomic commit protocols, we first discuss the differences between homogeneous, heterogeneous and multidatabase systems. Then, in Section 3.2, we overview the major protocols and optimizations that have been proposed in the literature in the context of homogeneous distributed database systems. In Section 3.3, we overview the methods that have been proposed to ensure the atomicity of transactions, in the context of heterogeneous multidatabase systems.

3.1 Distributed Database Systems: A Taxonomy

A *distributed database system* (DDBS) is a collection of database sites that are interconnected via a communication network. The data objects in a DDBS might be stored as disjoint partitions at different sites, replicated across sites or a combination of both. Individual database sites are responsible for the management of their databases while the control and coordination of transaction processing that ensure global data consistency can be either *centralized* or *distributed.*

As shown in Figure 1, we classify the different DDBSs based on three dimensions. These dimensions are: (1) *locality of control*, (2) *degree of integration* and (3) *degree of heterogeneity.* In a centralized control DDBS, only a particular site is responsible for the consistency of the distributed data by controlling and coordinating the transaction processing activities across the different sites. On the other hand, in a distributed control DDBS, the different sites share the responsibility of control over the execution

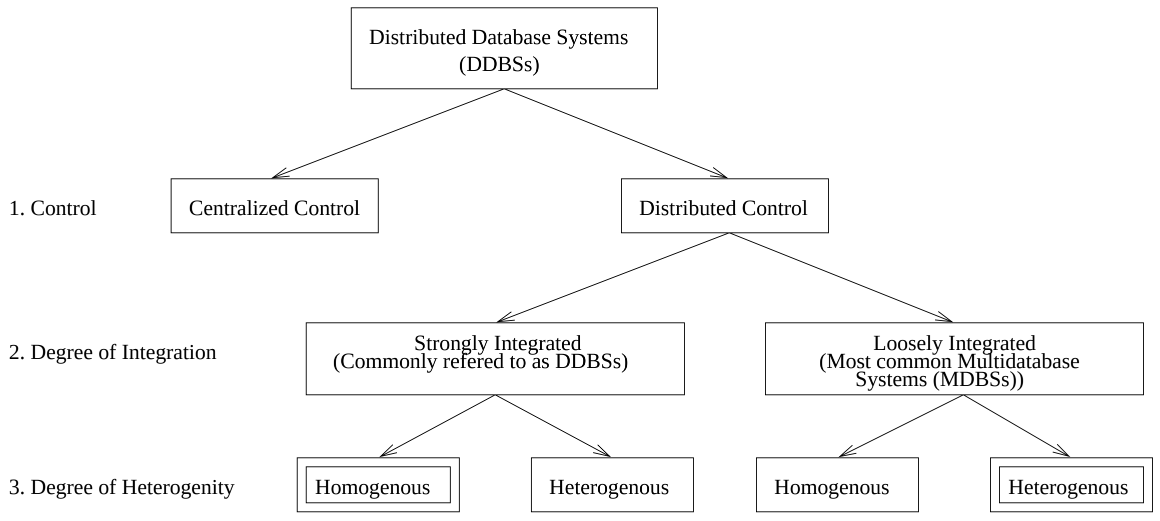

Distributed Database Systems (DDBSs)
1. Control
Centralized Control
Distributed Control
2. Degree of Integration
Strongly Integrated (Commonly refered to as DDBSs)
Loosely Integrated (Most common Multidatabase Systems (MDBSs))
3. Degree of Heterogenity
Homogenous
Heterogenous
Homogenous
Heterogenous

Figure 1 Taxonomy of distributed database systems.

of transactions and interact cooperatively to achieve data consistency.

With respect to the second dimension, i.e., degree of integration, a distributed control DDBS can be regarded as *strongly integrated* or *loosely integrated.* In a strongly integrated, distributed control DDBS, each database site shares sufficient control information regarding the state of its database and the state of the locally executing transactions with the other sites in ensuring global consistency. In a loosely integrated, distributed control DDBS, each site preserves some degree of *autonomy* and it might not be willing to exchange any control information with any other site. A special type of loosely integrated, distributed control DDBSs is *multidatabase systems* (MDBSs). A MDBS responds to the needs of different human organizations to interoperate their database systems already in service supporting their own applications and users, without making any modifications to the way they operate.

With respect to the third dimension, i.e., degree of heterogeneity, the database sites in a distributed control DDBS, whether strongly or loosely integrated, can be *homogeneous* or *heterogeneous.* In the former case, the database sites employ identical mechanisms for the management of data and transaction processing whereas in the latter case, the different sites employ different mechanisms for either the management of data, transaction processing or both.

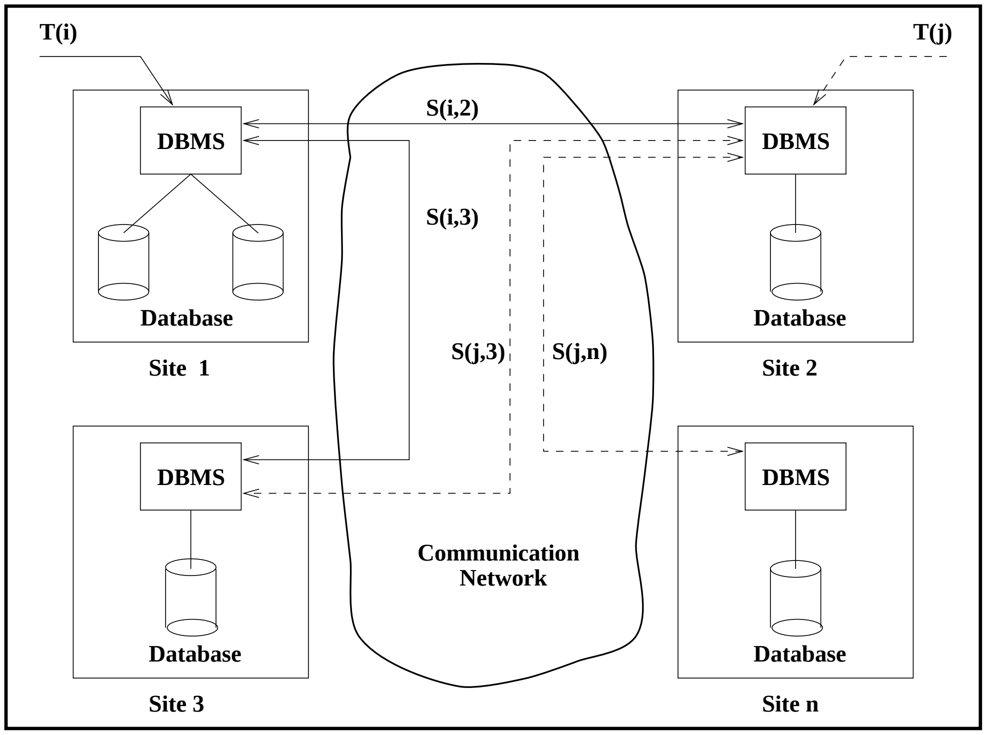


Figure 2 A distributed database environment.

The contributions of this dissertation are in the context of homogeneous, strongly integrated, distributed control database systems, commonly referred to as distributed database systems (DDBSs), and in the context of heterogeneous, loosely integrated, distributed database systems, commonly referred to as multidatabase systems (MDBSs). Before reviewing the different protocols and methods that have been proposed in the literature for both of these environments to ensure the atomicity of distributed transactions, we briefly discuss a typical distributed transaction processing environment.

In a distributed database environment, each transaction is associated with a *coordinator* which is responsible for coordinating the different aspects of the transaction execution. The coordinator of a transaction, which is assumed, without loss of generality, to be the *transaction manager* at the site where the transaction has been initiated. For example, in Figure 2, the coordinator of transaction T_i is the transaction manager, which is a component of the database management system (DBMS), at site 1. In the figure, T_i accesses data located at sites 1, 2 and 3 and transaction T_j accesses data located at sites 2, 3 and n. The data distribution is transparent to submitted transactions. A transaction accesses data by submitting its data operations to its coordinator. Depending on the location of the data objects, the coordinator determines the appropriate participant site to which it submits each data operation it receives from the transaction for execution. Hence, each transaction is decomposed by its coordinator into several *subtransactions*, each of which executes at a single site. For example, the subtransactions of T_i are $S_{i,1}$, $S_{i,2}$ and $S_{i,3}$.

The atomicity property of a distributed transaction cannot be guaranteed without taking additional measures beside concurrency control and recovery protocols. Revisiting our fund transfer example in Section 2.2, consider the case where a fund transfer transaction has finished its execution by debiting one account at one site and crediting another one in another site, and has submitted its final commit primitive to its coordinator. Now, assume that the coordinator forwards the commit primitive of the transaction to both sites that have participated in the execution of the transaction. However, one of the two participants fails before receiving the commit primitive. In this case, the participant that has failed will not find a commit record for the transaction in its log during its recovery procedure and will consider the transaction as aborted, undoing all its effects as part of its recovery procedure. On the other hand, the site that is still operational, after receiving the commit primitive of the transaction, will commit the transaction and make it effects permanent on the state of the database at its site. Thus, in this example, the atomicity of the transaction has been violated because the transaction has ended up committing at one site and aborting at the other.

To prevent such atomicity violations of transactions despite site and communication failures, an *atomic commit protocol* is usually employed in a DDBS to coordinate the commitment of distributed transactions across multiple sites. We discuss various proposals and implementations of atomic commit protocols in the next section.

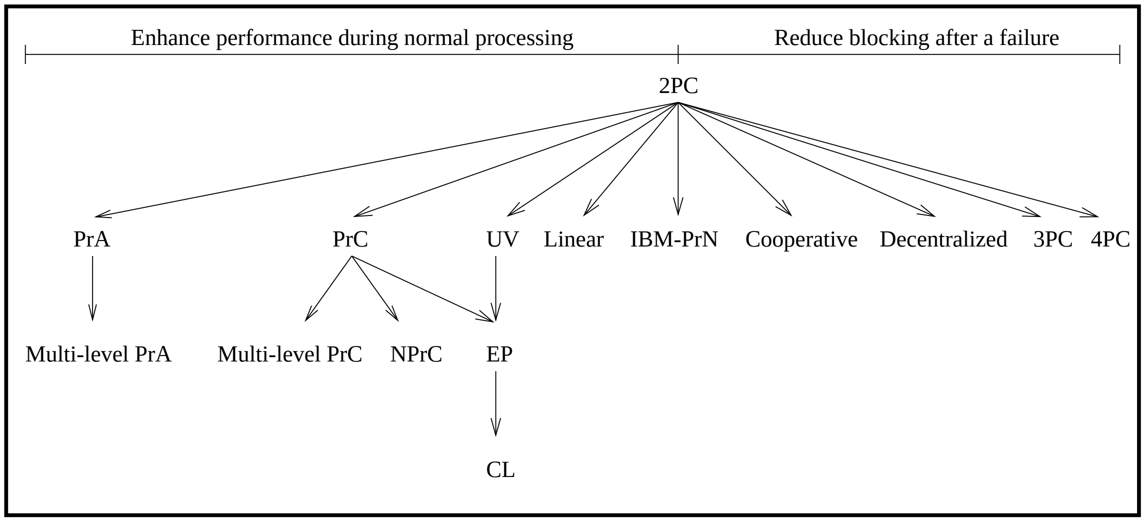


Figure 3 Significant steps in the evolution of ACPs.

3.2 Atomic Commitment in Distributed Database Systems

Since the first three contributions of this dissertation are in the context of performance of *atomic commit protocols* (ACPs) in DDBSs, in this section, we review the major ACPs and optimizations that have been proposed in the context of DDBSs. Considering that the *two-phase commit* (2PC) protocol is the first proposed ACP, all the other ACPs have evolved from 2PC and are geared towards enhancing its performance. As shown in Figure 3, some of these 2PC variants have been designed to enhance commit processing for the normal (i.e., non-failure) case while the others have been designed to reduce the cost of recovery after a failure.

Since the motivation behind the design of all ACPs is to enhance the performance of commit processing for the normal processing case or to reduce the cost of recovery after a failure, all ACPs can be regarded as optimizations to the basic 2PC. However, we distinguish between a 2PC variant and a 2PC optimization. As it will become evident below, only a single 2PC variant can be used in a DDBS with any number of optimizations. Unless stated otherwise, all ACPs discussed in this section are based on the assumptions that (1) each site is *sane* (29, 30) and (2) each site can cause only *omission* failures. That is, each site is assumed to be *fail stop* (31) where it never deviates from the specification of the protocol that it is using and when it fails, it will, eventually, recover.

In this section, we review ACPs that have been designed to enhance performance during normal processing, based on the assumption that failures are rare; and ACPs that have been designed to reduce the cost of recovery after a failure, based on the assumption that failures are frequent. We also discuss protocols that have been proposed in order to deal with *commission* failures in which a site may never recover or it may deviate from its protocol specification resulting in a non-atomic execution of transactions. In this way, we cover the whole spectrum of ACPs and the reasons behind their designs.

This section is structured as follows: In the next three subsections, we discuss the basic two-phase commit protocol and two variants, the *presumed abort protocol* and *presumed commit protocol*. In Subsection 3.2.4, we discuss the *new presumed commit protocol* that has been proposed to enhance the performance of the presumed commit commit protocol. In Subsection 3.2.5, we review some other two-phase commit protocol variants while in Subsection 3.2.6 we review some other ACPs. In Subsection 3.2.7, we discuss three important atomic commit protocol optimizations. In Subsection 3.2.8, we evaluate the performance of presumed abort and presumed commit protocols using the traditional method of performance evaluation. We also review some of the efforts that have been made in the area of analyzing and evaluating different atomic commit protocols that go beyond the traditional method.

3.2.1 The Basic Two-Phase Commit Protocol

In this section, we first describe the basic two-phase commit protocol. Then, we discuss its recovery aspects.

3.2.1.1 Description of the Basic Two-Phase Commit Protocol. As shown in Figure 4, the basic *two-phase commit* protocol (2PC) (1, 2), as the name implies, consists of two phases, namely a *voting phase* and a *decision phase*. During the voting

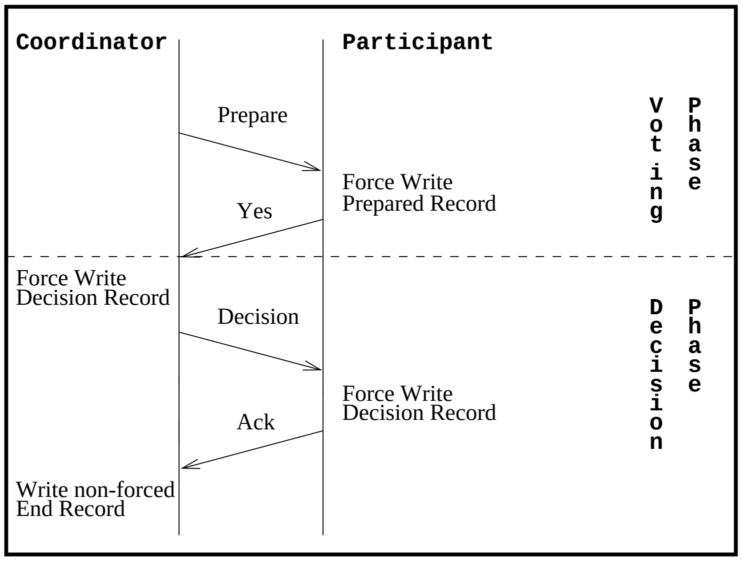

Coordinator
Participant
Prepare
Yes
Force Write Prepared Record
Voting Phase
Force Write Decision Record
Decision
Ack
Force Write Decision Record
Decision Phase
Write non-forced End Record

Figure 4 The basic two-phase commit protocol.

phase, the coordinator of a distributed transaction requests all the sites participating in the transaction's execution to *prepare to commit* whereas, during the decision phase, the coordinator either decides to commit the transaction if *all* the participants are *prepared to commit* (voted "yes"), or to abort if any participant has decided to abort (voted "no"). If a participant has voted "yes", it can neither commit nor abort the transaction until it receives the final decision from the coordinator. When a participant receives the final decision, it complies, *acknowledges* the decision and releases all the resources held by the transaction (i.e., releases the locks held by the transaction, removing the transaction control block from its table, etc.). The coordinator completes the protocol when it receives acknowledgments from all the participants.

The resilience of 2PC to system and communication failures is achieved by recording the progress of the protocol in the logs of the coordinator and the participants. The coordinator force writes a *decision* record prior to sending its final decision to the participants. Since a force write ensures that a log record is written into a stable storage that survives system failures, the final decision is not lost if the coordinator fails[1]. Similarly, each participant force writes a *prepared* record before sending its

[1]In contrast, a non-forced log write is written into the log buffer in main memory and its cost is

"yes" vote and a *decision* record before acknowledging a final decision. When the coordinator completes the protocol, it writes a non-forced *end* record, indicating that the log records pertaining to the transaction can be garbage collected when necessary. The end log record indicates to the garbage collection procedure that it can garbage collect all the log records pertaining to the transaction from the stable log.

In addition to recording the progress of 2PC in its log, each coordinator maintains a *protocol table* in its main memory. The coordinator of a transaction documents the progress of 2PC in its protocol table as well as the identities of the sites participating in the transaction's execution. This table enables the coordinator to respond to the inquiries of the participants regarding the status of a transaction, in the case of a communication or a participant failure, very quickly and without having to access its stable log. Once the 2PC protocol regarding a transaction is completed, the coordinator *forgets* the transaction by discarding all information pertaining to the transaction from its protocol table.

Clearly, in the absence of failures, 2PC ensures the atomicity of each transaction because all the sites participating in a transaction's execution will reach a consistent decision regarding the transaction and enforce it. Now, let us consider the behavior of 2PC in the case of communication and site failures.

3.2.1.2 Recovery in Two-Phase Commit Protocol. Site and communication failures are usually detected by *timeouts*. In 2PC there are four situations where a communication failure might occur. The first situation is when a participant is waiting for a prepare to commit message from the coordinator which occurs before the participant has voted. In this case, the participant may unilaterally decide to abort if it times out while waiting for the prepare to commit message. The second situation is when the coordinator is waiting for the votes of the participants. Since the coordinator has not made a final decision yet and no participant could have decided to commit, the coordinator can decide to abort. The third situation is when a

negligible compared to a forced write that requires a disk access. However, a non-forced log write might be lost in the case of a site failure.

participant has voted “yes” but has not received a commit or an abort final decision message. In this case, the participant cannot make any unilateral decision because it is uncertain about the coordinator’s final decision. The participant, in this case, is *blocked* until it re-establishes communication with the coordinator. The forth situation is when the coordinator is waiting for the acknowledgments of the participants. In this case, the coordinator re-submits its final decision to those participants that have not acknowledged the decision once it re-establishes communication with them. Notice that the coordinator cannot simply discard the information pertaining to a transaction from its protocol table or its stable log until it receives acknowledgments from all the participants.

To recover from site failures, there are two cases to consider: coordinator’s failure and participant’s failure. In the case of a coordinator’s failure, the coordinator, upon its restart, scans its stable log and re-builds its protocol table to reflect the progress of 2PC for all the pending transactions prior to the failure. The coordinator has to consider only those transactions that have started the protocol and have not finished it prior to the failure (i.e., transactions associated with decision log records without corresponding end log records in the stable log). Once the coordinator re-builds its protocol table, it completes the protocol for each of these transactions by re-submitting its final decision to all the participants whose identities are recorded in the decision record and waiting for their acknowledgment. Since some of the participants might have already received the decision prior to the failure and enforced it, these participants might have already forgotten that the transaction had ever existed. In this case, these participants simply reply with *blind* acknowledgments, indicating that they have already received and enforced the final decision.

In the case of a participant’s failure, the participant, as part of its recovery procedure, checks whether there exists any transaction in a prepared to commit state (i.e., has a prepared log record without a corresponding final decision log record). For each prepared to commit transaction, the participant inquires the transaction’s coordinator about its final decision. Once the participant receives the final decision from the coordinator, it enforces the decision and completes the protocol by acknowledging the coordinator.

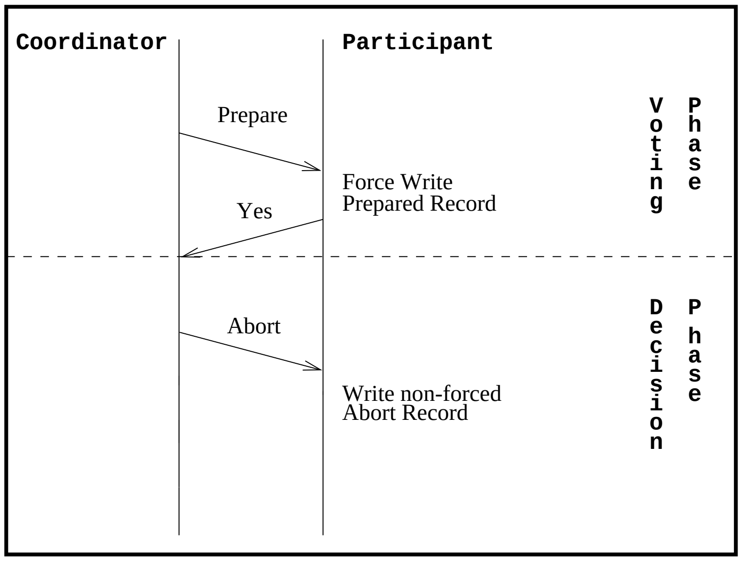


Figure 5 The presumed abort 2PC protocol (abort case).

3.2.2 The Presumed Abort (PrA) Protocol

The basic 2PC protocol is also referred to as the *presumed nothing* 2PC (PrN) protocol (7) because it treats all transactions uniformly, whether they are to be committed or aborted, requiring information to be explicitly exchanged and logged at all times. However, in case of a coordinator's failure, there is a hidden presumption in PrN by which the coordinator considers all active transactions at the time of the failure as aborted transactions. This presumption allows a coordinator in 2PC not to force write any log records prior to the decision phase. Note that a force write involves a disk access that suspends the protocol until the disk access is completed. If a participant inquires the coordinator about an active transaction after the coordinator has failed and recovered, the coordinator, not remembering the transaction, will direct the participant to abort the transaction, by presumption.

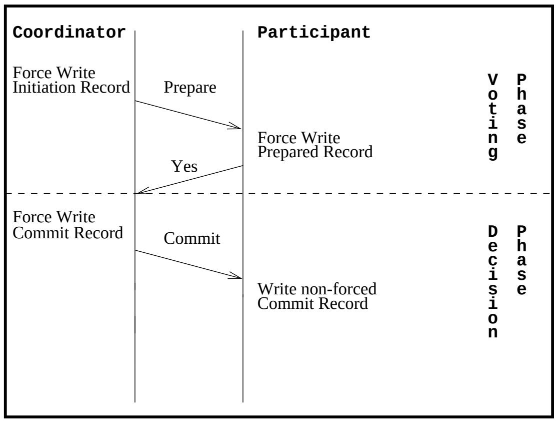
The *presumed abort* (PrA) protocol is derived from PrN to reduce the cost associated with aborted transactions by making the abort presumption explicit (3, 4). When the coordinator of a transaction decides to abort the transaction, in PrA, the coordinator discards all information about the transaction from its protocol table and sends out abort messages to all the participants without having to log an abort decision record as it would be the case in PrN (see Figure 5). After a coordinator failure, if a participant inquires about the outcome of a transaction, the coordinator, not finding any information regarding the transaction will direct the participant to abort the transaction. Furthermore, in PrA, the coordinator of a transaction does not require abort acknowledgments from the participants because it can discard all information pertaining to the transaction from its protocol table once an abort decision is made. Since the participants are not required to acknowledge abort decisions, they do not have to force write abort log decisions either. Instead, they write non-forced abort records.

Compared to PrN, PrA saves a forced log write at the coordinator's site and, a forced log write and an acknowledgment message from each participant for the abort case. For the commit case, the cost of PrA remains the same as in PrN. Failures in PrA are handled as in PrN.

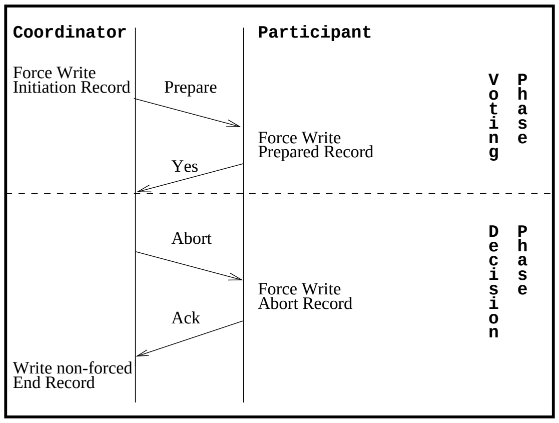
3.2.3 The Presumed Commit (PrC) Protocol

As opposed to PrA protocol that favors aborted transactions, the *presumed commit* (PrC) protocol is designed to reduce the cost of committed transactions (3, 4). It is based on the assumption that a transaction is most probably going to be committed once it has finished its execution and submitted its commit request to its coordinator.

In PrC, instead of interpreting missing information about transactions as abort decisions, which is the case in PrA, coordinators interpret missing information about transactions as commit decisions. However, in this 2PC variant, a coordinator of a transaction has to force write an *initiation* record for the transaction before sending out prepare to commit messages to the participants (Figure 6). The initiation record ensures that missing information about a transaction will not misinterpreted as a commit after a coordinator's site failure. Thus, this record is necessary for the correctness of this variant. In addition, the initiation record contains the identities of the participants that facilitate recovery after a coordinator failure, which in the



(a) Commit case.



(b) Abort case.

Figure 6 The presumed commit 2PC protocol.

case of PrN and PrA is recorded in the decision records.

As shown in Figure 6 (a), to commit a transaction, the transaction's coordinator force writes a commit record to logically eliminate the initiation record of the transaction and then sends out its commit decision to all the participants. When a participant receives the commit message, it writes a non-forced commit record and commits the transaction releasing all its resources. Since the coordinator can discard all information about a committed transaction without the acknowledgments of the participants, a participant does not acknowledge a commit decision.

As shown in Figure 6 (b), to abort a transaction, the transaction's coordinator does not force write an abort record. Instead, the coordinator, sends out abort messages to all the participants and waits for their acknowledgments. Once the coordinator receives the acknowledgments, it discards all information pertaining to the transaction from its protocol table and writes a non-forced end record. Each participant, in this case, force writes an abort record and then acknowledges the coordinator's abort decision.

In the case of a coordinator's site failure, the coordinator re-builds its protocol table by scanning its log as part of its recovery procedure and includes each transaction with an initiation record that is without an associated end record. For each of these transactions, the coordinator sends out an abort message to each participating site and waits for acknowledgments. A participant has either received and enforced the abort decision prior to the coordinator's failure or has been left blocked awaiting the final decision. In the former case, the participant will not have any recollection about the transaction and it will blindly acknowledge the decision. In the latter case, on the other hand, the participant force writes an abort log record, as if the coordinator did not fail, and then acknowledges the decision. Once all the required acknowledgments arrive, the coordinator writes a non-forced end record and forgets the transaction.

In the case of a participant failure, the participant inquires about the outcome of each transaction that has a prepare log record but without a decision record. When a coordinator receives an inquiry message pertaining to a transaction, the transaction either has an entry including an initiation record in the coordinator's protocol table, or it does not have any entry which means that the coordinator has forgotten the transaction. In the former case, the coordinator responds with an abort message and waits for an acknowledgment. In the latter case, not remembering the transaction, the coordinator responds with a commit message (hence, the presumed commit presumption holds).

Compared to PrN, PrC saves a forced log write and an acknowledgment message from each participant for the commit case at the expense of a single extra forced log write at the coordinator (i.e., the initiation log record). For the abort case, PrC incurs one extra forced log write at the coordinator compared to PrN.

3.2.4 The New Presumed Commit Protocol

The *new presumed commit* 2PC (NPrC) variant (7), eliminates the initiation record of the PrC by (1) giving up full knowledge about those active transactions prior to a coordinator's site failure and (2) giving up garbage collecting the log records pertaining to those transactions. To ensure the correctness of the PrC without having to force write initiation log records, the NPrC introduces the concept of *recent transactions* (RECT) and *potentially initiated transactions* (PIT). As shown in Figure 7

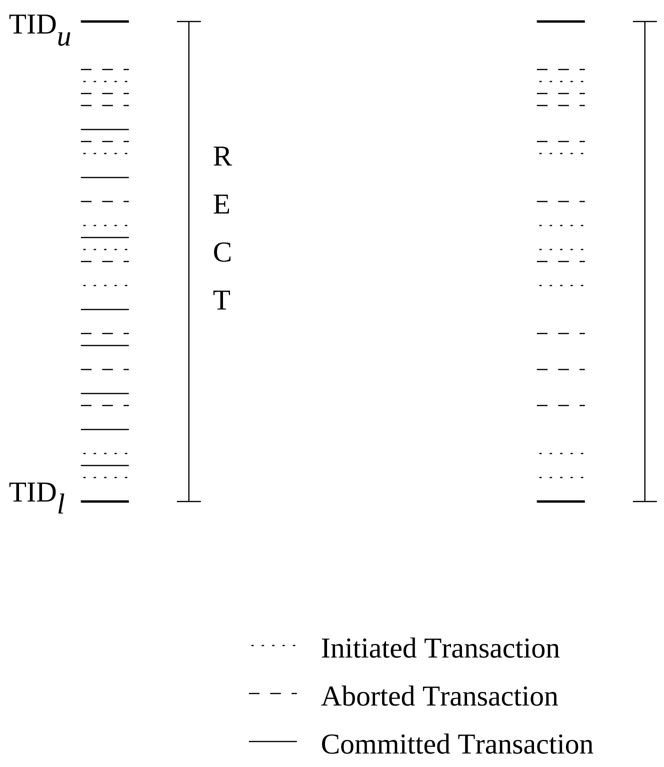


Figure 7 RECT and PIT in NPrC.

RECT is the set of transactions that contains all the transactions with *transaction identifiers* (*tids*) that are confined between a lower bound tid_l and an upper bound tid_u such that all transactions with *tids* less than tid_l have finished the NPrC protocol and no transaction with a *tid* higher than tid_u has started its execution. PIT is a subset of RECT such that no transactions in PIT has a commit final decision. This means, at the time of a failure, all transactions in PIT are either aborted transactions or transactions that were possibly active at a coordinator's site by the time of its failure.

As shown in Figure 7, the basic idea behind the NPrC protocol is to bound RECT by stably recording tid_l and tid_u. This is done by assigning transactions monotonically increasing *tids*, advancing tid_l whenever the transaction that has this *tid* completes the protocol, and ensuring that no transaction with a *tid* greater than tid_u can start its execution. When the tid_l is advanced, it is either forced written as part of the transaction log record if the transaction that caused it to advance is to be committed, or written into the log buffer if the transaction that caused it to advance is to be aborted. In the latter case, the tid_l will be propagated to the stable log when the log buffer is flushed into stable storage when the site runs out of main memory space, or when a subsequent force log write is performed. tid_u, on the other hand, is either chosen to be fixed or by periodically logging candidate tid_u.

tid_l and tid_u are used to determine the PIT set after a coordinator's failure. After a failure, the coordinator, by scanning its log, determines all committed transactions prior to the failure and their *tids* lies between tid_l and tid_u. These committed transactions have explicit commit log records in the stable log and are excluded from PIT, as shown in Figure 7. In this way, PIT contains only those transactions that are either aborted or were possibly active prior to the failure and for which it is safe to respond with an abort message if any participant inquires about any of these transactions.

For each coordinator's failure, there is a PIT set. Once the PIT set is determined after a coordinator crash, it is recorded in the stable storage and the coordinator starts with a new tid_l, that is greater than the tid_u of the previous crash, and a new tid_u. When stable memory space becomes full, the PIT sets can be garbage collected by propagating them to all possible participants. When all the participants acknowledge the reception of PIT sets, the coordinator can discard them knowing that no participant will inquire about the outcome of any of the transactions contained in these sets.

3.2.5 Other Two-Phase Commit Variants

In this section, we discuss other 2PC variants that have been proposed in the literature and adopted in some commercial systems. The first three protocols exploits the characteristics of the communication networks to enhance the performance of 2PC protocol. Then, we review the IBM-PrN protocol that has been designed taking into consideration the necessity of heuristic decision. The next three protocols have been designed to reduce message, log or time complexities by exploiting the semantics of transactions and the distributed environment. The analysis of some of these protocols can be found in the book by Bernstein, Hadzilacos and Goodman (5), while the others, such as IBM-PrN can be found in Samaras *et al.* (8).

The *linear* 2PC (1, 11) exploits the communication network characteristics to reduces message complexity at the expense of time complexity compared to the basic 2PC, making it suitable for token ring local area networks. In linear 2PC, the participants are linearly ordered with the coordinator being the first in the linear order. The coordinator sends a message to the site that follows it in the linear order. The message tells the participant the coordinator's vote and also indicates to the participant that it is time to vote. Thus, if the vote is a "yes", the coordinator prepares itself to commit before sending the message. When a participant receives a vote from its predecessor in the linear order, it prepares itself to commit if the vote that it has received is a "yes" vote and its own vote is also a "yes" vote, and sends a message to its successor. If a participant receives a "no" vote or its own vote is a no, it aborts the transaction and sends an abort message to its predecessor if it has votes "yes". The participant also sends a "no" vote to its successor if it has one. Eventually, the last participant will receive the collective vote of all its predecessors. On a commit decision, the participant commits the transaction and sends a commit message to its predecessor which in turn commits the transaction and sends a commit message to its predecessor, and so on. If the last participant decides to abort the transaction, it sends an abort message to its predecessor which in turn aborts the transaction and sends an abort message to its predecessor, and so on. The commit or abort acknowledgments are also propagated to the site which has made the final decision (i.e., the last site in the linear order) in a manner similar to the way the vote messages are propagated.

By reducing the time complexity from three messages which is the case in 2PC to two messages, it becomes less likely for a participant to be blocked during commit processing in the case of a coordinator failure. This is the motivation behind the design of *decentralized* 2PC (12). In decentralized 2PC, the interconnecting communication network is assumed to be fully connected and it reduces time complexity at the expense of message complexity. In decentralized 2PC, depending on its own vote, the coordinator sends a message to all participants. As in linear 2PC, the vote message of the coordinator has a dual role. It tells the participants that it is time to vote and, at the same time, the coordinator's vote. When a participant receives the coordinator's vote, it broadcasts a "yes" vote to the other participants only if the coordinator's and its own vote are "yes" votes. Otherwise, the participant aborts the transaction and broadcasts a "no" vote to the other participants. Hence, in decentralized 2PC, there are two rounds of messages for a participant to make a final decision. The first round is the coordinator's vote while the second is the other participants' votes.

The *cooperative* 2PC (32) also reduces the likelihood of blocking in case of a coordinator failure. In cooperative 2PC, in the case of a coordinator or a communication failure, a site does not block waiting until it re-establishes communication with the coordinator. Instead, it inquires the other operational sites that have participated in the transaction's execution. If any of the operational sites have received the final decision from the coordinator prior to the failure, it informs the inquiring site about the final decision. Thus, reducing the time for which a participant is blocked waiting for recovery from a failure.

The IBM-PrN, which is part of the SNA LU6.2 architecture (33, 8) that defines the *peer-to-peer* distributed transaction environment, the commit protocols and their synchronization in this type of an environment, is another 2PC variant. IBM-PrN has been designed taking into consideration the necessity of *heuristic decisions*. That is, in IBM-PrN, a participant site might unilaterally decide to commit or abort a transaction to avoid any unbearable delays while in a prepared to commit state, especially in case of failures. Hence, a coordinator in IBM-PrN does not make any presumptions about the outcome of a prepared to commit transaction after a site failure. This is because some participants might have decided to commit while the others have decided to abort the transaction. Therefore, in IBM-PrN, a coordinator force writes an initiation record before it sends out the prepare to commit messages, and each participant has to acknowledge the final decision regardless of whether the decision is to commit or to abort the transaction. In this way, the coordinator will be able to detect any heuristic decision and to correct it. Of course, the intervention of a human (or automated) operator is required for the restoration of consistency in the event of of a heuristic decision (34).

The *unsolicited vote* protocol (UV) (35) eliminates the voting phase of 2PC based on the assumption that each participant knows when it has executed the last operation on behalf of a transaction. In this case, a participant does not have to wait for the prepare to commit message. Instead, it sends its vote in its own initiative once it recognizes that the transaction has no more operations to process. It means that the coordinator either submits to a participant all the operations at the same time (which is a form of predeclaration) or indicates to the participant the last operation at the time that the operation is submitted. The latter implies that each transaction has knowledge about data distribution and can indicate to its coordinator when has finished accessing a participant site.

Another 2PC variant that also eliminates the voting phase of 2PC is the *early prepare* protocol (EP) (36, 9). EP is based on the assumption that the cost of accessing a stable storage in some systems is as cheap as accessing main memory. EP combines UV with PrC without assuming that a participant can recognize the last operation of a transaction. Since PrC requires the identities of the participants to be explicitly recorded at the coordinator's log in a forced initiation record, the number of forced initiation records pertaining to a transaction is equal to the number of the participants that executed the transaction. This is because the initiation record has to be updated and forced written each time a new participant is involved in the execution of the transaction. Furthermore, the participant has to prepare the transaction each time it executes an operation of the transaction and prior to acknowledging the operation. This means that the number of forced prepared records pertaining to a transaction at a participant is equal to the number of operations submitted by the transaction and executed by the participant.

The *coordinator log* protocol (CL) (36, 9) is another 2PC variant that eliminates the voting phase of 2PC. CL builds on EP. However, CL is based on the assumption that transactions are most probably going to commit and are (very) short (i.e., transactions deal with negligible amounts of data). CL eliminates the need for the forced logging activities required by EP at the participants' sites by having the coordinators maintain the logs and using *distributed write-ahead logging* (DWAL) (37). That is, the stable log of a participant site is distributed (i.e., scattered) across multiple-coordinator sites. The CL protocol also eliminates the need for the initiation log records of EP at the coordinator sites at the expense of having to communicate with all the participants in the system in case of a coordinator site failure. This is in order to determine the set of active transactions prior to the coordinator's failure and abort them instead of wrongly assuming commitment.

Thus far, we have reviewed some of the protocols that have been proposed in order to minimize the cost of commit processing during normal processing by reducing the message complexity, the log complexity or the time complexity. Among them were two protocols (i.e., decentralized and cooperative) that reduce the blocking aspects of 2PC in the case of a coordinator failure. All these protocols have two phases. Some of them have an explicit voting phase while the others implicit. In the next section, we review some of the efforts that have been made in order to eliminate the blocking aspects of 2PC by adding extra coordination messages and forced log writes.

3.2.6 Other Atomic Commit Protocols

All the protocols that we have discussed thus far have been designed to enhance the performance of 2PC during normal processing and without assuming *commission* failures. That is, they are based on the assumption that when a coordinator fails, it will recover and become operational again. Furthermore, the coordinator never deviates from its protocol specification that might lead to inconsistencies by sending different decision messages to the participants. In this section, we review other ACPs that have been proposed to reduce the cost of recovery after a failure and to enhance reliability in the presence of commission failures. Specifically, we will briefly discuss the *three-phase*, *four-phase* and *open* commit protocols.

Even though the cooperative 2PC, that we discussed in the previous section, reduces the likelihood of blocking in the case of a coordinator failure, it is still subject to blocking in the event of a coordinator's site failure when all other participants are in their prepared to commit state. The *three-phase commit* (3PC) (12) and the *four-phase commit* (4PC) (38) ACPs eliminate the blocking aspects of 2PC that are due to site failures. That is, if a coordinator fails, the participants can make their own decision. In 3PC, an intermediate buffering state between the prepared to commit and the final commit (or abort) states at the participants' sites is introduced. If the coordinator fails during commit processing, the operational sites exchange the status of the transaction among themselves and elect a new coordinator. The new coordinator commits the transaction if any operational site has the transaction in the intermediate commit state. Otherwise, the new coordinator aborts the transaction. In 4PC, on the other hand, the coordinator of a transaction initiates 2PC with a number of back-up sites that are linearly ordered. The back-up sites do not participate in the transaction execution per se but they increase the number of sites that might have status information about the transaction in the case of a coordinator's failure. Once the back-up sites have acknowledged the commitment of the transaction, the coordinator initiates 2PC with rest of the participants. Thus, in the case of a coordinator failure, the back-up site with the least identifier in the order that is still operational takes over as the new coordinator and commits the transaction as in 2PC.

The *open commit protocols* (29, 30) (OCPs) have been proposed in the context of *open distributed systems* in which the different sites have different reliability characteristics. In such an environment, a participant site is classified as *trusted* or *non-trusted* node. A trusted node is a one that fails only transiently and when it fails it does not send misleading messages. Otherwise, the node is considered non-trusted in the sense that it may never recover or it may deviates from the algorithm of the commit protocol by sending different messages including misleading ones, causing a commission failure. The theme behind OCPs is to ensure the atomicity of transactions at least across trusted nodes and despite the existence of non-trusted ones. This goal is achieved by delegating the commit processing (i.e., transferring the commit responsibilities) from a non-trusted node to a trusted one and transforming the execution tree of a transaction, through restructuring, into a different commit tree. In this way, OCPs guarantee that all trusted nodes will reach an agreement about the outcome of transactions despite the participation of non-trusted nodes.

3.2.7 Atomic Commit Protocol Optimizations

To reduce the cost associated with commit processing, a number of optimizations have been proposed in the literature and implemented in commercial systems. Some of them can be used with a number of 2PC variants as well as other ACPs. Samaras *et al.* (33, 8) survey some of the most common optimizations that are implemented in commercial transactional environments. A recent optimization is *optimistic* (OPT) (19) in which transactions do not adhere to the strict two-phase locking rules. Since transactions tend to commit when they reach their commit points, in OPT, a transaction may borrow data that has been modified by another transaction that has entered a prepared to commit state. That is, when a transaction enters its prepared to commit state, other transactions can observe its effects at the expense of aborting them if the prepared to commit transaction is aborted. When combined with an ACP, OPT enhances the overall system performance due to the early release of data held by prepared to commit transactions.

In this section, we first discuss the traditional read-only optimization (3, 4) which is one of the most important optimizations that have been adopted by the distributed transaction processing standards (20) and implemented in a wide range of commercial systems (8). Then, we mention two other optimizations due to their significance and incorporation into a number of commonly known commercial systems (e.g., DEC Dtm (10)). The first optimization is the *last agent* (33, 8) while the second one is *group commit* (39, 40) and its *lazy commit* generalization (6).

Traditionally, a transaction is termed (completely) *read-only* if all the operations it has submitted to all the participants are read operations. On the other hand, a transaction is termed *partially* read-only if only some of the participants in its execution have executed read operations. Otherwise, a transaction is termed an *update* transaction.

In the read-only optimization, when a participant that has executed only read operations on behalf of a transaction receives a prepare to commit message from the transaction's coordinator, it either replies with a "no" or "read-only" vote instead of a “yes” and immediately releases all the resources held by the transaction without writing any log records.

From a coordinator’s perspective, the “read-only” vote means that the transaction has read consistent data. Furthermore, the read-only participant does not need to be involved in the second phase of the protocol because it does not matter whether the transaction is finally committed or aborted to ensure its atomicity at the participant.

If a transaction is *read-only* (i.e., all the operations it has submitted to all the participants are read operations), it does not matter whether the transaction is finally committed or aborted since it has not modified any data. Hence the coordinator of a read-only transaction, in both PrA and PrC, treats the transaction as an aborted one. This is because it is cheaper to abort than to commit a read-only transaction with respect to logging. Recall that a coordinator does not write any log records in PrA whereas abort records are written in a non-forced manner in PrC.

The last agent optimization, being part of the IBM LU6.2 architecture, has been implemented by a number of commercial products. The last agent optimization reduces the cost of commit processing in the presence of a *single remote* participant. In the last agent, a coordinator prepares itself and the nearby participants to commit a transaction, and delegates the responsibility of making the final decision to the remote participant. This optimization has a significant performance enhancement when it is very costly for the coordinator to communicate with the remote participant in terms of messages (e.g., the communication medium with the last agent is a satellite link). By using this optimization, there is a saving of a single round of messages which is the last agent’s vote compared to the case where only the coordinator is the one which makes the final decision.

The group commit optimization has been also implemented by a number of commercial products to reduce the cost associated with the forcing of the log records. In the context of centralized database systems, a commit record pertaining to a transaction is not forced on an individual basis. Instead, a single force write to the log is performed when a number of transactions are to be committed or when a timer has expired. The latter technique is used in order to limit the response time of a transaction when the system becomes lightly loaded (i.e., not many activities are going on in the system). Thus, the cost of a single access to the stable log is amortized among several transactions.

In the context of distributed database systems, this technique is used at the participants' sites *only* for the commit records of transactions during commit processing. The *lazy commit* optimization is a generalization of *group commit* in which not only the commit records at the participants are forced in a group fashion but *all* log records are lazily forced written on stable storage during commit processing. In addition, the coordination messages pertaining to different transactions are also propagated in a grouped fashion. For example, a single message from a participant might contain the acknowledgments of several decisions pertaining to different transactions as well as votes for some other transactions. In this way, the cost of sending a single message is also amortized among several transactions.

3.2.8 Performance Evaluation of Atomic Commit Protocols

As we mentioned in the introduction of the dissertation, traditionally, the performance of ACPs is measured using the three metrics, namely, *message complexity*, *log complexity* and *time complexity*. Based on these metrics, in this section, we evaluate the performance of PrC and PrA using the above three metrics. We also evaluate their performance in the context of read-only transactions and using the traditional read-only optimization. We choose to evaluate PrC and PrA in this section just to show how performance evaluation of ACPs are usually conducted and to establish a base for our comparison of these two protocols with the ones that we will present in this dissertation (Chapters 4 and 6). In this section, we also review some of the efforts that have been made to capture the behavior of different ACPs using advanced methods that go beyond the traditional one.

3.2.8.1 Basic Performance Evaluation. Traditionally, the counting of messages and log records is depicted in a tabular form by considering a pair of a coordinator and a participant, as Table 1 shows. The table shows the cost associated with update transactions for the cases of both committed and aborted transactions assuming a "yes" vote from each participant. In the table, m denotes the total number of log records, n denotes the number of forced log writes, p denotes the number of messages received from the coordinator and q denotes the number of messages sent back to the coordinator.

Given Table 1, during normal processing, the cost to commit a transaction executing at N participants in PrA is $2N + 1$ forced log writes and $4N$ coordination messages whereas, in PrC, the cost is $N + 2$ forced log writes and $3N$ coordination messages. On the other hand, the cost to abort a transaction in PrA is N forced log writes and $3N$ coordination messages whereas, in PrC, the cost is $2N + 1$ forced log writes and $4N$ coordination messages. Thus, it is cheaper to use PrA in a system where transactions are most probably going to abort while it is cheaper to use PrC if transactions have higher probability of being committed. In a system where transactions have the same probability of being aborted as of being committed, it is cheaper to use PrA. This is because the costs of the two variants are not symmetric. Whereas the cost to commit a transaction in PrA is the same as to abort a transaction in PrC, the cost to abort a transaction in PrA is less than to commit a transaction in PrC. To abort a transaction in PrA, the coordinator does not write any log records whereas, to commit a transaction in PrC, the coordinator has to force write two log records. For a similar reason, it is cheaper to terminate a read-only transaction using PrA rather than using PrC.

For read-only transactions, a coordinator, in both PrA and PrC, aborts a read-only transaction since it is cheaper than committing it. As shown in Table 2, both PrA and PrC require the same number of coordination messages to terminate a read-only transaction. However, with respect to the logging activities, a coordinator in

Table 1 The costs associated with update transactions in 2PC and its two common variants.

Variants	Commit Decision						Abort Decision					
	Coordinator			Participant			Coordinator			Participant		
	m	n	p	m	n	q	m	n	p	m	n	q
2PC	2	1	2	2	2	2	2	1	2	2	2	2
PrA	2	1	2	2	2	2	0	0	2	2	1	1
PrC	2	2	2	2	1	1	2	1	2	2	2	2

PrA does not write any log records whereas in PrC, a coordinator has to write two log records, one of which is forced. Not knowing whether a transaction is going to be read-only, a coordinator in PrC has to force write an initiation record. To forget the read-only transaction, the coordinator also writes a non-forced end log record when it receives the read-only votes of the participants.

For a *partially* read-only transaction (i.e., only some of the participants in its execution have executed only read operations), a coordinator in both PrA and PrC behaves as in the case of an update transaction discussed above, considering only update participants in the second phase of the protocol. However, a transaction that has performed only read operations at a participant site in PrC will hold the resources at that site longer than in PrA. This is because a read-only participant in PrC has to wait until the coordinator has forced the initiation record before it receives the prepare to commit message which allows it to release the resources held by the transaction.

From the above performance analysis, we conclude that, generally, PrA is better than PrC because their costs are not symmetric. By factoring in the effects of read-only transactions, PrA is definitely the best. The is due to the cost associated with the forced initiation log records of PrC. In Chapter 6, we will propose two new presumed commit variants and a new read-only optimization that reduce the cost associated with the forcing of initiation records.

Table 2 Cost of read-only transactions using the traditional read-only optimization.

	Coordinator			Participant		
Variants	m	n	p	m	n	q
PrA	0	0	1	0	0	1
PrC	2	1	1	0	0	1

3.2.8.2 Advanced Performance Evaluations. Since site failures might lead to network partitioning, all the ACPs that we have discussed thus far are blocking. In fact, all ACPs are blocking in case of multiple partitions due to site and communication failures. Skeen and Stonebraker (41) formally modeled crash recovery in distributed database systems and proved the non-existence of non-blocking ACPs in presence of even a single network partition. Cooper (42) introduced the notion of the *window of uncertainty* which defines the conditions under which a site is vulnerable to blocking in the event of a network partition. Based on a probabilistic model, Cooper also evaluated and compared the expected number of blocked sites in the case of network partitioning for 2PC, linear 2PC, cooperative 2PC, 3PC and 4PC with three back-up sites. Not surprisingly, the results of the evaluation supports the intuition behind the design of these protocols. For example, in his evaluation, 3PC had the minimal expected number of blocked sites after a partition while 2PC had the maximal expected number of blocked sites. It should be pointed out that due to the increased cost that is incurred during normal processing with those protocols that optimize commit processing for the failure case, such as 3PC and 4PC, these protocols have never been implemented, at least in their original form, in commercial systems. In this respect, their significance is merely theoretical.

With respect to performance evaluation, until recently (18, 19), there was no comprehensive and comparative experimental studies among different ACPs that go beyond the use of the traditional analytical method. This is despite the fact that traditional performance evaluation techniques fail to show the impact on the magnitude of performance differences on a system's overall performance when using one ACP versus another one. Even in these recent studies, a number of important aspects have not been considered. These aspects include the modeling of main memory buffer management, the type of write-ahead logging and the type of the communication networks. In our work, that we present in Chapter 5, we consider some of these aspects. By factoring in the effects of these aspects, we reveal the hidden overhead of the ACPs that we evaluate in our study. Consequently, our results reflect more accurately the magnitude of performance differences on a system's performance when choosing one ACP versus another.

Thus far, we reviewed some of the related work that has been conducted by the different researchers in the context of DDBSs. Since this dissertation also addresses the issue of atomicity in the context of MDBSs, in the next section, we review some of the related work that has been done in this context.

3.3 Atomic Commitment in Heterogeneous Multidatabase Systems

In this section, we first describe a multidatabase environment and its constituent components. Then, we overview the different efforts that have been made to ensure the atomicity of transactions in the context of multidatabase systems and relate our work to these efforts.

As shown in Figure 8, a *multidatabase system* (MDBS) is a software system that facilitates interoperability across multiple pre-existing and heterogeneous database systems. A MDBS allows each database system to continue to operate in an independent fashion and (ideally) does not require any changes to existing databases, applications, and the local database management systems (LDBMSs).

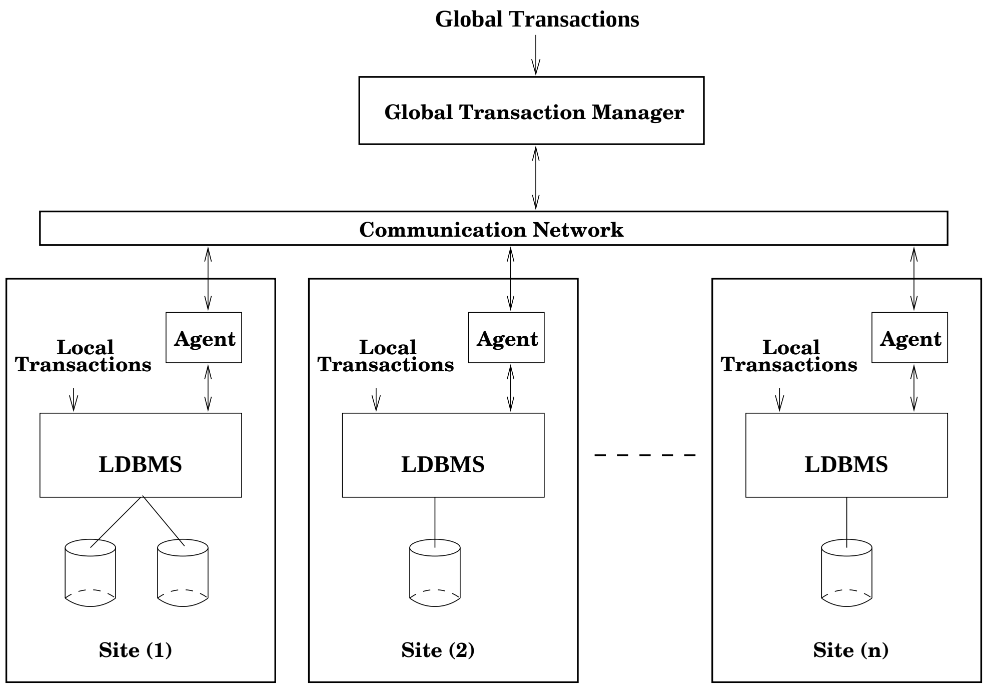


Figure 8 The MDBS model.

As shown in Figure 8, two types of transactions execute in a MDBS:

- *local transactions* that access data located at the sites where these transactions are initiated. These transactions execute only under the control of the LDBMSs and the MDBS is not aware of their existence.
- *global transactions* that access data located at multiple databases. Global transactions are submitted to and executed under the control of the *global transaction manager* (GTM) of the MDBS.

As in the case of (homogeneous) DDBSs, a global transaction is decomposed by the GTM into several *subtransactions*, each of which executes as a local transaction at some site. An *agent* which resides above each LDBMS, is responsible for the different aspects of the execution of subtransactions at its site and in particular, of the termination protocol needed to ensure the atomicity of global transactions.

As shown in Figure 9, there are three approaches that have been proposed in the literature to ensure the atomicity of global transactions. The figure captures

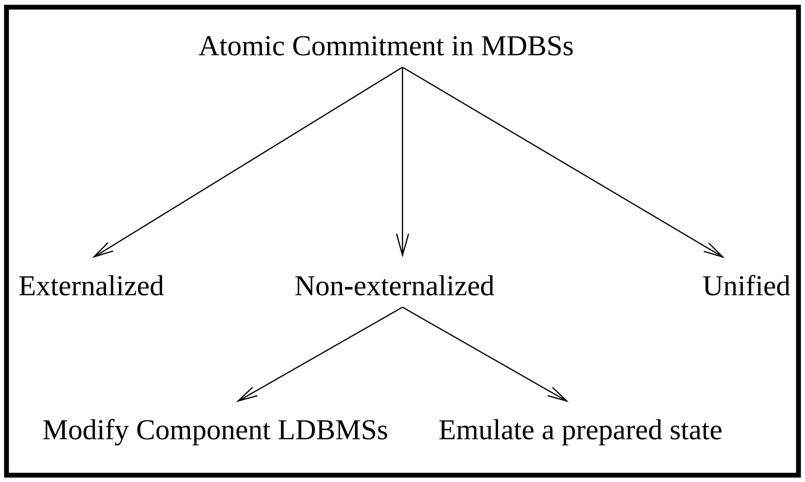


Figure 9 Taxonomy of atomic commitment in MDBSs.

these approaches in a form of a taxonomy along the same lines as previous taxonomies (43, 44, 45). In the figure, we classify these three approaches as *externalized*, *non-externalized*, and *unified* ACPs. This classification is based on the assumption about whether each LDBMS supports a visible prepared to commit state or not. That is, whether or not each LDBMS uses and externalizes a commit protocol. An externalized LDBMS makes its commit protocol public to the outside world through its interface by accepting and responding to system call pertaining to its commit protocol. On the other hand, a non-externalized LDBMS does not accept or respond to an atomic commit protocol system calls. In an MDBS it might happen that there are some LDBMSs that externalize their ACPs while others do not. Hence, the two types of termination protocols complement each other. Therefore, a third type of termination protocol, referred to as *unified*, which combines the other two types has been also proposed in the literature. Even though our work fits in the first category (i.e., externalized ACPs), we also briefly survey the ACPs that have been proposed in the other two categories for the sake of completeness.

3.3.1 Externalized Atomic Commit Protocols

Given the current standardization efforts (20, 21), the research in this direction is based on the assumption that future LDBMSs will support ACPs with externalized prepared to commit states. Thus, the challenge is to integrate LDBMSs that use different and incompatible ACPs. The incompatibility of ACPs means that the semantics of the coordination messages and the actions that are taken by a LDBMS that uses one ACP might be completely different than their counterparts in another ACP. Integrating LDBMSs that use incompatible ACPs in a MDBS is not as trivial a task as it was previously believed (46, 43, 47, 48). It is not simply the case that once a LDBMS supports a visible prepared to commit state, it can be integrated in a MDBS regardless of the ACPs used by the other LDBMSs (12, 49).

Pu *et al.* (46) concentrated on integrating *asymmetric* commit protocols and *symmetric* ones in their Harmony prototype which integrates centralized LDBMSs where each of which supports one of the two classes of ACPs. In asymmetric protocols, such as PrN, only the coordinator is responsible for making the final decision whereas in symmetric protocols, such as decentralized 2PC, all participating sites have the same right in the execution of the protocol and can commit independently. In the Harmony project, a component system called *Supernova* is responsible for translating agreement protocols among the different LDBMSs. In the event that some LDBMSs use asymmetric commit protocols while the others employ symmetric ones in a transaction execution, Supernova is the coordinator of all asymmetric ACPs and a member of the symmetric ones. To ensure the atomicity of a transaction, Supernova does not send out its vote to the symmetric members until it hears from all LDBMSs that use asymmetric ACPs. Thus, the symmetric participants are prohibited from making a unilateral decision that might jeopardize atomicity until they receive the Supernova's vote.

Tal and Alonso (47, 48) take the problem one step further. In their work, they assume that each LDBMS is a DDBS that supports an ACP. In this case, ensuring the atomicity of transactions is further complicated because a LDBMS, being distributed, might reach a different decision about the outcome of a transaction than the other participating LDBMSs in the transaction's execution. For example, assume that the LDBMSs participating in a global transaction's execution are using decentralized 2PC. Furthermore, assume that the GTM has sent prepare to commit messages to all participating sites. Since a prepare to commit message in decentralized 2PC has a dual role (i.e., it tells each LDBMS that the transaction has finished its execution and at the same time the GTM's vote), one LDBMS might commit the transaction if all the participants at its site have exchanged "yes" votes and another LDBMS might abort the same transaction if any participant at its site has decided to abort (i.e., has sent a "no" vote). By using *auxiliary* participant processes at each LDBMS, Tal and Alonso interoperate the basic 2PC, linear 2PC, decentralized 2PC and 3PC. For example, in the case of a decentralized 2PC-based LDBMS, the auxiliary participant (which can be thought of as the agent in our model) is used to prohibit the other participants at its site from making a decision by not sending its vote to the participants at its site until it receives the final decision from the GTM. In this way, the participants are blocked from being able to make a commit decision that might latter jeopardize the atomicity of the transaction.

Mohan *et al.* (50, 51) designed a new 2PC variant called the *generalized presumed abort* (GPA) protocol. The motivation behind the design of GPA protocol is to achieve the efficiency of presumed abort protocol and the generality of the *peer-to-peer* distributed transaction environment. In contrast to the *client-server* environment in which only the coordinator can initiate commit processing, in the peer-to-peer environment, any participant can initiate commit processing. To allow existing sites to continue to use IBM-PrN without having to modify any of the software applications at these sites, the GPA is designed as a superset of IBM-PrN. As in presumed abort protocol, in GPA, the coordinator of a transaction (i.e., the participant that initiates commit processing) does not force write any log records before the voting phase in the event that all other participants are using GPA. Thus, GPA is similar to PrA with respect to logging in the absence of IBM-PrN participants. However, with respect to the coordination messages, participants have to acknowledge both commit as well as abort decisions. This modification to the original presumed abort protocol, in GPA, is in order to cope with heuristic decisions that IBM LU6.2 supports. On the other hand, if some participants are using IBM-PrN, the coordinator, in GPA, force writes an initiation record as in the case of IBM-PrN. These modifications to the presumed abort protocol, in GPA, makes GPA a superset of IBM-PrN that allows for a smooth migration from the old protocol (i.e., IBM-PrN) to the more efficient GPA protocol.

When interoperating different ACPs, there are two issues that we need to deal with, as we discuss in Chapter 7. The first issue is *functional correctness*, in which all participating sites in a transaction's execution should reach the same decision about the transaction. The second issue is *operational correctness*, in which that all participating sites (including the coordinator) should also be able to, eventually, forget a transaction and garbage collect its log records. The work by the first two research groups discussed above has been concentrated on reaching functional correctness when interoperating different ACPs. That is, reaching a consistent agreement regarding the outcome of global transactions irrespective of the cost of having to remember the outcome of terminated transactions forever. In our work, we concentrate on the second part of the problem which deals with operational correctness. This is a more pragmatic approach that is implicitly adopted in all commercial database systems. From this point of view, a practical integration of ACPs should also allow all sites participating in a transaction's execution to, eventually, forget about the outcome of the transaction and garbage collect their log. The third research group above also focused on this issue *but* concentrated on the implementation aspects of PrA in order to make it fit into a pre-existing commercial architecture. For this reason, the original PrA protocol was modified such that it acknowledges both commit as well as abort decisions. In our work that we present in Chapter 7, we interoperate PrN, PrA and PrC without modifying any of them hence, preserving the autonomy requirement in MDBSs.

In the next section, we briefly overview the work that has done in the context of non-externalized ACPs.

3.3.2 Non-Externalized Atomic Commit Protocols

Most of the existing literature on MDBSs is based on the assumption that each LDBMS does not externalize its ACP. This assumption means that each LDBMS is either centralized in nature, hence does not support any form of an ACP, or it does not externalize its ACP even though it supports one (i.e., it does not expect or respond to system calls pertaining to its ACP from other systems). Thus, the challenge is to ensure the atomicity of global transactions despite the fact that each LDBMS does not externalize an ACP. The protocols reported in the literature can be classified into two categories. The first category of protocols suggest modifying component LDBMSs to support an externalized ACP while the second category of protocols achieve the atomicity of global transactions by emulating a prepared to commit state to ensure the atomicity of global transactions.

With respect to autonomy, it should be pointed out that all the protocols that have been proposed in this direction impose autonomy violations in one way or another. They only differ in the way they impose autonomy violations and the degree of such violations. That is, some of them require major changes to LDBMSs while others require changes to local applications, impose restrictions on the data access pattern, or the initiation of transactions. The autonomy violations that some of these methods impose have been formally analyzed by Chrysanthis and Ramamritham (52), explicitly showing the effects of these methods on autonomy and the trade offs between different types of autonomy.

In the next section, we discuss why modifying LDBMSs in order to support an externalized ACP is not a reasonable alternative.

3.3.2.1 Modify Component Local Database Management Systems. To ensure the atomicity of global transactions, some researchers suggested modifying the source code of LDBMSs to support an externalized ACP (49, 46). These proposals do not only violate the autonomy requirement of LDBMSs; but might be impossible to achieve. This is because even if the man power required to make such modifications is available, LDBMSs come from different vendors and their source code is usually not available for such modifications.

An alternative to this solution is to emulate a prepare to commit state that ensures the atomicity of global transactions which we discuss next.

3.3.2.2 Prepared State Emulation. The other alternative that ensures the atomicity of global transactions is to emulate a prepared to commit state at each LDBMS site without having to modify the source code of the LDBMS. Emulation-based protocols still violate the autonomy of the database sites with various degrees but less than modifying the source code of the component LDBMSs discussed above. In this direction, guaranteeing the atomicity of global transactions can be achieved using one of three approaches that we discuss below.

The three approaches are: (1) *commitment after*, (2) *commitment before* and (3) *hybrid*. The first two approaches are based on the relative commitment of the subtransactions pertaining to a global transaction at the participating LDBMSs with respect to the commitment the global transaction by the GTM (53). That is, in the first approach, commitment after, the LDBMSs participating in a global transaction's execution commit the transaction locally after the GTM has made the final commit decision. Thus, in the event that a LDBMS aborts the transaction after the final commit decision is made but before it has received the decision, the effects of the transaction should be *redoable* or *retriable*. In a redo-based method, only the write operation of an aborted subtransaction are redone again whereas in a retry-based method, all the operations, including the read operations, are re-executed again. A number of methods have been proposed to realize the commitment after approach. These methods are based on either *data partitioning* (54, 55, 56, 57), *re-routing* (53, 58), *MDBS exclusive right* (59, 60, 61, 62, 63) or *reservations* (64, 45). For example, in data partitioning, the type of a transaction (i.e., local versus global) determines which data items the transaction can access and in which mode (i.e., update versus read). By restricting the access pattern of transactions to data, the subtransactions pertaining to a global transaction can be redone.

In the second approach, commitment before, the LDBMSs commit a transaction before the GTM has made its final commit decision. Thus, in the event that the GTM has finally decided to abort the transaction, the effects of the transaction should be *undoable*. The methods that realize commitment before are based on either *compensation* (1) or *reservations* (64, 45). Compensation-based methods guarantee *semantics atomicity* which is a weaker notion than the traditional atomicity of transactions, whereas reservations can be used to ensure either notions of atomicity. In a compensation-based method, unlike traditional atomicity, the state of a data item after the transaction is undone might not be the same as it was before the transaction had modified the data item. For example, in a hotel reservation application, the number of single rooms available in the hotel after a transaction has reserved a room (for a client) might not be the same after its associated compensating transaction has executed (i.e., if the client has decided to cancel his/her reservation after a couple of days). This is because other transactions and compensating transactions might have executed in between the client's two transactions.

The third approach, hybrid, is basically a combination of the other two approaches. For example, in the *pivot* method, some subtransactions of the same global transaction can be retriable while others are compensatable (65). The pivot subtransaction, on the other hand, is neither retriable nor redoable. To ensure the atomicity of whole transaction, all compensatable subtransactions should always commit before the pivot subtransaction while all retriable subtransactions should commit after the pivot subtransaction.

As we mentioned above, the unified approach, which we discuss next, combines both externalized and non-externalized methods to ensure the atomicity of global transactions.

3.3.3 Unified Atomic Commit Protocols

With minor differences with respect to their classifications and terminologies, both Nodine (44) and Mullen (45) proposed a unifying approach in which they integrate different methods that ensure global transaction commitment. Both researchers attempt to provide the most general and flexible framework that ensures the atomicity of transactions despite the diversity of the semantics of transactions and data. However, in both of these works, only functional interoperation is achieved with respect to the traditional atomicity of transactions, since the only requirement is the support of a visible prepare to commit state by each site that employs an ACP. Otherwise, each site must use the same ACP if operational interoperation is to be achieved. Our work differs from their work in that we do not assume a single externalized 2PC variant but rather we integrate LDBMSs that use different externalized 2PC variants in a practical manner.

3.4 Summary

In this chapter, we classified the different types of distributed database systems based on a three-dimension taxonomy. These dimensions are: (1) locality of control, (2) degree of integration and (3) degree of heterogeneity. Based on our taxonomy, we emphasized that our contributions lie within homogeneous DDBSs and heterogeneous MDBSs. Then, we reviewed related work in both areas of research. In the context of DDBSs, we have related the evolution of the different ACPs to the first published and known ACP namely, the two-phase commit protocol. In the context of MDBSs, we have classified the different methods that have appeared in the literature based on their assumptions about whether the local database management systems externalize ACPs or not, and briefly surveyed some of these methods.

In the next chapter, we present the first contribution of this dissertation, namely, the *implicit yes-vote* and *implicit yes-vote with a commit coordinator* protocols which are targeted towards future distributed database systems interconnected via high speed networks.

4.0 THE IMPLICIT YES-VOTE COMMIT PROTOCOL

An efficient ACP that scales well for future gigabit-networked distributed database systems should minimize message, log and time complexities. In this chapter, we present such a protocol, called *implicit yes-vote* (IYV), that minimizes all three metrics by eliminating the (explicit) voting phase of the two-phase commit protocol, in particular of presumed abort (PrA) protocol. IYV achieves this performance enhancement by exploiting (1) the semantics of the database management mechanism, i.e., the use of strict two-phase locking (S2PL) and write-ahead logging (WAL), and (2) the characteristics of gigabit-networks, i.e., the propagation latency of messages is more of an issue than the size of messages (66, 67). Based on these two assumptions, IYV not only effectively *eliminates* the (explicit) voting phase of the two-phase commit (2PC) protocol but it also supports the notion of *forward* recovery. Through forward recovery, IYV enables partially executed transactions at a participant site, after the participant has failed and recovered, to resume their execution on the failed participant without having to abort them.

Based on the first assumption (i.e., S2PL), it is not possible for a participant to abort a transaction after it has executed and acknowledged all the operations received from the transaction. In IYV, rather than forcing a prepared log record at the end of a transaction's execution at a participant which indicates the *explicit* prepared to commit state which is the case in the 2PC protocol, a transaction enters an *implicit* prepared to commit state after the execution of each of its operations without force writing any prepared to commit log records. That is, since the transaction is guaranteed to be serializable after the execution of the last operation at a participant site, the transaction is assumed to have entered a prepared to commit state at the participant site once the last operation is executed and acknowledged. Two fundamental questions still need to be answered in order to be able to utilize the implicit prepared state idea: (1) How would a participant be able to recover the state of a transaction with respect to the control information over its database, in the case of a site failure, without having to explicitly prepare the transaction before acknowledging each of its operations i.e., without force writing a prepared log record? (2) How would a participant be able to forward recover a transaction if it cannot re-acquire the read locks that were held by the transaction prior to a failure to fully reconstruct the control state over the database?

IYV answers both questions through a partial replication of the control information of the participant sites at the coordinators, given that (1) migrating large amounts of data from a participant to a coordinator (and vice versa) in gigabit networks will not pose a problem (68, 69, 70); and (2) the cost of forcing a log is practically independent of its size, i.e., the number of records to be written, and is due to queueing delays. We call this replication of control information *replicated write-ahead logging* (RWAL). In RWAL, the redo part of a participant's log is transferred and logged at the coordinators' sites. To facilitate forward recovery, each participant's lock table is also partially replicated at the sites of the coordinators.

In the next section, we describe IYV and discuss its behavior in the presence of failures. The basic IYV assumes that all participant sites are highly reliable. However, due to pragmatic considerations, it is expected that, for some time, future distributed database sites might contain sites that use old technologies with less reliability characteristics. For this reason, in Section 4.2, we propose an IYV variant called the *implicit yes-vote with a commit coordinator* (IYV-WCC) that takes into consideration the reliability characteristics of individual database sites (71). In Section 4.3, we discuss the correctness of the IYV assumption about the semantics of S2PL and explain the need for the propagation of the read locks to the coordinators to support forward recovery in more detail. In Section 4.4, we discuss the differences and similarities of IYV with other major 2PC variants whereas, in Section 4.5, we analytically evaluate the performance of IYV and IYV-WCC and compare it with the performance of the same 2PC variants that we discussed in Chapter 3.

4.1 The Implicit Yes-Vote (IYV) Protocol

In this section, we present the details of our protocol and discuss what kind of information is needed to be logged and where. Then, in Section 4.1.2, we discuss the recovery aspects of IYV.

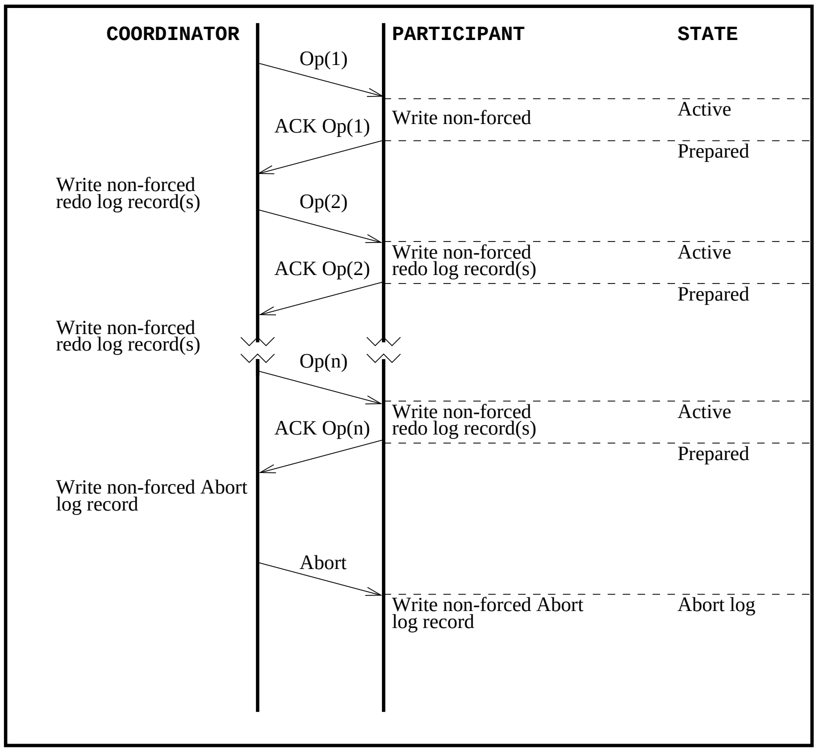
4.1.1 Description of IYV Protocol

As in the case of 2PC, a coordinator records information pertaining to the execution of a transaction in its protocol table in main memory. Specifically, a coordinator keeps for each transaction the identities of the participants and any pending request at a participant.

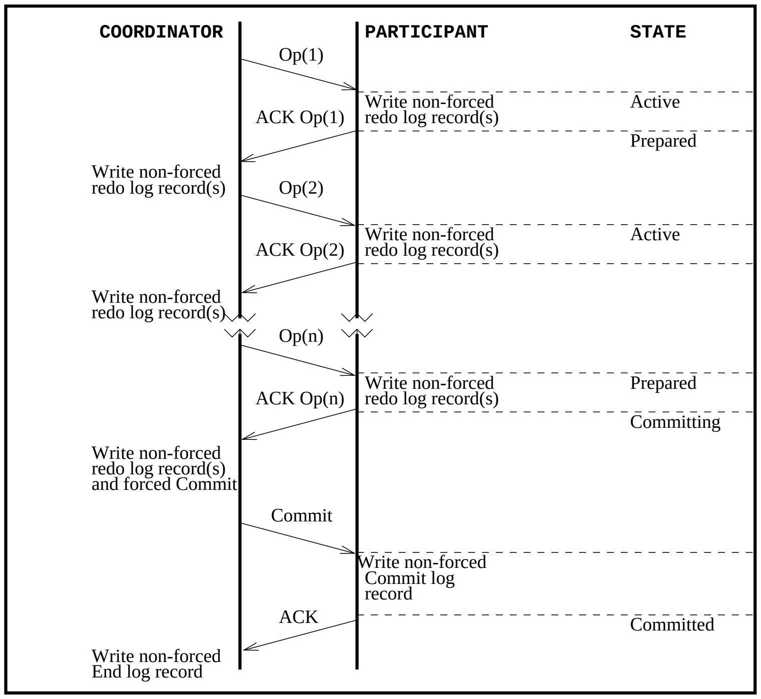
An operation submitted by a transaction can be either an *update* or a *read* operation. Following the S2PL protocol, before the execution of an operation, a participant places a write lock on each data item that is to be updated and a read lock on each data item that is to be read by the operation. The locks are kept in a lock table in main memory.

Once an operation is executed successfully, the participant acknowledges (ACK) the coordinator with a message that contains the results of the operation. In IYV, the participant also includes all the read locks that have been acquired during the execution of the operation. For an update operation, a participant also includes in the acknowledgment message all the *redo* log records that have been generated during the execution of the operation with their corresponding *log sequence numbers* (LSNs). As shown in Figure 10, a participant does not force its log into stable storage prior to acknowledging an operation. If a participant fails to process an operation, it aborts the transaction and sends a *negative* acknowledgment (NACK) to the transaction's coordinator.

When a coordinator receives an ACK from a participant, it writes a non-forced log record containing the received redo records along with the participant identity. Hence,


COORDINATOR
PARTICIPANT
STATE
Op(1)
Write non-forced
Active
ACK Op(1)
Prepared
Write non-forced redo log record(s)
Op(2)
Write non-forced redo log record(s)
Active
ACK Op(2)
Prepared
Write non-forced redo log record(s)
Op(n)
Write non-forced redo log record(s)
Active
ACK Op(n)
Prepared
Write non-forced Abort log record
Abort
Write non-forced Abort log record
Abort log

(a) Abort case


COORDINATOR
PARTICIPANT
STATE
Op(1)
Write non-forced redo log record(s)
Active
ACK Op(1)
Prepared
Write non-forced redo log record(s)
Op(2)
Write non-forced redo log record(s)
Active
ACK Op(2)
Write non-forced redo log record(s)
Op(n)
Write non-forced redo log record(s)
Prepared
ACK Op(n)
Committing
Write non-forced redo log record(s) and forced Commit
Commit
Write non-forced Commit log record
ACK
Committed
Write non-forced End log record

(b) Commit case

Figure 10 The IYV coordination messages and log writes.

the coordinator's log contains a partial image of the redo part of each participant's log which can be used to reconstruct the redo part of a participant's log in case it is corrupted due to a system failure. Also, as part of its protocol table, the coordinator maintains a *participants' lock table* (PLT) in which it records the read locks included in the ACK message along with the identity of the participant. As a result, the coordinator's PLT contains a partial image of the lock table of each participant. The PLT is used in the case of a participant failure in order to enable the participant to recover its state *exactly* as it was prior to the failure, thereby allowing partially executed transactions that are still active in the system to forward recover and resume their execution after the participant has recovered, without violating serializability (see subsection 4.3).

As shown in Figure 10 (a), if the coordinator receives either an abort request from the transaction or a NACK regarding the transaction from a participant, the coordinator aborts the transaction. In Figure 10 (a), we assume that the transaction has requested to abort. On an abort decision, the coordinator writes a non-forced abort log record and sends abort messages to all the participants. On the other hand, when the coordinator of a transaction receives a commit primitive from the transaction after all the transaction's operations have been successfully executed and acknowledged, as shown in Figure 10 (b), it commits the transaction. On a commit decision, the coordinator force writes a commit log record that contains the identities of all participants and then, sends commit messages to all participants.

When a participant receives a commit message regarding a transaction, it writes a non-forced commit log record and commits the transaction, releasing all the transaction's resources. A participant acknowledges a commit decision only after the decision log record is placed into stable storage as a result of a log buffer overflow. If a participant receives an abort message, it writes an abort record and aborts the transaction, releasing all the resources held by the transaction. Unlike commit decisions, a participant in IYV does not acknowledge an abort decision, similar to the presumed abort protocol (that we discussed in Section 3.2.2).

For a commit decision, when the coordinator receives acknowledgments from all the participants, it writes a non-forced end log record and discards all information pertaining to the transaction from its protocol table including the list of locks that are stored in the PLT, knowing that no participant will inquire about the transaction's status in the future.

4.1.2 Recovery in IYV Protocol

As shown in Figure 11, IYV is resilient to both communication and site failures. As is the case in the basic two-phase commit protocol (2PC) and all its variants, site and communication failures are detected by timeouts.

4.1.2.1 Communication Failures. Although communication failures are assumed to be rare in high speed networks, there are three situations in IYV where a communication failure might occur while a site is waiting for a message. These situations represent checkpoints at which a communication failure might occur during the course of the protocol, as depicted earlier in Figure 10. The first situation is when a participant has no pending acknowledgments and is waiting for a new operation or a final decision. This is shown as the first case of the communication failure in the participant algorithm in Figure 11. In this case, the participant is blocked until the communication with the coordinator is re-established. Then, the participant inquires the coordinator about the transaction's status. The coordinator replies with either a final decision or a *still active* message. In the former case, the participant enforces the final decision and then acknowledges it if it is a commit decision, while in the latter case, the participant waits for further operations.

The second situation is when the coordinator of a transaction is waiting for an operation acknowledgment from a participant. This is shown as the first case of communication failures in the algorithm of the coordinator in Figure 11. In this case, the coordinator may abort the transaction and submit a final abort decision to the rest of the participants. Similarly, the participant may abort the transaction if the communication failure has occurred while the participant has a pending acknowledgment. This is shown as the second case of communication failures in the participant algorithm in Figure 11. Notice that the coordinator of a transaction may commit the transaction despite communication failures with some participants as long as these participants have no pending acknowledgments.

The third situation is when the coordinator of a transaction is waiting for the acknowledgments of a commit decision. Since the coordinator needs the acknowledgments in order to discard the information pertaining to the transaction from its protocol table and its log, it re-submits the decision once these communication failures are fixed (the second case of communication failures in the coordinator algorithm). When a participant receives the commit decision after a failure, it either just acknowledges the decision if it has already received and enforced the decision prior to the failure[1], or enforces the decision and then sends back an acknowledgment.

4.1.2.2 Site Failures. As mentioned above, we are assuming that each site employs physical logging and uses a Undo/Redo crash recovery protocol in which the undo phase *precedes* the redo phase (that we discussed in Section 2.2.2).

Coordinator Failure

Upon a coordinator restart, after a failure, the coordinator re-builds its protocol table by scanning its stable log. The coordinator needs to consider only those transactions that have commit decision records without a corresponding end records. As shown in the coordinator recovery algorithm (the first step after a site failure in Figure 11), for each of these transactions, the coordinator creates an entry in its protocol table that includes the identities of the participants as recorded in the transaction's decision record. Then, it restarts the decision phase of IYV for each of these transactions by re-submitting its decision to all the participants and resumes normal operation.

[1]A participant without any memory regarding the transaction is assumed to have already enforced the decision and discarded all information pertaining to the transaction.

Coordinator's Algorithm

In case of a communication failure:

1. Abort each active transaction that has a pending acknowledgment at an inaccessible site or no participant site can be found to process one of the transaction's operations.

2. Re-submit the commit decision of each committed transaction without and end record after the failure is fixed.

In case of a site failure:

1. For each transaction that has a commit decision record in the stable log without a corresponding end record, include the transaction in the protocol table and restart the decision phase.

2. Abort all active transactions (i.e., transactions without decision log records).

3. Do not consider transactions with end records already in the stable log.

4. Resume normal processing.

Participant's Algorithm

In case of a communication failure:

1. Wait until the failure is fixed and then inquire about the status of all active transactions without pending acknowledgments.
    - Either a *decision* or a *still active* message will be received for each of these transactions.

2. Abort all active transactions (i.e., transactions with pending acknowledgments).

In case of a site failure:

1. Analysis phase: identify committed, aborted and active transactions. Also, determine the largest LSN.

2. For each coordinator, send a *recovering* message containing the largest LSN.

3. Undo the effects of aborted and active transactions.

4. Once the *repair* messages arrive, repair the log, update the list of committed and still-active transactions and re-build the lock table.

5. Complete the redo phase.
    - Redo committed transactions and release their locks.
    - Redo still-active transactions and retain their locks.

6. Resume normal processing.

Figure 11 Recovery in IYV Protocol.

If a participant has already received and enforced a final commit decision prior to the failure, as in the case of a communication failure, the participant simply responds with an acknowledgment. If the participant has not received the decision, it must have been waiting for the decision and once it receives the decision, it writes a non-forced commit record and then sends an ACK message when the decision record is in the stable log.

For those transactions without final decision records (i.e., those transactions that were active prior to the failure or their non-forced abort records did not make it to the stable log before the failure), the coordinator can safely forget them and consider them as aborted transactions (the second case of the coordinator recovery algorithm). If a participant in the execution of one of these transactions has a pending acknowledgment, when it times out due to the coordinator site failure, it will abort the transaction, as in the case of a communication failure that we discussed above. On the other hand, if the participant is left blocked (i.e., the participant has acknowledged all a transaction's operations and is in the implicit prepared to commit state), when the coordinator recovers, the participant will inquire about the status of the transaction. The coordinator, not remembering the transaction after its recovery, will respond with an abort message by presumption. For those transactions that are associated with decision records as well as end records (the third case in the coordinator recovery algorithm), the coordinator can safely discard all information about these transactions, knowing that no participant will inquire about their outcome in the future.

Participant Failure

Also shown in Figure 11 are the steps of the participant recovery after a site failure. Since the entire log might not be written into a stable storage until after the log buffer overflows, the log may not contain all the redo records of the transactions committed by their perspective coordinators after a failure of a participant. Thus, during the *analysis phase* of the restart procedure, the participant determines the largest LSN that is associated with the last record written in its log that survived the failure (the first step in the participant recovery algorithm). Then, the participant sends a *recovering* message that contains the largest LSN to all coordinators in the system (the second step in the recovery algorithm). In the mean time, the participant recovers those aborted and committed transactions that have decision records pertaining to them already stored in its stable log (the third step in the algorithm). That is, while waiting for the reply messages to arrive from the coordinators, the *undo phase* can be performed, even potentially completed, and the *redo phase* can be initiated. This ability of overlapping the undo phase with the resolution of the status of active transactions and the repairing of the redo part of the log, partially masks the effects of dual logging and communication delays. Note that because of the use of write-ahead logging (WAL) (that we discussed in Section 2.2.2), all the required undo log records that are needed to eliminate the propagated effects of any transaction on the database are always available in the participant's stable log and never replicated at the coordinators' sites.

When a coordinator receives a recovering message from a participant, it will know that the participant has failed and is recovering from the failure. Based on this knowledge, the coordinator checks its protocol table to determine each transaction that the participant has executed some of its operations and the transaction is either still active in the system (i.e., still executing at other sites and no decision has been made about its final status, yet) or has committed but did not finish the protocol (i.e., a final decision has been made but the participant has not acknowledged the decision prior to its failure). For each transaction that is finally committed, the coordinator responds with a commit status along with a list of all the transaction's redo records that are stored in its log and have LSNs greater than the one that was included in the recovering message of the participant.

For each active transaction that is still in progress in other sites, the coordinator responds with a *still-active* status containing, as in the case of a committed transaction, a list of the redo records associated with LSNs greater than the one included in the recovering message of the participant. The message also contains all the read locks that were held by the transaction at the participant's site prior to its failure.

All these responses and redo log records are packaged with the read locks acquired by active transactions in a single *repair* message and sent back to the participant. If a coordinator has no active transactions and all committed transactions have been acknowledged as far as the failed participant is concerned, the coordinator sends an ACK repair message, indicating to the participant that there are no transactions to be recovered as far as this coordinator is concerned.

Once the participant has received reply messages from all the coordinators (the third step in the participant recovery algorithm in Figure 11), the participant repairs its log and completes the redo phase. The participant also re-builds its lock table by re-acquiring the update locks during the redo phase in conjunction with the read locks received from the coordinators. Once the redo phase is completed (the fourth step in the participant recovery algorithm), the participant acknowledges all commit decision responses once these commit decisions are in its stable log, as in the case of normal processing. Then the participant resumes its normal processing (the last step in the participant recovery algorithm). Thus, in IYV's recovery algorithm, a long-executing transaction is not necessarily aborted as a result of a participant failure as would be the case in all other ACPs.

Simultaneous Coordinator and Participant Failures

The case of an overlapped coordinator and participant failure is handled using the same procedure as we discussed above. However, in this case, the participant is left blocked and cannot proceed in its recovery procedure until the coordinator has recovered. Even though this is a reasonable trade off considering the expected high reliability characteristics of future database sites, we still need a protocol that can be used with less reliable sites (for example, sites that use old technologies with less reliability). In the next section, we present an IYV protocol variant that we designed for this purpose. This IYV protocol variant is compatible with IYV but it is more costly than than IYV. However, as we show when we evaluate IYV and other protocols, its cost remains below the 2PC variants that require an explicit voting phase.

4.2 The IYV with a Commit Coordinator (IYV-WCC) Protocol

IYV reduces the time required to commit a distributed transaction at the expense of independent recovery of failed participant sites. In this section, we propose a novel coordination scheme for IYV that reduces the window of vulnerability to blocking and minimizes the time required for the sites to become operational after a failure. The new scheme combines the delegation of commitment technique with a timestamp synchronization mechanism. Although this new scheme incurs extra coordination messages and log writes, it enhances the performance of IYV during recovery in the presence of less reliable sites while still maintaining the cost of commit processing during normal processing below that of 2PC and its other well known variants.

By separating the execution of operations from commit processing of transactions, a participant in 2PC is able to save at its local stable storage all the log records pertaining to a prepared to commit transaction using a single force log write. Thus, after a system crash, a participant has all the necessary information in its log to recover independently and resume normal processing while waiting to resolve the status of in-doubt transactions (i.e., transactions with prepare log records but without associated decision records). In IYV, forcing the log every time a transaction enters the prepared to commit state would have been prohibitively expensive. Instead, the redo part of the log of a participant is replicated at the coordinators' sites. Therefore, a recovering participant in IYV needs to communicate with all the coordinators in order to determine which of the active transactions in its site have been committed and which are still in progress as well as their accessed data items. Because of this, a participant cannot independently recover and it has to block any access to its *entire* database until it receives replies from all the coordinators. Thus, in order to deal with unreliable coordinators, it is imperative that all participants become operational in a bounded amount of time, in a similar manner as in 2PC. Towards this end, we develop an IYV variant that utilizes two sites as coordinators and to involve delegation of commitment.

In this new IYV variant, an unreliable coordinator can be responsible for the execution of transactions initiated at its site which is cheaper than using a remote, reliable coordinator. When a transaction finishes its execution, the coordinator at the site where the transaction has been initiated, termed the *execution coordinator* (EC), prepares itself to commit the transaction and delegates the final commit decision to a more reliable site (i.e., a site that fails rarely and if it fails it recovers very quickly), termed the *commit coordinator* (CC). The delegation of commitment is achieved by sending a message from the EC to CC. The delegation message includes the identities of the participants as well as all the redo log records generated during the execution of the transaction with their associated LSNs. In this way, a recovering participant can inquire both coordinators and will be able to finish its recovery process as soon as it receives a reply from either of the two coordinators. Since the participant will receive a response from at least the reliable CC, the participant will be able to recover in a bounded amount of time, allowing new transactions to execute at its site. Thus, reducing the overall cost of recovery in IYV.

4.2.1 Description of IYV–WCC Protocol

As mentioned above, in this IYV variant, each unreliable coordinator is paired with a more reliable commit coordinator that is responsible for the commit processing of the transactions initiated at the unreliable site. All such pairings are known to both EC and CC coordinators and all participants in the system.

As in IYV, when a participant receives an operation pertaining to a transaction from an EC, it includes the redo records generated and the read locks acquired during the execution of the operation in the ACK message. The participant also generates a *timestamp* based on its local clock for the transaction after successfully executing the transaction's first *update* operation and includes the timestamp in the ACK message as well. That is, each transaction is associated with a *begin timestamp vector* (BTV) that contains an entry for each participant that has executed an update operation. As it will become clear below, timestamping is the key for the correctness of this IYV variant because it ensures that a CC will always make consistent decisions about the outcome of transactions delegated to it by any EC and despite failures. If a participant fails to execute an operation, it aborts the transaction and sends a

NACK message. If an EC receives a NACK message in response to an operation request from a participant or an abort primitive from the transaction, the EC aborts the transaction and sends an abort message to each participant (except the one which has sent a NACK message, if any) and forgets the transaction.

Now, let us examine in detail the commitment of a transaction in the presence of delegation of commitment. During the discussion, we will refer to Figure 12 when numbering the actions taken in each step of the commit process. When a transaction finishes its execution at all participating sites successfully and submits its final commit primitive to its EC, the EC prepares itself to commit the transaction by force writing a prepare log record which includes the identities of all participants as well as the timestamp at each update participant (i.e., the BTV). Notice that the forced prepare log record also causes all redo log records received from the participants to be forced into the stable log (action 1 in Figure 12). Then, the EC delegates the commit responsibilities to its associated CC. The delegation action is achieved by sending an *intention to commit* message (action 2). This message contains the identities of the participants, all the redo log records generated during the execution of the transaction with their corresponding LSNs, and the timestamps at the update participants. Thus, the CC is involved in a transaction's commitment process only when the transaction has finished its execution successfully at all participating sites and invoked the commit transaction management primitive.

A CC keeps track of the time of the most recent crash of each participant by maintaining a *failure timestamp list* (FTSL). The FTSL contains the local times of the participant sites and it does not require any global clock synchronization. When a CC receives an intention to commit message pertaining to a transaction, the CC compares the transaction's BTV with the FTSL. Specifically, the CC compares the timestamp that has been assigned to the transaction by each participant to the participant's most recent crash timestamp. If the timestamp of the transaction at a participant is *greater than* the failure timestamp of the participant, it means that the transaction has been initiated after the participant has recovered from its most recent site failure and the participant has executed all the transaction's operations without a failure since its most recent failure. If this is the case for all the entries in

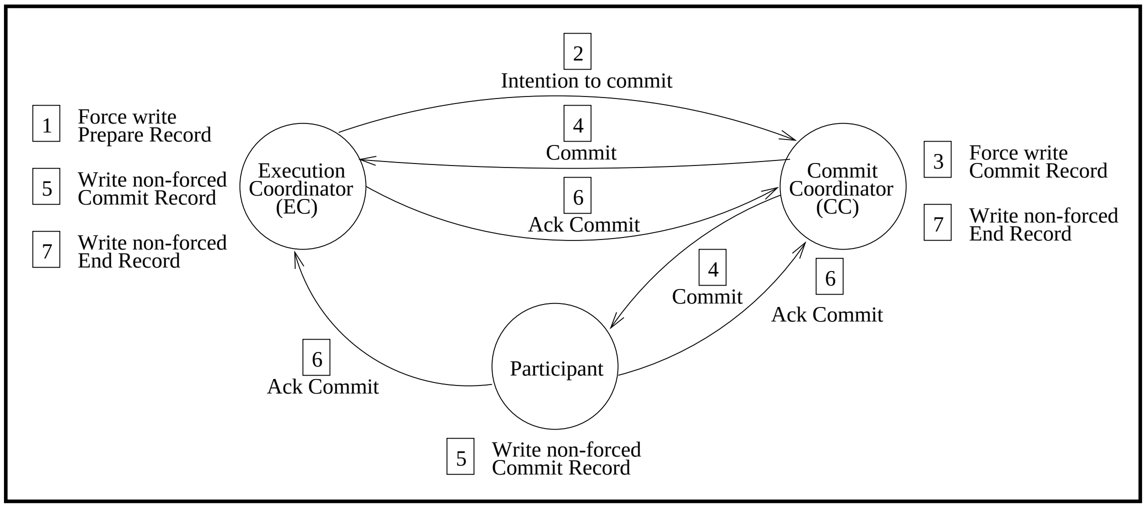

1 Force write Prepare Record
5 Write non-forced Commit Record
7 Write non-forced End Record
2 Intention to commit
4 Commit
6 Ack Commit
Execution Coordinator (EC)
Commit Coordinator (CC)
3 Force write Commit Record
7 Write non-forced End Record
4 Commit
6 Ack Commit
6 Ack Commit
Participant
5 Write non-forced Commit Record

Figure 12 The coordination messages and log writes in IYV-WCC.

the BTV, the CC commits the transaction. That is, since all the participants have executed all the operations of the transaction without any site failure in between the time the transaction has been initiated and the time the CC has received the intention to commit message, the CC commits the transaction. Otherwise, the CC aborts the transaction.

On a commit decision, the CC force writes a commit log record (action 3) and sends a commit message to each participant (including the EC) (action 4). When a participant receives such a message, it writes a non-forced commit log record (action 5) and sends a commit acknowledgment to both of the CC and EC (action 6), only when the commit record has been propagated to the stable log. Similarly, when the EC receives a commit message, it writes a non-forced commit record and sends a commit acknowledgment to the CC. When the CC and EC receive the required acknowledgments, they write non-forced end log records (action 7) and forget the transaction. The commit acknowledgment sent to the EC by the participants has a dual role. First, it tells the EC that the transaction has committed which is necessary in the case that the CC fails after making the decision but before sending it to the EC. Second, it allows the EC to take over during the recovery of the CC after a failure and direct any participant that inquires about the outcome of the transaction to commit the transaction. That is, if any participant has received the commit decision before the CC has failed, the EC will know about the decision and direct any other participant that inquires about the transaction status to commit the transaction.

On an abort decision, on the other hand, the CC sends an abort message to the EC and to each participant where the transaction is still active (i.e., has its failure timestamp less than the transaction's timestamp) and forgets the transaction without writing any log records. There is no need to send abort messages to participants that have failure timestamps greater than the transaction's timestamp because these participants have already aborted and undone the effects of the transaction during their recovery, as we will show in the next section. When a participant receives an abort message from a CC pertaining to a transaction, it aborts the transaction and releases all the resources held by the transaction, without writing any log records or acknowledging the decision.

4.2.2 Recovery in IYV–WCC

In this section, we discuss the recovery aspects of IYV–WCC. IYV–WCC is resilient to both communication and site failures that are detected by timeouts, as it is the case in all other ACPs.

4.2.2.1 Communication Failures. In IYV–WCC, there are five situations where a site is waiting for a message. The first situation is when the EC has forced a prepare log record for a transaction and has sent an intention to commit message to the CC. In this situation, the EC is left blocked. It cannot determine the final status of the transaction until it receives the final decision from the CC or an acknowledgment message from a participant indicating that the transaction has committed. While in the first situation, the EC inquires the CC once it re-establishes communication with the CC if it has not heard from any of the participants. The CC replies with an abort message if it has no recollection about the outcome of the transaction. Otherwise, the CC responds with a commit message that has to be acknowledged by the EC.

The second situation is when the EC is waiting for the acknowledgments from the participants pertaining to a committed transaction. While in this situation, the EC re-submits a commit message to each participant that has not acknowledged the commitment of the transaction. When a participant receives such a message, the participant has either received a similar message from the CC and committed the transaction or it has been left blocked awaiting for the final decision. In the former case, the participant will have no re-collection about the transaction and will respond with an acknowledgment. On the other hand, if the participant is left blocked, it will commit the transaction by writing a non-forced commit log record and will acknowledge the EC once the log record is propagated to the stable log. Once the EC receives the required acknowledgment messages, it completes the protocol by writing a non-forced end log record and forgets the transaction.

The third situation is similar to the second one but with respect to the CC. That is, when the CC has already made its final commit decision and some participants have not acknowledged the decision. This situation is handled in a manner similar to the previous one.

The fourth situation is when a participant has timed out and it does not have any pending acknowledgments (i.e., all operations have been acknowledged). In this case, the participant inquires both the EC and the CC about the status of the transaction. While in this situation, a participant may receive one out of twelve different combinations of responses as shown in Table 3. A *No–response* in the table indicates that a participant did not receive a response from a coordinator during a specified amount of time while a *No–info* indicates that the responding coordinator has no recollection about the transaction at the time it has received the inquiry message from the participant. A *Still–active* response from an EC indicates that the transaction is still active at other sites and no final decision has been made regarding its final outcome. The rest of responses in the table are self explanatory.

In Table 3, response combinations numbered 1 to 4 indicate to the participant that the transaction has been committed. For example, response combination 2 indicates that the transaction has been committed even though no response is received from its

Table 3 Responses to a communication failure.

No.	EC Response	CC Response	Participant Conclusion
1	Commit	No–response	Commit
2	No–response	Commit	Commit
3	Prepared	Commit	Commit
4	Commit	Commit	Commit
5	No–info	No–response	Abort
6	No–info	No–info	Abort
7	No–response	No–response	Wait
8	Prepared	No–response	Wait
9	Still–active	No–response	Wait
10	No–response	No–info	Wait
11	Prepared	No–info	Wait
12	Still–active	No–info	Wait

EC, thereby, a participant is not blocked as it would have been the case in the basic IYV. Response combinations 5 and 6 indicate to the participant that the transaction has been aborted because it is not possible for an EC to forget about a committed transaction without receiving commit acknowledgments from all participants. Since it is not possible for a participant to inquire about the transaction if the transaction has been committed and the participant has already acknowledged the commitment of the transaction, the only remaining possibility is that the transaction has been aborted which is the right conclusion for a participant to make when it receives a No-info from the EC. The rest of the response combinations (i.e., 7 to 12) indicate to the participant that it cannot do anything regarding the transaction except to wait until it receives further instructions form either the EC or CC.

The fifth situation is when a participant times out and it has a pending acknowledgment. That is, the participant realizes that a communication failure has occurred with the EC before acknowledging an operation pertaining to a transaction. In this case, the participant may abort the transaction. Similarly, an EC may abort a transaction if it times out without receiving an operation acknowledgment from a participant.

4.2.2.2 Participant Failure. During its recovery process, a participant sends *recovering* messages to all (EC and CC) coordinators in the system and creates a *non-responding coordinators list* (NRCL). As in IYV, a recovering message contains the LSN associated with the latest record written into the participant's stable log as well as the value of the participant's time clock (i.e., the restart time). While waiting for the reply messages to arrive, the participant starts the undo phase by undoing the effects of each aborted as well as each partially executed transaction, i.e., each transaction without a commit record in its own log. Once the undo phase is completed, the participant starts the redo phase by redoing all transactions that have been committed prior to the failure according to its log. Then, the participant blocks awaiting the reply messages to arrive from the coordinators.

When an EC receives the recovering message, it responds with a message that contains all the redo log records with LSNs greater than the one received in the recovering message for each prepared to commit, committed and still active transaction. For each still active transaction, the EC also includes all the read locks held by the transaction at the participant prior to the failure. The EC also indicates, in its repair message, which transactions are in their prepared to commit states (even though a prepared to commit transaction might not have redo log records with LSNs greater than the one received from the participant), which transactions have been committed and which transactions are still in their active state. Then, the EC waits for an acknowledgment message from the participant before it can submit any further operations for execution to the recovering participant. If the recovering participant has not participated in any of the prepared to commit, committed or active transactions at the EC, the EC acknowledges the recovering message of the participant, implying to the participant that there is *nothing for you* to consider during recovery. This message indicates to the participant that it has recorded in its log and acknowledged the commitment of all transactions initiated at the EC prior to the failure.

Table 4 Responses to a site failure.

No.	EC Response	CC Response	Participant Conclusion
1	No–response	No–response	Block
2	Commit	No–response	Redo and commit
3	No–response	Commit	Redo and commit
4	Prepared	Commit	Redo and commit
5	Commit	Commit	Redo and commit
6	Prepared	Nothing for you	Abort
7	Nothing for you	No–response	No recovery actions needed
8	No–response	Nothing for you	No recovery actions needed
9	Nothing for you	Nothing for you	No recovery actions needed
10	Prepared	No–response	Redo and wait
11	Still–active	No–response	Redo and wait
12	Still active	Nothing for you	Redo and wait

Similarly, when a CC receives a recovering message from a recovering participant, it responds with a message that contains all the redo log records of committed transactions that have LSNs greater than the one received from the participant. The CC also indicates which transaction have been committed and that the participant did not acknowledge its commitment prior to the failure. This is also the case even if the transaction did not have any redo log records associated with LSNs greater than the one contained in the recovering message. If each committed transaction that the participant has participated in its execution has been acknowledged prior to the failure, the CC acknowledges the recovering message that it has received from the participant. The CC also updates its FTSL to reflect the time at which the participant has restarted and force writes the list into the stable storage prior to acknowledging the recovering message that it has received from the participant.

As shown in Table 4, there are twelve possible response combinations that a recovering participant may receive from each pair of EC-CC coordinators. We represent these response combinations, in the table, on a per transaction basis for ease of exposition even though a reply to a recovering message from a coordinator may include status information that pertain to more than one transaction. As the table shows, there is only a single response combination out of the twelve possible combinations where a recovering participant is blocked (i.e., the first row in the table). Thus, unlike the case in IYV, a recovering participant does not block in the case of a single coordinator failure.

When a recovering participant receives a message in response to its recovering message from a coordinator, the participant extracts the control information contained in the message and updates its log and lock table accordingly. For example, response combination number 2 indicates to the participant that the transaction has been committed. Therefore, the participant uses the redo log records contained in the response of the EC to repair its log if there are any missing log records for the committed transaction from its log. Similarly, response combination number 11 allows the participant to repair its log as well as its lock table to reflect the read locks that were held by the transaction prior to the failure and forward recover the transaction.

When the participant receives a reply message from a coordinator, it removes the coordinator from its NRCL. If the coordinator is an EC, the participant sends back an acknowledgment message indicating to the EC that it has responded in a timely manner before the redo phase has finished. Therefore, the EC updates the BTV of each still active transaction to reflect the restart timestamp of the participant so that the transaction will not subsequently get aborted by the CC because its begin timestamp is less than the participant most recent failure timestamp. That is, since each CC will update its FTSL to reflect the failure of the participant, an active transaction will be aborted by the CC if its timestamp is not updated by its EC. Therefore, an EC updates the timestamp of each active transaction at its site once it receives an acknowledgment from the recovering participant. The recovering participant finishes its recovery process only when it receives a reply from either the EC or the CC for each pair of EC–CC coordinators. This is the *minimum* number of replies that allow a recovering participant to finish its recovery process. Otherwise, the participant suspends its recovery process.

Once the minimum reply messages arrive, the participant continues its redo phase and re-builds its lock table. Once the redo phase is finished, the participant resumes its normal processing. During normal processing, if the participant receives an operation message from an EC or a commit message from a CC that are still in its NRCL, the participant declines to execute the operation and sends back a *decline* message that contains the most recent participant failure timestamp. The decline message is interpreted by the coordinator to mean that it has not responded in a timely fashion to the most recent participant site failure. If the message is received by an EC, the EC ignores the timestamp contained in the message and aborts all active transactions that have performed update operations at the participant's site. This is because the participant has already recovered and aborted each such transaction based on a nothing for you reply message from the associated CC. For each transaction that has performed only read operations at the participant's site, the EC aborts the transaction only if it attempts to access (i.e., read or write) any data object at the participant. That is, a transaction that has performed only read operations is not aborted if it does not send any operation to be performed at the participant's site after the participant has recovered. This is because it does not matter whether the read–only transaction is committed or aborted at the participant's site as long as it preserves serializability. Since the transaction was serializable prior to the participant's failure (using S2PL), it can be committed as long as it does not submit a new operation for execution that might consequently violate serializability. Once the EC has acted upon the decline message, it includes an acknowledgment flag in the next operation it sends to the participant. When the participant receives an operation with an acknowledgment flag, it removes the EC from its NRCL and executes the operation, knowing that the EC has complied with its decline message.

If on the other hand, the decline message is received by a CC, the CC updates its FTSL and force writes the list into stable storage. Then, the CC re–sends any commit operation that has been declined by the participant including, as in the case of an EC, an acknowledgment flag that indicates to the participant that the coordinator has updated its FTSL and it can be removed from the participant's NRCL.

4.2.2.3 Coordinator Failure. During its recovery after a site failure, an EC aborts each transaction without a prepare log record and forgets the transaction. On the other hand, for each transaction with a prepare log record, the EC inquires its associated CC. If the CC has decided to commit the transaction, the CC responds with a commit message and waits for the acknowledgment of the EC. If the CC does not remember the transaction, it presumes that the transaction has been aborted and tells the EC by sending back an abort message. Recall that IYV is based on the presumed abort protocol.

In the event of a CC site failure, the CC re-builds its FTSL and protocol table during its recovery using its own log. After adding each transaction with a commit record but without a corresponding end log record into the protocol table, the CC sends a commit message to each participant in the execution of each transaction including the transaction's EC. Once the required acknowledgments arrives, the CC writes a non-forced end record and forgets the transaction.

In the next section, we discuss the correctness of the assumptions behind the design of IYV and its IYV-WCC variant and explain, further, the need for propagating the read locks held at a participant site by a transaction to the transaction's coordinator.

4.3 Correctness of IYV Protocol Assumptions

The essence of 2PC that ensures the atomicity of a distributed transaction is that it prevents a transaction from unilaterally committing or aborting at a site while it is in the prepared to commit state. A participant may be required to abort a transaction either for correctness reasons, such as ensuring serializability, or for performance reasons, such as minimizing transaction blocking. Regarding the latter, given that transactions are finite, we assume that a participant does not abort a transaction because it has not received an operation from the transaction for some time. It is the responsibility of the coordinator to decide whether or not it is necessary to abort a long-executing transaction.

As mentioned in Chapter 2, most commercial database management systems use strict two-phase locking (S2PL) for concurrency control and physical write-ahead logging (WAL) for recovery. Now, consider a distributed system in which all the sites employ S2PL. In such a distributed system, participants *never* abort transactions to ensure atomicity and *only* abort transactions in active state, (transactions having outstanding operation acknowledgments,) to resolve deadlocks.

Theorem 1: If each participant employs S2PL for concurrency control, it is not possible for a transaction to be involved in a non-serializable execution, a local deadlock at a participant, or a global deadlock when all the operations that were submitted by the transaction to the participants have been executed and acknowledged.

Proof: The proof proceeds by contradiction. Assume that all the operations submitted by a transaction have been executed and acknowledged and the transaction is involved in (1) a non-serializable execution or (2) a deadlock.

According to the S2PL rules, an operation submitted by a transaction is executed only after the locks required for the execution of the operation on the data items are acquired. This rule implies that an operation is not acknowledged until after the required locks for the execution of the operation are acquired.

The first part (1) contradicts the fact that S2PL schedulers produce serializable histories (5). If the transaction is involved in a non-serializable execution, at least one of its operations would have been blocked rather being acknowledged which contradicts the assumption that all the transaction's operations have been executed and acknowledged.

The second part (2) contradicts the fact that if a transaction is involved in a deadlock, at least one of its operations is blocked awaiting to hold some locks on some data items which again contradicts the assumption that all the operations pertaining to the transaction have been acknowledged. □

Corollary 1: A local deadlock at a participant site that employs S2PL involves only active transactions (i.e., transactions with pending operations).

Proof: As above, the proof proceeds by contradiction. For a deadlock to occur, the *hold-and-wait* condition must exist. If we assume that a transaction is involved in a local deadlock at a participant site after all the operations submitted to the participant have been executed and acknowledged, it means that the transaction is holding locks on some data items and is waiting to hold locks on other data items. However, since all the operations of the transaction have been executed and acknowledged by the participant, all the locks required for the execution of the operations submitted to the participant have been acquired. Hence, the hold-and-wait condition cannot exist after all the operations submitted by a transaction to a participant have been acknowledged and a local deadlock can only involve active transactions. □

Note that participants using an *optimistic* concurrency control protocol (which we mentioned in Chapter 2) do not exhibit the above property. That is, they might abort a transaction even though the transaction is not in an active state in order to ensure serializability (5). Hence, IYV and its IYV-WCC variant are not applicable in this case. On the other hand, it can be shown, as in Theorem 1, that participants using a *pessimistic* concurrency control protocol (other than S2PL) that avoids cascading aborts *never* abort transactions to ensure atomicity and *only* abort transactions in active state to ensure serializability or to resolve deadlocks. However, in this dissertation, we are considering only S2PL combined with physical WAL because of the wide acceptance of this combination.

As discussed above, recovery in IYV is based on the traditional Undo/Redo schemes in which the undo phase precedes the redo phase because it allows the analysis phase at a participant that involves communication with the coordinators to proceed concurrently with the undo phase and potentially part of the redo phase. This is not possible in the case of recovery schemes such as ARIES (28), in which the redo phase precedes the undo phase. In the case of a recovery scheme where the redo phase precedes the undo phase, a participant is blocked and cannot initiate recovery until it receives responses from the required coordinators (i.e., all coordinators in the system in the case of IYV). That is, IYV and IYV-WCC can incorporate such a recovery scheme but it will not offer the same efficiency during recovery in this case.

Now, let us also make the need for replicating the read locks that are held by the transactions at the coordinators' sites clear, a need which allows the support of forward recovery without violating consistency. Assume that we have two transactions T_1 and T_2, submitted at two different coordinators. T_1 reads data item x, writes data item y and then commits, whereas T_2 writes both data items x and y, and then commits. Here, $r_i[x]$ ($w_i[x]$) denotes a read (write) operation performed by transaction T_i on item x and c_i denotes the commit primitive of T_i.

$$T_1\text{: } r_1[x]\ w_1[y]\ c_1$$

$$T_2\text{: } w_2[x]\ w_2[y]\ c_2$$

Furthermore, assume that the first operation of T_1, $r_1[x]$, has been executed successfully and acknowledged. After that, the participant where the data items are stored fails. The resulting *history* of execution is as follows:

$$H_1\text{: } r_1[x]\ Crash$$

At this point, assume that the participant has received acknowledgment messages in response to its recovering messages from all coordinators including the coordinator of T_1 which has indicated that T_1 is still active in its acknowledgment message. Based on these replies, the participant finishes its recovery procedure using its own log and knowing that T_1 is still in progress but there is no redo actions that are needed to be used with this transaction since the transaction has performed only a read operation. Now, if we allow T_1 to forward recover after the participant has recovered without being able to reconstruct the exact lock table of the participant as it was before the failure, we might end up with the following non-serializable history.

$$H_2\text{: } r_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ c_1$$

H_2 is not serializable because it is cyclic (i.e., $T_1 \rightarrow T_2 \rightarrow T_1$) (as we discussed in Chapter 2). H_2 is possible because the participant after loosing its lock table, could not re-acquire the read lock on x on behave of T_1 as it was the case prior to the failure, allowing T_2 to acquire a write lock and change x after the participant has recovered. Once T_2 has committed, the participant receives a new operation pertaining to T_1 that also modifies the value of x. Since T_2 has already committed and all its locks has been released, T_1 can acquire a write lock on x and modifies it, resulting in a non-serializable execution. However, duplicating the read locks at the coordinators in IYV and its IYV-WCC variant allows a participant to re-build its lock table after a failure and to prevent execution histories similar to the one described above from occurring.

Here, it should be pointed out that a transaction always executes at the site of its coordinator. Thus, forward recovery is only possible in the case of participants' failures. A transaction cannot be forward recovered in the case of the (execution) coordinator's site failure because the state of the transaction (i.e., the program state which includes all local variables) is lost and cannot be restored, whereas the state and control information of the database at a participant can be restored with the help of the coordinator.

4.4 Comparison between IYV Protocol and the other Atomic Commit Protocols

IYV combines the advantages of UV, EP, and CL protocols (that we discussed in Chapter 3). Here, we compare IYV with these protocols and in particular with CL which shares the same basic idea with IYV in order to eliminate the voting phase of 2PC and to reduce the number of log force writes.

To avoid force writing the log records that are generated during the execution of each and every operation prior to acknowledging them, UV assumes that each site knows when it has executed the last operation on behalf of a transaction [35]. As we discussed in Section 3.2.5, this means that the coordinator either submits to a site all the operations at the same time (which is a form of predeclaration) or indicates to the participant the last operation at the time that the operation is submitted. The former is possible in very special cases. The latter is only possible if each transaction has knowledge about the data distribution in the system and indicates to the coordinator the last operation to be executed at a participant.

In contrast to UV, IYV does not make any assumption about the structure of transactions and does not assume any knowledge by the transactions about the data distribution. Thus, IYV is more general compared to UV. In the special cases in which UV is applicable, IYV and UV would exhibit similar behavior during normal processing.

In EP which is derived from PrC, the number of forced log writes pertaining to a transaction is equal to the number of the participants that executed the transaction since EP requires the identities of the participants to be explicitly recorded at the coordinator's log in a forced initiation log record. This is because an initiation log record has to be forced written each time a new participant is about to execute an operation of the transaction. In contrast, in IYV, a coordinator does not force write any initiation record.

To alleviate the drawback of the initiation records of EP, CL uses distributed write-ahead logging (DWAL) (as we discussed in Section 3.2.5). In the case of CL, since a participant might inquire a coordinator about the latest forced log write (i.e., to ensure the DWAL), CL might become very costly and less efficient when compared with any of the 2PC variants. Consider the case when a number of long-living transactions execute at a participant without excessive main memory. In this case, the participant might request a transaction's coordinator (explicitly) to force write its log more than once resulting in a great number of sequential messages. Also, rolling back aborted transactions has to be performed completely over the network. This means that when a participant aborts a transaction, it cannot release the resources held by the transaction until it communicates with the transaction's coordinator and receives the undo log records pertaining to the transaction, which is a significant overhead.

Another problem with CL is that the log records of transactions cannot be garbage collected by the coordinators and have to be remembered forever. This situation is similar to NPrC which also eliminates the initiation records that are required by PrC. However, a novel global garbage collection procedure is combined with NPrC to alleviate this drawback. In CL, garbage collection is given up for committed as well as aborted transactions even though abort decisions are acknowledged by the participants. (In this case, there is no actual benefit from the acknowledgment messages except that they contain the undo log records of aborted transactions. Notice that the underlying recovery scheme, in CL, is ARIES in which an undo is an operation that has to be performed logically and which will generate a *compensation log record* (CLR) that needs to be logged, too.)

Another significant difference between IYV and CL is the case of a coordinator's recovery after a failure. A coordinator in IYV can recover independently without communicating with any participant. In contrast, a recovering coordinator, in CL, has to communicate with all possible participants in the system in order to determine the set of unknown transactions in order to abort them instead of presuming their commitment since CL is a descendant from PrC protocol (36, 9). Furthermore, a recovering participant in CL has to wait until it receives all the log records from the coordinators and until all active transactions have been decided upon. In IYV, however, using the "still active" message, a participant can recover its state up to the point prior to its failure and resume its normal processing without having to wait until all active transactions have been decided upon, allowing long-living transactions to forward recover and resume their execution. Aborted transactions in IYV are handled locally by a participant without any communication with the coordinators (i.e., the undo log records do not have to be requested from the coordinators).

Even though IYV requires that the redo log records generated during the execution of a transaction's operation be logged both at its coordinator as well as at the participant that executed the operation, such a duplicate logging should incur negligible overhead because the log records are written in a non-forced manner and without involving any extra coordination messages. The only overhead is that IYV requires more buffer space for the log of the coordinator so that logging do not cause frequent flushing to the log buffer. As mentioned earlier, we believe that, in general, the overhead associated with the duplication of logs and the extra information contained in the commit and still-active messages is well offset by the reduction in the number of sequential coordination messages and the gain of being able to support forward recovery of interrupted, possibly long-lived, transactions due to participant and communication failures.

It should be pointed out that not forcing commit records at the participants in IYV differs from the group commit optimization in two ways. First, there is no notion of a group or a timer that determines when a force should take place. Second, group commit trades off performance during normal processing for increased blocking after a failure whereas in IYV the blocking of a site is the same irrespective of whether commit records are forced or not. This is because, in IYV, a participant cannot determine all transactions that were active at its site prior to a failure or their final status without contacting all coordinators in the system.

4.5 Analytical Evaluation

In this section, we use the same traditional analytical method that we used in Section 3.2.8.1 to evaluate the performance of IYV and IYV-WCC. We also compare their performance with the performance of 2PC, PrA, PrC, EP and CL, that we discussed in Chapter 3. This method, as we mentioned earlier, is based on evaluating the log, message and time complexities. However, we will tabulate the performance results differently in order to reflect the *sequentiality* of the overhead associated with the three performance metrics on the performance of the different protocols.

In our evaluation, we use best (ideal) and worst case scenarios (36, 9) to highlight the performance differences among the various ACPs. Also, we consider the number of coordination messages and forced log writes that are due to the protocols only (e.g., we do not consider the number of messages that are due to the operations and their acknowledgments). The cost of the protocols in both scenarios are evaluated during normal processing and in absence of failures.

Figure 13 graphically illustrates the sequence of coordination messages and forced log writes involved in 2PC, IYV and IYV–WCC to reach a decision point and to release the resources held at the participants for the commit as well as the abort case. The figure shows how we will evaluate the performance of the ACPs that we listed above, considering the sequential effects of coordination messages and forced log writes.

Tables 5 and 6 compare the different protocols under the worst case scenario. It should be pointed out that this "worst" case scenario is very close to the expected average behavior of transactions and the distributed environment. This is because the assumptions that we make in this case are more realistic than the assumptions that we make in the best case scenario. We denote by n the number of participants that executed a transaction and by d the number of data items that have been accessed by the transaction. In this scenario, we assume the following:

- A transaction has more than one write operation at each participant it accesses (i.e., $d > n$).
- Transactions execute serially (e.g., an operation is submitted by a transaction only when the previous operation has been executed and acknowledged).
- The participants are not known at the beginning of transactions.
- The participants in a transaction execution do not have excessive main memories and each operation generates a single log record. This means that each and every log record that is generated due to the execution of an operation has to be forced written into the stable log as a worst case scenario. Note that we do not include the number of forced log writes that are due to the operations and which are the same in all the protocols except for EP where the log records have to be forced written all the time. In CL, on the other hand, operations

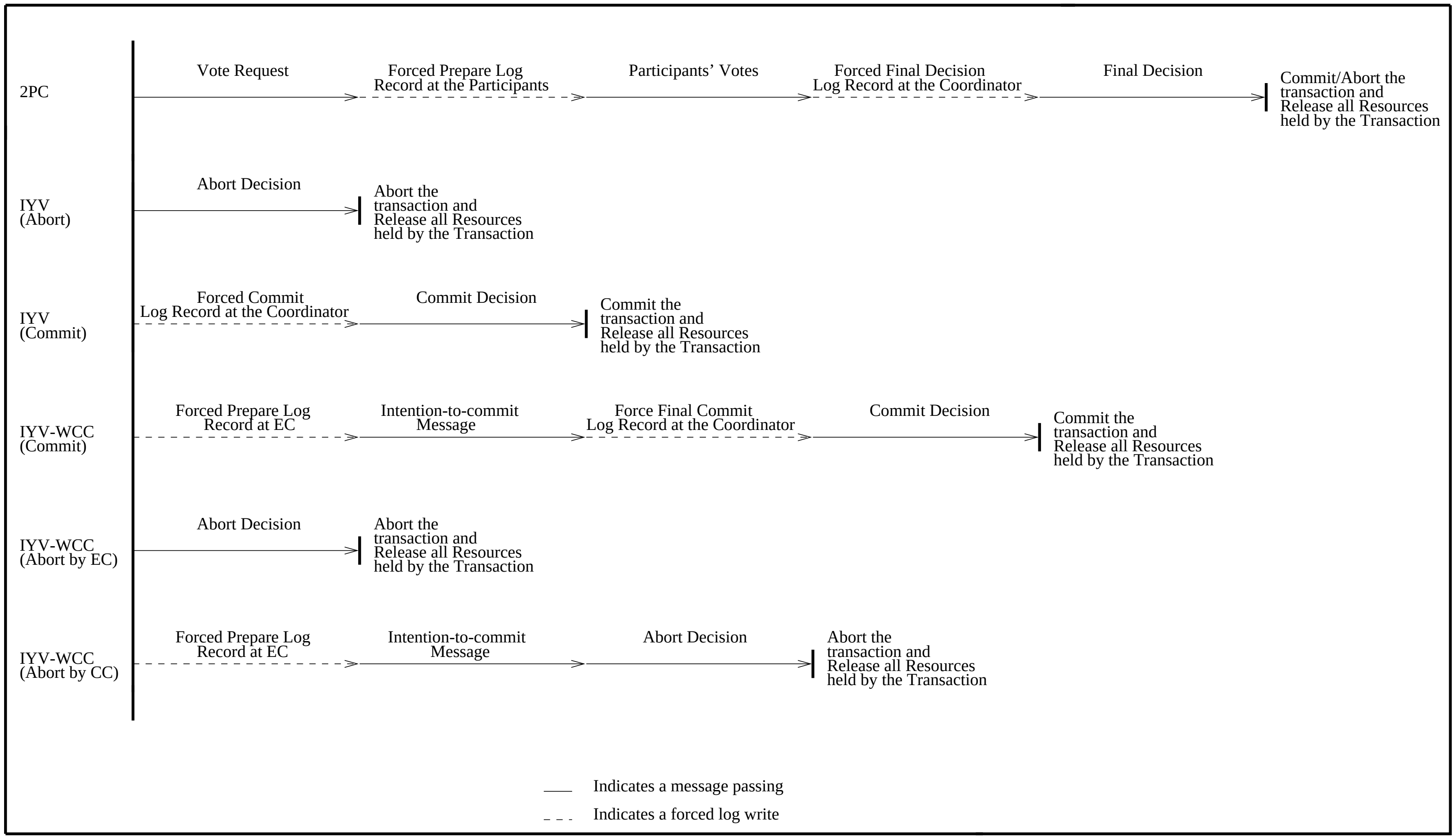

2PC
Vote Request
Forced Prepare Log Record at the Participants
Participants' Votes
Forced Final Decision Log Record at the Coordinator
Final Decision
Commit/Abort the transaction and Release all Resources held by the Transaction
IYV (Abort)
Abort Decision
Abort the transaction and Release all Resources held by the Transaction
IYV (Commit)
Forced Commit Log Record at the Coordinator
Commit Decision
Commit the transaction and Release all Resources held by the Transaction
IYV-WCC (Commit)
Forced Prepare Log Record at EC
Intention-to-commit Message
Force Final Commit Log Record at the Coordinator
Commit Decision
Commit the transaction and Release all Resources held by the Transaction
IYV-WCC (Abort by EC)
Abort Decision
Abort the transaction and Release all Resources held by the Transaction
IYV-WCC (Abort by CC)
Forced Prepare Log Record at EC
Intention-to-commit Message
Abort Decision
Abort the transaction and Release all Resources held by the Transaction
Indicates a message passing
Indicates a forced log write

Figure 13 The sequence of coordination messages and forced log writes required during normal processing.

Table 5 Committing a transaction assuming the worst case scenario.

	2PC	PrC	PrA	EP	CL	IYV	IYV–WCC
Log force delays	2	3	2	d+n+1	1	1	2
Total log force writes	2n+1	n+2	2n+1	d+n+1	1	1	2
DWAL Message delays	0	0	0	0	2d	0	0
Message delays (Commit)	2	2	2	0	2d	0	1
Message delays (Locks)	3	3	3	1	2d+1	1	2
Total messages	4n	3n	4n	n	2d+n	2n	3n+3
Total messages with piggybacking	3n	3n	3n	n	2d+n	n	n+2

Table 6 Aborting a transaction assuming the worst case scenario.

	2PC	PrC	PrA	EP	CL	IYV	IYV–WCC	
							EC Abort	CC Abort
Log force delays	2	2	1	d+n	0	0	0	1
Total log force writes	2n+1	2n+1	n	d+2n	0	0	0	1
DWAL Message delays	0	0	0	0	4d	0	0	0
Message delays (Abort)	2	2	2	0	2d	0	0	1
Message delays (Locks)	3	3	3	1	4d+1	1	1	2
Total messages	4n	4n	3n	2n	4d+n	2n	n	n+2
Total messages with piggybacking	3n	3n	3n	n	4d+n	n	n	n+2

add extra overhead because each force write is explicitly associated with two messages due to the distributed write ahead logging (DWAL).

The rows labeled "Log force delays" contain the sequence of forced log writes that are required by the different protocols up to the point that the commit/abort decision is made. The rows labeled "Message delays (commit/abort)" contain the number of sequential messages up to the commit/abort point, and the rows labeled "Message delays (Locks)" contain the number of sequential messages that are involved in order to release all the locks held by a committing/aborting transaction. For example, in Table 5 , the "Log force delays" for the 2PC protocol is two because there are two force log writes between the beginning of the protocol and the time a commit decision is made by a transaction's coordinator, as shown in Figure 13. Also, "Message delays (Commit)" and "Message delays (Locks)" are 2 and 3 respectively, because the 2PC involves two sequential messages in order for a coordinator to make its final decision regarding a transaction (i.e., the first phase of the protocol), and three sequential messages to release all the resources (i.e., locks) held by the transaction at the participants. In the row labeled "Total message with piggybacking", we apply *piggybacking* of the acknowledgments of the decision messages, which is a special case of the lazy commit optimization that we discussed in Section 3.2.7, to eliminate the final round of messages.

It is clear from Tables 5 and 6, that IYV and CL outperform all other 2PC variants with respect to the number of log force delays to reach a decision as well as the total number of log force writes. For the commit case, the two protocols require only one log force write whereas for the abort case neither IYV nor CL force write any log records. In this respect, EP is the most expensive of all protocols while IYV-WCC has the same log force delays complexity as 2PC and PrA but less by an order of n in the total log force complexity compared to 2PC, PrA and PrC.

CL becomes more expensive than IYV and IYV-WCC when message delays and total number of messages are considered. Due to DWAL, CL requires two explicit sequential messages to be exchanged between a participant and the coordinator of a transaction for each operation executed by the participant for the commit case (thus, the 2d in "DWAL Message delays"). For the abort case, four messages are needed to be exchanged between the participant and the coordinator of an aborted transaction. This is because undoing an operation using the recovery scheme of CL, ARIES, is another operation that has to be executed and logged. Since CL uses a DWAL logging protocol, undoing an operation requires two more explicit messages to be exchanged between the coordinator and the participant in the worst case scenario. Note that because of its dependency on the number of data operations, CL can potentially involve more messages to commit or abort a transaction than any of the 2PC variants in the case of long-transactions. On the other hand, with respect to messages, IYV and EP perform better in all aspects than any other protocol. IYV-WCC comes in the second place. For the commit case, EP incurs the least number

Table 7 Committing a transaction assuming the ideal case scenario.

	2PC	PrC	PrA	EP	CL	IYV	IYV-WCC
Log force delays	2	3	2	3	1	1	2
Total log force writes	2n+1	n+2	2n+1	n+2	1	1	n+2
Message delays (Commit)	2	2	2	0	0	0	1
Message delays (Locks)	3	3	3	1	1	1	2
Total messages	4n	3n	4n	n	n	2n	3n+3
Total messages with piggybacking	3n	3n	3n	n	n	n	n+2

Table 8 Aborting a transaction assuming the ideal case scenario.

	2PC	PrC	PrA	EP	CL	IYV	IYV–WCC	
							EC Abort	CC Abort
Log force delays	2	2	1	2	0	0	0	1
Total log force writes	2n+1	2n+1	n	2n+1	0	0	0	1
Message delays (Abort)	2	2	2	0	0	0	0	1
Message delays (Locks)	3	3	3	1	1	1	1	2
Total messages	4n	4n	3n	2n	2n	2n	n	n+2
Total messages with piggybacking	3n	3n	3n	n	2n	n	n	n+2

of total messages. This situation changes when piggybacking is considered.

Piggybacking can be used to eliminate the final round of messages for the commit case in 2PC, PrA, IYV, and IYV-WCC. That is not the case for PrC, EP and CL because a commit final decision is never acknowledged in these protocols. Similarly, this optimization can be used in the abort case with 2PC, PrC, and EP but not with PrA, CL, IYV or IYV-WCC. In PrA, IYV and IYV-WCC, an abort decision is never acknowledged while in CL, the acknowledgment is sent immediately because it contains the undo log records of the aborted transaction.

Table 7 and Table 8, compare the number of messages and forced log writes that are needed to commit and abort a transaction, respectively, for the different protocols based on the ideal case scenario. In this ideal scenario, we make the following assumptions:

- A transaction performs at most one write operation per each site it accesses (i.e., $n = d$).
- The operations of a transaction execute in parallel.
- The participants are known at the beginning of transactions.
- The participants have limited main memory which causes each log record to be forced to stable log.

In the ideal case, CL and IYV have the same cost with respect to the number of sequential force log writes and messages for the commit case, dominating the other ACPs with CL dominating IYV with respect to the total number of messages. When piggybacking is considered, CL and IYV variants have exactly the same cost. For the abort case, both CL and IYV have the same costs with respect to the number of sequential force writes and the total number of forced log writes. Similarly, they have the same cost considering the number of sequential and total number of messages.

Comparing the two scenarios, we note that the cost associated with EP is highly dependent on the number of operations submitted by the transactions while CL is also dependent on the characteristics of the distributed database system (e.g., the propagation latency of the communication network, the log buffer size, and the main memory size participants, etc.) All of these factors are due to CL's DWAL. This makes EP and CL completely inefficient in distributed database systems with relatively long-living transactions where a transaction executes a large number of operations at a small number of sites (i.e., $d >> n$), a situation common in advanced distributed database applications.

With respect to IYV-WCC, Tables 5 and 7 show that, for the commit case, IYV–WCC has increased both of the number of *sequential* forced log writes and coordination messages by one, to reach a commit decision and to release the locks held by a committing transaction, when compared with IYV. The cost of sequential forced log writes remains the same as in 2PC and PrA, but one less than PrC. The cost of sequential coordination messages to reach a commit point and release the locks held by a committing transaction is less by one when compared with 2PC, PrA and

PrC. In the case of IYV–WCC, there are $3n+3$ total messages. This is because there are n commit messages, one for each participant, and $2n$ acknowledgment messages, two from each participant. The other 3 messages are the intention to commit, the commit final decision sent to the EC and the EC's acknowledgment.

For the abort case, Tables 6 and 8, the cost to abort a transaction in IYV–WCC remains the same as in IYV when the abort decision is made by the EC. On the other hand, the cost to abort a transaction by the CC incurs an extra sequential coordination message to reach an abort decision as well as to release the locks held by the aborting transaction. Notice that if a transaction is to be aborted, it will be aborted by its EC rather than the CC in the absence of failures. Hence, the cost of aborting transactions, in the absence of failures, remains the same as in IYV, the best alternative.

Figure 14 compares the cost associated with 2PC, IYV and IYV–WCC during the recovery process of a participant. After the analysis and the undo phase of the recovery procedure, both IYV and IYV–WCC incur a bounded delay in the case that all coordinators respond to the recovering inquiry messages during the recovery process of a participant. This delay increases the time required for the recovery procedure in IYV and IYV-WCC when compared with the 2PC, which does not require any coordinator's response for a participant to become operational after a failure. However, the extra recovery time required in IYV and IYV–WCC will be, at least partially, offset by the ability of the two protocols to forward recover any still active transaction at the coordinators. In the case that a participant encounters a failed coordinator during its recovery process in IYV, the participant will suspend its recovery for an unbounded amount of time until the coordinator has recovered. In IYV-WCC, on the other hand, a participant will be able to recover and become operational again within a bounded delay as long as one of the coordinators in each EC-CC pair is still operational during the recovery process of the participant.

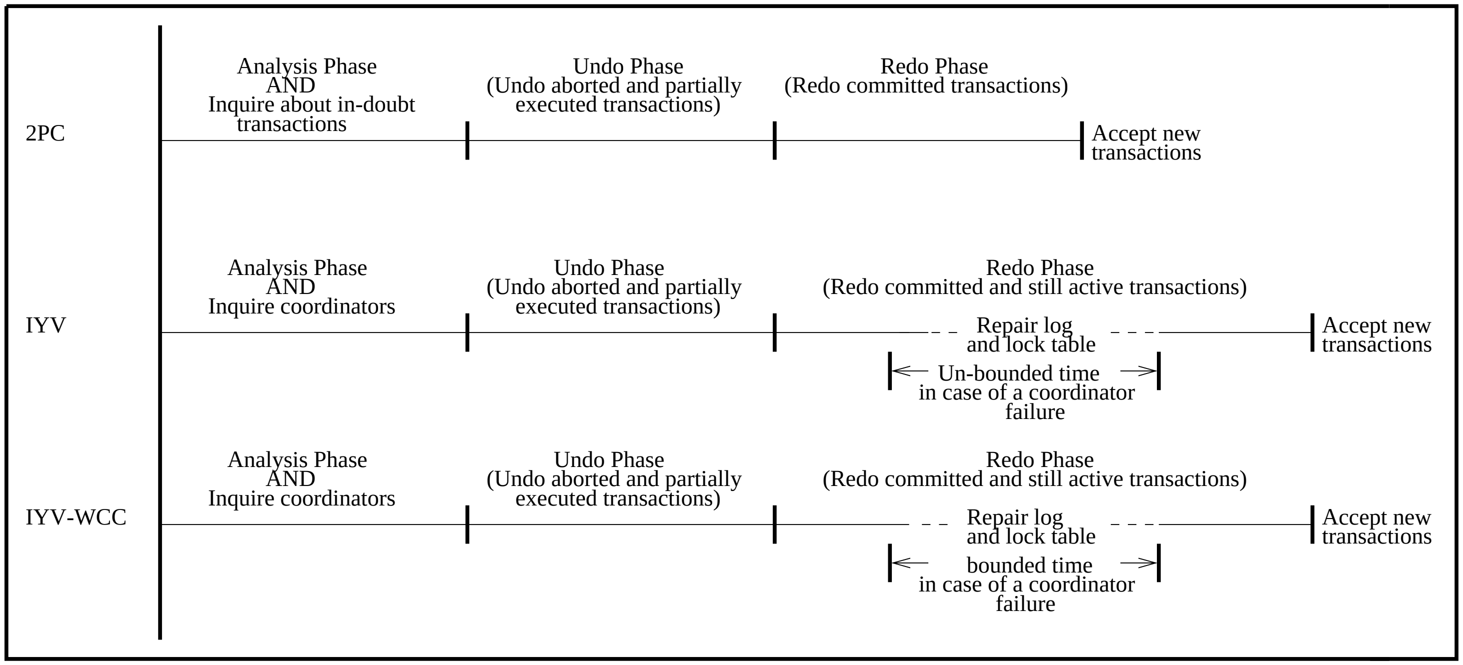

2PC
Analysis Phase AND Inquire about in-doubt transactions
Undo Phase (Undo aborted and partially executed transactions)
Redo Phase (Redo committed transactions)
Accept new transactions
IYV
Analysis Phase AND Inquire coordinators
Undo Phase (Undo aborted and partially executed transactions)
Redo Phase (Redo committed and still active transactions)
Repair log and lock table
Un-bounded time in case of a coordinator failure
Accept new transactions
IYV-WCC
Analysis Phase AND Inquire coordinators
Undo Phase (Undo aborted and partially executed transactions)
Redo Phase (Redo committed and still active transactions)
Repair log and lock table
bounded time in case of a coordinator failure
Accept new transactions

Figure 14 The amount of time required to become operational after a failure.

4.6 Summary

In this chapter, we presented *implicit yes-vote* protocol (IYV), a new ACP for future gigabit-networked distributed databases. IYV exploits the semantics of (1) strict two-phase locking of the database management mechanisms and (2) the characteristics of gigabit-networks to enhance performance over the other well known ACPs. IYV reduces the cost of commit processing during normal operation and, after a participant failure, it allows partially executed transactions at the failed participant that are still active in the system to resume their execution after the participant has recovered, a situation that is not possible in any other ACP.

To reduce the blocking aspects of IYV in the presence of less reliable sites, we also developed the *implicit yes-vote with a commit coordinator* (IYV-WCC) atomic commit protocol. As in IYV, IYV-WCC allows forward recovery of transactions while at the same time it enhances independent recovery compared with IYV at the expense of extra coordination messages and forced log writes. This makes it suitable in the presence of less reliable database sites. The performance enhancement in both protocols is achieved through a low-cost partial replication of each participant's log and lock table at the coordinators' sites.

To highlight the performance enhancement of our new protocols, we compared the performance of IYV and IYV-WCC to the performance of other known protocols based on the traditional analytical method. This method is based on evaluating the performance of the different protocols under worst and ideal case scenarios. Our evaluations reveals that the performance of our proposed protocols is very promising. In the next chapter, we evaluate the performance of the different protocols based on a simulation model to better establish both the relative and absolute performance enhancements of our IYV protocol compared to the other well known protocols.

Even though IYV seems to be a very promising protocol with respect to performance, it should be noted that it is not always applicable. First, it cannot be applied in systems that supports *deferred consistency constraints validation* (e.g., SQL triggers [(6)]) because it eliminates the (explicit) voting phase of 2PC. However, in these systems, some integrity constraints are validated at commit time of transactions which require from a coordinator to explicitly request the participants in a transaction's execution to prepare to commit the transaction by validating any deferred integrity constraints. Second, IYV cannot be used in resource-constrained systems (e.g., low bandwidth networks, small main memory or high cost disk access systems). This motivates us to search for another alternative ACP that can be used when IYV is not applicable and to revisit PrA and PrC protocols. Before we present the results of our investigations in this direction, which is the topic of Chapter 6, let us first evaluate the performance of IYV, CL, PrA and PrC based on simulation, which constitutes the second contribution of this dissertation and the topic of the next chapter.

5.0 PERFORMANCE OF ATOMIC COMMIT PROTOCOLS IN GIGABIT-NETWORKED DATABASE SYSTEMS

Although the traditional method of performance evaluation has been useful in analyzing the best and worst case scenarios as we showed in Chapter 4, it can neither be used to completely characterize the respective efficiency of the implicit yes-vote (IYV) and coordinator log (CL), nor provide a basis to compare them with other atomic commit protocols. This is because the performance impact of these protocols on a system's performance depends on many factors including the propagation latency of the communication network, the log buffer size and the percentage of the flushing of data from main memory to the stable storage. Furthermore, the traditional method of performance evaluation fails when one attempts to capture the magnitude in the impact of one ACP versus another on a system's overall performance. For this reason, we have implemented a comprehensive simulator for a distributed database system in order to study the performance implications of atomic commit protocols on *transaction throughput* under different system configurations and transaction behaviors (72).

In this chapter, based on our simulation results, we report on the performance implications of the two-phase commit (2PC) protocol and its most commonly known two variants (namely, presumed abort (PrA) and presumed commit (PrC)), as well as CL and IYV protocols, on transaction throughput in wide-area, gigabit-networked distributed database systems. In contrast to other recent comparative performance evaluations of two-phase commit variants in local area networks (18, 19), we explicitly model (1) the propagation latency of the communication network, (2) the overhead of the management of the database buffer and of flushing the transaction and protocol execution log records and (3) the overhead of recovery from site failures. Previous studies did not consider these aspects based on the belief that these aspects should not affect the relative performance of the common 2PC variants. On the other hand, our results show that these aspects do have a direct impact on the performance of IYV and CL. Previous studies have also adopted a parallel model of execution of transactions' operations at the participant sites. In our study, we adopt the more realistic sequential execution model of the operations of transactions since transactions, being programs that are usually written in *ad hoc* fashion, have behaviors that cannot be determined a priori (4) with respect to either their execution patterns or the sites participating in their execution.

Since read-only transactions are the majority of transactions in general database systems, we also study the performance gains when the traditional read-only (which we discussed in Section 3.2.7) and unsolicited update-vote (which we present in the next chapter (Section 6.6)) optimizations are incorporated into the five protocols evaluated. To isolate the impact of ACPs on the overall performance of the system, we also simulate the behavior of the system when *distributed-execution centralized-commit* (DECC) is used. As in a previous study (19), DECC simulates the distributed execution of operations and centralized commit processing (i.e., no ACP is used). Though artificial, DECC allows us to evaluate the *highest attainable* system performance in the absence of failures. In this way, we can better relate the performance enhancement of ACPs and optimizations to the highest attainable performance while at the same time comparing their performance to each other.

Our results are based on the assumption that a transaction will commit when it reaches its commit point. That is, in the absence of site failures, a participant always votes "yes" for a transaction during the course of commit processing. Salient results of our study show that IYV is, in general, better than all the other evaluated protocols during both normal processing and in the presence of one or two failed sites at any given time. IYV is matched by CL under some circumstances whereas, under others, CL is the worst among all the evaluated protocols. Interestingly, with respect to the two-phase commit variants, the choice of a protocol has very little impact on performance for the case of long transactions as opposed to short ones. Further, performance enhancements due to a read-only optimization are more pronounced with short transactions. Finally, we show that there is a cross-over point between the performance curves of presumed abort and presumed commit protocols even under the assumption that all transactions are to be committed when they reach their commit points. This result cannot be shown using the traditional method of performance evaluation which we used in Section 3.2.8.1, showing that presumed commit protocol is, in general, better than presumed abort protocol.

The rest of the chapter is structured as follows: In Section 5.1, we present our simulation system model and its associated parameters. In Section 5.2, we present the results of our study during normal processing and in the absence of failures whereas in Section 5.3, we present the results in the presence of failures.

5.1 Simulation System

In this section, we discuss our simulation system model and its parameters. We also discuss the transaction execution model and the workload model. We have implemented our simulator in *C* using the CSIM simulation library (by Mesquite Software Inc.) on a UNIX Ultra SPARC workstation.

5.1.1 Simulation System Model

We modeled our system in a manner similar to other database simulation models (73, 74, 75, 19). Table 9 contains our simulation model parameters which we divide them into four logical sets of parameters: (1) *database parameters*, (2) *transaction parameters*, (3) *site parameters* and (4) *resource parameters*.

In our model, a database is a collection of objects that are uniformly distributed across a number of sites without data replication. A data object in our model is uniquely identified by the tuple $< Site_{id}, Object_{No} >$. The database parameters which are the number of sites (*NumSites*) and objects (*NumObjs*) are specified as parameters to the simulating system.

The sites are interconnected via a *high speed* wide-area communication network. The propagation latency (*PropLatency*) of the network is specified as a resource parameter.

Table 9 Simulation parameters.

Database Parameters		
1.	*NumSites*	The number of database sites
2.	*NumObjs*	The number of data items per database site
Transaction Parameters		
3.	*ExecPattern*	Sequential
4.	*DistDegree*	Number of participants
5.	*ParticipantSize*	Transaction's average access per participant
6.	*ThinkTime*	Think time between database operations
7.	*PercRead-OnlyTrx*	percentage of read-only transactions
Site Parameters		
8.	*NumCPUs*	Number of CPUs
9.	*NumDisks*	Number of disks
10.	*MPL*	Degree of multiprogramming per site
11.	*HitRate*	Buffer pool hit probability
12.	*LogFlushRate*	Log pool flush probability due to WAL
13.	*LogSize*	Maximum log buffer size in pages
14.	*TBF*	Time between failures
15.	*TTR*	Time to repair
Resource Parameters		
16.	*CPUTime(MESG)*	CPU Time for processing a message
17.	*CPUTime(READ)*	CPU Time for processing a read operation
18.	*CPUTime(WRITE)*	CPU Time for processing a write operation
19.	*DiskTime*	Disk access time
20.	*DiskTransfTime*	Page transfer time
21.	*PropLatency*	Propagation time for a message
22.	*Timeout*	Message timeout

Each site in our system consists of (1) a *transaction manager* (TM), (2) a *data manager* (DM), (3) a *lock manager* (LM), (4) a *communication manager* (CM), (5) a *resource manager* (RSM), (6) a *database cache manager* (DCM) and (7) a *recovery manager* (RM).

At a site, the TM manages transaction identifiers, dispatches operations for execution to the appropriate DMs and coordinates the commit processing for transactions initiated at its site. A TM maintains its own log for those transactions that it coordinates.

A DM receives operation requests from both the local TM and remote TMs, accesses the resources necessary to fulfill these requests, acknowledges the requesting TM upon the completion of the request, and participates in the commit processing of those transactions that have performed operations at its site. A DM maintains a log for all database operations that it executes and for transactions in which it participates in their commitment.

For concurrency control, we use *strict two-phase locking*, that we briefly discussed in Chapter 2, the de facto standard of the industry. A LM at a site is responsible for the granting and releasing of locks at its site in accordance to the used ACP. If the lock manager cannot satisfy a request for a lock on a data object, the requesting transaction is immediately aborted. Through *immediate abort* (which is also called *immediate-restart* (73)), deadlocks are avoided since transactions never wait for requested locks to be released. This deadlock avoidance strategy simplifies the simulation model without affecting the relative performance of the evaluated protocols.

A RSM is a logical entity the represents the set of physical resources available at any given site. Access to all physical resources within a site is served on a first-come-first-serve (FCFS) basis without any preference to the type of service requested from a resource. In our system, the physical resources available at a site consists of a number of CPUs (*NumCPUs*) and disks (*NumDisks*). All CPUs within a site share a common queue and are responsible for the processing of messages and database operations. When a message is received or about to be sent by a CM, it consumes some $CPUTime(MESG)$ of CPU time. Furthermore, the receipt of a message may require additional CPU time. For example, receiving a message requesting a database operation will require $CPUTime(READ)$ of CPU time for a read operation, or $CPUTime(WRITE)$ for a write operation. Additionally, some messages will be need to be acknowledged requiring another $CPUTime(MESG)$ of CPU time.

At a site, there are one or more disks dedicated to storing data, and separate disks dedicated to storing the logs. The RSM maintains a separate queue for each disk at its site. For log disks, the log buffer may be limited to $LogSize$. When the log buffer reaches $LogSize$, the log buffer must be flushed to disk. The cost of flushing to or reading from disk is represented by the access time ($DiskTime$), and a transfer rate ($DiskTransfTime$) for each page moved to/from the disk. Thus, the cost associated with disk services can be summarized as follows:

$$Cost(Disk) = DiskTime + (DiskTransfTime * NumberOfPages)$$

A DCM at a site is responsible for the management of the data transfer between database cache and data disk(s). A DCM determines whether a page resides in the database cache or needs to be fetched from the data disks based on a *HitRate* parameter. Similarly, a DCM is responsible for locating an available slot in the cache to swap the requested database page in the case of a miss. If the page to be replaced in the cache is dirty, the page must first be flushed to disk before it is replaced. However, before flushing the replaced page to disk, the DCM must insure that WAL has been performed for the dirty page. Based on the $LogFlushRate$, the DCM determines whether WAL needs to be performed or not. If WAL must be performed, the DCM requests that the DM at its site flush its log. Once the DM has flushed its log, the DCM flushes the dirty page to disk and fetches the requested database page from disk.

The RM at a site is another logical entity that captures the behavior of the site after a failure. When failures are enabled in our simulation, they are separated by time between failures (TBF) milliseconds. Once a site has failed, the site remains down for time to repair (TTR) milliseconds in order to repair the failure. Any message sent to a failed site will timeout after *Timeout* milliseconds. Timeouts are only used for actual failures. That is, in our system, we incorporated a low level mechanism that checks for a failure in the case of a timeout. Thus, transactions are not aborted in the case that a site becomes slow due to high congestion on its resources. After the site has been repaired, the TM and the DM at that site must recover. As part of its recovery procedures, the TM completes commit processing for each incomplete transaction. On the other hand, the RM consults its log and performs the necessary undo and redo phases for aborted and committed transactions, restoring the database state to a consistent state. For IYV and CL, the recovery process involves contacting all other sites for recovery whereas only the coordinators of in-doubt transactions are contacted for the other 2PC variants.

In next subsection, we describe the transaction model and the execution model adopted in this study while in subsection 5.1.3, we describe the workload that is applied to the system.

5.1.2 Transactions and their Execution Model

While still adhering to the traditional ACID (i.e., Atomicity, Consistency, Isolation and Durability) properties of transactions, a distributed transaction is modeled as a sequence of read and write operations that is terminated by a commit or an abort transaction management primitive. The execution model of distributed transactions can be either *sequential*, *participant-sequential* or *parallel*. In the sequential execution model, before a transaction submits an operation, it waits until the previous submitted operation has been executed and acknowledged by the corresponding participant. In other words, a transaction submits an operation only if it has no other operations pending, irrespective of the type of the pending operation. When a transaction receives the results of an operation, it spends some *ThinkTime* which represents the processing time of the received results before it sends the next operation for execution.

In the participant-sequential execution model, a transaction submits all its operations to a participant at the same time. When the transaction receives the results of the operations that it has submitted to a participant, it also spends some *ThinkTime* before it sends the next set of operations to another participant.

In the parallel execution model, a transaction submits all its operations to all participants at the same time without any *ThinkTime* between its operations.

In all execution models, once the last pending operation pertaining to a transaction is acknowledged and a commit primitive is received from the transaction, the coordinator of the transaction initiates an atomic commit protocol. In the participant-sequential and parallel execution models it is assumed that all the operations of a transaction are known at the time that the transaction is submitted to the system whereas in the sequential execution model there is no such an assumption. Hence, the latter execution model is more general than the other two and, in our study, we consider only this model (*ExecPattern*).

5.1.3 Workload Model

Each site is associated with a multiprogramming level (MPL) that is specified as a parameter to the system. The MPL parameter is used to limit the number of active transactions at a site at any given time. At the beginning of a simulation run, a trace of transactions is generated and used with all protocols. The trace is generated based on the *ExecPattern* of transactions, the number of sites participating in a transaction's execution, which is specified by the *DistDegree* parameter, the number of data operations that a transaction performs at each participant site, which is uniformly distributed between 0.5 and 1.5 of the *ParticipantSize* parameter, and the percentage of read-only transactions (*PercRead-OnlyTrx*).

The simulator is run at full capacity (i.e., peak load). That is, when a transaction terminates, a new transaction enters the system and starts executing at the site where the previous transaction has terminated. For aborted transactions, we use *fake* restarts where an aborted transaction is restarted as an independent transaction after a delay time that is equal to the mean response time of transactions. For each run, the simulator executes until 10,000 transactions are committed. This translates to a total of 12,000 to 40,000 transactions that are processed by the system, depending on the transaction length. We confirmed that this number of transactions makes our system operate within its steady state by comparing runs with 10,000, 12,000 and 15,000 committed transactions. Our comparison did not show any statistically significant differences between these runs. Hence, in all our experiments, we run our system for 10,000 committed transactions. The performance curves in all our experiments represent the statistical mean of three independent runs with a confidence half-length interval of no more than 2.7 at the 90% confidence level and no more than 3.5% relative precision (i.e., relative error).

5.2 Performance of Atomic Commit Protocols (ACPs) During Normal Processing

In this section, we evaluate the performance of ACPs in the absence of failures. The parameter settings for these experiments are shown in Table 10. Since there are no failures, in our experiments, we assume that when a transaction reaches its commit point (i.e., all its operations have been executed and acknowledged), the transaction will be committed. Also, since 2PC and PrA behave exactly the same under in the absence of failures for committing transactions, for the clarity of our figures, we include only the performance curves of PrA. For the non-failure case, we conducted three sets of experiments which are shown in Table 11. The first set of experiments, experiments 1 and 2, deal with the impact of ACPs on the system's performance in the case of, relatively, long and short update transactions, respectively. Given the size of the simulated database, long transactions execute, on average, 6 operations at each participant site while short transactions execute, on average, 2 operation at each participant site. The second set of experiments, experiments 3 and 4, deal with the impact of ACPs on the system's performance in the presence of read-only transactions for long and short transaction sizes, respectively, without the use of read-only optimizations. The third set of experiments, experiments 5 and 6, show

Table 10 Simulation parameters for the non-failure case.

Database Parameters		
1.	*NumSites*	8
2.	*NumObjs*	1000
Transaction Parameters		
3.	*ExecPattern*	Sequential
4.	*DistDegree*	3
5.	*ParticipantSize*	6 (long) and 2 (short)
6.	*ThinkTime*	0
7.	*PercRead-OnlyTrx*	0 (update), 70 %
Site Parameters		
8.	*NumCPUs*	1
9.	*NumDisks*	1 for each log and 2 for data
10.	*MPL*	4-14 (long), 5-50 (short)
11.	*HitRate*	80 %
12.	*LogFlushRate*	50 %
13.	*LogSize*	10 pages
Resource Parameters		
14.	*CPUTime(MESG)*	1 msec
15.	*CPUTime(READ)*	5 msec
16.	*CPUTime(WRITE)*	5 msec
17.	*DiskTime*	20 msec
18.	*DiskTransfTime*	0.1 msec
19.	*PropLatency*	50 msec

Table 11 Non-failure experiments.

Set No. 1	
Experiment 1	Long, update transactions
Experiment 2	Short, update transactions
Set No. 2	
Experiment 3	Long, 70% read-only transactions
Experiment 4	Short, 70% read-only transactions
Set No. 3	
Experiment 5	Long, 70% read-only transactions with read-only optimizations
Experiment 6	Short, 70% read-only transactions with read-only optimizations

the effects of read-only optimizations on the performance of ACPs for long and short, 70% read-only transactions, respectively.

5.2.1 Experiment 1: How do ACPs perform with long transactions?

In this experiment as well as all the other experiments that we report in this dissertation, we measure the system throughput which is the total number of committed transactions per second with varying multiprogramming levels (MPLs). The MPL represents the total number of transactions executing at any given site and at any given point in time (since the system operates at full capacity). As indicated in other studies that use a closed-queuing system model (e.g., Gupta *et al.* (19)), the performance curves of the response time of transactions is the inverse of the system throughput curves. Hence, we do not report on the response time of transactions in our study.

As shown in Figure 15, the x axis is used for the MPL while the y axis is used for the system throughput. As shown in the figure, the performance curves of all ACPs start to increase from MPL 4 up to the peak MPL (i.e., MPL 8) and then they start to decline. This *thrashing* behavior of the system is due to the contention of

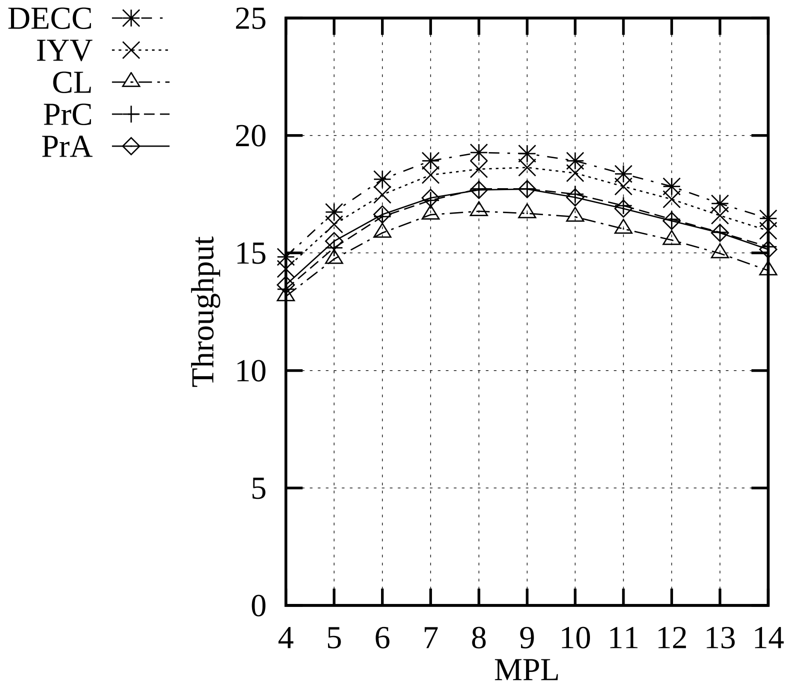


Figure 15 The performance of ACPs for long update transactions.

transactions over the data objects as well as the system resources (i.e., CPU, Disk, log buffer, and the flushing and fetching of data objects) and appears in all our experiments as well as other simulation studies $^{(73,\ 18,\ 19)}$. Due to this contention, at high MPLs, transactions tend to abort because of the high percentage of conflicts over the data objects, reducing the overall system performance.

In this experiment, we examine the performance of the different protocols when all the operations of transactions are update operations and transactions are relatively long, given the size of our database. At the peak MPL (MPL 8), the difference in the performance of the system in the ideal case (i.e., distributed-execution centralized-commit (DECC)) and the worst case (i.e., using the coordinator log (CL) protocol), in this experiment, is 2.5 transactions per second which translates to about 15% performance difference. In the case of implicit yes-vote (IYV), DECC outperforms IYV by about 5%. At the same time, IYV outperforms all other protocols. Also, all three 2PC variants have about the same throughput. IYV outperforms two-phase commit variants at the peak performance by about 5% performance enhancement in throughput with no less than 2.5% enhancement across all multiprogramming levels. Similarly, IYV outperforms CL by about 10% at the peak performance with no less than 9% performance enhancement across all multiprogramming levels.

One interesting observation in this experiment and all the experiments that we report in this dissertation is the existence of a cross-over point between the performance curves of PrA and PrC even though all transactions are committed once they reach their commit point. Based on the traditional performance evaluation, this point should not exist since PrC will always have the least number of coordination messages and forced log writes. However, our simulation system reveals that under low system loads, the initiation records of PrC affect its performance and makes it worse than PrA. After a certain point (at higher MPLs), the effects of the forced log writes at the participants in PrA as well as the acknowledgment messages of the commit decisions overshadow the cost of the initiation records of PrC, making PrC performance better than PrA performance. Our results also show that the location of this cross-over point as well as the magnitude in the performance difference before the cross-over point changes from one experiment to another, depending on the transaction mix, the length of transactions and whether a read-only optimization is being used or not.

5.2.2 Experiment 2: How do ACPs perform with short transactions?

Figure 16 shows the impact of the different ACPs on the performance of the system when all the operations of transactions are update operations and transactions are relatively short, given the size of our database. Comparing the different protocols to DECC, DECC outperforms IYV by about 12% while it outperforms CL by about 13%. With respect to PrA and PrC, DECC outperforms PrC by about 27% whereas DECC outperforms PrA by about 83%.

The results of this experiment also show three interesting observations. The first observation is that CL is a clear winner compared to the three 2PC variants

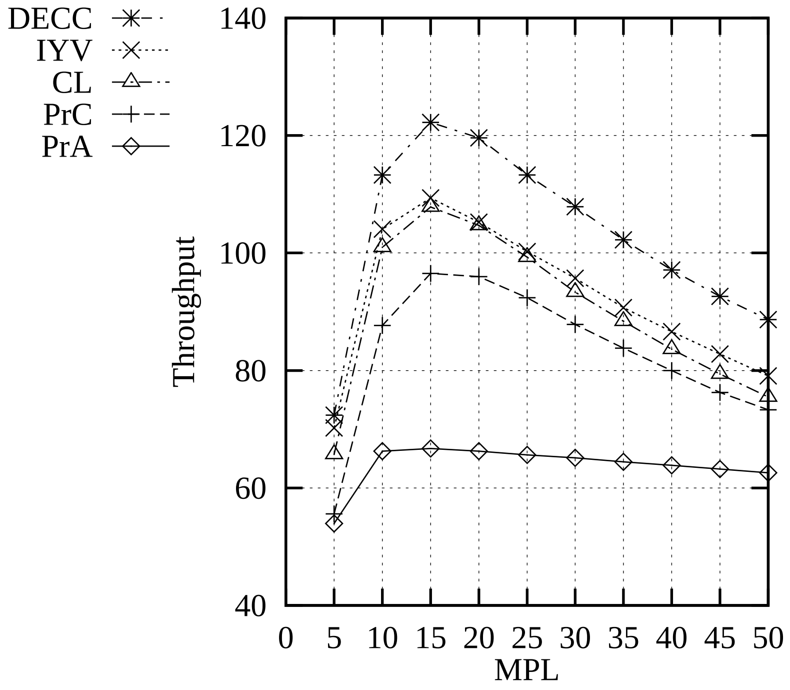


Figure 16 The performance of ACPs for short update transactions.

as opposed to being the loser in the case of long, update transactions (Experiment 1). This result clearly supports the motivation behind the design of CL (9) which assumes short transactions with high probability of being committed once they reach their commit point. However, we note that the performance of CL starts to degrade more quickly after MPL 25 where its performance enhancement over PrC is about 3% at MPL 50 after it was about 12% at the peak MPL. On the other hand, IYV performance enhancement over PrC degrades to about 8% at MPL 50 from about 13% at MPL 15. The reason behind the quick degradation in CL's performance is due to its *distributed write-ahead logging* (DWAL) which requires a participant that aborts a transaction to wait until it receives the undo log records pertaining to the transaction from the transaction's coordinator before it can release the locks held by the transaction. In contrast, the other protocols do not suffer from such an overhead since the undo records of an aborting transaction at a participant are available locally in its own log.

The second observation is that the performance of PrC has increased from a negligible one in the case of long transactions (Experiment 1) to about a 45% enhancement over the performance of PrN and PrA at peak performance (MPL 15). Similarly, by comparing the results of the first experiment and this one, we notice that the maximum performance difference in the first experiment was about 15% (DECC versus CL) whereas in this experiment it is about 83% (DECC versus PrA). Thus, not only the relative performance order of the protocols have changed (CL became a winner in this experiment compared to the 2PC variants after it was a looser in the previous experiment), bout also the magnitude in the performance differences have greatly changed. These two results clearly support our claim that the traditional way of evaluating the performance of ACPs does not only fail to reflect their relative performance but it also fails to reflect the magnitude in performance differences. The magnitude in the performance differences between Experiment 1 and Experiment 2 can be justified once the ratio of the cost associated with commit processing over the cost of a transaction's execution is considered. In the case of long transactions, commit processing is less costly than in the case of short transactions compared to the overall transaction execution cost. Thus, any extra coordination messages or forced writes incurred in an ACP are more severely reflected on its performance with short transactions.

The third observation is regarding PrA's low thrashing behavior compared with the other protocols after it reaches its peak performance. With respect to this issue, we note that PrA reaches its performance peak very quickly because the system becomes highly congested due to the excessive forced log writes and coordination messages. This has a consequence that makes PrA less sensitive to increased MPL compared to the other protocols.

5.2.3 Experiment 3: How do ACPs perform with long, read-only transactions?

Figure 17 shows the performance of ACPs for log, majority read-only transactions. Since read-only transactions are the majority of transactions in any general

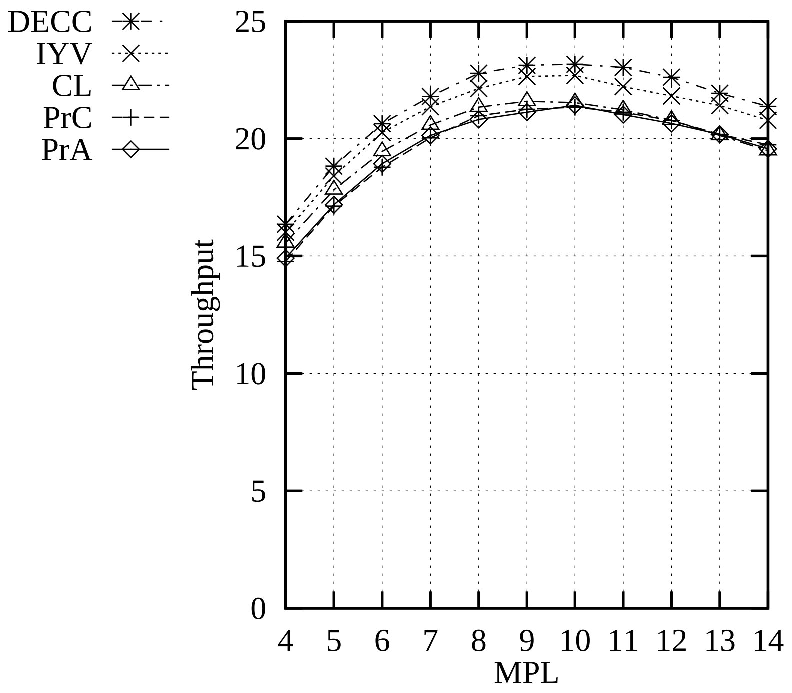


Figure 17 The performance of ACPs with long 70% read-only transactions.

database system, in this experiment, we generated a trace of long transactions that contains 70% read-only transactions. By comparing Figure 17 to Figure 15, we notice that, when read-only transactions are introduced, the performance of all the evaluated ACPs has been enhanced by at least 10%, across all multiprogramming levels. Furthermore, the peak performance point of all protocols has been shifted from MPL 9 to MPL 10. This is consistent with the fact that the system resources are still under utilized and transactions do not conflict at the same rate as in Experiment 1.

In this experiment, IYV still exhibits the best performance over all other protocols and across all multiprogramming levels. In addition, CL's performance has been enhanced to become better than all three two-phase commit variants at low MPLs and about the same as PrC at peak performance (i.e., around MPL 10) due to the reduced distributed write-ahead logging (DWAL) of CL when the majority of transactions are read-only.

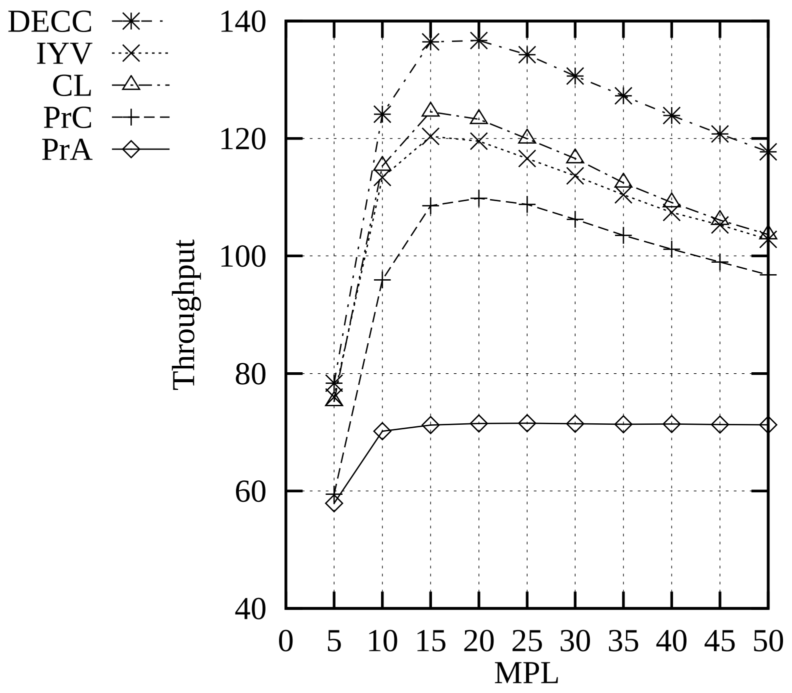


Figure 18 The performance of ACPs with short 70% read-only transactions.

Finally, we again notice a cross-over between the performance curves of PrC and PrA with PrC performing better around the peak MPL. As mentioned above, even though we noticed a cross-over point in each of our experiments, in some figures for the sake of clarity, the cross-over points are not shown.

5.2.4 Experiment 4: How do ACPs perform with short, read-only transactions?

As in experiment 3, in this experiment we introduced a 70% read-only transactions in the transaction trace. Figure 18 shows the performance of ACPs for short transactions where the performance of all ACPs has been enhanced by at least 7% across all multiprogramming levels (which is the case in the PrA protocol). Interestingly, the performance of CL became better than IYV in this experiment. This is because DWAL does not add much extra overhead in the case of short transactions dominated by read-only transactions and the extra non-forced (commit) log records of IYV become more significant given the limited log buffers of the participants.

We also note that PrA reaches a steady state in this experiment for a period longer than that in the previous experiment where all transactions are update transactions (Experiment 2). This is because the 70% read-only transactions do not conflict over locks with each other and, therefore, read-only transaction do not get aborted unless it conflicts over a lock with an update transaction, resulting in a less aborted transactions with less system thrashing.

5.2.5 Experiment 5: How do read-only optimizations affect the performance of ACPs for long transactions?

In this experiment, we used traces containing 70% read-only as in experiment 3 and factored in the effects of the traditional read-only (TRO) optimization, that we discussed in Section 3.2.7 and our *unsolicited update-vote* (UUV) optimization, that we will discuss in the next chapter (Section 6.6), on the behavior of the 2PC variants. We also applied a special case of UUV in the case IYV and CL. Since a coordinator in both protocols can determine if a transaction is read-only at a participant's site based on whether it has received any log records from the participant during the execution of the transaction, participants do not have to send unsolicited update-votes. Thus, in this special case, which we will call *RO*, the coordinator sends a read-only message to each read-only participant without waiting until the commit record is in its stable log, thereby releasing the resources at read-only participants earlier than their update counterparts.

As shown in Figure 19, neither IYV nor CL have benefited significantly from RO (i.e., about 1% performance enhancement) while the 2PC variants have benefited more from TRO and UUV reducing the performance gap with IYV at high MPLs from about 7% (which is the case in Experiment 3) to 3%. For CL, at low MPLs, its performance was about the same as the 2PC variants whereas, at high MPLs, CL performance became worse than PrN, PrA and PrC due to DWAL.

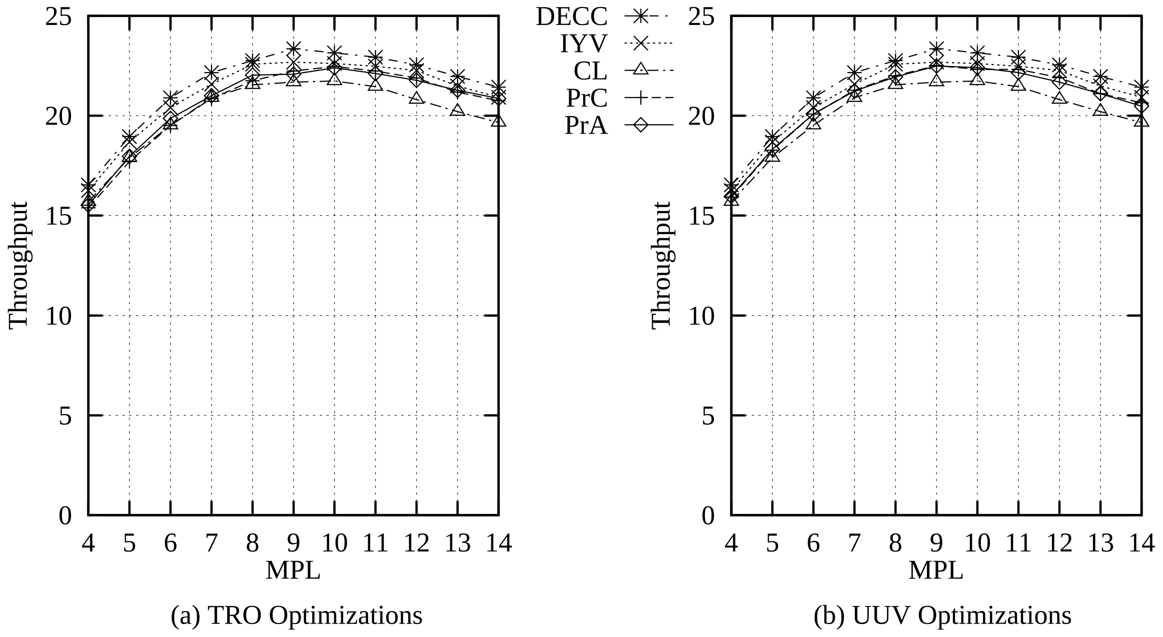


Figure 19 The performance of ACPs for long transactions with read-only optimizations.

5.2.6 Experiment 6: How do read-only optimizations affect the performance of ACPs for short transactions?

As in the previous experiment, in this experiment we have also factored in the effects of TRO, UUV and RO on the behavior of ACPs. As we have reasoned in experiment 2, any extra coordination messages or forced log writes in the case of short transactions by an ACP, have a significant impact on its performance compared to long transactions. Conversely, any reduction in the coordination messages or forced log writes greatly enhances the performance of an ACP in the case of short transactions as opposed to long ones. Thus, unlike the results of experiment 5, the performance of all protocols has been enhanced with PrA gaining the most and CL the least, as shown in Figure 20. PrA has gained about 60% performance enhancement using TRO, bringing its performance comparable to PrC. It also made PrA as sensitive as the other protocols to the MPL level as opposed to its behavior in experiments 2 and 4. By factoring in the effects of RO, the performance of IYV is again better than CL since it has gained more by the reduction of the logging activities

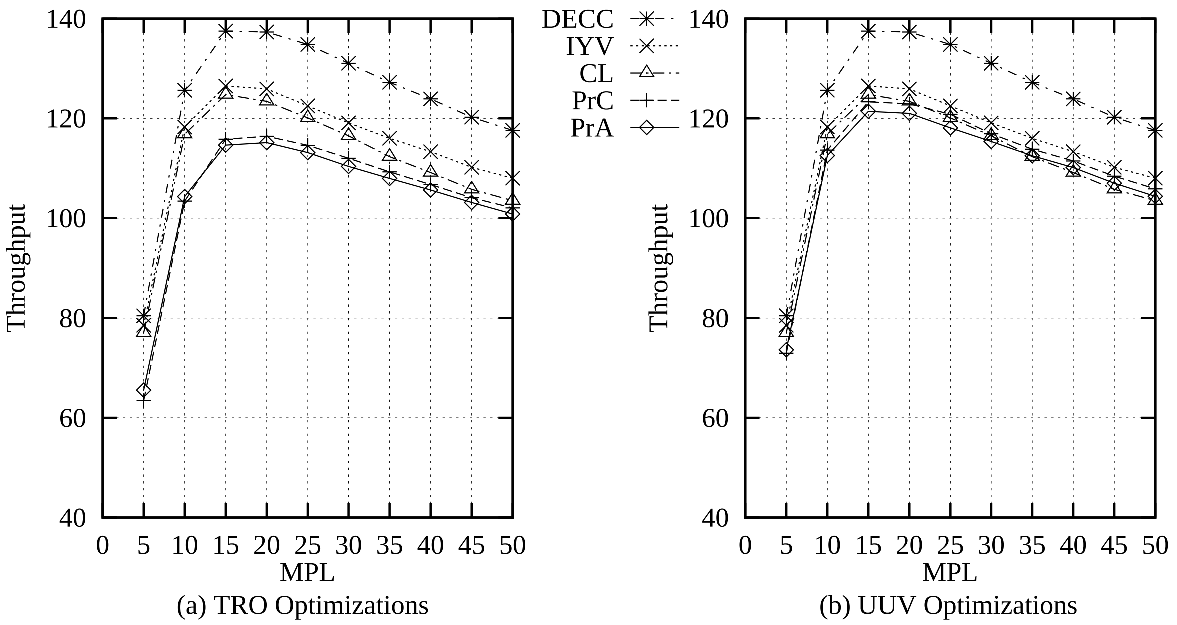


Figure 20 The performance of ACPs for short transactions with read-only optimizations.

than CL using RO. By comparing UUV with TRO, PrA has gained about 70% with UUV instead of 60% with TRO while PrC has gained 12% using UUV instead of 6% using TRO. As a result, UUV have closed the gap between the performance of PrC and IYV to about 3% in favor of IYV.

5.3 Performance of ACPs in Case of Failures

In this section, we evaluate the performance of ACPs in the presence of failures. We investigate the effects of failures on both update transactions and 70% read transactions with long and short transaction sizes by presenting the results of three experiments. The first two experiments deal with single failures while the third experiment deals with double failures.

For our failure runs, the time between failures was set to 10,000 milliseconds, the repair time for a failure was set to 200 milliseconds, and the message timeout to

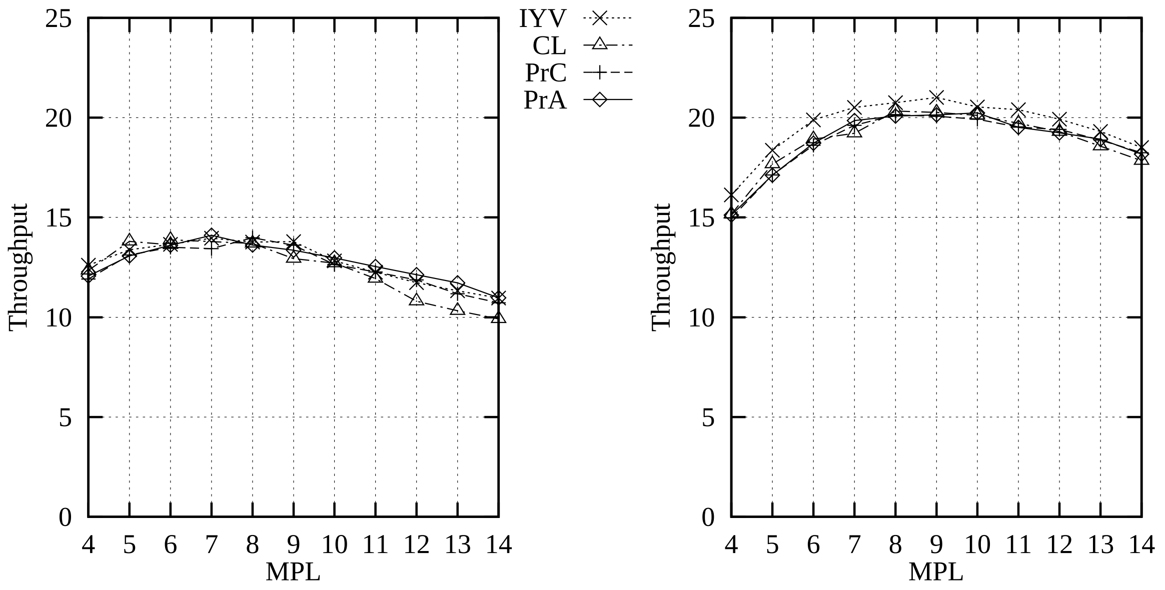
400 milliseconds. This means that on average for long transaction about 25 failures occurred per 10,000 committed transactions while for short transactions about 8 failures occurred per run. It has to be noted that this is an extreme situation even for a relatively high failure-prone system. However, this exaggerated failure-prone tests will show us the robustness or vulnerability of the different protocols to system failures.

5.3.1 Experiment 7: How do single failures affect the performance of ACPs in case of long transactions?

In the case of long, update transactions, Figure 21 (a), the presence of single failures seems to cause a 25% performance degradation for all protocols. Whereas, in the case of long, 70% read-only transactions, Figure 21 (b), the performance degradation for all protocols is between 10-15%. Obviously, a 70% read-only transaction mix will cause less logging and therefore speed system recovery time for a failed site causing system throughput to remain higher. Furthermore, there is very little or no performance gain in using IYV compared with the three two-phase commit variants in the case of long transactions. The added expense, for a recovering participant in IYV, to contact all the coordinators in the system in order to recover any missing redo log records is most likely responsible for this phenomenon. Similarly, both the participant and coordinators for CL must contact all other sites in the system in order to gain any missing or incomplete information. Therefore, the performance of CL remains lower than the performance of IYV and the three two-phase commit variants.

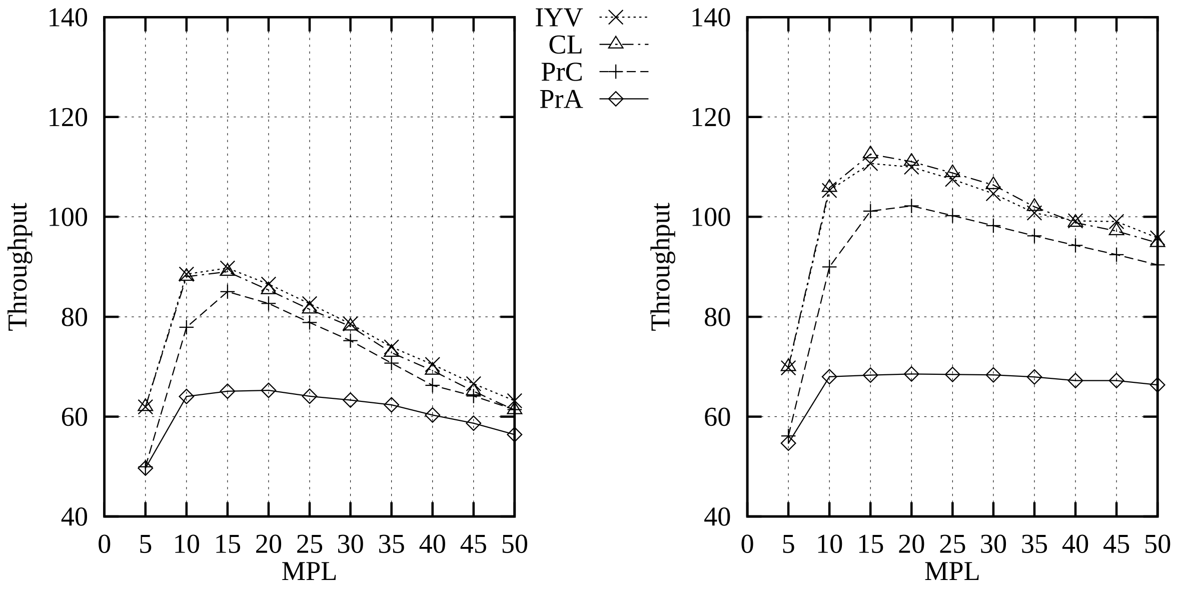
5.3.2 Experiment 8: How do single failures affect the performance of ACPs in case of short transactions?

In the case of short, update transactions, Figure 22 (a), the presence of single failures cause between a 15-20% performance degradation for IYV, CL, and PrC. Interestingly, PrA is affected less strongly by the presence of failures in this case



(a) Update Transactions (b) 70% Read-Only Transactions

Figure 21 Single failures for long transactions.



(a) Update Transactions (b) 70% Read-Only Transactions

Figure 22 Single failures for short transactions.

since PrA is already severely congested on the system resources and committing less transactions. In the case of 70% read-only transactions Figure 22 (b), the performance degradation of IYV, CL, and PrC is only between 5-10%. Also IYV and CL retains a sizable advantage over the two-phase commit variants in both single failure cases due to their far superior performance in normal processing. At peak throughput, IYV retains about a 7% advantage over PrC while CL's advantage over PrC is slightly less.

In summary, for short transactions, the performance of IYV and CL in normal processing is enough to offset the additional costs of their recovery schemes and still provide a substantial performance improvement.

5.3.3 Experiment 9: How well can IYV perform in the case of simultaneous failures?

We have performed two experiments for simultaneous failures in the case of short transactions. The total number of failures should be double the single failure case for short transactions since two sites are now failing simultaneously. In this experiment, we choose to compare IYV with PrC, and we did not consider CL since IYV outperforms CL in all the previous failure experiments. Similarly, we chose PrC to represent the 2PC variants since PrC has been shown in to out perform PrA and PrN in previous experiments.

Figure 23 (a) represents the case where both failures occur at the same moment, and are repaired concurrently. While throughput degrades 17% from the single failure case, IYV achieves a 10% performance advantage over PrC.

The second case represents the situation of two sequential/overlapping failures of a coordinator and participant site. In effect, both failed sites are affected by the overlapping failures in the case of IYV while for PrC, the failed sites recover independently. In Figure 23 (b), the two failures occur at the same time, but the repair time for one of the failures is the normal 200 ms while the repair time for

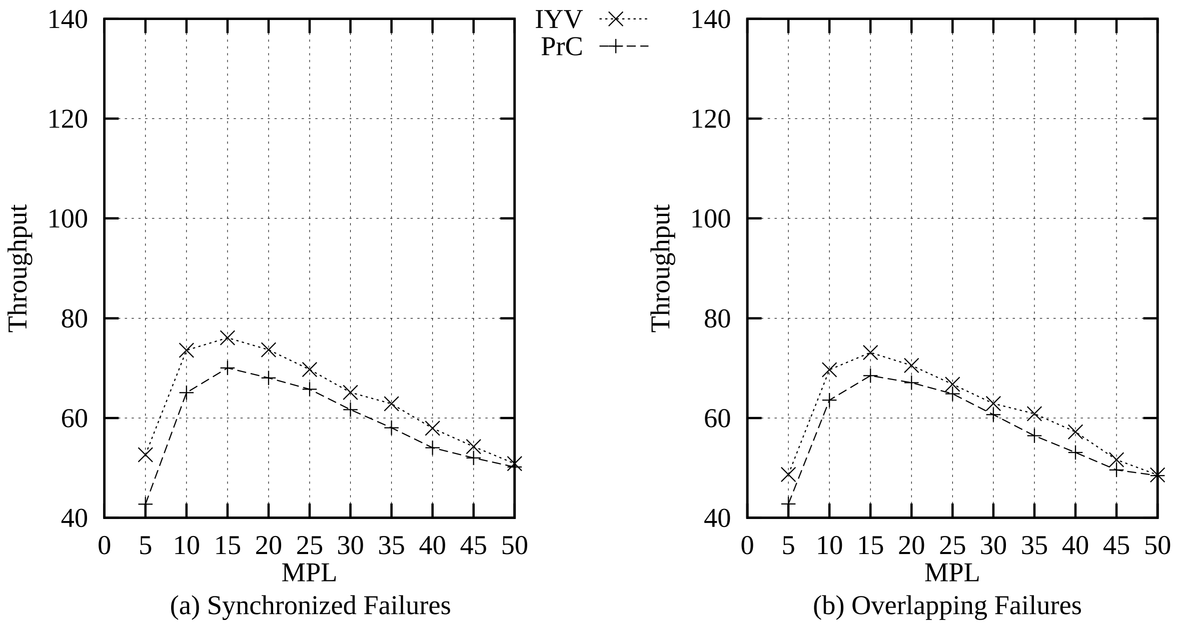


Figure 23 Simultaneous failures for short transactions.

the second failure is 2000 ms. IYV holds only a 4% advantage over PrC at peak throughput, and the performance of IYV degrades 5% from the synchronized failure case while PrC degrades very little.

5.4 Summary of Results

The results of our experiments show that the IYV protocol is better or performs equally to the other evaluated protocols in almost all cases of long and short, update transactions as well as of long and short, majority read-only transactions with and without using a read-only optimization. This also holds in the presence of single and two overlapping site failures. The exception was CL performing better than IYV in the case of short, majority (above 70%) read-only transactions without a read-only optimization. This leads us to the conclusion that implicit yes-vote is, in general, better than all the other evaluated protocols making it the choice for the future gigabit-networked distributed database systems.

Our results also showed that the performance of CL is greatly influenced by transactions' length and the degree of multiprogramming. CL performance degrades significantly for long transactions and high multiprogramming levels. CL generally performs worse than the three 2PC variants for long transactions except for the case of majority read-only transactions without a read-only optimization.

Another very interesting result is that, when there is a performance difference between 2PC variants, PrC is always the winner. This is especially the case for short transactions. This result is in contrast with the general belief that PrA is better than PrC. This is only true in low multiprogramming levels where the initiation log records associated with PrC has a major impact on its performance. In fact, we have observed in all our experiment a cross-over in their performance in favor of PrC. The location of the cross-over point varies depending on the length of transactions, the transaction mix (i.e., the percentage of read-only transactions) and whether or not a read-only optimization is used.

Finally, we note the difference between these evaluation results and those in Chapter 4 which are based on the traditional performance evaluation method and which did *not* capture neither the relative performance of ACPs nor the magnitude in the performance differences.

Since IYV is not applicable in systems that require an explicit voting phase (as we mentioned in the previous chapter) and based on our performance evaluation results, in the next chapter, we investigate techniques that enhances the performance of PrC in the context of the more general multi-level transaction execution model since PrC is the best alternative. We also present our new read-only optimization (i.e., UUV) that we have evaluated its performance in this chapter and show how it eliminates the initiation records of PrC variants from read-only participants and transactions.

6.0 AN ARGUMENT IN FAVOR OF PRESUMED COMMIT PROTOCOL

In the previous chapters, we considered atomic commit protocols (ACPs) in the context of a two-level transaction execution model. Current distributed transaction processing standards and commercial systems adopt a more general multi-level transaction execution model. Although implicit yes-vote (IYV) is a highly efficient protocol that can be extended to the multi-level transaction execution (MLTE) model in a straight forward manner, it might not always be applicable as we mentioned in the Chapter 4. For this reason and motivated by the superior performance of presumed commit (PrC) protocol over presumed abort (PrA) protocol, as our simulation results showed in the previous chapter, in this chapter, we revisit PrC and PrA protocols in the context of MLTE model.

In Section 3.2.8.1, based on our analytical evaluation, we concluded that performance argument that favors the choice of PrA protocol rather than PrC to be the standard ACP is due to the major drawback of PrC which requires forcing initiation records for both read-only and update transactions. Thus, if there is a way to eliminate or reduce the cost associated with the initiation records, the argument would go in favor of PrC, especially given the fact that high speed networks and computing systems are becoming highly reliable and distributed transactions will most probably commit after all their operations have been successfully executed and acknowledged. The same intention has been behind the design of the *new presumed commit* protocol for the two-level transaction execution model (7) that we briefly discussed in Section 3.2.4.

In this chapter, we present two new PrC variants that force write at most a single initiation record, at the root coordinator, effectively eliminating *all* the intermediate initiation records from cascaded coordinators in the MLTE model (76). The first PrC variant is called the *rooted presumed commit* (RPrC) protocol where only the root coordinator force write an initiation record. The second PrC variant is called the *re-structured presumed commit* (ReSPrC) protocol which is based on the idea of *flattening* the transaction execution trees (8). The difference between the two variants is that RPrC protocol is more general with respect to applicability, whereas ReSPrC protocol is more efficient. Specifically, since RPrC protocol does not require the transformation of a transaction's execution tree into a two-level commit tree, which is the case in ReSPrC protocol, RPrC protocol is more applicable than ReSPrC protocol. On the other hand, when the ReSPrC protocol is applicable, it provides savings in time complexity during commit processing that ReSPrC protocol does not provide.

Furthermore, in order to completely eliminate the effects of the initiation records from read-only participants and read-only transactions, we develop a new read-only optimization called the *unsolicited update-vote* (UUV) (77, 76). UUV can be used with RPrC, ReSPrC, as well as the other PrC variants.

The rest of this chapter is structured as follows: In the next section, we first discuss the multi-level transaction execution (MLTE) model. Then, we discuss multi-level PrA (4) and extend PrC to the MLTE model. In Section 6.2, we compare the performance of multi-level PrA and multi-level PrC. Then, we present RPrC in Section 6.3 and ReSPrC in Section 6.4. In Section 6.5, we evaluate the two new PrC variants with respect to applicability and performance. In Section 6.6, we present UUV while in Section 6.7, we apply UUV to both PrA and PrC (including the two proposed PrC variants) and show how UUV eliminates the cost of the initiation log records from read-only participants and transactions.

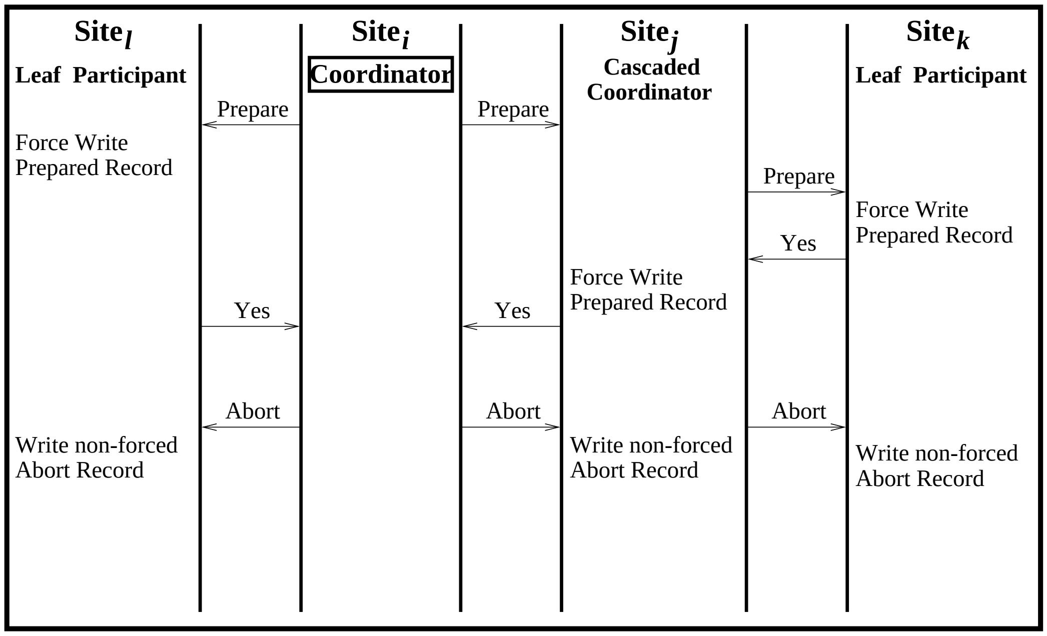
6.1 Multi-Level Presumed Abort and Presumed Commit Protocols

In this section, we first describe multi-level presumed abort and presumed commit protocols. Then, we discuss the recovery aspects in both protocols.

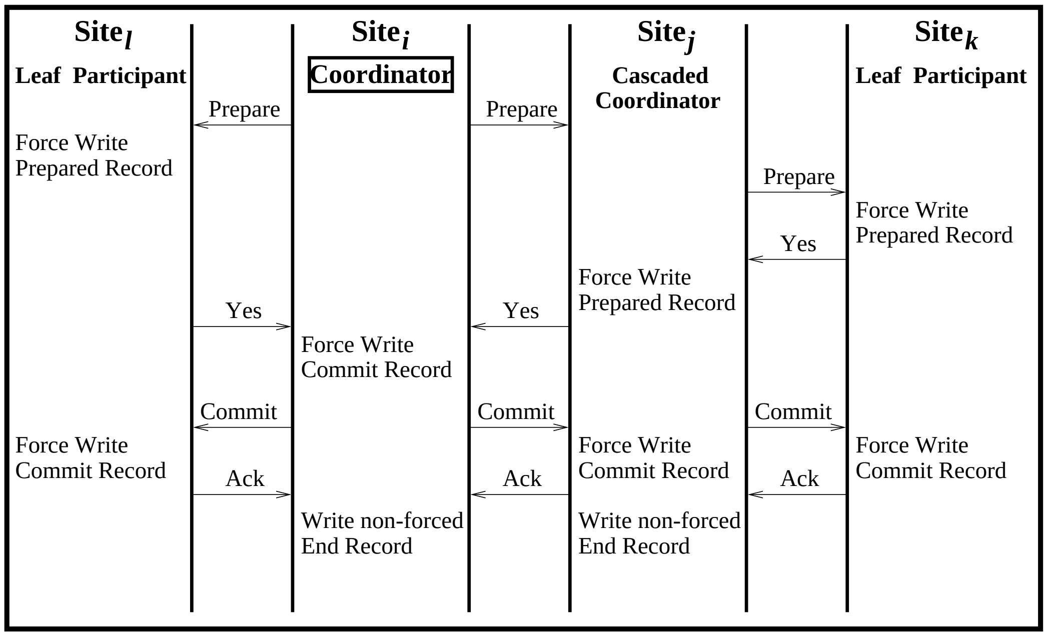
6.1.1 Description of Multi-Level Presumed Abort and Presumed Commit Protocols

The *multi-level transaction execution* (MLTE) model is similar to the tree of processes model [(4)]. In this model, a participant is a process that is able to decompose a subtransaction further. Thus, a participant can initiate other participant processes at its site or different sites. Hence, the processes pertaining to a transaction can be represented by a multi-level execution tree where the coordinator process resides at the root of the tree. In this model, the interactions between the coordinator of the transaction and any process have to go through all the intermediate processes, called *cascaded coordinators*, that have caused the creation of a process.

In the MLTE model, the behavior of the root coordinator and each leaf participant in the transaction execution tree, in both 2PC variants (PrA and PrC), remains the same as in two-level transactions. The only difference is the behavior of cascaded coordinators (i.e., non-root and non-leaf participants) which behave as leaf participants with respect to their direct ancestors and root coordinators with respect to their direct descendants. Specifically, when a cascaded coordinator receives a prepare to commit message, in *multi-level PrA* (Figure 24), it forwards the message to its descendent participants and waits for their votes. As shown in the figure, if all descendants have voted "yes", the cascaded coordinator force writes a prepare log record and then sends a "yes" vote to its coordinator. If any descendant has voted "no", the cascaded coordinator sends an abort decision to its descendants and a "no" vote to its coordinator. When a cascaded coordinator receives an abort decision (Figure 24 (a)), it writes a non-forced abort record, forwards the decision to its direct descendants and forgets the transaction. On the other hand, when a cascaded coordinator receives a commit decision (Figure 24 (b)), it forwards the decision to its direct descendants and force writes a commit record. Afterwards, the cascaded coordinator sends an acknowledgment to its coordinator. Once the direct descendants of the cascaded coordinator acknowledge the decision, it writes a non-forced end record and forgets the transaction.


Site$_l$
Leaf Participant
Site$_i$
Coordinator
Site$_j$
Cascaded Coordinator
Site$_k$
Leaf Participant
Prepare
Prepare
Force Write Prepared Record
Prepare
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Yes
Abort
Abort
Abort
Write non-forced Abort Record
Write non-forced Abort Record
Write non-forced Abort Record

(a) Abort case.


Site$_l$
Leaf Participant
Site$_i$
Coordinator
Site$_j$
Cascaded Coordinator
Site$_k$
Leaf Participant
Prepare
Prepare
Force Write Prepared Record
Prepare
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Yes
Force Write Commit Record
Commit
Commit
Commit
Force Write Commit Record
Force Write Commit Record
Force Write Commit Record
Ack
Ack
Ack
Write non-forced End Record
Write non-forced End Record

(b) Commit case.

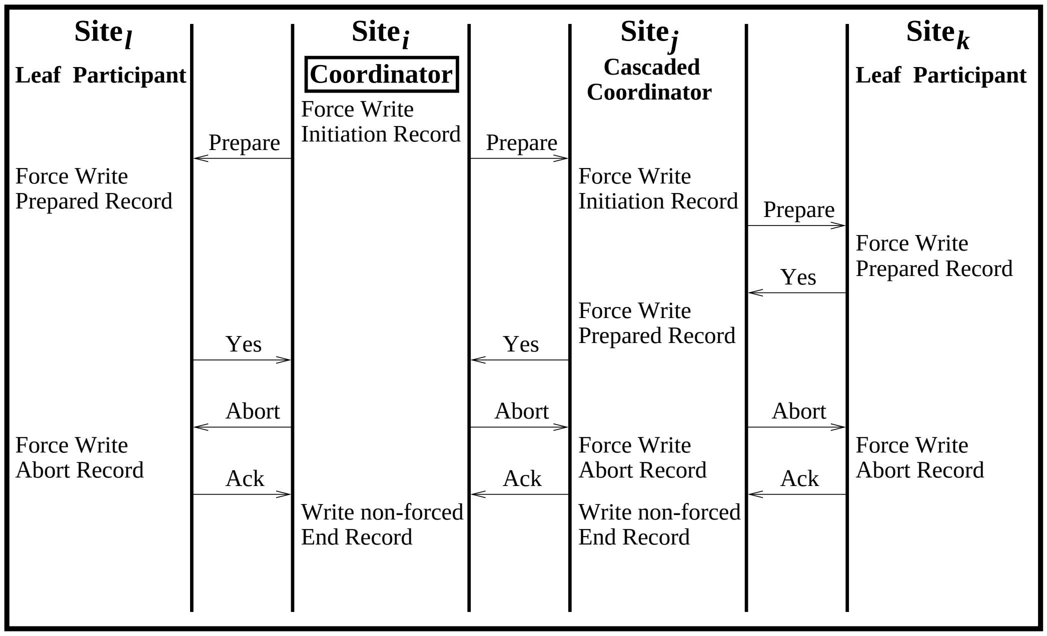
Figure 24 Multi-level PrA.

PrC can be extended in the MLTE model in a manner similar to PrA. While the extension of PrC to MLTE model has been highlighted by Mohan *et al.* (4), here we work out the details of the extension. We also adopt the *early* acknowledgment strategy rather than the *late* acknowledgment strategy (8) when extending both PrA and PrC to the MLTE model since we do not consider ***heuristic decisions*** (33, 8, 34) in this dissertation.

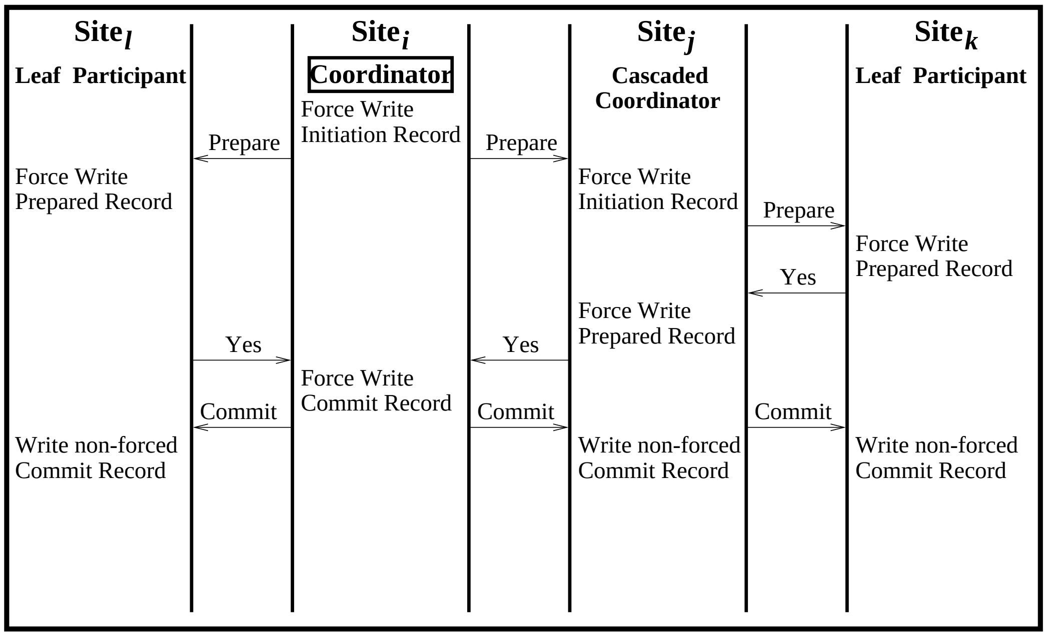
As shown in Figure 25, each cascaded coordinator in *multi-level PrC* has to force write an initiation record before propagating the prepare to commit message to its descendent participants. If the final decision is to abort the transaction (Figure 25 (a)), a cascaded coordinator propagates the decision to its descendants, force writes an abort record and, then, acknowledges its ancestor. Once the acknowledgments arrive from the descendants, a cascaded coordinator writes a non-forced end record and forgets the transaction. If the final decision is a commit decision (Figure 25 (b)), a prepared to commit cascaded coordinator propagates the decision to its direct descendants, writes a non-forced commit record and, then, forgets the transaction.

6.1.2 Recovery in Multi-Level Presumed Abort and Presumed Commit Protocols

The behavior of the root coordinator and leaf participants in case of failures remain the same as in the two-level transaction execution model. Again, only the behavior of cascaded coordinators is different which behaves, in both multi-level PrA and multi-level PrC, as both a leaf and a root participant. Hence, in the case of a failure, the presumption of PrA or PrC, depending on which protocol is used, holds between any two adjacent levels in the transaction tree. That is, if a participant inquires its direct ancestor about the outcome of a transaction, the ancestor replies according to the presumption of the protocol used in the event that it does not remember the transaction.



(a) Abort case.



Site$_l$
Leaf Participant
Site$_i$
Coordinator
Site$_j$
Cascaded Coordinator
Site$_k$
Leaf Participant
Force Write Initiation Record
Prepare
Prepare
Force Write Prepared Record
Force Write Initiation Record
Prepare
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Yes
Force Write Commit Record
Commit
Commit
Commit
Write non-forced Commit Record
Write non-forced Commit Record
Write non-forced Commit Record

(b) Commit case.

Figure 25 Multi-level PrC.

6.2 Evaluating Multi-Level Presumed Abort and Presumed Commit Protocols

As we discussed in Chapter 3, PrA is, in general, better than PrC in the two-level transaction execution model due to the latter's excessive force writes even for read-only transactions. In this section, we evaluate the performance of multi-level PrA and multi-level PrC.

In the MLTE model, multi-level PrA and multi-level PrC retain the relative advantages of PrA and PrC. They also retain the relative message complexity of PrA and PrC. However, due to the extra forced initiation log records at the cascaded coordinators, the difference between the cost of aborting a transaction in multi-level PrC and multi-level PrA is greater than the difference between PrC and PrA. For this reason, the difference between the cost of committing a transaction in multi-level PrC and multi-level PrA is less than the difference between PrC and PrA. Let us illustrate this by considering a transaction with N participants of which C are cascaded coordinators and L are leaf participants. Our evaluation is based on the behavior of the different participants that we illustrated in Figures 24 and 25.

Multi-level PrA involves $L + C$ (or N) forced log writes to abort a transaction whereas multi-level PrC involves $2L + 3C + 1$ (or $2N + C + 1$). That is, multi-level PrC incurs $N + C + 1$ more forced log writes than multi-level PrA while PrC incurs only $N + 1$ more forced log writes than PrA to abort a transaction. To commit a transaction, multi-level PrC involves $L + 2C + 2$ (or $N + C + 2$) forced log writes whereas multi-level PrA incurs $2L + 2C + 1$ (or $2N + 1$). That is, multi-level PrA requires $N - C - 1$ more forced log writes than multi-level PrC while, as we showed in Section 3.2.8.1, PrA incurs $N - 1$ more forced log writes than PrC.

In addition to reducing the relative performance advantage of multi-level PrC over multi-level PrA in committing transactions, the fact that these extra forced initiation records are written *sequentially* during the voting phase gives rise to another undesirable effect. In a lightly loaded system where there is less congestion over participant log disks, a coordinator in multi-level PrC experiences more delays to reach a final decision than in multi-level PrA. This is because the coordinator, in PrC, has to wait until all the initiation records are forced in all its descendants, in a sequential fashion (Figure 25), before it receives their votes and makes the final decision. Consequently, participants in multi-level PrC receive a final decision later than in multi-level PrA. Therefore, participants hold the resources longer in multi-level PrC than in multi-level PrA. This increased turnaround time of transactions in multi-level PrC compared to multi-level PrA favors the usage of multi-level PrA. It also means that in the case of transactions with deep trees, the tradeoff between reducing conflicts over data items in multi-level PrA and reducing extra forced commit records at every participant (cascaded coordinator or leaf participant) in multi-level PrC goes in favor of multi-level PrA.

The force writing of initiation log records sequentially has the same negative effect on read-only transactions as for update ones. This is because a read-only participant in multi-level PrC has to suffer as well from the delays associated with the forced initiation records in all its ancestors in the transaction tree before it can vote "read-only" and release any resources.

From the above discussion, it becomes clear that the PrC variants are the best choice for committing transactions only in highly loaded systems and whose the majority of update transactions are finally committed. However, in general, and in systems in which the majority of the transactions are read-only in particular, the PrA variants perform better. This is because the costs of aborting a transaction in PrA variants are less than the costs of committing a transaction in PrC variants. This asymmetry in their costs is due to initiation log records forced in PrC variants for both update and read-only transactions. Since read-only transactions are the dominant type of transactions in any general database system, PrA protocol has become the choice of the current distributed transaction processing standards.

6.3 The Rooted Presumed Commit (RPrC) Protocol

To address the problems listed above, we have developed a new protocol called the *rooted presumed commit* (RPrC) protocol. As opposed to multi-level PrC, RPrC does not realize the two-level presumption of PrC on every adjacent level because it structures cascaded coordinators as leaf participants with respect to logging. That is, cascaded coordinators do not force write initiation records and, consequently, do not presume commitment in the case that they do not remember transactions. We describe this protocol in the following section.

6.3.1 Description of the RPrC Protocol

In RPrC, the root coordinator needs to know *all* the participants at all levels in a transaction's execution tree. Similarly, each participant needs to know *all* its ancestors in the transaction's execution tree. The former allows the root coordinator to determine when it can *safely* forget a transaction while the latter allows a prepared to commit participant at any level in a transaction's execution tree to find out the final *correct* outcome of the transaction, even if intermediate cascaded coordinators have no recollection about the transaction due to a failure.

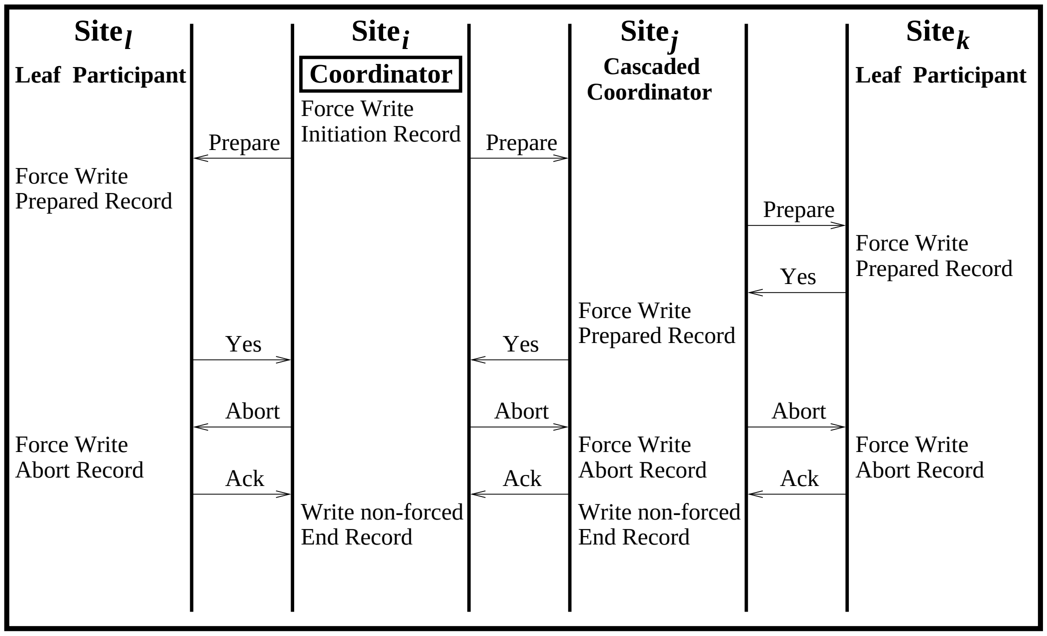
In order for the root coordinator to know the identities of all participants in RPrC, each participant includes its identity in the acknowledgment of the *first* operation. When a cascaded coordinator receives an acknowledgment of a first operation from a participant, it also includes its identity in the acknowledgment message. In this way, the identities of all participants and the chain of their ancestors are propagated to the root coordinator. As shown in Figure 26, when the transaction submits its commit request, the coordinator, force writes an initiation record that includes the identities of all participants in the transaction execution tree. Then, it sends out prepare to commit messages to its direct descendants.

The root coordinator sends its identity as part of the prepare to commit message. When a cascaded coordinator receives the prepare to commit message, it appends its own identity before propagating the message to its direct descendants. When a leaf participant receives a prepare to commit message, it copies the identities of its ancestors in the prepare log record before sending its "yes" vote. When a cascaded coordinator receives "yes" votes from all its direct descendants, the cascaded coordinator also records the identities of its ancestors as well as its descendants in its prepare log record before sending its collective "yes" vote to its direct ancestor.

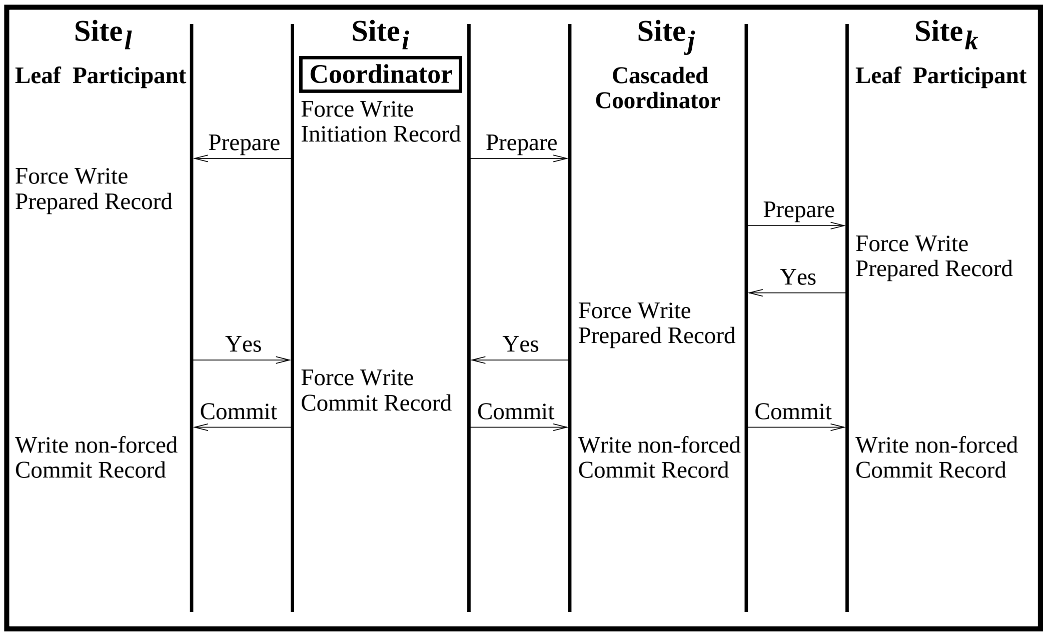
If any direct descendant has voted "no", the cascaded coordinator force writes an abort log record, sends a "no" vote to its direct ancestor and an abort message to each direct descendant that has voted "yes" and waits for their acknowledgments. Once all the abort acknowledgments arrive, the cascaded coordinator writes a non-forced end record and forgets the transaction.

If the root coordinator receives a "no" vote, it propagates an abort decision to all direct descendants that have voted "yes" and waits for their acknowledgments (Figure 26 (a)), knowing that all the descendants of a direct descendant that has voted "no" have already aborted the transaction. When the coordinator receives the acknowledgments of its abort decision, it writes a non-forced end record and forgets the transaction. When a cascaded coordinator receives the abort message, it behaves as in multi-level PrC. That is, it propagates the message to its direct descendants and writes a forced abort record. Then, it acknowledges its direct ancestor. Once the cascaded coordinator has received acknowledgments from all its direct descendants, it writes a non-forced end record and forgets the transaction. When a leaf participant receives the abort message, it first force writes an abort record and, then, acknowledges its direct ancestor.

As in multi-level PrC, when the root coordinator receives "yes" votes from all its direct descendants, it force writes a commit record (Figure 26 (b)), propagates its decision to its direct descendants and forgets the transaction. When a cascaded coordinator receives a commit message, it propagates the message to its direct descendants, writes a non-force commit record and forgets the transaction. When a leaf participant receives the message, it commits the transaction and writes a non-forced commit record.



(a) Abort case.


Site$_l$
Leaf Participant
Site$_i$
Coordinator
Force Write Initiation Record
Site$_j$
Cascaded Coordinator
Site$_k$
Leaf Participant
Prepare
Prepare
Force Write Prepared Record
Prepare
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Yes
Force Write Commit Record
Commit
Commit
Commit
Write non-forced Commit Record
Write non-forced Commit Record
Write non-forced Commit Record

(b) Commit case.

Figure 26 The rooted presumed commit (RPrC) protocol.

6.3.2 Recovery in RPrC Protocol

As in all other atomic commit protocols, site and communication failures are detected by *timeouts*. If the root coordinator times out while awaiting the vote of one of its direct descendants, the root coordinator makes an abort final decision, sends abort messages to all its direct descendants and wait for their acknowledgments to complete the protocol.

Similarly, if a cascaded coordinator times out while awaiting the vote of one of its direct descendants, it makes an abort decision. In this case, the cascaded coordinator force writes an abort log record, sends a "no" vote to its direct ancestor and abort messages to all its direct descendants and waits for their abort acknowledgments.

In the event of a leaf participant site failure, during its recovery process, the participant inquires its direct ancestor about the outcome of each prepared to commit transaction. In its inquiry message, the participant includes the identities of its ancestors recorded in the prepare log record. In this way, unlike the case of PrC, if the direct ancestor of the prepared participant does not remember the transaction, it uses the list of ancestors included in the inquiry message to inquire its own direct ancestor about the transaction's outcome rather than replying with a commit message by presumption. Eventually, either one of the cascaded coordinators in the path of ancestors will remember the transaction and provide a reply, or the inquiry message will finally reach the root coordinator. The root coordinator will respond with the appropriate decision if it remembers the outcome of the transaction or will respond with a commit decision by presumption. Once the participant receives the reply message, it enforces the decision and acknowledges it only if it is an abort decision.

In the event that the root coordinator fails, during its recovery process, the root coordinator records in its protocol table each transaction with an initiation record without a corresponding commit or end record. These transactions have not finished their commit processing by the time of the failure and need to be aborted. Thus, for each of these transactions, the coordinator sends an abort message to its direct descendants, as recorded in the initiation record, along with their lists of descendants in the transaction execution tree. The recipient of the abort message can be either a cascaded coordinator or a leaf participant. In the case of a cascaded coordinator, if it is in a prepared to commit state, the cascaded coordinator behaves as in the case of normal processing discussed above. Otherwise, it responds with a blind acknowledgment, indicating that it has already aborted the transaction. Similarly, if the abort message is received by a leaf participant, the participant behaves as in the case of normal processing if it is in a prepared to commit state or replies with a blind acknowledgment.

In the case of a cascaded coordinator failure, during its recovery process, the cascaded coordinator adds to its protocol table each *undecided* transaction (i.e., a transaction that has a prepare record without a corresponding final decision record) and each aborted transaction that has not been fully acknowledged (i.e., a transaction that has an abort log record without a corresponding end record) by its direct descendants prior to the failure. For each undecided transaction, the cascaded coordinator inquires its direct ancestor about the outcome of the transaction. As in the case of a leaf participant failure, the inquiry message contains the identities of all ancestors as recorded in the prepare record. Once the cascaded coordinator receives the final decision, it completes the protocol as in the normal processing case discussed above. For each aborted but not fully acknowledged transaction, the cascaded coordinator re-sends abort messages to its direct descendants and waits for all their acknowledgments before writing a non-forced end log record.

6.4 The Re-Structured Presumed Commit (ReSPrC) Protocol

In this section, we present ReSPrC which involves the restructuring of a multi-level transaction execution tree, and in particular, combining PrC with the *flattening* technique (8) to generate a *two-level transaction commit tree.*

The *re-structuring* of a transaction tree has been previously used to enhance the reliability of commit processing by reducing the blocking effects of atomic commit protocols in case of failures (30). Also, the *flattening* of a distributed transaction's tree has been suggested to reduce the cost of commit processing that is due to the serialization of messages in a transaction's tree (8). That is, instead of sending the coordination messages during commit processing in a sequential fashion from one process at one level to another at the next level in a transaction tree, the flattening technique allows the coordinator of the transaction to send messages directly to the participant processes without having to go through intermediate processes. This technique significantly reduces the cost of commit processing especially in deep trees (8). We propose to use the same technique to eliminate the initiation log records from cascaded coordinators and to reduce the cost that is associated with their serialization.

In ReSPrC, when the root coordinator receives a commit request from a transaction, it sends prepare to commit messages directly to all participants. To be able to communicate directly with all the participants, the root coordinator needs to know the identities of all participants. In ReSPrC, this is achieved in a manner similar to the one used in the RPrC. That is, each participant propagates its identity in the acknowledgment of the first operation it executes. Also, each participant needs to know the identity of the root coordinator to be able to communicate with root coordinator directly during the course of commit processing. This is achieved by having the direct ancestor of a participant to propagate the identity of the root coordinator in the first operation it forwards to the participant for execution. In this way, ReSPrC dynamically generates a two-level transaction commit tree for each transaction irrespective of the depth of the transaction's execution tree.

Thus, in addition to achieving our initial goal, that is reducing the number of initiation records in multi-level PrC, with ReSPrC we have enhanced the performance of commit processing in PrC in two ways. First, the forced log records in ReSPrC are performed in parallel rather than sequentially (e.g., the prepare log records). Second, we have reduced the total number of log writes. That is, a cascaded coordinator in ReSPrC neither force writes an initiation record nor writes an end record for an aborted transaction.

Furthermore, the use of the flattening technique provides some performance enhancement in the presence of *loopbacks* (50). A loopback occurs when a process, for example P_1 at site $Site_1$ creates another process P_2 at $Site_2$, which in turn creates P_3 back at $Site_1$. Assuming P_1 is a coordinator, by using ReSPrC, rather than communicating with P_3 though P_2 located at a different site, the coordinator communicates directly and locally with P_3 without the cost of having to exchange messages with P_3 via an external communication medium. In database systems where loopbacks are predominant, the performance enhancement becomes significant.

6.5 Evaluation of RPrC and ReSPrC Protocols

Although both ReSPrC and RPrC eliminate the initiation records of multi-level PrC from cascaded coordinators, ReSPrC is clearly more efficient than RPrC since ReSPrC allows for maximum parallelism during commit processing whereas RPrC suffers from the serialization of messages and forced log writes at each level of the commit tree.

However, ReSPrC cannot always be used. ReSPrC cannot be used in an environment where a participant is prohibited from directly communicating with the root coordinator or vice versa for security reasons. In general, ReSPrC also cannot be used when the communication topology does not support direct interaction between a root coordinator and the leaf participants. Similarly, the use of ReSPrC is limited when the establishment of new direct communication channels (i.e., sessions) between the coordinator and the participants are expensive and should be avoided as much as possible. A situation that exists in some commercial systems (6).

On the other hand, RPrC does not suffer from the applicability limitations of ReSPrC even for security reasons. Although RPrC requires the propagation of the participants' identities through the branches of the trees, by applying some basic encryption techniques to the identities of the participants, RPrC provides sufficient security to prohibit a participant from being able to identify the other participants. For example, if a key-based encryption technique is to be used, each cascaded coor-

Table 12 The costs associated with multi-level PrA, multi-level PrC, RPrC and ReSPrC protocols.

	Commit		Abort	
	Forced Log Writes	Messages	Forced Log Writes	Messages
PrA	$2L + 2C + 1$	$4N$	$L + C$	$3N$
PrC	$L + 2C + 2$	$3N$	$2L + 3C + 1$	$4N$
RPrC	$L + C + 2$	$3N$	$2L + 2C + 1$	$4N$
ReSPrC	$L + C + 2$	$3N$	$2L + 2C + 1$	$4N$

dinator in a transaction tree would use a different key to encipher the identity of its direct ancestor before propagating it to its direct descendants. Similarly, a cascaded coordinator enciphers the identities of its direct descendants, using the same key, before propagating them to its direct ancestor.

The flattening technique can also be applied to multi-level PrA resulting in *restructured PrA* (ReSPrA). When ReSPrC and ReSPrA are applicable, the tradeoff between them is reduced to the tradeoff between PrC and PrA as we discussed in Section 3.2.8.1. Similarly, the relative advantage of RPrC and multi-level PrA is reduced to the relative advantage of PrC and PrA. For instance, to illustrate the latter, consider N participants of which C are cascaded coordinator and L are leaf participants ($N = L + C$). As shown in Table 12, to commit a transaction in RPrC requires $L+C+2$ (or $N+2$) forced log writes whereas, to abort a transaction requires $2L + 2C + 1$ (or $2N + 1$) forced log writes, which is the same as in PrC. Thus, the decisive factor in selecting one over the other is the cost associated with read-only transactions which, as we show in the next section, can be efficiently handled using the *unsolicited update-vote* optimization.

6.6 The Unsolicited Update-Vote Optimization (UUV)

The cost associated with read-only participants can be reduced further if the coordinator of a transaction knows, at the end of a transaction and before the initiation of the commit protocol, which participants are read-only in the execution of the transaction. However, this does not mean, in general, that the coordinator can commit a transaction by sending a single decision message to each read-only participant and without voting as suggested in (7). Before presenting how this can be achieved, let us clarify this point with the following scenario. Note that here we do not assume any particular concurrency control protocol as we did in Chapter 4 when we presented the implicit yes-vote and implicit yes-vote with a commit coordinator protocols.

Assume two transactions T_1 and T_2, initiated at two different coordinators. T_1 is a (completely) read-only transaction that reads a data item x located at a participant site S_1 and a data item y located at site S_2 and then commits, whereas T_2 writes both data items x and y and then commits. A read (write) operation performed by transaction T_i on a data item x is denoted by $r_i[x]$ ($w_i[x]$) and the commit primitive of T_i is denoted by c_i.

$$T_1\text{: } r_1[x]\ r_1[y]\ c_1$$

$$T_2\text{: } w_2[x]\ w_2[y]\ c_2$$

Furthermore, assume that the first operation of T_2, $w_2[x]$, has been executed and acknowledged. Following that, the two operations of T_1 are executed and acknowledged. Then, T_2 submits its second operation, $w_2[y]$, to S_2 which is executed and acknowledged. Assuming that the coordinator of T_1 knows that the transaction has performed only read operations at both sites, the coordinator will send a c_1 to both S_1 and S_2. When c_1 is received by either S_1 or S_2, it would mean that T_1 has been terminated and it is the time to release the resources held by the transaction. If S_1 and S_2 are using an optimistic concurrency control protocol, the following execution histories are possible.

$$H_{S_1}\colon\ w_2[x]\ r_1[x]\ c_1\ c_2$$

$$H_{S_2}\colon\ r_1[y]\ w_2[y]\ c_1\ c_2$$

The global history of execution is neither serializable nor recoverable. It is not serializable because its serialization graph is cyclic (i.e., $T_2 \rightarrow T_1 \rightarrow T_2$) and non-recoverable because T_1 reads the value of x written by T_2 and commits before T_2 does. We have reached the above scenario because, using a single message without a reply, we have effectively prohibited the participants from validating the read-only transaction with respect to serializability and recoverability. Thus, sending a single termination message from the coordinator to a read-only participant without a reply vote from the participant, does not, generally, work since it might lead to inconsistencies.

Therefore, we need a method that allows a coordinator to determine which participants are read-only in a transaction's execution during run time without having to explicitly poll their votes, while still avoiding scenarios that might lead to inconsistencies similar to the one given above. This is the essence of the *unsolicited update-vote* (UUV) that we describe next.

6.6.1 Description of UUV Optimization

In UUV, when a transaction starts executing, its coordinator marks the transaction as a read-only one in its protocol table. Each time the transaction needs access to data at a new participant, the coordinator adds the identity of the participant to its protocol table and marks the participant as read-only before sending the request to the participant. When a participant executes the *first* update operation (which is recognized by the generation of undo/redo log record(s)) on behalf of the transaction, the participant sends an unsolicited update-vote to the coordinator. This is a flag that is set as part of the operation's acknowledgment to the coordinator. Hence, UUV piggybacks control information in the acknowledgment messages of the operations in order to determine update participants.

When the coordinator receives an unsolicited update-vote from a participant, it changes the status of the participant from read-only to update and resets the status of the transaction.

In the case that each participant site employs a pessimistic concurrency control protocol that also *avoids cascading abort* (5), such as strict two-phase locking (that we discussed in Chapter 2), a transaction is guaranteed to be *serializable* and *recoverable* after all its operations have been executed and acknowledged (5). Thus, the coordinator of a transaction is guaranteed that the transaction is serializable and recoverable at each read-only participant after the execution of each read operation. However, in the case that a participant employs an *optimistic* concurrency control protocol, this is not true and the participant has to validate the transaction before acknowledging each read operation as long as it has not already sent an unsolicited update-vote as part of a previous operation's acknowledgment.

When a transaction finishes its execution and submits its final commit request, the transaction's coordinator checks its protocol table to determine which participants have sent unsolicited update-vote as part of their operations' acknowledgments. For each participant that has sent an unsolicited update-vote, the coordinator knows that the participant is an update participant and sends to the participant a prepare to commit message. For each participant that has not sent an an unsolicited update-vote, the coordinator excludes the participant from voting by sending a read-only message indicating to the participant that the transaction has been terminated and it can release all the resources held by the transaction. When a read-only participant receives a read-only message, it releases all the resources held by the transaction without writing any log records.

In the next section, we discuss three other methods that we have considered to determine read-only participants without having to (explicitly) poll their votes.

6.6.2 Other Methods

For completeness, we discuss in this section three other methods that we have considered to determine read-only participants and point out their limitations.

1. By predeclaration in which each transaction indicates that it will perform only read operations (4).
2. By analyzing each submitted (high level) operation of each transaction.
3. By assuming that each participant knows when it has executed the last operation on behalf of a transaction. In this case, the participant sends its vote proclaiming itself as read-only in its own initiative once it recognizes that the transaction has no more operations to process.

The first method is very restrictive because transactions are written in an ad hoc fashion and their behavior cannot be determined a priori except in very special cases. The second method assumes that a coordinator is able to process and analyze high level operations as well as the return results from the participants. To realize this method requires expansion in the functionality of coordinators in current database management systems as opposed to UUV which requires the interpretation of a bit in acknowledgment messages. The third method assumes that the coordinator either submits to a participant all the operations at the same time, which is again a form of predeclaration, or indicates to the participant the last operation at the time the operation is submitted, as it is the case in the *unsolicited vote* optimization (35). As we discussed in Section 3.2.5, the latter is possible if each transaction has knowledge about data distribution and indicates to the coordinator the last operation to be executed at a participant.

6.7 Applying UUV to Presumed Commit Protocol variants

In this section, we apply UUV with PrC variants. In the next subsection we apply UUV to two-level PrC while in Subsection 6.7.2, we apply UUV to multi-level PrC, RPrC and ReSPrC. In both subsections, we compare the performance PrC variants when combined with UUV to the performance PrA variants when combined with the same optimization.

6.7.1 UUV with Presumed Commit Protocol

By combining UUV with PrC, a coordinator does not have to poll or wait for the votes of read-only participants. Therefore, for read-only transactions, UUV not only saves a message from each participant but it also eliminates the waiting time for all the votes to arrive and, hence acknowledges the transaction commitment earlier when compared with the traditional read-only optimization. For a partially read-only transaction, on the other hand, acknowledging the transaction commitment might become faster than the standard read-only optimization. This is possible, for example, in the case that some read-only participants are connected with the coordinator via low speed communication links while their update counterparts are connected with the coordinator via high speed communication links. In this case, the read-only participants become the bottleneck in the commit processing using the traditional read-only optimization. For this reason, a final decision pertaining to a partially read-only transaction is reached faster with fewer coordination messages by using UUV compared to the traditional read-only optimization. Hence, by combining UUV with PrC (similarly with ReSPrC) the cost associated with read-only transactions is cheaper than in PrA combined with the traditional read-only optimization.

The cost of PrA combined with UUV is the same as in PrC combined with UUV. This is because, using UUV, both PrA and PrC will incur the same coordination message complexities without any logging activities. Specifically, using the UUV, a coordinator that uses PrC should not force an initiation log record because it will know that the transaction is read-only by the time the transaction submits its final commit request. In this case, the coordinator discards any information pertaining to the transaction, acknowledges the commitment of the transaction and sends out a read-only final decision to each participant.

For partially read-only transactions, in the two-level transaction execution model, it is cheaper to use PrC with UUV if these transactions are most probably going to commit even though there is an extra forced log write at the coordinator's site (i.e., the initiation record). This is because PrC allows for a reduction of one forced log write (i.e., the commit decision record) and a message from each update participant. In addition, a read-only participant does not suffer from the cost associated with the forcing of the initiation record as it would have been the case if the traditional read-only optimization were used. Therefore, it is cheaper to use PrC with UUV even if there is only a single site where a transaction has submitted update operation(s) and will finally be committed.

6.7.2 UUV with Multi-Level Presumed Commit Protocol Variants

For a read-only transaction, neither the root coordinator nor any cascaded coordinator force writes initiation records for the transaction by using UUV with the multi-level PrC. Hence, the cost associated with commit processing of read-only transactions becomes the same in both multi-level PrA and multi-level PrC when they are combined with UUV while multi-level PrC combined with UUV is cheaper than multi-level PrA combined with the traditional read-only optimization.

For a partially read-only transaction, a cascaded coordinator in multi-level PrC has to send an unsolicited update-vote if any of its descendants has performed an update operation. Such a cascaded coordinator participates in the voting phase and force writes an initiation record. However, a leaf read-only participant does not suffer from the cost of forcing the initiation record at its direct ancestor. This is because the direct ancestor will send a read-only message to the participant without having to wait for the forced record to be in the stable log. Thus, none of the participants in a read-only branch in a transaction's execution tree will suffer from the cost of any initiation records if the whole branch up to the root coordinator is read-only.

By combining UUV with RPrC, the root coordinator of a read-only transaction also does not force write an initiation record. For a partially read-only transaction, since RPrC eliminates intermediate initiation records, a read-only participant will suffer from at most a single forced write (i.e., an initiation record at the root coordinator). Hence, the cost of commit processing for a committing, partially read-only transaction in RPrC is less than in multi-level PrA considering the saving in the total number of acknowledgment messages and the number of forced log writes at the participants. The savings in the number of acknowledgment messages and forced log writes are further magnified for update transactions. For example, there are N extra messages and $N - 1$ forced log writes in multi-level PrA compared with RPrC for a committing transaction where N is the number participants in the transaction tree excluding the root coordinator[1].

6.8 Summary

The presumed abort protocol (PrA) and the presumed commit protocol (PrC) are two competing two-phase commit variants. The former reduces the cost associated with aborting transactions while the latter reduces the cost associated with committing transactions. This makes only one variant appropriate at any given time depending on the behavior of transactions and the reliability of the distributed environment. Given the traditional networking environment and the behavior of transactions, the argument has been in favor of PrA rather than PrC. This is due to the cost of the forced initiation records associated with PrC even for read-only transactions. However, given the reliability characteristics of modern distributed environments and the high probability of a transaction of being committed rather than aborted after all its operations have been executed and acknowledged, we argued in favor of PrC by proposing two new PrC variants. Namely, *rooted PrC* (RPrC) and *re-structured PrC* (ReSPrC).

[1]Notice that the root coordinator force writes two log records in RPrC compared with one in multi-level PrA, hence we have $N - 1$ extra forced log writes in multi-level PrA.

Both RPrC and ReSPrC eliminate *all* intermediate initiation records from cascaded coordinators in the multi-level transaction execution model, which is the model adopted by the current transaction processing standards and commercial systems. Furthermore, regardless of the depth of a transaction's execution tree, there is at most a single forced initiation record in both variants compared to multi-level PrC, while the new PrC variants still maintain the low count in the total number of messages and forced log writes for a committing transaction compared to multi-level PrA.

For read-only transactions, we proposed a new read-only optimization that is called the unsolicited update-vote optimization and showed that the cost associated with read-only transactions in PrC and the newly proposed variants is *exactly* the same as in PrA. This is also true for read-only participants of partially read-only transactions in both PrC and PrA as well as ReSPrC. For a partially read-only transaction in RPrC, a read-only participant might suffer from the cost of a single forced initiation record at the root coordinator. In general, however, PrC variants involve lower number of coordination messages and total forced log writes compared with PrA variants when committing update as well as partially read-only transactions.

In conclusion, this work nullifies the basis for the argument that exclusively favors PrA, i.e., the low cost associated with read-only transactions and transactions in the multi-level transaction execution model, and makes the case that PrC should become part of future protocol standards. In addition, ReSPrC and RPrC as well as the unsolicited update-vote optimization provide sufficiently appealing efficiency characteristics that make them very attractive to be adopted in commercial systems that use a two-phase commit variant that also force writes initiation records such as the one's based on IBM SNA LU 6.2 architecture, the de facto standard of the industry (6).

In the next chapter, we further strengthen our argument by showing how we can interoperate database systems that uses PrN, PrA and PrC protocols in the context of multidatabase systems, despite their conflicting presumptions about the outcome of transactions and without violating the autonomy of the constituent database sites.

7.0 DEALING WITH INCOMPATIBLE PRESUMPTIONS OF TWO-PHASE COMMIT PROTOCOLS

In this chapter, we discuss the issue of compatibility among *atomic commit protocols* (ACPs) in a *multidatabase system* (MDBS) environment. In MDBSs, in which all local sites support an ACP, the incompatibility of the various ACPs may be due to the differences in the semantics of their coordination messages or actions. In the case of the three commonly known two-phase commit variants (namely, the presumed nothing (PrN), presumed abort (PrA) and presumed commit (PrC) protocols), as we show, the incompatibility arises because of their conflicting presumptions about the outcome of transactions in the presence of failures. Furthermore, we show that supporting a *visible prepare to commit state* in which a participant is prohibited from unilaterally committing or aborting a transaction after it has voted "yes", is not enough for a successful integration of ACPs in an operational fashion, because the outcome of some transactions might have to be remembered forever.

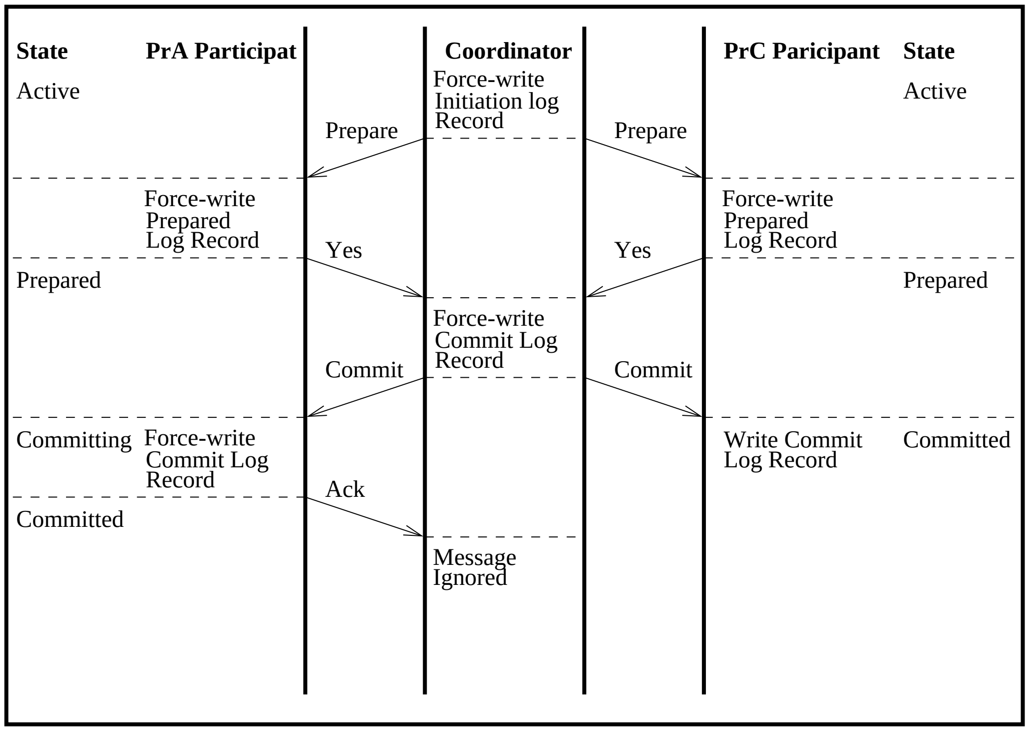
Thus, in order to be able to integrate these ACPs in a MDBS, we define an *operational* correctness criterion that allows terminated transactions to be forgotten. We propose *Presumed Any* (PrAny), a two-phase commit protocol variant that successfully integrates PrN, PrA and PrC protocols (78). We choose to integrate PrN and PrA because they have been widely implemented in existing systems, and PrC because it is expected to become part of the standards, as we argued in Chapter 6.

In the next section, we discuss the compatibility of PrN, PrA, and PrC, showing, by means of a protocol called *union two-phase commit protocol* (u2PC), that supporting a visible prepare to commit state is not sufficient for a practical integration of ACPs. In Section 7.2, we first derive another protocol from u2PC called u2PC* in which the coordinator ignores protocol violations due to "unexpected" (extra) messages but not due to "expected" (missing) messages. Even though u2PC* guarantees the atomicity of global transactions, we show that the coordinator has to remember the outcome of some transactions forever. Then, based on the two protocols, we distinguish *functional correctness* from *operational correctness* and express the practical requirements of a commit protocol with a criterion, called *operational correctness*. In Section 7.3, we present PrAny and prove its correctness with respect to this operational correctness criterion.

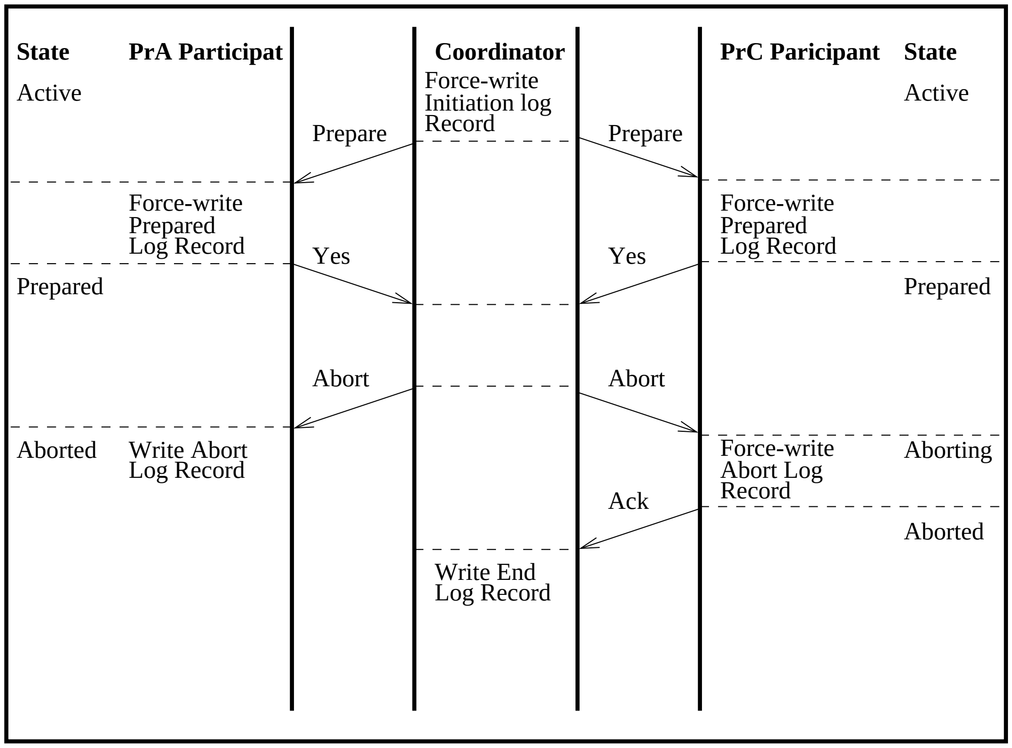
7.1 Compatibility of Two-Phase Commit Protocol Variants

In this section, we examine the compatibility of PrN, PrA and PrC by assuming that all three can coexist in a MDBS and that the global transaction manager (GTM) which is responsible for the atomic commitment of global transactions (see Section 3.2.1), can use any of the three variants. GTM also knows the protocol used by each participant. That is, the coordinator knows what messages to expect from each participant and interoperates with participants implementing a different protocol than its own protocol by ignoring message violations of its protocol. A protocol violation with respect to messages occurs when (1) GTM receives an unexpected type of a message from a participant, or (2) GTM does not wait to receive a type of a message from a participant whose protocol does not generate it. We call this integrated GTM protocol *union two-phase commit protocol* (u2PC), indicating the protocol used by the coordinator in parenthesis, e.g., u2PC(PrC) indicates that the coordinator employs PrC.

As shown in Figure 27 (a), let us consider the commitment of a transaction that has executed at two participants using u2PC. Assume that the coordinator and one of the participants employ PrC while the other participant employs PrA. The voting phase is the same in both variants. The only difference between the two variants, as far as the coordination messages are concerned, occurs in the decision phase. In the event that the coordinator of the transaction makes a commit final decision, in accordance with PrC, the coordinator does not expect any commit acknowledgment messages. However, the PrA participant will acknowledge the commit decision. By knowing that this participant will send an acknowledgment, the coordinator will not consider this message, although it is a violation of its protocol. With respect to



(a) Commit case.



(b) Abort case.

Figure 27 The union two-phase commit protocol (u2PC(PrC)).

the logging activities at the coordinator, once the coordinator makes the commit final decision, it forgets the transaction, discarding all information pertaining to the transaction from its protocol table and if necessary, garbage collects the transaction's log records. Since the coordinator employs PrC, it will always be able to respond to the inquiries of the participants, in case of a failure, with a commit final decision, using the PrC presumption.

Now, consider another transaction that has finished its execution at the same two participants and the coordinator has decided to abort the transaction, as shown in Figure 27 (b). In this case, the PrA participant never acknowledges an abort decision. Hence, the coordinator forgets the outcome of the transaction once it has received the acknowledgment of the PrC participant and without waiting for an acknowledgment from the PrA participant. This achieves atomicity as long as there are no failures. Otherwise, the atomicity of the transaction might be violated. For example, if the PrA participant fails after it has received the final outcome but before writing it in its stable log, the participant will inquire about the outcome of the transaction as part of its recovery procedure. If the coordinator has already received the acknowledgment from the PrC participant and forgotten about the transaction, the coordinator will wrongly respond with a commit final decision (using the PrC presumption) which clearly violates the atomicity of the transaction since the transaction has been aborted at the PrC participant.

A similar situation occurs if the coordinator employs PrN or PrA. However, in this case, the atomicity of committed transactions might be violated. All three situations can be generalized in the following theorem:

Theorem 2: It is impossible to ensure *global atomicity* in a MDBS using u2PC when transactions execute at both PrA and PrC participants.

Proof: We prove the above theorem by considering all three possible cases of GTM using PrC, PrA and PrN and a transaction executing on two participants, one using PrA and the other PrC. For each case, we provide an example that leads to an atomicity violation, showing that it is impossible to ensure global atomicity using u2PC.

Case I – u2PC(PrC): Assume that the GTM uses PrC. We have proven this case in our motivating example above.

Case II – u2PC(PrA): Assume that the GTM uses PrA and decides to commit a transaction. In this case, the PrA participant will acknowledge the commit decision but not the PrC participant. Hence, the GTM forgets the transaction once it receives the acknowledgment from the PrA participant. Now, it is possible for the PrC participant to fail before receiving the commit decision and for its inquiring message to arrive after the GTM has received the acknowledgment from the PrA participant and forgotten the transaction. In this case, the GTM will respond with an abort decision (by the PrA presumption) which violates the atomicity of the transaction.

Case III – u2PC(PrN): Assume the GTM uses PrN and the GTM decides to commit a transaction. This is similar to case II above since PrN uses an implicit PrA presumption if it does not remember a transaction. □

Clearly, u2PC might violate transaction atomicity because a coordinator prematurely forgets the outcome of transactions and is forced to respond with a decision using the presumption of its protocol to an inquiring message, although the semantics of the inquiring message assume a coordinator who uses an opposite presumption. In the next section, we consider a variation of u2PC that ensures atomicity but has the drawback that it has to remember the outcome of some transactions forever.

7.2 Operational Correctness

As mentioned in the previous section, u2PC cannot ensure transaction atomicity because a coordinator infers the outcome of a terminated transaction in the absence of any record about the transaction based on its own protocol presumptions, ignoring the potentially conflicting presumption of the participants. This suggests that a coordinator can avoid dealing with conflicting presumptions if it forgets a transaction only after receiving the necessary acknowledgments from all participants in accordance to its protocol. Such an integrated protocol can be derived from u2PC, denoted as u2PC*, by requiring a coordinator to wait for expected messages according to its protocol and only ignore protocol violations due to unexpected (extra) messages.

Let us consider the case of using u2PC*(PrC) in the example MDBS of the previous section, in which GTM uses PrC and a transaction executes on two participants one employing PrA and the other PrC. If the coordinator decides to abort the transaction, the coordinator will expect acknowledgment messages from both the participants according to u2PC*(PrC) before forgetting the transaction. But, a PrA participant never acknowledges an abort decision and hence the coordinator can never forget the transaction. Since the coordinator always remembers the outcome of the transaction in spite of failures, the coordinator will always be able to respond to any inquiring message with the correct outcome of the transaction. Thus, u2PC*(PrC) ensures the atomicity of a transaction.

Clearly, u2PC* is correct but not practical since a coordinator has to maintain a record of some transactions forever. This leads us to distinguish correctness of ACPs into *functional correctness* which only captures the atomicity–guarantee requirement and *operational correctness* which requires, in addition, that the coordinator and participants to be able to eventually forget about the outcome of terminated transactions.

Definition 1: The integration of different ACPs is *operationally correct* if and only if

1. The coordinator and all the participants reach consistent decisions regarding the outcome of transactions and regardless of failures.
2. The coordinator can, eventually, discard all the information pertaining to terminated transactions from its protocol table and garbage collect its log.

3. All participants can, eventually, forget about transactions and garbage collect their logs.

Theorem 3: It is impossible to achieve *operational correctness* if the coordinator is using u2PC* in the presence of transactions that execute at both PrA and PrC participants.

Proof: As in theorem 2, we prove this theorem by considering all three possible cases of GTM using PrC, PrA and PrN and a transaction executing on two participants, one using PrA and the other PrC. For each case, we provide an example that shows that the GTM needs to remember the outcome of a transaction forever.

Case I – u2PC*(PrC): Assume that the GTM uses PrC. We have proven this case in our example above, showing that, in u2PC*(PrC), the GTM has to remember the outcome of aborted transactions forever.

Case II – u2PC*(PrA): Assume that the GTM uses PrA and decides to commit a transaction. In this case, the PrA participant will acknowledge the commit decision but not the PrC participant. Hence, the GTM, in u2PC*(PrA), has to remember the outcome of committed transactions forever.

Case III – u2PC*(PrN): Assume the GTM uses PrN and the GTM decides to commit a transaction. This is similar to Case II above. That is, since the PrC participant will not acknowledge the commit decision and the GTM, in u2PC*(PrN), has to remember the outcome of committed transactions forever. Similarly, the GTM, in u2PC*(PrN), has to remember the outcome of an aborted transaction as in Case I since the PrA participant will not acknowledge an abort decision. □

Thus, to maintain operational correctness in a MDBS, a coordinator should be able to, eventually, forget the outcome of transactions without violating the consistency of its decisions. We call this a *safe state*. Intuitively, a coordinator is in a safe state with respect to a transaction if (1) it forgets a transaction after all participants have acknowledged its decision (as in PrN) or (2) it forgets a transaction without receiving acknowledgments from some participants but can use a single presumption that is consistent with the transaction's final outcome.

Realizing the first condition that satisfies the safe state concept requires modifying either PrA or PrC to acknowledge abort or commit decisions, respectively. This is indeed the case of the generalized presumed abort protocol that we discussed in Section 3.3.1 in which PrA protocol is modified to interoperate it with the IBM-PrN. However, this is not an acceptable solution because it violates the autonomy of the constituent local database systems in the multidatabase environment. For this reason, we propose a *safety criterion* that realizes the second condition of the safe state concept without modifying neither PrA nor PrC protocols, preserving the autonomy of local database management systems.

Definition 2: (Definition of safe state)
A coordinator is in a safe state with respect to the outcome of a global transaction G, if G has been aborted and only the presumed abort presumption holds, or G has been committed and only the presumed commit presumption holds.

The above safety criterion can be formally expressed in a first order predicate logic with a precedence relation (79, 80, 81). In the following definition, H represents the complete history of the execution of a transaction containing all the events pertaining to the transaction and indicating the (partial) order in which these events occur. The significant events for the definition of the safety criterion are: $Decide_C(Abort_G)$ which denotes that the coordinator C decides to abort a global transaction G and $Decide_C(Commit_G)$ which denotes that the coordinator decides to commit G. $DeletePT_C(G)$ denotes that the information pertaining to G is deleted from the protocol table of the coordinator. INQ_{g_i} denotes an inquiry message from a participant regarding a subtransaction g_i of G. $Respond_C(Outcome_{g_i})$ denotes the reply of the coordinator to the inquiry message. The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ *precedes* event ϵ' in H. It is false, otherwise.

Definition 3: (Formal definition of safe state)

$$
\begin{aligned}
&SafeState_C(G) \Rightarrow \\
&\qquad ((Decide_C(Abort_G) \in H \wedge \forall g_i \in G\ (DeletePT_C(G)) \rightarrow INQ_{g_i}) \Rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad Respond_c(Abort_{g_i}) \in H) \vee \\
&\qquad ((Decide_C(Commit_G) \in H \wedge \forall g_i \in G\ (DeletePT_C(G)) \rightarrow INQ_{g_i}) \Rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad Respond_C(Commit_{g_i}) \in H)
\end{aligned}
$$

Our safety criterion implies that some information including the outcome of transactions has to be remembered as long as more than one presumption is possible. In the next section, we present the presumed any (PrAny) protocol which implements the safety criterion. PrAny assumes that a coordinator knows the protocols used by the different LDBMSs and uses this knowledge to decide when to discard the information about a transaction.

7.3 The Presumed Any (PrAny) Protocol

In this section, we describe the PrAny protocol that integrates PrN, PrA and PrC according the operational correctness criterion that we have defined above. First, we describe PrAny during normal processing. Then, in Section 7.3.2, we discuss the recovery aspects of PrAny in the case of failures. Finally, in Section 7.3.3, we prove the correctness of the PrAny protocol.

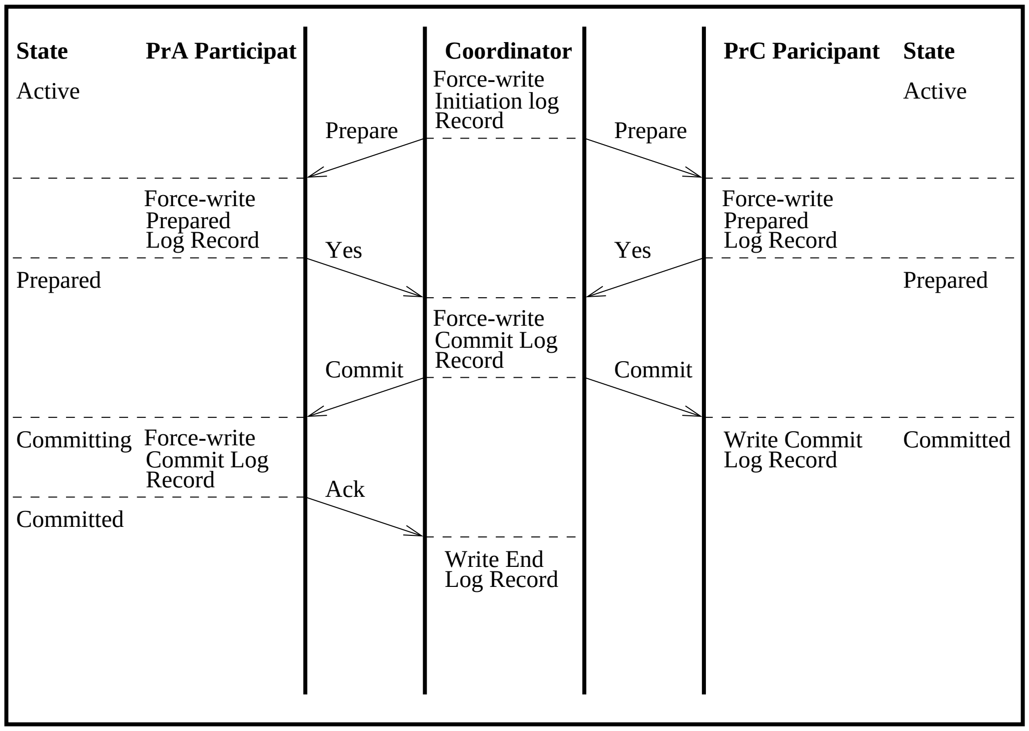
7.3.1 Description of the PrAny Protocol

In PrAny, a coordinator records the 2PC protocol employed by each participant with active transactions in a table called *active participant protocols* (APP) table that is maintained in main memory. The APP is updated either from a system table, called the *participants' commit protocol* (PCP), or based on responses from the agents of the participating sites.

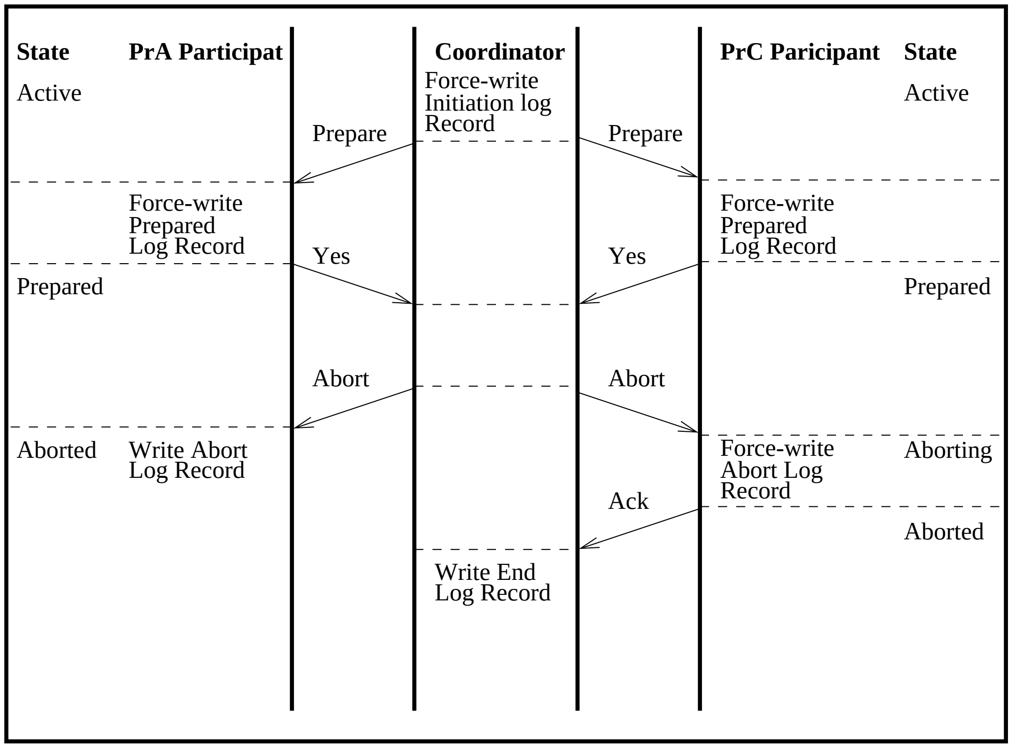
A coordinator refers to its APP to decide which protocol to use with the participants in the execution of a transaction. The coordinator selects PrN if all the participants use PrN. Similarly, it selects PrA if all the participants use PrA whereas it decides to use PrC if all the participants use PrC. By using PrN, PrA or PrC with all the participants, the coordinator will always be in a safe state if it does not remember the final outcome of a transaction.

In the event that some of the participants employ PrA while the others employ PrN or PrC, the coordinator selects PrAny. From the coordinator's perspective, PrAny consists of the same two phases, i.e., the voting phase and the decision phase, as in PrN, PrA and PrC, as shown in Figure 28. The only distinction between PrAny and the other 2PC variants is in the logging activities at the coordinator's site and the time at which the coordinator can safely forget about the outcome of transactions.

As shown in Figures 28 (a) and 28 (b), in PrAny, the coordinator starts the voting phase by force writing an initiation record which includes the identities of the participants, as is the case in the PrC variant. The initiation record also includes the protocol used by each participant. Then, the coordinator sends to each participant a prepare to commit request. Once the coordinator receives the votes from all the participants, it force writes a commit record if the decision is commit (Figure 28 (a)). If the decision is abort, no decision record is written into the log (Figure 28 (b)). Then, the coordinator sends its final decision to all the participants. On a commit final decision, the coordinator writes a non-forced end record once all the PrN and PrA participants acknowledge the decision. On an abort final decision, on the other hand, the coordinator writes an end record once all the PrN and PrC participants acknowledge the decision. After that, the coordinator writes an end record in its log and discards all information pertaining to the transaction from its protocol table.



(a) Commit case.



(b) Abort case.

Figure 28 The presumed any protocol.

7.3.2 Recovery in the PrAny Protocol

As in all other 2PC commit protocols, communication and site failures are detected by timeouts. The recovery procedure in case of communication and participants' failures are handled in a manner similar to the way they are handled in PrN, PrA and PrC protocols. According to the behavior of PrN, PrA and PrC, the coordinator expects those participants that employ PrN and PrA to acknowledge commit final decisions but not those participants that employ PrC (Figure 28 (a)). The coordinator can forget about the outcome of a committed transaction once the PrN and PrA participants acknowledge the commit decision, knowing that only a participant that employs PrC might inquire about the decision in the future. If a PrC participant inquires about a (commit) final decision after the coordinator has forgotten the transaction, the coordinator, knowing that the participant uses PrC, will direct the participant to commit the transaction, by the presumption of PrC.

Similarly, if a coordinator makes an abort final decision, it expects only those participants that employ PrN and PrC to acknowledge the decision but not those employing PrA (Figure 28 (b)). Hence, the coordinator can forget about the outcome of an aborted transaction once the PrN and PrC participants acknowledge the abort decision. If a PrA participant inquires about an (abort) final decision after the coordinator has forgotten the transaction, the coordinator, knowing that the participant uses PrA, will direct the participant to abort the transaction, by the presumption of PrA.

After a failure, at the beginning of its recovery procedure, the coordinator rebuilds its protocol table by analyzing its stable log. For each transaction that has a decision log record without an initiation record, it means that PrN or PrA has been used for its commitment. For each such transaction without an end record, the coordinator adds the transaction in its protocol table and re-initiates the decision phase with the recorded decision in the log. In the case of PrA, the decision is always commit since PrA requires only commit decisions to be recorded in the log. In the case of PrN, the decision could be either commit or abort.

For each transaction that has an initiation record, it means that PrC or PrAny has been used for its commitment. Depending on the identities of the participants recorded in the initiation record and the protocols that they use, the coordinator determines which of the protocol was used for the commitment of the transaction. For each such transaction that has used PrC for its commitment and has no commit or end log record, the coordinator adds the transaction in its protocol table and re-initiates the decision phase with an abort decision in accordance with PrC.

Finally, for each transaction that has used PrAny for its commitment and has only an initiation record, or has initiation and commit records but no end record, the coordinator adds the transaction in its protocol table. In the former case, since either no decision was made or abort was decided before the failure, the coordinator submits an abort decision to the PrN and PrC participants. It does not include the PrA participants in accordance with PrA[1]. In the latter case, since a commit decision record is found, the coordinator submits a commit decision to the PrN and PrA participants but, in accordance to PrC, not to PrC participants.

As during normal processing, after sending out a decision, the coordinator waits for acknowledgments from PrN and PrC participants in the case of an abort decision and from PrN and PrA participants in the case of a commit decision. When a participant receives a final decision, it enforces and acknowledges the decision if it has not already enforced the decision. Otherwise, the participant simply acknowledges the decision[2]. When all the expected acknowledgments arrive, the coordinator writes an end log record and forgets about the transaction.

[1]A coordinator in PrA never re-submits an abort decision to the participants after its failure because it will not have any recollection about aborted transactions. It is the responsibility of the participants to inquire about the outcome of such transactions. Similarly, a coordinator in PrC never re-submits commit decisions to the participants after its failure.

[2]A participant without any memory regarding a transaction is assumed to have already received and enforced the decision and discarded all information pertaining to the transaction.

7.3.3 Proof of Correctness

In Chapter 3, we thoroughly discussed the behavior of PrN and how it recovers after failures. That discussion provides a proof of correctness for PrN since we have exhaustively considered all possible cases of failures. That is, what would happen if a failure occurs and at what point during the course of PrN. We will use the same strategy to show the correctness of PrAny.

Theorem 4: The PrAny protocol satisfies the operational correctness criterion.

Proof:

To show the correctness of PrAny, we need to show that all the three requirements of operational correctness are satisfied. That is, (1) The coordinator and all the participants reach consistent decisions regarding the outcome of transactions and regardless of failures, (2) the coordinator can, eventually, discard all the information pertaining to terminated transactions from its protocol table and garbage collect its log and (3) all participants can, eventually, forget about transactions and garbage collect their logs, are satisfied.

PrAny consists of the same two phases as PrN. Hence, the first and the third requirements of the operational correctness criterion are satisfied since all participants will reach an agreement. The only remaining requirement that needs to be proven is the third one which requires that the coordinator should eventually be able to forget about the outcome of transactions. We prove the second requirement by considering the two possible outcomes of transactions. The proof proceeds by contradiction.

<u>Commit Case:</u> Assume that the coordinator has made a commit decision and after forgetting the outcome of the transaction, it replies to an inquiry message with an abort decision.

If the inquiring participant is PrC, then the coordinator will use the commit presumption of PrC and will respond with a commit decision which contradicts the initial assumption.

In order to reply with an abort, it means that coordinator has used the abort presumption. This means that the message is from a PrA participant, but this is impossible since all PrA and PrN participants must have acknowledged the commit decision in order for the coordinator to forget the outcome of the transaction. Similarly, it is impossible for the inquiry message to be from a PrN participant.

Abort Case: Assume that the coordinator has made an abort decision and after forgetting the outcome of the transaction, it replies to an inquiry message with a commit decision.

If the inquiring participant is PrA, then the coordinator will use the abort presumption of PrA and will respond with an abort decision which contradicts the initial assumption.

In order to reply with an commit, it means that the coordinator has used the commit presumption. This means that the message is from a PrC participant, but this is impossible since all PrC and PrN participants must have acknowledged the abort decision in order for the coordinator to forget the outcome of the transaction. Similarly, it is impossible for the inquiry message to be from a PrN participant. □

7.4 Summary

In this chapter, we showed that it is possible to integrate incompatible atomic commit protocols in a multidatabase system from a functional point of view as long as these protocols support a visible prepare to commit state. However, this result is not enough for practical integration because the outcome of some transactions might have to be remembered forever. Therefore, we defined an operational correctness criterion for integration that allows transactions to be forgotten.

Based on our proposed operational correctness criterion, we developed a multidatabase two-phase commit (2PC) protocol, called *Presumed Any* (PrAny), that integrates the *presumed nothing*, *presumed abort* and *presumed commit* 2PC variants despite their conflicting presumptions about the outcome of transactions and without violating the autonomy of the local database systems.

This chapter concludes the contributions of this dissertation. In the next chapter, we summarize our contributions, discuss our expected future work in the context of atomic commit protocols and conclude this dissertation.

8.0 SUMMARY AND CONCLUSIONS

Research in the area of database systems is one of the most active areas in computer science and technology. This is due to the fact that current and future application software systems require controlled access to data with enhanced reliability guarantees despite concurrency and failures. These guarantees are provided by database management systems that support the traditional ACID (i.e., atomicity, consistency, isolation and durability) transaction properties.

The atomicity property of distributed transactions can only be ensured with the use of an atomic commit protocol. Atomic commit protocols received extensive work in the late seventies to the mid eighties. After that, the database system industry took over and the standardization organizations picked, what was seemingly the best choice among the available atomic commit protocols at that time.

Due to the great impact of atomic commit protocols on the performance of any distributed database system, we were motivated to investigate this area given recent advances in hardware, software and network technology. The results of our investigations do not only contain theoretical characterizations, but also contain practical and well engineered atomic commit protocols.

In this chapter, we first summarize the contributions of this dissertation. Then, in Section 8.2, we discuss expected future work in the area of atomic commit protocols. In Section 8.3, we conclude this dissertation.

8.1 Summary

The focus of this dissertation was on *atomic commit protocols* (ACPs) in two distributed database environments, namely, distributed database systems (DDBSs)

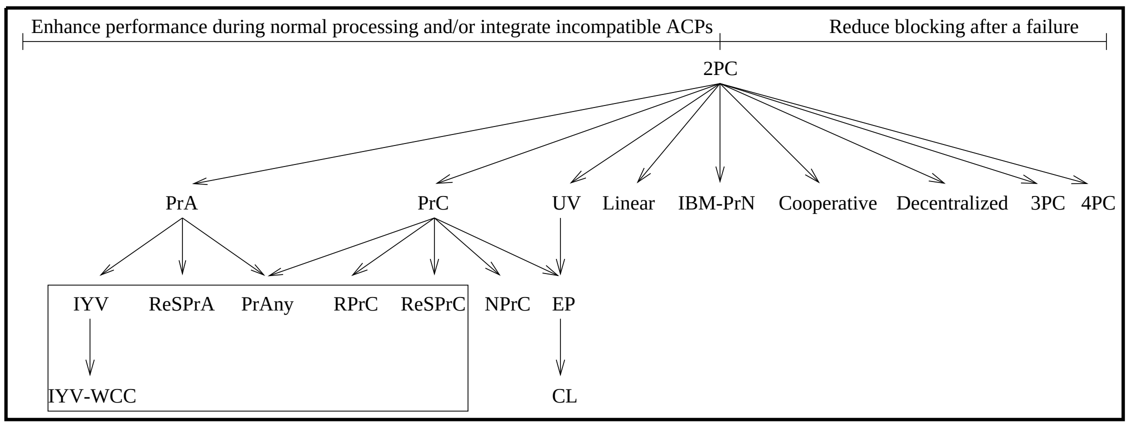

Enhance performance during normal processing and/or integrate incompatible ACPs
Reduce blocking after a failure
2PC
PrA
PrC
UV
Linear
IBM-PrN
Cooperative
Decentralized
3PC
4PC
IYV
ReSPrA
PrAny
RPrC
ReSPrC
NPrC
EP
IYV-WCC
CL

Figure 29 Our contributions in the evolution of ACPs.

and heterogeneous multidatabase systems (MDBSs). In the context of DDBSs, the contributions of this dissertation dealt with the issue of *performance* of ACPs whereas, in the context of MDBSs, its contributions dealt with the issue of *compatibility* of ACPs. Figure 29 depicts (in the area covered with a box), the new two-phase commit variants that this dissertation introduced. We list the four major contributions of this dissertation below.

1. *We developed two highly efficient ACPs by exploiting the characteristics of future gigabit-networked distributed database systems.*

The first protocol is called the *implicit yes-vote* (IYV) protocol which is a two-phase commit protocol variant that is targeted towards future distributed database systems (Chapter 4). In IYV, we exploit the semantics of the strict two-phase locking concurrency control protocol to eliminate the (explicit) voting phase of the two-phase commit protocol, and the characteristics of high-speed networks to facilitate recovery through our notion of *replicated write-ahead logging* (RWAL). In RWAL we replicate the redo parts of the participants' logs at the coordinator sites, given that the propagation latency is the dominant component of the overall communication cost in high-speed networks, while the migration of large amounts of data is not a problem.

Furthermore, after a participant failure, the IYV protocol allows partially executed transactions that are still active at other participants to resume their execution after the participant has recovered. We achieve forward recovery in IYV by replicating read locks. That is, when a transaction acquires read locks at a participant, the participant propagates the read locks to the transaction's coordinator. In this way, after a failure, a participant can re-acquire all read locks pertaining to a transaction from the transaction's coordinator and forward recover the transaction, if it is still active at other participants.

Since it is expected that some database sites will be less reliable than others, because they might use old technologies, we have also proposed *implicit yes-vote with delegation of commitment* (IYV-WCC) for such less reliable sites. IYV-WCC incorporates a novel coordination scheme that reduces the window of vulnerability of IYV to blocking and minimizes the time required for the sites to become operational after a failure. The new scheme combines the delegation of commitment technique with a timestamp synchronization mechanism that does not require global clock synchronization. Although this new scheme incurs extra coordination messages and log writes compared to IYV, it enhances the performance of IYV during recovery in the presence of less reliable sites. At the same time, it maintains the cost of commit processing during normal processing below that of two-phase commit and its other well known variants. We showed the performance of IYV and IYV-WCC and compared it with other ACPs using the traditional analytical method that is based on evaluating message, log and time complexities.

2. *We evaluated the performance of IYV and four other ACPs as well as read-only optimizations using simulation.*

In our simulation study, we explicitly modeled (1) the propagation latency of the communication network, (2) the overhead of the management of the database buffer and of flushing the transaction and protocol execution log records and (3) the overhead of recovery from site failures. By factoring in the effects of these aspects, we revealed the hidden overhead of the ACPs that we evaluated in our study. Consequently, our results reflect more accurately, compared to other simulation studies, the magnitude of performance differences on a system's performance when choosing one ACP versus another.

Salient results of our study show that IYV is, in general, better than all the other evaluated protocols during both normal processing and in the presence of one or two failed sites at any given time. IYV is matched by coordinator log (CL) protocol under some circumstances whereas, under others, CL is the worst among all the evaluated protocols. Interestingly, with respect to the two-phase commit variants, the choice of a protocol has very little impact on performance for the case of long transactions as opposed to short ones. Further, performance enhancements due to a read-only optimization are more pronounced with short transactions. Finally, we showed that there is a cross-over point between the performance curves of presumed abort and presumed commit protocols even under the assumption that all transactions are to be committed when they reach their commit points. This result cannot be shown using the traditional analytical method of performance evaluation which we used in Section 3.2.8.1, showing that presumed commit protocol is, in general, better than presumed abort protocol.

3. *We showed that it is possible to design ACPs and optimizations that scale well in future distributed database systems.*

Although IYV is a highly efficient ACP and can be extended to the more general multi-level transaction execution model, it is not suitable for future distributed database environments that require an explicit voting phase. Since future database systems are expected to have a high probability of transactions being committed rather than aborted, our investigations led us to revisit the presumed commit protocol in the context of the more general multi-level transaction execution model and to develop two new presumed commit protocol variants that are called *rooted presumed commit* (RPrC) and *re-structured presumed commit* (ReSPrC) (Chapter 6). Both RPrC and ReSPrC eliminate *all* intermediate initiation log records of the original presumed commit protocol at cascaded coordinators in the multi-level transaction execution model. Whereas RPrC is more general than ReSPrC with respect to applicability, RPrC is less efficient than ReSPrC.

For read-only transactions, we proposed a new read-only optimization that is called the unsolicited update-vote optimization and showed that the cost associated with read-only transactions in the original presumed commit protocol (PrC) and the newly proposed variants is *exactly* the same as in the presumed abort protocol (PrA). This is also true for read-only participants of partially read-only transactions in both PrC and PrA as well as ReSPrC. For a partially read-only transaction in RPrC, a read-only participant might suffer from the cost of a single forced initiation record at the root coordinator. In general, however, PrC variants involve lower message and log complexities compared to PrA variants, when committing update transactions as well as partially read-only transactions.

Furthermore, our work nullifies the basis for the classical argument that exclusively favors PrA, i.e., the low cost associated with read-only transactions and transactions in the multi-level transaction execution model, and makes the case that PrC should become part of future protocol standards. Our arguments in favor of PrC are further strengthened by the results of our simulation study that we presented in Chapter 5. Our results show that the choice of a two-phase commit variant has very little impact on performance in the case of long transactions as opposed to short ones. In the case of short transactions, our results show that the presumed commit protocol is, in general, better than presumed abort.

4. *We showed how to interoperate ACPs that have incompatible presumptions about the outcome of terminated transactions in the context of multidatabase systems.*

In Chapter 7, we showed that it is possible to integrate incompatible atomic commit protocols in a multidatabase system from a functional point of view as long as these protocols support a visible prepare to commit state. However, this result is not sufficient for a practical integration because the outcome of some transactions might have to be remembered forever. Therefore, we defined an operational correctness criterion for integration that allows transactions to be forgotten. Based on our operational correctness, we introduced the concept of a *safe state* that defines when a coordinator to be able to forget the outcome of terminated transactions.

Based on the characterization of the safe state concept, we developed a multidatabase two-phase commit (2PC) protocol, called *Presumed Any* (PrAny), that integrates the *presumed nothing*, *presumed abort* and *presumed commit* 2PC variants despite their conflicting presumptions about the outcome of transactions and without violating the autonomy of the local database systems.

Our research in the area of atomic commit protocols that ensures the traditional notion of transaction atomicity has highlighted some problems and provided answers to them. As in an any ongoing research, more needs to be done, and we believe that we have opened some areas that need further investigation. We list these areas in the following section.

8.2 Future Work

This dissertation contributed a number of solutions to the issues of performance and compatibility of ACPs in distributed database environments. Furthermore, it opened new areas that require future investigation, some of which constitute our plans for future work.

- As we mentioned at the end of Chapter 3, the implicit yes-vote protocol is not applicable in some distributed database systems such as systems that involve deferred consistency constraint validation that is performed at the commit time of transactions. These systems require an explicit voting phase as part of the prepare to commit state rather than an implicit one, which is the essence of IYV. Since IYV is a highly efficient ACP, as our results presented in Chapter 5 indicated, we need a method that makes IYV applicable in such systems while still retaining its performance enhancement when compared to the presumed abort and the presumed commit protocols that use an explicit voting phase.
- We have implemented a comprehensive simulation system to study the impact of ACPs on the overall performance of a distributed database system. As part of our future work, we expect to expand our studies to include the multi-level

transaction execution model. We would also like to conduct experiments on forward recovery to measure the magnitude of performance gain when using this notion.

- With respect to the traditional notion of atomicity of transactions, until recently, there has been no atomic commit protocol tailored for real-time distributed database systems (82). The search for an efficient real-time ACPs has just begun, and much work is needed to be done in this area. We have recently investigated the applicability of implicit yes-vote in real-time distributed database systems due to its efficiency characteristics (83). The efficiency metric in real-time database systems differs from the efficiency metric in non-real time distributed database systems. In the former environment, efficiency depends on the number of mission-critical transactions that are committed per unit time rather than the total number of transactions that are committed per unit time, which is the case in latter environment. Thus, the performance of implicit yes-vote protocol needs further investigation in this direction.

- With respect to integrated ACPs, we need to extend our safety criterion and to investigate tools and methods that will allow us to integrate other protocols.

- As mentioned above, the implicit yes-vote is not applicable in some systems. Similarly, our simulation results (Chapter 5) indicate that the presumed abort protocol is more efficient than the presumed commit protocol at low system loads. These two results reveal that there is no single ACP that can be considered the best with respect to performance and applicability even within the same environment. That is, one ACP might be better than another during some periods of time while the situation is reversed at other periods of time. Thus, as part of our future work, we intend to investigate *adaptive* ACPs that allow distributed database systems to switch from one protocol to another to enhance efficiency (78).

8.3 Conclusions

The contributions of this dissertation lead us to draw the following conclusions.

- By exploiting the semantics of the underlying communication networks as well as the operating and database management systems, we can develop highly efficient transaction processing protocols. For example, by factoring in the semantics of data, we can develop high performance concurrency control and recovery protocols. Similarly, by factoring in the semantics of concurrency control and recovery protocols adopted by a transaction management system as well as the characteristics of communication networks, we can develop highly efficient atomic commit protocols. Semantics-based techniques are promising for the development of highly efficient protocols in the context of distributed database systems.
- By conducting extensive experimental studies, we can reveal systems behavior that cannot be captured analytically. For example, experimental studies allow us to compare different techniques and point out their hidden overhead that degrades systems performance which cannot be captured analytically to determine either the relative or the absolute performance of the different techniques.
- By factoring in the state-of-the-art technology, a method that might not seem to be feasible at a given point in time, might become the ultimate option at another point in time. This means that the re-evaluation of all previous ideas is an essential first step before seeking new solutions to a problem. Furthermore, this has the implication that the database standards should evolve as older methods again become applicable and newer methods are introduced.

Based on the contributions of this dissertation, we believe that the area of atomic commit protocols, within the context of traditional atomicity of transactions, requires further and extensive investigations from the research community and we hope that this dissertation will provide the stimulus for further research and development in this important area of transaction processing systems.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Gray, J. "Notes on Data Base Operating Systems". In Bayer, R., Graham, R. M., and Seegmuller, G., editors, *Operating Systems: An Advanced Course, Lecture Notes in Computer Science.* Vol. 60, pp. 393–481, Springer-Verlag, Berlin, 1978.

[2] Lampson, B. W. "Atomic Transactions". In Lampson, B. W., Paul, M., and Siegert, H. J., editors, *Distributed Systems - Architecture and Implementation: An Advanced Course, Lecture Notes in Computer Science.* Vol. 105, pp. 246-265, Springer-Verlag, Berlin, 1981.

[3] Mohan, C. and Lindsay, B. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions". In *Proceedings of the 2nd ACM SIGACT/SICOPS Symposium on Principles of Distributed Computing*, pp. 76–88, August 1983.

[4] Lindsay, C. Mohan B. and Obermarck, R. "Transaction Management in the R^* Distributed Data Base Management System". *ACM Transactions on Database Systems*, Vol. 11, No. 4, pp. 378–396, December 1986.

[5] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Adison-Wesley, Reading, Massachusetts, 1987.

[6] Gray, J. N. and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, California, 1993.

[7] Lampson, B. and Lomet, D. "A New Presumed Commit Optimization for Two Phase Commit". In *Proceedings of the 19th International Conference on Very Large Databases*, pp. 630–640, Dublin, Ireland, August 1993.

[8] Samaras, G., Britton, K., Citron, A., and Mohan, C. "Two-Phase Commit Optimizations in a Commercial Distributed Environment". *Distributed and parallel Databases*, Vol. 3, No. 4, pp. 325–360, October 1995.

[9] Stamos, J. and Cristian, F. "Coordinator Log Transaction Execution Protocol". *Distributed and Parallel Databases*, Vol. 1, No. 4, pp. 383–408, 1993.

[10] Spiro, P., Joshi, A., and Rengarajan, T. K. "Designing an Optimized Transaction Commit Protocol". *Digital Technical Journal*, Vol. 3, No. 1, Winter 1993.

[11] Rosenkrantz, D. J., Stearns, R. E., and Lewis II, P. M. "System Level Concurrency Control for Distributed Database Systems". *ACM Transactions on Database Systems*, Vol. 3, No. 2, pp. 178–198, June 1978.

[12] Skeen, D. "Non-blocking Commit Protocols". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 133–147, Ann Arbor, Michigan, April 1981.

[13] Schwartz, P. M. and Spector, A. "Synchronizing Shared Abstract Data Types". *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 223–250, August 1984.

[14] Weihl, W. "Commutativity–Based Concurrency Control for Abstract Data Types". *IEEE Transactions on Computers*, Vol. 37, No. 12, pp. 1488–1505, December 1988.

[15] Herlihy, M. P. and Weihl, W. "Hybrid Concurrency Control for Abstract Data Types". In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pp. 201–210, March 1988.

[16] Badrinath, B. and Ramamritham, K. "Semantics–Based Concurrency Control: Beyond Commutativity". *ACM Transactions on Database Systems*, Vol. 17, No. 1, pp. 163–199, March 1992.

[17] Chrysanthis, P. K., Raghuram, S., and Ramamritham, K. "Extracting Concurrency from Objects: A Methodology". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 108–117, Denver, Colorado, May 1991.

[18] Liu, M., Agrawal, D., and Abbadi, A. El. "The Performance of Two-Phase Commit Protocols in the Presence of Site Failures". In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing Systems*, 1994.

[19] Gupta, R., Haritsa, J., and Ramamritham, K. "Revisiting Commit Processing in Distributed Database Systems". In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, (To Appear) May 1997.

[20] Braginski, E. "The X/Open DTP Effort". In *Proceedings of the 4th International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1991.

[21] Upton IV, F. "OSI Distributed Transaction Processing, An Overview". In *Proceedings of 4th International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1991.

[22] Gray, J. "The Transaction Concept: Virtues and Limitations". In *Proceedings of the 7th International Conference on Very Large Databases*, pp. 144–154, Cannes, France, September 1981.

[23] Haerder, T. and Reuter, A. "Principles of Transaction-oriented Database Recovery". *ACM Computing Surveys*, Vol. 15, No. 4, pp. 87–317, December 1983.

[24] Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. "Granularity of Locks and Degrees of Consistency in a Shared Database". In *Proceedings of the 1st International Conference on Very Large Databases*, pp. 25–33, Framingham, Massachusetts, September 1975.

[25] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. "The Notion of Consistency and Predicate Locks in a Database System". *Communications of the ACM*, Vol. 19, No. 11, pp. 624–633, November 1976.

[26] Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. "Granularity of Locks and Degrees of Consistency in a Shared Database". In Stonebraker, M., editor, *Readings in Database Systems*, pp. 94–121. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1988.

[27] Dijkstra, E. W. "Cooperating Sequential Processes". Technical Report EDW–123, Technological University, Eindhoven, The Netherlands, 1965.

[28] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". *ACM Transactions on Database Systems*, Vol. 17, No. 1, pp. 94–162, March 1992.

[29] Rothermel, K. and Pappe, S. "Open Commit Protocols for the Tree of Processes Model". In *Proceedings of the 10th International Conference on Distributed Computer Systems*, pp. 236–244, Paris, France, 1990.

[30] Rothermel, K. and Pappe, S. "Open Commit Protocols Tolerating Commission Failures". *ACM Transactions on Database Systems*, Vol. 18, No. 2, pp. 289–332, June 1993.

[31] Schlighting, R. and Schneider, F. "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems". *ACM Transactions on Computing Systems*, Vol. 1, No. 3, pp. 222–238, September 1983.

[32] LeLann, G. "Error Recovery". In Lampson, B. W., Paul, M., and Siegert, H. J., editors, *Distributed Systems: Architecture and Implementation - An Advanced Course, Lecture Notes in Computer Science.* Vol. 105, pp. 371– 376, Springer-Veralg, Berlin, 1981.

[33] Samaras, G., Britton, K., Citron, A., and Mohan, C. "Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment". In *Proceedings of the 9th International Conference on Data Engineering*, pp. 520–529, Vienna, Austria, February 1993.

[34] Samaras, G. and Nikolopoulos, S. "Algorithmic Techniques Incorporating Heuristic Decisions to Commit Protocols". In *Proceedings of the 21st Euromicro Conference*, Como, Italy, September 1995.

[35] Stonebraker, M. "Concurrency Control and Consistency of Multiple of Data in Distributed INGRES". *IEEE Transactions on Software Engineering*, Vol. SE–5, No. 3, pp. 188–194, May 1979.

[36] Stamos, J. and Cristian, F. "A Low-Cost Atomic Commit Protocol". In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, pp. 66–75, 1990.

[37] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., and Rasmussen, R. "The Gamma Database Machine Project". *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, pp. 44–69, 1990.

[38] Hammer, M. and Shipman, D. "Reliability Mechanisms for SDD–1: A System for Distributed Databases". *ACM Transactions on Database Systems*, Vol. 8, No. 4, pp. 431–466, December 1980.

[39] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D. "Implementation Techniques for Main Memory Database Systems". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1–8, Boston, Massachusetts, 1984.

[40] Gawlick, D. and Kinkade, D. "Varieties of Concurrency Control in IMS/VS Fast Path". *IEEE Database Engineering*, Vol. 8, No. 2, June 1985.

[41] Skeen, D. and Stonebraker, M. "A Formal Model of Crash Recovery in a Distributed System". *IEEE Transactions on Software Engineering*, Vol. SE–9, No. 3, pp. 219–228, May 1983.

[42] Cooper, E. "Analysis of Distributed Commit Protocols". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 175–183, Orlando, Florida, June 1982.

[43] Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. "Overview of Multidatabase Transaction Management". *VLDB Journal*, Vol. 1, No. 2, pp. 181–239, October 1992.

[44] Nodine, M. H. *Interactions: Multidatabase Support for Planning Applications.* PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island, May 1993.

[45] Mullen, J. G. *Atomic Commitment in Multidatabase Systems.* PhD thesis, Department of Computer Science, Purdue University, West Lafayette, Indiana, December 1993.

[46] Pu, C., Leff, A., and Chen, Shu-Wie F. "Heterogeneous and Autonomous Transaction Processing". *IEEE Computer*, Vol. 24, No. 12, pp. 64–72, December 1991.

[47] Tal, A. and Alonso, R. "Integration of Commit Protocols in Heterogeneous Databases". In *Proceedings of the International Conference on Information and Knowledge Management*, pp. 27–34, Baltimore, Maryland, November 1992.

[48] Tal, A. and Alonso, R. "Commit Protocols for Externalized–Commit Heterogeneous Databases". *Distributed and Parallel Databases*, Vol. 2, No. 2, pp. 209–234, April 1994.

[49] Gligor, V. D. and Lunckenaugh, Gary L. "Interconnecting Heterogeneous Database Management Systems". *IEEE Computer*, Vol. 17, No. 1, pp. 33–43, January 1984.

[50] Mohan, C., Britton, K., Citron, A., and Samaras, G. "Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols". In *Proceedings of the 5th International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1993.

[51] Samaras, G., Briton, K., Citron, A., and Mohan, C. "Enhancing SNA's LU 6.2 Sync Point to Include Presumed Abort Protocol". Technical Report TR# 29.1751, IBM, IBM Research Triangle Park, August 1993.

[52] Chrysanthis, P. K. and Ramamritham, K. "Autonomy Requirements in Heterogeneous Distributed Database Systems". In *Proceedings of the Conference on the Advances on Data Management*, pp. 283–302, December 1994.

[53] Muth, P. and Rakow, T. "Atomic Commitment for Integrated Database Systems". In *Proceedings of the 7th International Conference on Data Engineering*, pp. 296–304, Kobe, Japan, April 1991.

[54] Breitbart, Y., Silberschatz, A., and Thompson, G. "Reliable Transaction Management in a Multidatabase System". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 215–224, Atlantic City, New Jersey, May 1990.

[55] Breitbart, Y. and Silberschatz, A. "Strong Recoverability in Multidatabase Systems". In *Proceedings of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pp. 170–175, Phoenix, Arizona, February 1992.

[56] Breitbart, Y., Silberschatz, A., and Thompson, G. "Transaction Management Issues in a Failure-Prone Multidatabase System Environment". *VLDB Journal*, Vol. 1, No. 1, pp. 1–39, July 1992.

[57] Mehrotra, S., Rastogi, R., Breitbart, Y., Korth, H., and Silberschatz, A. "Ensuring Transaction Atomicity in Multidatabase Systems". In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 164–175, June 1992.

[58] Soparkar, N., Korth, H., and Silberschatz, A. "Failure–Resilient Transaction Management in Multidatabases". *IEEE Computer*, Vol. 24, No. 12, pp. 28–36, December 1991.

[59] Wolski, A. and Veijalainen, J. "2PC Agent Method: Acheiving Serializability in Presense of Failures in a Heterogeneous Multidatabase". In *Proceedings of the IEEE PARBASE*, pp. 321–330, Miami Beach, Florida, March 1990.

[60] Wolski, A. and Veijalainen, J. "2PC Agent Method: Acheiving Serializability in Presense of Failures in a Heterogeneous Multidatabase". In Rishe, N., Navathe, S., and Tal, D., editors, *Databases: Theory, Design, and Applications.* pp. 268–287, IEEE Computer Society Press, Los Alamitos, California, 1991.

[61] Georgakopoulos, D. *Transaction Management in Multidatabase Systems.* PhD thesis, Department of Computer Science, University of Houston, Houston, Texas, December 1990.

[62] Georgakopoulos, D. "Multidatabase Recoverability and Recovery". In *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, pp. 348–355, Kobe, Japan, April 1991.

[63] Barker, K. and Ozsu, M. T. "Reliable Transaction Execution in Multidatabase Systems". In *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, pp. 344–347, Kobe, Japan, April 1991.

[64] Mullen, J. G., Jing, J., and Sharif-Askary, J. "Reservation Commitment and its use in Multidatabase Systems". In *Proceedings of the 4th IEEE International Conference on Database and Expert Systems Applications (DEXA)*, pp. 116–121, Prague, Czech Republic, September 1993.

[65] Mehrotra, S., Rastogi, R., Korth, H., and Silberschatz, A. "A Transaction Model for Multidatabase System". In *Proceedings of the International Conference on Distributed Computing Systems*, pp. 56–63, June 1992.

[66] Al-Houmaily, Y. J. and Chrysanthis, P. K. "Two–Phase Commit in Gigabit-Networked Distributed Databases". In *Proceedings of the 8th ISCA International Conference on Parallel and Distributed Computing Systems*, pp. 554–560, Orlando, Florida, September 1995.

[67] Al-Houmaily, Y. J. and Chrysanthis, P. K. "An Atomic Commit Protocol for Gigabit-Networked Distributed Databases". *Journal of Systems Architecture, The EUROMICRO Journal*, (To Appear) 1997.

[68] Kleinrock, L. "The Latency/Bandwidth Tradeoff in Gigabit Networks". *IEEE Communications Magazine*, Vol. 30, No. 4, pp. 36–40, 1992.

[69] Banerjee, S., Li, V., and Wang, C. "Distributed Database Systems in High-Speed Wide-Area Networks". *IEEE Journal on Selected Areas in Communications*, Vol. 11, No. 4, pp. 617–630, May 1993.

[70] Banerjee, S. and Chrysanthis, P. K. "Data Sharing and Recovery in Gigabit-Networked Databases". In *Proceedings of the 4th International Conference on Computer Communications and Networks*, September 1995.

[71] Al-Houmaily, Y. J. and Chrysanthis, P. K. "The Implicit Yes-Vote Commit Protocol with Delegation of Commitment". In *Proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems*, pp. 804–810, Dijon, France, September 1996.

[72] Al-Houmaily, Y. J., Conticello, R., and Chrysanthis, P. K. "Performance of Atomic Commit Protocols in Gigabit-Networked Database Systems". Technical Report TR-97-15, Department of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania, March 1997.

[73] Agrawal, R., Carey, M., and Livny, M. "Concurrency Control Performance Modeling: Alternatives and Implications". *ACM Transactions on Database Systems*, Vol. 12, No. 4, pp. 609–654, December 1987.

[74] Carey, M. and Livny, M. "Distributed Concurrency Control Performance: A Study of Algoritms, Distribution, and Replication". In *Proceedings of the 14 International Conference on Very Large Data Bases*, pp. 13–25, Los Angeles, California, August 1988.

[75] Carey, M. and Livny, M. "Parallelism and Concurrency Control in Distributed Database Machines". In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 122–133, Portland, Oregon, June 1989.

[76] Al-Houmaily, Y. J., Chrysanthis, P. K., and Levitan, S. P. "An Argument in Favor of the Presumed Commit Protocol". In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pp. 255–265, Birmingham, U. K., April 1997.

[77] Al-Houmaily, Y. J., Chrysanthis, P. K., and Levitan, S. P. "Enhancing the Performance of Presumed Commit Protocol". In *Proceedings of the 12th ACM Annual Symposium on Applied Computing, Special Track on Database Technology*, pp. 131–133, San Jose, California, March 1997.

[78] Al-Houmaily, Y. J. and Chrysanthis, P. K. "Dealing with Incompatible Presumptions of Commit Protocols in Multidatabase Systems". In *Proceedings of the 11th ACM Annual Symposium on Applied Computing, Special Track on Database Technology*, pp. 186–195, Philadelphia, Pennsylvania, February 1996.

[79] Chrysanthis, P. K. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, September 1991.

[80] Chrysanthis, P. K. and Ramamritham, K. "A Formalism for Extended Transaction Models". In *Proceedings of the 17th International Conference on Very Large Databases*, pp. 103–112, Barcelona, Spain, September 1991.

[81] Chrysanthis, P. K. and Ramamritham, K. "Synthesis of Extended Transaction Models Using ACTA". *ACM Transactions on Database Systems*, Vol. 19, No. 3, pp. 450–491, September 1994.

[82] Gupta, R., Haritsa, J., Ramamritham, K., and Seshadri, S. "Commit Processing in Distributed Real-Time Database Systems". In *Proceedings of the 17th Real-Time Systems Symposium*, pp. 220–229, Washington, D.C., December 1996.

[83] Al-Houmaily, Y. J. and Chrysanthis, P. K. "In Search for an Efficient Real-Time Atomic Commit Protocol". In *Proceeding of the 17th Real-Time Systems Symposium on Work in Progress*, pp. 51–54, Washington, D.C., December 1996.