Chapter 13

Recovery and Performance of Atomic Commit Processing in Distributed Database Systems

P. K. Chrysanthis, G. Samaras, and Y. J. Al-Houmaily

Abstract

A transaction is traditionally defined so as to provide the properties of atomicity, consistency, integrity, and durability (ACID) for any operation it performs. In order to ensure the atomicity of distributed transactions, an atomic commit protocol needs to be followed by all sites participating in a transaction execution to agree on the final outcome, that is, commit or abort. A variety of commit protocols have been proposed that either enhance the *performance* of the classical *two-phase commit* protocol during normal processing or reduce the cost of *recovery* processing after a failure. In this chapter, we survey a number of two-phase commit variants and optimizations, including some recent ones, providing an insight in the performance trade-off between normal and recovery processing. We also analyze the performance of a representative set of commit protocols both analytically as well as empirically using simulation.

13.1 INTRODUCTION

Transactions are powerful abstractions that facilitate the structuring of database systems, and distributed systems in general, in a reliable manner. Each transaction represents a task or a logical function that involves access to a shared database and

assumes that it executes as if no other transactions were executing concurrently and as if there were no program and system failures. In this way, programmers are relieved from dealing with the complexity of concurrent programming and failures, and can focus on designing the applications and developing correctly the individual transactions of the applications.

A transaction provides reliability guarantees by implementing a state transformation with four important properties, commonly known as ACID properties [23, 27, 24]: (1) *atomicity*, (2) *consistency*, (3) *isolation* and (4) *durability*. *Atomicity* ensures that either all or none of the transaction's operations are performed. Thus, all the operations of a transaction are treated as a single, indivisible, atomic unit. *Consistency* requires that a transaction maintains the integrity constraints on the database. *Isolation* demands that a transaction executes (as though) without any interference from other concurrent transactions. *Durability* ensures that all the changes made by a successfully terminated (committed) transaction become permanent in the database, surviving any subsequent failures.

The ACID properties are usually ensured by combining two different sets of algorithms. The first set, referred to as *concurrency control protocols*, ensures the isolation property, whereas the second one, referred to as *recovery protocols*, ensures atomicity and durability properties. Commonly, consistency is satisfied by designing transactions such that each transaction preserves the consistency of the database at its boundaries and is enforced by specifying integrity constraints on a database using *triggers* and *alerters*.

In a *distributed database system* (DDBS) in which the data items are stored at multiple sites interconnected via a communication network, transactions are executed in a distributed fashion at different sites based on the location of the data that they require to access. Since sites and communication links can fail independently, the atomicity property of a distributed transaction cannot be guaranteed without taking additional measures besides concurrency control and recovery protocols. Specifically, for a distributed transaction that executes across multiple sites, the sites need to agree about *when* and *how* the transaction should terminate. That is, all the sites participating in a transaction execution need to (1) *eventually* reach an agreement; and (2) all agree to either *commit* the transaction, making all its effects persistent, or *abort* the transaction, obliterating all its effects as if the transaction had never executed. A protocol that achieves this kind of agreement is called an *atomic commit protocol* (ACP).

Three important performance issues are associated with ACPs.

- *Efficiency during Normal Processing*: This refers to the cost of an ACP to provide atomicity in the absence of failures. Traditionally, this is measured using three metrics [10, 38, 39]. 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

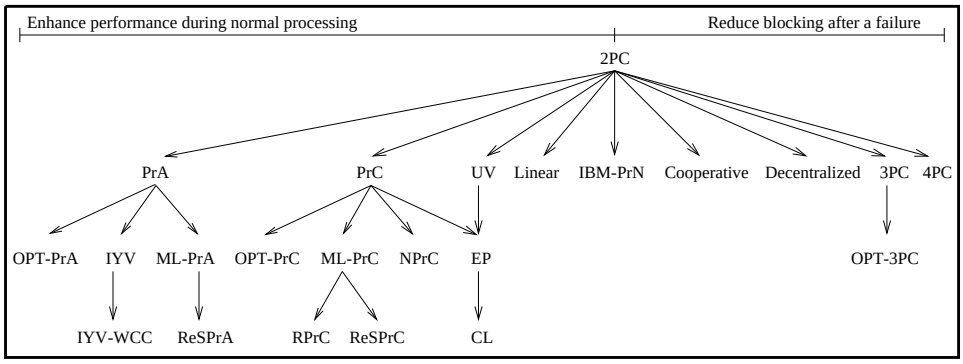


Figure 13.1. Significant steps in the evolution of ACPs.
2PC: Two-Phase Commit [1978], Linear 2PC [1978], UV: Unsolicited-vote [1979], Decentralized 2PC [1981], 3PC: Three-Phase Commit [1981], 4PC: Four-Phase Commit[1981], Cooperative 2PC [1981], PrA: Presumed Abort [1983], ML-PrA: multi-level PrA [1983], PrC: Presumed Commit [1983], ML-PrC: multi-level PrC [1983], IBM-PrN: IBM's 2PC [1990], EP: Early Prepare [1990], CL: Coordinators Log [1990], NPrC: New PrC [1993], IYV: Implicit YES-Vote [1995], IYV-WCC: IYV with Commit Coordinator [1996], RPrC: Rooted PrC [1997], ReSPrC: Restructured PrC [1997], OPT: Optimistic ACPs [1997].

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 or sequential exchanges of messages that are required in order for a decision to reach the participants.

- *Resilience to Failures:* This refers to the failures an ACP can tolerate and the effects of failures on operational sites. A ACP is *non-blocking* if it allows transactions being processed at a failed site to be terminated at the operational sites without waiting for the failed site to recover [55, 24].

- *Independent Recovery*: This refers to the speed of recovery, that is, the time required for a site to recover its database and become operational accepting new transactions after a system crash. A site can *independently* recover, if it has all the necessary information for recovering stored locally (in its log) and does not require any communication with any other site.

The *two-phase commit* protocol (2PC) [22, 32] is the first proposed and simplest ACP. The basic idea behind the design of all other ACPs is to enhance either (1) the efficiency of the 2PC for the normal processing case or (2) the reliability of 2PC by either reducing 2PC's blocking aspects or enhancing the degree of independent recovery. In this chapter, we survey a number of key ACPs and discuss the rationale behind their designs, hence tracing the significant steps in the evolution of ACPs

(Figure 13.1). In a way, 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 in what follows, 2PC variants make conflicting assumptions and hence, only a single 2PC variant can be used in a DDBS with any number of compatible optimizations. By this, we do not imply that conflicting 2PC variants cannot be made to interoperate [4, 66].

Although message complexity, log complexity and time complexity are very useful in analyzing the best and worst case performance behavior of ACPs, they neither can be used to completely characterize the performance of some protocols (e.g., Coordinator Log (CL) [60, 61], Implicit Yes-Vote (IYV) [3] and the set of OPT ACPs [26]) nor are able to capture the impact of ACPs on the *overall* performance of a system. To illustrate this, we present the results of a recent simulation study that shows the relative performance differences among CL, IYV, 2PC and two of 2PC's common variants in a distributed database system over a wide-area network [9, 2]. This study has been motivated by the fact that, in the near future due to the recent advances in networking technologies, it is expected that very large distributed databases systems will be deployed over high-speed wide-area networks. The study also captures the performance gains when read-only optimizations are used.

The rest of this chapter is structured as follows. After introducing the different distributed database system models in the next section, we discuss the basic 2PC in section 13.3. *Presumed-abort* (PrA) and *presumed commit* (PrC) are the best known 2PC variants. In section 13.4, we focus on PrA-based and PrC-based ACPs and analytically evaluate and compare them. In section 13.5, we discuss six more 2PC variants developed with specific environments in mind whereas in section 13.6, we discuss the six most significant optimizations that reduce the cost of commit processing. In section 13.7, we deal with the issue of blocking and present ways to reduce its negative effects. In section 13.8, we present an ACP simulator and discuss the results of the study of the five ACPs mentioned above with regard to their impact on transaction throughput. This chapter ends with some concluding remarks.

13.2 DISTRIBUTED DATABASE SYSTEM MODEL

In a distributed database environment, each transaction is associated with a *coordinator* that is responsible for coordinating the different aspects of the transaction execution. In a *client-server* environment, the coordinator of a transaction is assumed to be the *transaction manager* at the site where the transaction has been initiated. For example, in Figure 13.2, 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,

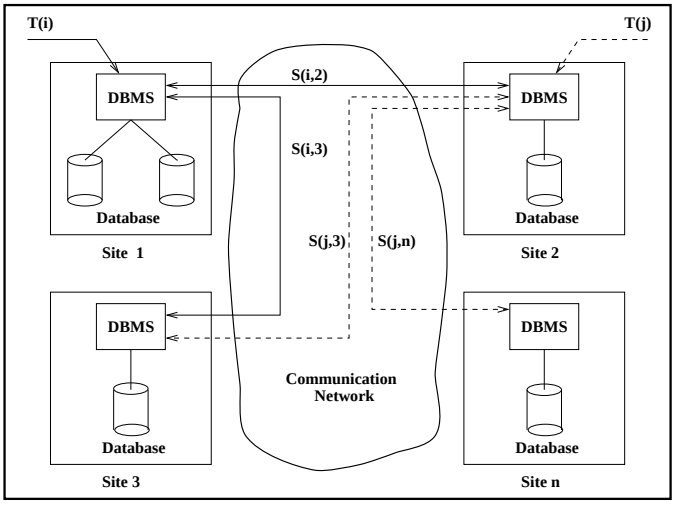


Figure 13.2. A distributed database environment.

the coordinator determines the appropriate *participant* site to which it submits each data operation received 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 $S_{i,1}$ executing at the site where T_i has been initiated and $S_{i,2}$, and $S_{i,3}$ sent for execution to sites 2 and 3, respectively. When a transaction finishes its execution and submits its commit request, its coordinator initiates an ACP that is carried out by the transaction managers at the participanting sites, to ensure that all its subtransactions either commit or abort at all sites.

In the standards and in a number of commercial database systems, the basic two-level distributed transaction execution model is generalized to a multi-level transaction execution model, called the *tree-of-processes* model [39]. In this model, a transaction manager at a participating site can decompose a subtransaction further into new subtransactions to be executed at its site or different sites. Hence, a distributed transaction can be represented by a multi-level execution tree where the coordinator resides at the root of the tree. The interactions between the coordinator and any participant transaction manager TM_i have to go through all the intermediate participants which are called *cascaded coordinators* between the coordinator and TM_i.

Each transaction manager maintains a *protocol table* in its main memory. For each (sub)transaction at its site, it records the identities of the sites that need to participate in the commitment of the transaction and the progress of the protocol once it is initiated. The protocol table also enables a root or cascaded coordinator to respond to any inquiries pertaining to a transaction very quickly. In order to be

able to recover after failures, part of the information in the protocol table is also recorded in a log on a stable storage (*stable log*) that survives system failures. This includes the identities of the sites that might need to be contacted during recovery and the different states of the ACPs in progress.

An alternative to the *client-server* environment is the *peer-to-peer* environment. In this environment, any participant in the transaction can decide to initiate the commit protocol and thus become the coordinator at the root of the transaction commit tree [50].

All ACPs discussed in this chapter with the exception of *IBM's presumed nothing* protocol (IBM-PrN) [48, 50] (Section 13.7.2), are only targeted for a client-server environment. Further, unless otherwise stated, all ACPs are based on the assumptions that (1) each site is *sane* [46, 47] and (2) each site can cause only *omission* failures. That is, each site is assumed to be *fail stop* [53] where it never deviates from the specification of the protocol that it is using, and when it fails, it will, eventually, recover.

13.3 THE BASIC TWO-PHASE COMMIT PROTOCOL

The basic *two-phase commit* protocol (2PC) [22, 32], as the name implies, consists of two phases, namely, a *voting phase* and a *decision phase* (Figure 13.3). During the voting phase, the coordinator of a distributed transaction requests that all sites participating in the transaction's execution *prepare-to-commit*, whereas, during the decision phase, the coordinator either decides to commit the transaction if *all* participants are *prepared to commit* (voted "yes") or to abort if any participant has decided to abort (voted "no"). On commit decision, the coordinator sends out a commit message to all participants whereas on abort decision, it sends out abort messages only to those participants which voted "yes".

When a participant receives a "prepare" request for a transaction, it validates the transaction with respect to data consistency (e.g., executes all integrity constraint evaluations deferred for commit time). If the transaction is validated, then the participant replies with a "yes" vote; otherwise it sends a "no" vote and aborts the transaction releasing all the resources held by the transaction. Even when a participant votes "yes", it may release some of the resources held by the transaction, in particular those not needed for rolling back the transaction in case of an abort decision (e.g., read locks).

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 with the decision, *acknowledges* the decision, and releases all the resources held by the transaction.

When the coordinator receives acknowledgments from all theparticipants that voted "yes", it completes the protocol and *forgets* the transaction by discarding all information pertaining to the transaction from its protocol table.

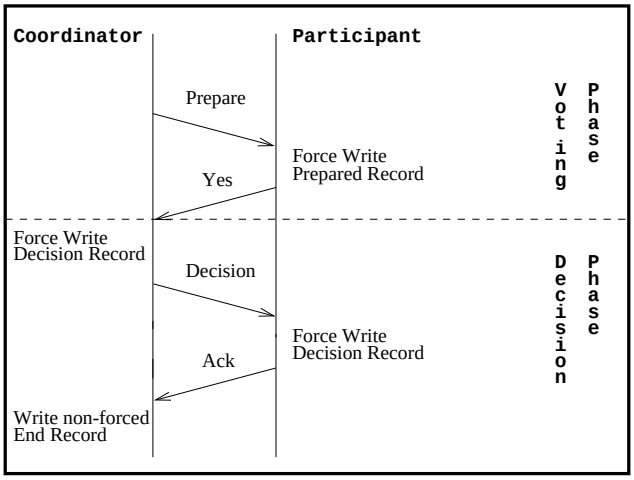


Figure 13.3. The basic two-phase commit protocol.

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 "yes" vote and force writes a *decision* record before acknowledging a final decision. In the case that different logs are used to record update operations and the progress of commit protocols, a participant must force all update log records pertaining to the transaction on a stable log before force writing a *prepare* log record. 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 from the stable log when necessary. Note that the end log record as well as the "acknowledgment" message is required for resource management purposes.

From the above it follows that, during normal processing, the cost to commit or to abort a transaction executing at n participants is the same. 2PC requires $2n+1$ forced log writes (log complexity), one forced write at the coordinator's site and two at each participant, and $4n$ messages (message complexity). The time complexity of 2PC is 3 rounds, first one is the sending of "prepare" requests, second one is the sending of votes and third one is the sending of decision messages.

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

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

decision regarding the transaction and enforce it. Now, let us consider the behavior of 2PC in the case of communication and site failures.

13.3.1 Recovery in 2PC 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" message from the coordinator. Since this can occur before the participant has voted, the participant may unilaterally decide to abort if it times out. 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 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 reestablishes communication with the coordinator. The fourth situation is when the coordinator is waiting for the acknowledgments of the participants. In this case, the coordinator resubmits its final decision to those participants that have not acknowledged the decision once it reestablishes 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 rebuilds 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 prior to the failure (that is, transactions associated with decision log records without corresponding end log records in the stable log). Once the coordinator rebuilds its protocol table, it completes the protocol for each of these transactions by resubmitting its final decision to all the participants whose identities are recorded in the decision record and are 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 (that is, has a prepared log record without a corresponding final decision log record). For each prepared-to-commit transaction, the participant reestablishes communication with the transaction's coordinator and inquires it about its final decision. When the participant receives the final decision from the coordinator, it enforces

the decision and completes the protocol by acknowledging the coordinator. Once the participant recovers the database, it re-acquires the locks of the prepared-to-commit transactions (in accordance to their write log records) and resumes normal transaction processing.

13.4 ENHANCING THE PERFORMANCE OF THE 2PC PROTOCOL

By using today's technology, a disk access requires 10 to 20 milliseconds, whereas the propagation latency of a message from one site to another is typically of the order of hundreds of milliseconds in 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. Because of this, 2PC has been found to consume a substantial amount of a transaction's execution time during normal processing [59]. 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 this section, we discuss the two most notable 2PC variants, namely, *presumed abort* (PrA) and *presumed commit* (PrC) that reduce the message complexity of 2PC by eliminating a single message from each participant site for aborting and committing transactions, respectively. Our discussion also includes the *new PrC* and *rooted PrC* protocols.

13.4.1 The Presumed Abort Protocol

The basic 2PC protocol is also referred to as the *presumed nothing* 2PC (PrN) protocol [33] 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 makes the abort presumption of PrN explicit in order to reduce the cost associated with aborted transactions further [38, 39]. 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 13.4). After a coordinator

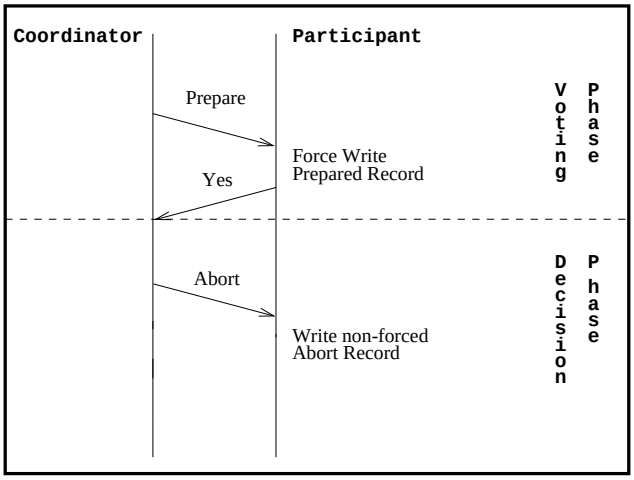


Figure 13.4. The presumed abort 2PC protocol (abort case).

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.

The PrA can be extended in the tree-of-processes model as follows. The behavior of the root coordinator and each leaf participant in the transaction execution tree remains the same as in two-level transactions. Cascaded coordinators (i.e., non-root and non-leaf participants) are made to behave as leaf participants with respect to their direct ancestors and as root coordinators with respect to their direct descendants. Specifically, when a cascaded coordinator receives a "prepare" message, in *multi-level PrA*, it forwards the message to its descendent participants and waits for their votes. If all descendants have voted "yes", the cascaded coordinator force writes a prepared 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, 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, 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.

It should be pointed that PrA is the current choice of the ISO-OSI [67, 30] and X/Open [12, 19] distributed transaction processing standards. Along with 2PC, PrA has been implemented in a number of commercial products such as DECdtm Services [59, 68], DEC VMS [11, 31], Transarc's Encina [54, 58] and TUXEDO of Unix System Laboratories [43].

13.4.2 The Presumed Commit Protocol

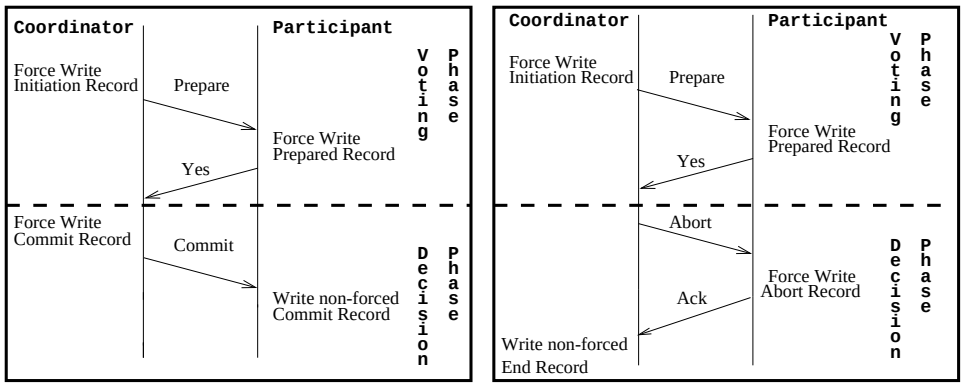
As opposed to PrA protocol that favors aborted transactions, the *presumed commit* (PrC) protocol is designed to reduce the cost of committed transactions [38, 39]. It is based on the assumption that a transaction is most probably going to be committed once it has finished its execution and submitted 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 (which is also called *collecting* record in [39]) for the transaction before sending out "prepare" messages to the participants (Figure 13.5). The initiation record ensures that missing information about a transaction will not be 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 facilitates recovery by recording the identities of the participants, which in the case of PrN and PrA are recorded in the decision records.

To commit a transaction (Figure 13.5(a)), 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 have to acknowledge a commit decision.

To abort a transaction (Figure 13.5(b)), on the other hand, the transaction's coordinator does not force write an abort record. Instead, the coordinator sends out abort messages to all the participants that voted "yes", 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 rebuilds its protocol



(a) Commit case. (b) Abort case.

Figure 13.5. The presumed commit 2PC 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, 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 about the transaction.

In the case of a participant failure, the participant inquires about the outcome of each transaction that has a prepared log record but without a final decision record. When a coordinator receives an inquiry message pertaining to a transaction, the coordinator either has an entry including an initiation record in its protocol table for this transaction or it does not have any entry, which means that the coordinator has forgotten the transaction. In the former case, the coordinator sends an abort and waits for an acknowledgment. In the latter case, not remembering the transaction, the coordinator sends a commit final decision 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 an extra forced log write at the coordinator (that is, the initiation log record). For the abort case, PrC incurs one extra forced log write at the coordinator compared to PrN.

PrC can be extended in the tree-of-processes model in two ways. The first extension provides for a speedy recovery in the case of failures accompanied by

slow commitment during normal processing, while the second one provides for fast commitment during normal processing accompanied by slow recovery in the presence of failures.

In both resulting protocols, the root coordinator and each leaf participant behave as in PrC, both during normal processing and in the presence of failures. The first protocol, called *multi-level PrC* [38, 39] extends PrC in a manner similar to PrA with the commit presumption to hold between any adjacent levels in the transaction tree in the case of failures. Since a cascaded coordinator behaves as a PrC coordinator for its direct descendent participants, each cascaded coordinator has to force write an initiation record before propagating the "prepare" message to its descendents. If the final decision is to abort the transaction, 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, a prepared-to-commit cascaded coordinator propagates the decision to its direct descendants, writes a non-forced commit record and, then, forgets the transaction.

The second protocol, called the *rooted presumed commit* protocol (RPrC) [8], eliminates *all* the intermediate initiation records from cascaded coordinators, and, consequently, reduces the cost of commitment during normal processing. But, as opposed to multi-level PrC, RPrC does not realize the two-level presumption of PrC on every adjacent level [8]. Consequently, in the presence of multiple failures, cascaded coordinators cannot provide a fast reply to inquiring messages from recovering participants by presuming commitment in the case that they do not remember a transaction due to a failure. If a cascaded coordinator does not remember a transaction, it needs to inquire its direct ancestor about the outcome of the transaction which in turn might have to inquire its own ancestor. 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. RPrC supports efficient propagation of inquiring messages, by requiring each participant to store the list of ancestors of a transaction as part of the transaction's prepare log record and include it in any inquiring message. In this way, if the direct ancestor of a prepared transaction does not remember the transaction, it uses the list of ancestors to find out its own direct ancestor.

It order to facilitate recovery after a coordinator failure and in particular, the failure of the root coordinator, RPrC also requires that, for each transaction, the root coordinator stores in the initiation record and each cascaded coordinator stores in the prepare record all the direct descendants along with their lists of descendants in the transaction's execution tree. Thus, for example, a recovering root coordinator can abort transactions that have not completed their commitment by sending an abort message to its direct descendants, as recorded in the initiation record, along

with their lists of descendants in the transaction execution tree to be used by any descendant cascaded coordinator which does not remember the transaction due to a failure, to propagate the abort decision further down the tree.

The list of descendants is constructed by requiring each participant to append its identifier in the acknowledgment of the *first* operation that executes on behalf of a transaction whereas the list of and ancestors is constructed by requiring each coordinator, including the root, to append its identifier in the "prepare" message.

13.4.3 The New Presumed Commit Protocol

Although PrC is more efficient than PrN, its major drawback, as alluded above, is the forcing of the initiation record which prolongs the voting phase and delays the sending of the "prepare" message. This means, for example, the participants in PrA receive the "prepare" message which allows them to release the read locks, earlier than the participants in PrC.

The *new presumed commit* 2PC variant (NPrC) [33], eliminates the initiation record of the PrC by (1) giving up precise knowledge about the active transactions prior to a coordinator's site failure and (2) making the garbage collection of the stable log more expensive. In order to distinguish between committed and aborted or active transactions in the absence of the initiation record after a failure, the coordinator in NPrC maintains two sets of transactions: the set of *recent transactions* (RECT) and the set of *potentially initiated transactions* (PIT). Specifically, in NPrC, a coordinator does not interpret lack of information in its log as commitment but instead, it presumes that a transaction is committed if it is not in a PIT set stored on its stable storage. A PIT contains those transactions that were either aborted or possibly active prior to a failure. After a coordinator crash, the corresponding PIT is derived from RECT as follows.

NPrC assigns transactions monotonically increasing identifiers. Any transaction whose identifier *tid* is less that the lower-bound tid_l of RECT has completed the commit protocol whereas no transaction with *tid* greater than the upper bound tid_u of RECT has started its execution. Whenever the transaction with tid_l completes the protocol, tid_l is advanced and is either forced written as part of the transaction's log records if the transaction is committed, or written into the log buffer if the transaction is aborted to be propagated to the stable log when a subsequent force log write is performed. On the other hand, tid_u is periodically increased and force written on the log.

After a failure, the coordinator reads from its log the values of tid_l and tid_u and, by scanning its log for explicit commit records, determines all committed transactions prior to the failure whose *tid* lies between tid_l and tid_u. By excluding these committed transactions from RECT, PIT is constructed. Once PIT is determined, it is recorded in the stable storage and the coordinator resumes processing 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

2PC Variants	Commit Decision						Abort Decision					
	Coordinator			Participant			Coordinator			Participant		
	r	f	p	r	f	q	r	f	p	r	f	q
PrN	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
NPrC	1	1	2	2	1	1	1	1	2	2	2	2

Table 13.1. The complexities of 2PC and its standard variants.

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.

13.4.4 Efficiency of Presumed Abort and Presumed Commit variants

The cost of the common 2PC variants discussed above during normal processing, as shown in Table 13.1, can be compared in terms of the message and log complexities of the coordinator and a participant which votes "yes". Recall that the basic 2PC is also referred to as Presumed Nothing (PrN). In the Table, r is the total number of log records, f is the number of forced log writes, p is the number of messages received by a participant from the coordinator and q is the number of messages sent back to the coordinator by the participant.

In section 13.3, we saw that, in PrN, the cost to commit or to abort a two-level transaction executing at n participants is the same. PrN has log complexity $2n + 1$, message complexity $4n$ and time complexity 3 rounds. PrA and the two PrC variants also have the same time complexity as PrN.

To abort a transaction in PrA requires n forced log writes, one at each participant, and $3n$ messages. The cost to commit a transaction in PrA is the same as in PrN. That is, $2n+1$ forced log writes and $4n$ messages. In PrC, the cost to commit a transaction is $n + 2$ forced log writes and $3n$ messages. To abort a transaction, the cost is $2n + 1$ forced log writes and $4n$ coordination messages. In NPrC, there is a reduction of a forced log write for both the commit as well as the abort case[2] (i.e., the initiation record at the coordinator's site) when compared with PrC.

It follows, from the above, that it is cheaper to use PrA in a system where transactions are most probably going to abort, while it is cheaper to use PrC, or preferably NPrC, if transactions have higher probability of being committed. In a

[2] In NPrC, an **abort** record is forced only if the transaction is the oldest (i.e., has the least tid in its RECT set) so that the tid_l is advanced. Here, we assume that an aborted transaction is always the oldest as a worst case scenario in order to simplify performance analysis.

	Commit		Abort	
	Forced Log Writes	#Messages	Forced Log Writes	#Messages
ML-PrA	$2l + 2c + 1$	$4n$	$l + c$	$3n$
ML-PrC	$l + 2c + 2$	$3n$	$2l + 3c + 1$	$4n$
RPrC	$l + c + 2$	$3n$	$2l + 2c + 1$	$4n$

Table 13.2. The complexities of multi-level PrA, multi-level PrC and RPrC.

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 exactly symmetric. Whereas the cost to commit a transaction in PrA is the same as to abort a transaction in PrC and NPrC, the cost to abort a transaction in PrA is not the same as to commit a transaction in neither PrC nor NPrC. PrC requires two forced log writes at the coordinator's site and NPrC requires one forced write for the commit case while PrA does not require any log writes for the abort case.

Multi-level PrA (ML-PrA) and multi-level PrC (ML-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, whereas 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.

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 PrA incurs $n - 1$ more forced log writes than PrC.

RPrC is more efficient that multi-level PrC and the relative advantage of RPrC and multi-level PrA is reduced to the relative advantage of PrC and PrA. 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. Table 13.2 summarizes the complexities of the multi-level 2PC variants.

13.5 OTHER TWO-PHASE COMMIT VARIANTS

In this section, we discuss six more 2PC variants that have been proposed for specific environments. The common characteristic of these protocols is that they exploit the semantics of the communication networks, the database management systems and/or the transactions to enhance the performance of 2PC.

13.5.1 Network Topology Specific 2PCs

The *linear* 2PC [22, 45] exploits the communication topology to reduce 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 initiates the commitment of a transaction by sending a "prepare" message to the site that follows it in the linear order. The message also includes the coordinator's 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 voted "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 that has made the final decision (that is, the last site in the linear order) in a manner similar to the way the vote messages are propagated. That is, each message is sent in a different round and thus, linear 2PC has the same message and time complexities $2n$, for n participants. Compared to 2PC, linear 2PC has the same log complexity as 2PC, reduces the message complexity of 2PC from $3n$ to $2n$ but it increases the time complexity of 2PC from 3 to $2n$ rounds.

In *decentralized* 2PC (d2PC) [55], the interconnecting communication network is assumed to be fully connected and contrast to linear 2PC, d2PC reduces time complexity at the expense of message complexity. As in linear 2PC, the coordinator in d2PC includes in the "prepare" message it own vote. Furthermore, it includes the identities of all the participants. When a participant receives the "prepare" message, it broadcasts its vote to all participants. If a participant has voted "yes", it can locally decide about the outcome of the transaction when a participant receives the

votes from all other participants. If all votes are "yes", the participant commits the transaction, otherwise it aborts it.

In d2PC, two rounds of messages are required for a participant to make a final decision. The first round is the coordinator's vote, whereas the second is the other participants' votes. By reducing the time complexity from three rounds of messages which is the case in 2PC, to two rounds, it becomes less likely for a participant to be blocked during commit processing in the case of a coordinator failure. This is another advantage of d2PC. On the other hand, d2PC requires $n^2 + n$ messages compared to $3n$ required by 2PC, where n is the number of participants.

13.5.2 Transaction Type Specific 2PCs

The common goal of all four protocols below is to improve both the message and time complexities of 2PC by eliminating the explicit voting phase of 2PC, which polls the votes of the participants. Specifically, these protocols eliminate the "prepare" messages and their corresponding sequential rounds of messages.

The *unsolicited-vote* protocol (UV) [62] shortens 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" message. Instead, it sends its vote on its own initiative once it recognizes that it has executed the last operation. When the coordinator receives the vote from each participant, it proceeds with the decision phase.

The applicability of UV is limited to database environments in which a coordinator either submits to a participant all the operations of a transaction at the same time or indicates to the participant the last operation at the time that this operation is submitted. The former is a form of predeclaration while the latter implies that each transaction has knowledge about data distribution and can indicate to its coordinator when it has finished accessing a participant site.

Another 2PC variant that eliminates the voting phase of 2PC is the *early prepare* protocol (EP) [60, 61] in which the "prepare" messages are traded for force written log records. EP is based on the following three assumptions: (1) the cost of accessing a stable storage in some systems is as cheap as accessing main memory, (2) each site implements the *strict two-phase locking* protocol [20] which is used in most commercial systems for concurrency control, and (3) transactions do not require deferred consistency constraints validation at commit time.

EP combines UV with PrC without assuming that a participant can recognize the last operation of a transaction. Every operation, therefore, is treated as the last operation and its acknowledgment is interpreted as a "yes" vote. This means that a participant has to prepare a transaction each time it executes an operation of that transaction and prior to acknowledging the operation[3] When the participant

[3] This is possible based on the assumption that each site employs the strict two-phase locking protocol in which it is impossible for a participant in a transaction's execution to abort the transaction due to a deadlock or serializability violation once all the operations received by the

receives a new operation for execution, the transaction becomes active again. While the transaction is active, the participant can abort it, for example, if it causes a deadlock. When a transaction is aborted by a participant, the participant responds to an operation with a *negative acknowledgment* (NACK) which the coordinator interprets as a "no" vote and decides to abort the transaction at all participants.

Since, EP requires each acknowledgment to be preceded by a force log write, the number of forced prepared records pertaining to a transaction at a participant is equal to the number of operations submitted sequentially by the transaction and executed by the participant. Furthermore, since PrC requires the identities of the participants to be explicitly recorded at the coordinator's log as part of a forced initiation record, the coordinator must update and force write an initiation record each time a new participant is involved in the execution of the transaction. In the worst case, when the participants become known dynamically, the number of forced initiation records pertaining to a transaction is equal to the number of the participants that executed the transaction.

Two other protocols, namely, the *coordinator log* protocol (CL) [60, 61] and the *implicit yes-vote* protocol (IYV) [2, 3, 6], share the same basic idea with EP but they eliminate the need for the forced log writes at the participants. CL which is derived from PrC, eliminates logging at the participants by implementing *distributed write-ahead logging* (DWAL) [17] whereas IYV which is based on PrA, by implementing *replicated write-ahead logging* (RWAL). Both CL and IYV assume high speed communication network and highly reliable sites.

In CL, only the coordinators maintain stable logs. When a participant executes an operation, as part of its acknowledgment, the participant propagates any redo and undo log records generated during the execution of the operation to the transaction's coordinator to write them in its log. This means that participants cannot independently recover after a system failure and need to communicate with all possible coordinators in order to reconstruct their logs. CL also eliminates the forced initiation log record of PrC at the expense of independent recovery. That is, after a failure, during its recovery, the coordinator needs to communicate with all possible participants in the system in order to determine the set of active transactions prior to the failure and to abort them instead of wrongly assuming commitment. Thus, in CL, the execution of transactions is distributed across multiple database sites, whereas WAL logging and commit processing are centralized at the coordinators' sites.

To commit a transaction, in CL, the coordinator first force writes a commit log record and then sends out commit messages to all participants. When a participant receives a commit message, it commits the transaction by releasing all the resources held by the transaction and without sending an acknowledgment according to PrC. On the other hand, when the coordinator of a transaction decides to abort the transaction, it writes a non-forced abort record and sends out abort messages to all participants. Each abort message includes the undo records of the

participant have been successfully executed and acknowledged [6, 2].

aborted transaction.[4] When a participant receives an abort message, it rolls back the transaction, releases all the resources held by the transaction, and sends an acknowledgment message that includes all the new log records generated during the transaction rollback. When the coordinator receives the acknowledgment messages, it writes both the received log records and an end record in in its log in a non-forced manner and forgets the transaction.

As opposed to CL, in IYV participants maintain a log and can independently rollback an aborted transaction. IYV eliminates only the forced prepare records at the participants by using RWAL, that is, replicating the redo part of its log pertaining to a transaction at the transaction's coordinator site. Furthermore, IYV proposes logging of state information (e.g., read locks) along with redo records at the coordinator site in order to support *forward recovery*.

On a commit decision, the coordinator in IYV first force writes a commit log record, then sends out commit messages to all participants and waits for their acknowledgments. When a participant receives a commit message, it commits the transaction releasing all its resources and writing a non-forced commit log record. When the commit log record is flushed into stable storage, due to a periodic flushing of the log, the participant acknowledges the commit decision since it follows PrA. When the coordinator receives acknowledgments from all participants, it writes a non-forced end log record and forgets the transaction.

On an abort decision, the coordinator sends out abort messages to all participant except for the participants that have sent NACKs and forgets the transaction without writing any log records. When a participant receives an abort message, it undoes the effects of the transaction using its local log and writes a non forced abort log record. Once the effects of the transaction are undone, the participant releases the resources held by the transaction without sending any messages back to the coordinator.

Although, as opposed to CL, coordinators in IYV can independently recover, participants in IYV, as in CL, cannot. Recovering participants need to communicate with all potential coordinators in order to reconstruct the redo part of their log and determine which of the active transactions in its site have been committed and which are still in progress. Because of this, a participant cannot resume normal transaction processing until it receives replies from all the coordinators. Thus, the applicability of IYV as well as CL is curtailed in the presence of unreliable and slow coordinators.

Even though, due to local logging, a recovering participant in IYV can overlap the undo phase with the resolution of the status of active transactions and the repairing of the redo part of the log that partially masks the effects communication delays, it is imperative that all participants become operational in a bounded amount of time, in a similar manner as in 2PC. For this reason, another IYV variant,

[4] The assumed underlying recovery scheme of CL is ARIES [40], in which transactions are undone logically (that is, undoing an operation is another operation that needs to be executed and acknowledged).

called *implicit yes-vote with delegation of commitment* (IYV-WCC) has been proposed in [5, 2] that reduces the blocking aspects of IYV by combining the delegation of commitment technique, found in open commit protocols [46, 47] (section 13.7), with a novel timestamp coordination mechanism. Although, the new IYV variant requires extra messages and log writes, it still maintains the cost of commit processing during normal processing below that of 2PC, PrA and PrC. (For a detail analytical comparison of EP, CL, IYV, IYV-WCC with 2PC and its common variants see [5, 2].)

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. In what follows, after discussing the most common 2PC optimizations, 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.

13.6 ATOMIC COMMIT PROTOCOL OPTIMIZATION

In this section, we focus on the six most significant optimizations proposed in the literature, namely, *read-only, last agent, group commit, sharing the log, flattening the transaction tree* and optimistic, that can reduce the cost associated with commit processing. With the exception of the optimistic optimization which was only recently proposed (1997), all others have been implemented in commercial systems. These can be combined and used together with a number of 2PC variants as well as other ACPs. For a survey of the most common optimizations implemented in commercial transactional systems, including within a peer-to-peer environment can be found in [48, 50].

13.6.1 Read-Only

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

In the traditional read-only optimization [39], when a participant that has executed only read operations on behalf of a transaction receives a "prepare" 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, 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 read-only optimization can be considered as the most significant optimization, given that read-only transactions are the majority in any general database system. In fact, the performance gains allowed by the read-only optimization provided the argument in favor of PrA to become the current choice of ACP in the standards and commercial systems.

From the evaluation in section 13.4.4, it is clear the PrC variants are the best choice for committing transactions only in systems in which the majority of the transactions are update transactions and are finally committed. However, in general, and in systems in which the majority of the transactions are read-only in particular, the PrA variants are the choice. 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 primarily to initiation log records forced in PrC variants for both update and read-only transactions. Not knowing whether a transaction is going to be read-only, a coordinator in PrC has to force write an initiation record.

This motivated the development of new read-only optimization, called *unsolicited update-vote* (UUV) [7, 8] that eliminates the initiation record for read-only transactions using a similar idea as the UV and EP protocols (see section 13.5). In UUV, a participant is assumed as a read-only one until it sends an *unsolicited update-vote* to the coordinator as part of the acknowledgement of the first update operation (which is recognized by the generation of undo/redo log record(s)) that it has executed. When a coordinator receives a final commit request for a transaction, it checks for any update votes pertained to the transaction and if none is found, the transaction is recognized as read-only and the coordinator sends out a read-only message to all the participants indicating that the transaction can be terminated. UUV can also be used to reduce the cost of committing partial read-only transactions by excluding read-only participants from voting by sending them a *read-only* message and allowing them to release all the resources held by the transaction without writing any log record.

From the performance point of view, the cost of PrA variants combined with UUV is the same as in PrC variants combined with UUV while PrC combined with UUV is cheaper than PrA combined with the traditional read-only optimization that requires the voting phase.

13.6.2 Last Agent

The *last agent* optimization has been implemented by a number of commercial systems to reduce the cost of commit processing in the presence of a *single remote* participant [50]. In this optimization, a coordinator first prepares itself and the nearby participants for commitment (fast first phase), and then delegates the responsibility of making the final decision to the remote participant. Once the remote participant receives the request for final decision, if it is prepared to commit, it sends out a commit decision, otherwise, it sends out an abort one. In this way, it reduces the communication required with the faraway participant to one slow round-trip message exchange.

This optimization yields the greatest benefit, especially if sending messages to the remote participant involves long network delays (i.e., connection through satellite). Clearly, its benefit is also depended on the time required to prepare the nearby participants. For example, in the case of a deep transaction tree, if the time required to prepare the nearby participants is sufficient to send a "prepare" message to the remote participant and get back its vote, last agent is not applicable. In such cases, it actually delays the decision by sequencing the "prepare" messages to the nearby participants and to the remote one.

13.6.3 Group Commit

The group commit optimization [18, 21] 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 (that is, 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 the *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.

13.6.4 Sharing the Log

A local data manager (DM) uses a log to keep track of updates so that it can either abort or commit a transaction. Before a DM votes "yes", it ensures that this information has been forced to stable storage. When it learns of a commit outcome, it also force writes a commit record.

The DM can share the same log with the transaction manager (TM) at its site [39]. With this optimization, the DM takes advantage of the sequential nature of the log and of the knowledge that the TM will force write a commit record. The DM does not force write the prepared record because the TM's force write of the commit record causes the local DM's earlier non-forced write to be written to the log. If the transaction successfully commits, the TM's commit record and the DM's prepared record will both be on the log. This ensures successful recovery processing. If the system fails before the commit is forced, the prepared record may be lost. This does not change the outcome of the transaction, since the TM aborts the transaction if it does not find a commit record on the log. Similarly, the DM does not need to force write the commit record. If the system fails and the non-forced commit record is lost, since TM's commit record and the DM's prepared record are both on the log, the recovery process will successfully commit the transaction.

This optimization saves two forced writes per DM that shared the log. The more DMs that share the log with the TM, the more savings per transaction. CL and IYV, discussed in section 13.5, can be viewed as extending this optimization to include remote data/resource managers.

13.6.5 Flattening the transaction tree

As discussed in the previous section, in a multi-level 2PC variant, coordination messages are propagated down and up the transaction tree in a sequential manner, level by level. Clearly, this serialization of messages increases the duration of commit processing as the tree depth grows.

The *flattening the transaction tree* optimization transforms the transaction execution tree of *any* depth into a two-level commit tree at commit initiation time [50]. In this way, the root coordinator sends coordination messages directly to, and receives messages directly from, any participant. This is achieved by (1) propagating the identity of the root coordinator to each participant as part of the first operation request and (2) propagating the identity of each participant to the root coordinator as part of the acknowledgement of the first operation the participant has executed.

Since messages in a flatten, two-level commit tree can be sent in parallel, this optimization avoids the propagation delays and can be a big performance winner in distributed transactions that contain deep trees. This optimization has been effectively used in [8] to optimize the performance of the multi-level PrC by reducing, in addition to time complexity, its log complexity. That is, it (1) reduces the number of initiation records, (2) allows forced log records to be performed in parallel and

(3) reduces the number of non-forced log writes.

This optimization cannot be used in an environment where a participant is prohibited to directly communicate with the root coordinator or vice versa for security reasons. In general, it also cannot be used when the communication topology does not support direct interaction between a root coordinator and the leaf participants. Similarly, it is not feasible 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 [24].

Another limitation is in protocols that do not require acknowledgement, such as some message based protocols, and conversational protocols (i.e., LU6.2 [29, 65]). In these, the identities of all the participants may not be known prior to the first phase. However, for these protocols it is possible to flatten the tree during the second phase.

13.6.6 Optimistic

The most recently proposed optimization is *optimistic* (OPT) [26] which enhances the overall system performance by reducing blocking arising out of locks held by prepared transactions.

OPT shares the same assumption as PrC, that is, transactions tend to commit when they reach their commit points. Under this assumption, OPT relaxes the *strictness* requirement of recovery [10, 44], allowing a transaction to *borrow* data that have been modified by another transaction that has entered a prepared to commit state and has not committed. 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. OPT prevents cascading aborts and complex recovery after failures by limiting the abort chain to one. It allows only prepared transactions to lend their data while borrowers cannot enter the prepared state until the borrowing is terminated, a rule which is consistent with the notion of prepared state in ACPs.

OPT can be combined with the other optimizations and integrated with most of the ACPs. Its power to enhance the overall system performance due to the early release of data held by prepared to commit transactions has been demonstrated through simulation using OPT-2PC, OPT-PrC, OPT-PrA and OPT-3PC. An interesting result of these simulations is that, although 3PC (three-phase commit protocol, discussed in the next section) is more expensive than any of the 2PC variants, OPT-3PC exhibits better peak throughput than all the 2PC variants in high contention situations. This means that in such situations is better to use OPT-3PC that offers superior performance during normal processing and non-blocking properties in the presence of failures.

13.7 REDUCING THE BLOCKING EFFECTS OF 2PC PROTOCOL

All the proposed ACPs involve blocking [56, 10]. They just differ in the size of the window during which a site might be blocked [16]. In this section, we focus on ACPs that have been designed with different non-blocking features that reduce the size of this window. These protocols can be classified into whether they preserve the prepare state or allow *unilateral or heuristic* decisions in the presence of unbearable delays. Below, we first briefly review four protocols in the first category which we call *non-blocking ACPs*, namely, *cooperative* 2PC, *three-phase*, *four-phase*, and *open* commit protocols. Interestingly, perhaps, due to their increased overheads, none of the non-blocking ACPs has been implemented in commercial systems. Then, we discuss IBM's Presumed Nothing protocol, which has been designed to take into consideration the necessity of heuristics decisions.

13.7.1 Non-blocking Atomic Commit Protocols

The *cooperative* 2PC [34] reduces the *likelihood* of blocking in case of a coordinator failure. In cooperative 2PC, the identities of all participants are included in the "prepare" message so that each participant becomes aware of the other participants. In the case of a coordinator or a communication failure, a participant does not block waiting until it reestablishes communication with the coordinator. Instead, it inquires the other operational participants in the transaction's execution. If any of the operational participants have received the final decision from the coordinator prior to the failure, it informs the inquiring participant about the final decision, thus, reducing the time for which a participant is blocked waiting for recovery from a failure.

Cooperative 2PC 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. In contrast, the *three-phase commit* (3PC) [55] and the *four-phase commit* (4PC) [28] protocols eliminate the blocking aspects of a 2PC that are due to site failures. That is, if a coordinator fails, the participants can make their own decision.

In 3PC, a preliminary decision is reached before the final decision is made. For this reason, in 3PC, an extra phase is inserted between the two phases of 2PC and the *precommit* state, 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 precommit state. Otherwise, the new coordinator aborts the transaction.

In 4PC, on the other hand, the coordinator of a transaction initiates the 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 the 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.

All the protocols that we have discussed thus far have been designed assuming no *commission* failures. But this assumption does not hold in *open distributed systems*. In such an environment, a participant site is classified as, *trusted* or *nontrusted* node. A trusted node is one that fails only transiently, and when it fails, it does not send misleading messages. Otherwise, the node is considered nontrusted in the sense that it may never recover or it may deviate from the algorithm of the commit protocol by sending different messages, including misleading ones, causing a commission failure.

The *open commit protocols* [46, 47] (OCPs) have been proposed in the context of *open distributed systems* to ensure the atomicity of transactions *at least* across trusted nodes and despite the existence of nontrusted ones. This goal is achieved by delegating the commit processing (that is, transferring the commit responsibilities) from a nontrusted node to a trusted one and transforming the execution tree of a transaction through restructuring into a different commit tree. Thus, OCPs guarantee that all trusted nodes will reach an agreement about the outcome of transactions despite the participation of nontrusted nodes.

13.7.2 IBM's Presumed Nothing Protocol

In some real-world applications, in the event that a transaction's outcome is bloked due to failures or long delays, certain participants might be required not to wait for recovery processing to discover the outcome because of valuable locks being held [36, 50]. Depending on business needs, these participants might unilaterally commit or abort the transaction. The decision to allow heuristic decisions involves business trade-offs between the cost of fixing database inconsistencies and the cost of missed opportunities. This heuristic decision may damage the consistency of the transaction and data. The IBM's *Presumed Nothing* protocol (IBM-PrN) [51, 65] is a 2PC variant that detects and reports heuristic damage (i.e., conflicting heuristic decisions) simplifying the task of identifying problems that must be fixed. IBM-PrN is part of the SNA LU6.2 architecture [48, 50] that defines the *peer-to-peer* distributed transaction environment, the commit protocols and their synchronization.

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 involving the coordinator. Once a heuristic decision is made, a participant force writes its decision for comparison during recovery and reports it to the site's systems control operator (since a heuristic decision may result in loss of synchronization among the distributed resources that has to be repaired

by an operator action).

The scope of heuristic decisions depends on the participant's role in the transaction commit protocol. In general, heuristic decisions should be propagated to subordinates (descendants) and reported to the coordinator. The subordinates take the same action as the participant that propagated the decision. If recovery is required with some of them, it is performed and completed (so the subordinates can be informed of the heuristic decision of their coordinator) before their coordinator's coordinator (if any) is informed. This permits full and accurate reporting to the root coordinator.

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, whereas 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 damages and to correct it. Of course, as mentioned above, the intervention of a human (or automated) operator is required for the restoration of consistency in the event of a heuristic decision. Hence, IBM-PrN retains its performance close to 2PC while, reducing the blocking effects of 2PC in the case of failures.

Different heuristic situations are described in [51, 65]. The detailed description of these scenarios demonstrated the complexity of incorporating heuristic decisions in the basic 2PC. This resulted in a very similar to IBM-PrN 2PC variant called in [51] Presumed Nothing (PN), that reliable handles Heuristic Decisions. In [52], it was showed that all commit variants (PrA- or PrC-based) once enhanced with heuristic decisions collapsed onto the PrN commit variant. This led to a classification of commit protocols around heuristic processing. This classification is quite significant in that it simplifies, for example, the task of evaluating the performance of any commit protocol variant when heuristic decisions are to be considered. One can simply evaluate the IBM-PrN. At IBM, another effort to enhance SNA's LU6.2 with PrA that also includes heuristic decisions resulted in a protocol, called *generalized presumed abort* (GPA) [49, 41], that when recognition of heuristic decisions is required collapses to PN. [51, 52] generalizes and extends that effort.

13.8 SIMULATION-BASED PERFORMANCE EVALUATION

The traditional method of performance evaluation is useful, as we have seen in the previous sections, in analyzing best- and worst-case scenarios to highlight the performance differences among different ACPs. However, to compare and determine the best ACP for a particular database environment, quantitative performance evaluations with respect to the overall system performance, such as *mean response time* or *peak throughput*, are required as is the case with other database protocols (e.g., [1, 14, 15, 42]). Surprisingly, as it was pointed out in [26], very few such performance

evaluations exists [35, 26, 9, 2].

In this section, we report on a simulation study [9, 2] that evaluates the performance implications of five ACPs on *transaction throughput* in wide-area, gigabit-networked, distributed database systems. These protocols are 2PC (section 13.3), PrA (section 13.4.1), PrC (section 13.4.2, CL and IYV (section 13.5.2). The study under consideration considers CL and IYV because they can exploit the characteristics of a gigabit network to enhance the performance of a database system. In particular, they can exploit the fact that, in gigabit networks, the size of a message is less of a concern than the number of sequential phases of message passing.

To isolate the impact of ACPs on the overall performance of the system, the study also simulates the behavior of the system when *distributed-execution centralized-commit* (DECC) is used as in [26]. DECC simulates the distributed execution of operations and centralized commit processing (that is, no ACP is used). Though artificial, DECC shows the *highest attainable* system performance in the absence of failures. In this way, one 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.

Since read-only transactions are the majority of transactions in any general database system, the study also evaluates the performance gains when the traditional read-only (TRO) and unsolicited update-vote (UUV) optimizations (section 13.6.1) are incorporated into the protocols under evaluation.

In contrast to other recent comparative performance evaluations of 2PC variants in local-area networks [35, 26], the study under consideration 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 issues based on the belief that these issues should not affect the relative performance of the common 2PC variants. On the other hand, the results of the study show that these issues 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, whereas the study that we discuss adopted 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 [39] with respect to either their execution patterns or the sites participating in their execution.

In the rest of this section, we first present the simulation system model that is used in the study and its associated parameters, and then present only the results of the study during normal processing and in the absence of failures. The results in the presence of one or two failed sites at any given time can be found in [9, 2]. We summarize these results in the last subsection.

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
Resource Parameters		
14	*CPUTime(MESG)*	CPU time for processing a message
15	*CPUTime(READ)*	CPU time for processing a read operation
16	*CPUTime(WRITE)*	CPU time for processing a write operation
17	*DiskTime*	Disk access time
18	*DiskTransfTime*	Page transfer time
19	*PropLatency*	Propagation time for a message
20	*Timeout*	Message timeout

Table 13.3. Simulation system parameters.

13.8.1 Simulation System Model

The simulation model can be expressed by four logical sets of parameters as shown in Table 13.3: (1) *database parameters*, (2) *transaction parameters*, (3) *site parameters*, and (4) *resource parameters*.

In the model which is similar to the database simulation model in [1], a database is a collection of objects that are uniformly distributed across a number of sites without data replication. A data object is uniquely identified by the tuple $< Site_{id}, Object_{No} >$. The database parameters that are the number of sites ($NumSites$) and objects ($NumObjects$) 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.

Each site 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), and (6) a *database cache manager* (DCM).

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 on 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.

Strict two-phase locking is used for concurrency control. An 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 to avoid deadlocks.

An RSM is a logical entity that 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. The physical resources available at a site consist 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 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.

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 ensure 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.

13.8.2 Transactions and Their Execution Model

While still adhering to the traditional ACID, 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 adopted in the study is *two-level sequential*. In the two-level sequential execution model, (1) a participant never decomposes a subtransaction further and (2) 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.

13.8.3 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 ($PercRead\text{-}OnlyTrx$).

The simulator is run at full capacity (that is, 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, *fake* restart is used. By using fake restart, 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. This number of transactions ensures that the system is operating within its steady state. This is confirmed by comparing runs of 10,000, 12,000, and 15,000 committed transactions. The comparison did not show any statistically significant differences between these

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 13.4. Simulation parameter settings.

runs. Hence, in all experiments, the system executes for 10,000 committed transactions. The performance curves of all 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 (that is, relative error).

13.8.4 Performance of Atomic Commit Protocols

This section includes discussion about the performance of ACPs. The parameter settings for the different experiments are as shown in Table 13.8.3. Since there are no failures, it is assumed that when a transaction reaches its commit point (that is, all its operations have been executed and acknowledged), the transaction commits. Also, since 2PC and PrA behave exactly the same in the absence of failures for committing transactions, for the clarity of the figures, only the performance curves

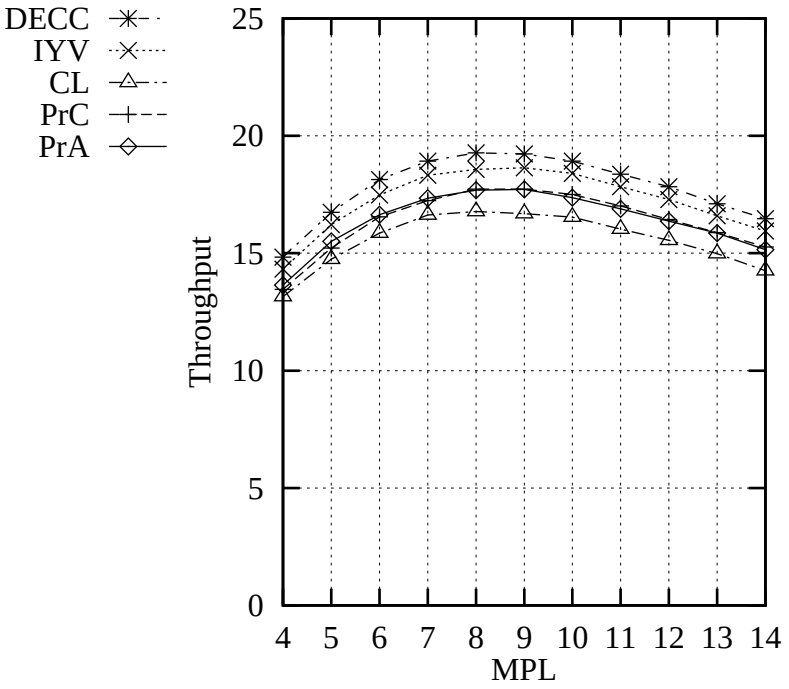


Figure 13.6. The performance of ACPs for long update transactions.

of PrA are included.

Given the size of the simulated database, long transactions execute, on average, 6 operations at each participant site, whereas short transactions execute, on average, 2 operation at each participant site. The first experiment deals with the performance impact of ACPs for long- and short-update transactions. The second experiment deals with the performance impact of ACPs when read-only transactions are introduced. The last experiment deals with the performance gains when read-only optimizations are used.

Performance of ACPs with Update Transactions

In this experiment, as well as all the other two experiments, the system throughput, which is the total number of committed transactions per second with varying multiprogramming levels (MPLs), is measured. 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 shown in Figure 13.6, the x axis is used for the MPL and the y axis is used for the system throughput. As shown in the figure, the performance curves of all ACPs start to increase from the MPL of 4 up to the peak MPL (that is, MPL of 8) and then they start to decline. This *thrashing* behavior of the system is due to the contention of transactions over the data objects as well as the system resources (that is, CPU, disk, log buffer, and the flushing and fetching of data objects) and appears in all three experiments. 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.

Figure 13.6 shows the performance of the different protocols when all the operations of transactions are update operations and transactions are long. At the peak MPL (MPL of 8), the difference in the performance of the system in the ideal case (that is, distributed-execution centralized-commit, DECC) and the worst case (that is, using the coordinator log CL protocol) is about 2.5 transactions per second which translates to about 15% performance difference. In the case of implicit yes vote (IYV), DECC outperforms IYV about by 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 over 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 the other two experiments 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, this result reveals that under low system loads, the initiation records of PrC affect its performance and makes it worse than PrA. 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.

Figure 13.7 shows the impact of the different ACPs on the performance of the system for short update transactions. 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 as opposed to being the loser in the case of long update transactions. This result clearly supports the motivation behind the design of CL [61] which assumes short transactions with high probability of being committed once they reach their commit point. However, the performance of CL starts to degrade more quickly after an MPL

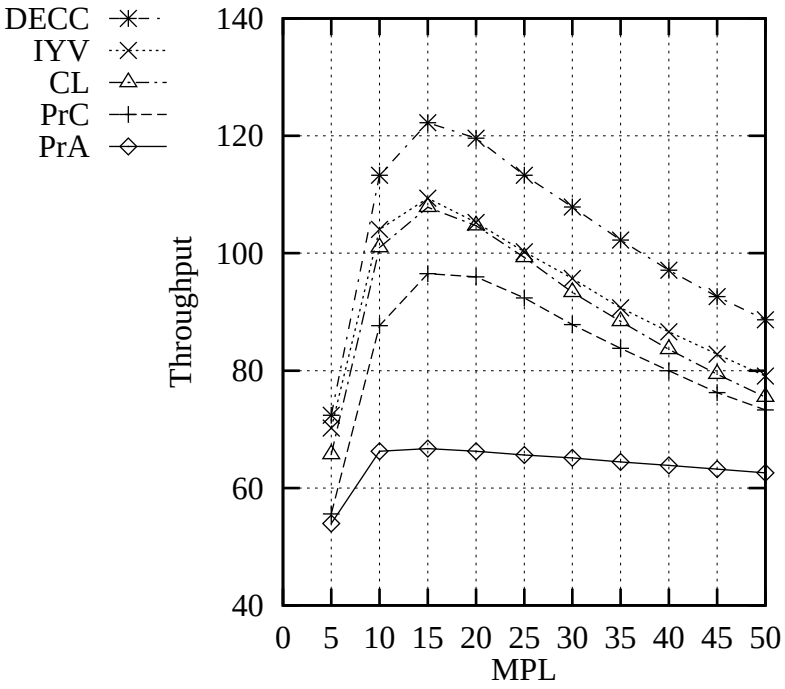


Figure 13.7. The performance of ACPs for short update transactions.

of 25 where its performance enhancement over PrC is about 3% at an MPL of 50 after it was about 12% at the peak MPL. On the other hand, IYV performance enhancement over PrC degrades to about 8% at an MPL of 50 from about 13% at an MPL of 15. The reason behind the quick degradation in the 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 is 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 to about a 45% enhancement over the performance of PrN and PrA at peak performance (MPL of 15). Similarly, by comparing the results of this experiment to the results of long transactions, one notices 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

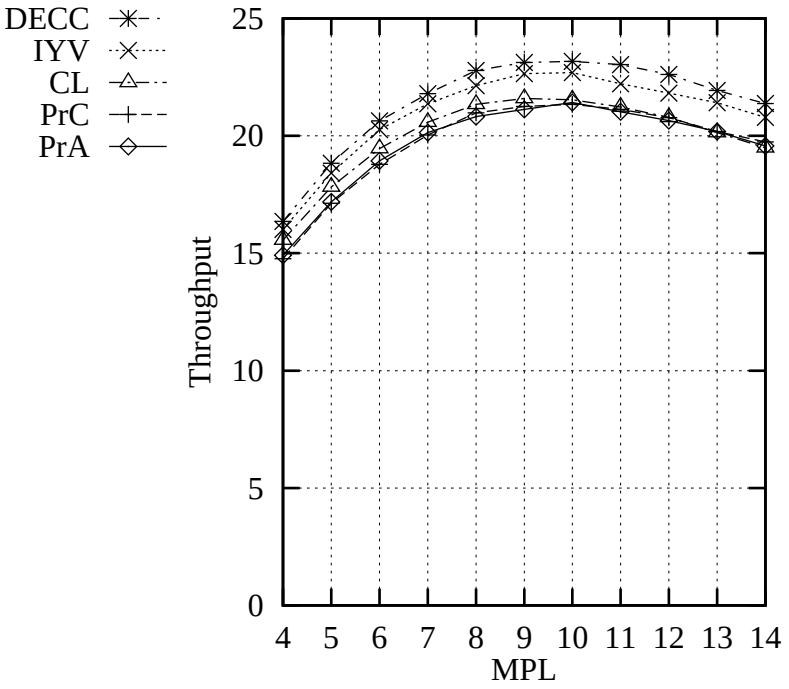


Figure 13.8. The performance of ACPs with long 70% read-only transactions.

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 loser in the previous experiment), but also the magnitude in the performance differences have greatly changed. These two results clearly show 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 third observation is regarding PrA's low thrashing behavior compared to the other protocols after it reaches its peak performance. With respect to this issue, it is noticed that the 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.

Performance of ACPs with Read-Only Transactions

Since read-only transactions are the majority of transactions in any general database system, traces of long and short transactions containing 70% read-only transactions were used in this experiment.

Figure 13.8 shows the performance of ACPs for log, majority read-only transactions. By comparing Figure 13.8 to Figure 13.6, one notices 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 an MPL of 9 to an MPL of 10. This is consistent with the fact that the system resources are still underutilized 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, the CL's performance has been enhanced to become better than all 3 two-phase commit variants at low MPLs and about the same as PrC at peak performance (i.e., around an MPL of 10) due to the reduced distributed write-ahead logging (DWAL) of CL when the majority of transactions are read-only.

Figure 13.9 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 the 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.

Another interesting result is that PrA reaches a steady state for a period longer than that in the case short update transactions. This is because the 70% read-only transactions do not conflict over locks with each other, and, therefore, a read-only transaction is not aborted unless it conflicts over a lock with an update transaction, resulting in fewer aborted transactions with less system thrashing.

Performance Gains When Using Read-Only Optimizations

As in the previous experiment, in this experiment, traces containing 70% read-only are used to evaluate the performance gains when the traditional read-only (TRO) optimization and unsolicited update-vote (UUV) optimization are incorporated into the system. For IYV and CL, a special case of UUV in the case IYV and CL is applied. 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 is called *read-only (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.

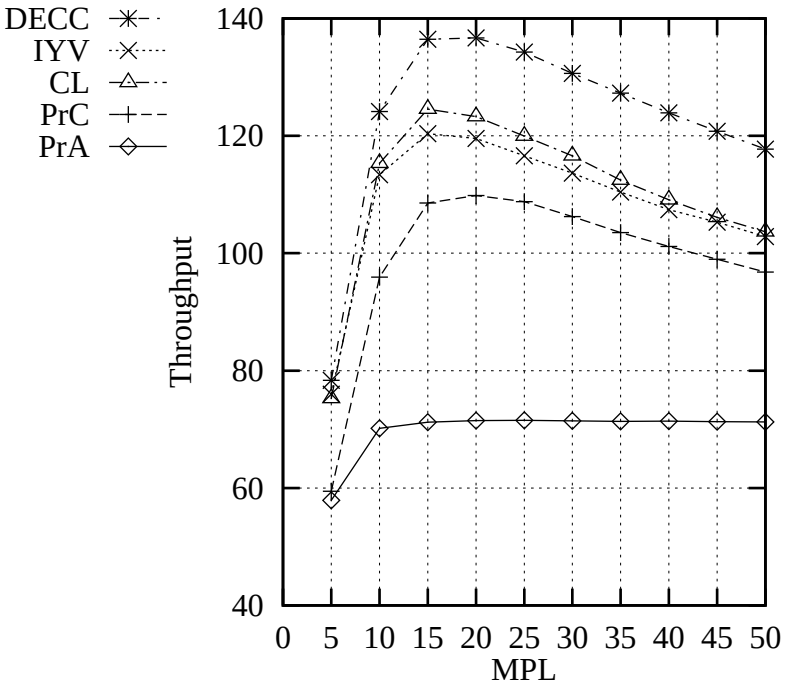


Figure 13.9. The performance of ACPs with short 70% read-only transactions.

As shown in Figure 13.10, neither the IYV nor the CL have benefited significantly from RO (that is, about 1% performance enhancement), whereas the 2PC variants have benefited more from TRO and UUV, reducing the performance gap with IYV at high MPLs to 3%. For the CL, at low MPLs, its performance is about the same as the 2PC variants whereas, at high MPLs, CL performance became worse than PrN, PrA, and PrC due to DWAL.

As reasoned in the previous experiments, 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 long transactions, the performance of all protocols has been enhanced with a PrA gaining the most and CL the least, as shown in Figure 13.11. 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

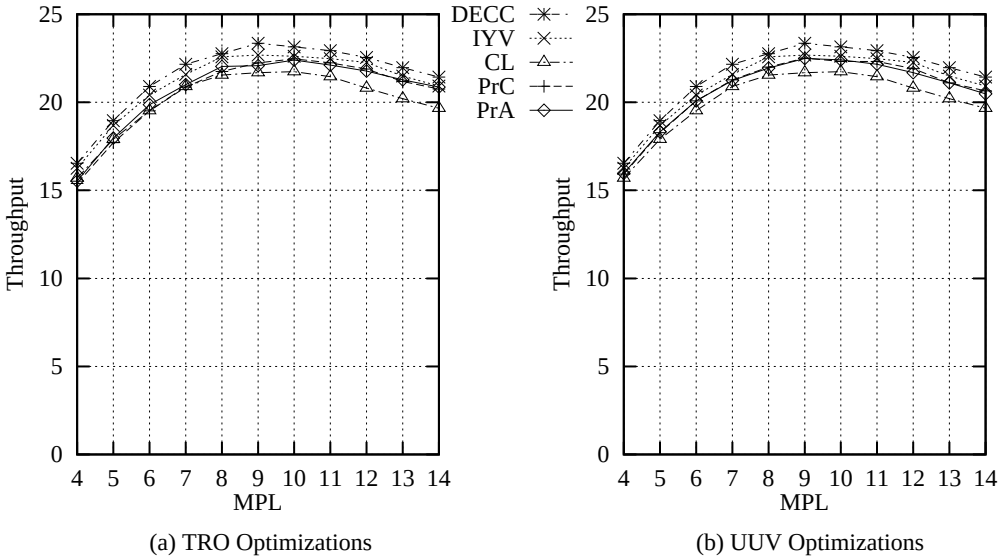


Figure 13.10. The performance of ACPs for long transactions with read-only optimizations.

of RO, the performance of IYV is again better than CL since it has gained more by the reduction of the logging activities than CL using RO. By comparing UUV with TRO, PrA has gained about 70% with UUV instead of 60% with TRO, whereas PrC has gained 12% using UUV instead of 6% using TRO. Hence, the UUV has closed the gap between the performance of PrC and IYV to about 3% in favor of IYV.

13.8.5 Summary of results

The results of the above study show that IYV 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, although the results in the present of failures were not discussed here.

With the exception of the case of short, majority read-only transactions without a read-only optimization (in which CL performed better than IYV), CL is the worst among all the evaluated protocols under the specified conditions and in particular,

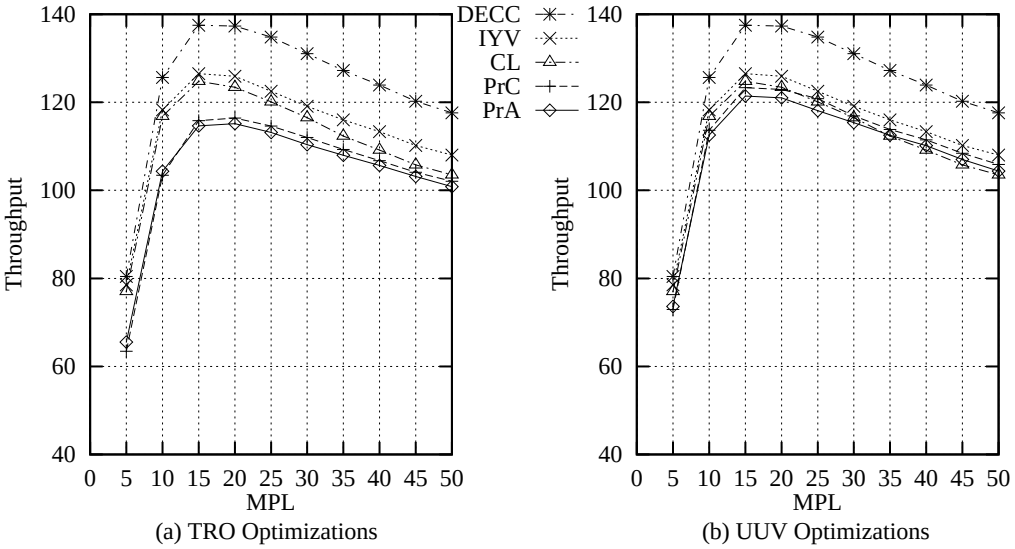


Figure 13.11. The performance of ACPs for short transactions with a read-only optimization.

the usage of the immediate abort strategy for deadlock avoidance. This has revealed the fact that in CL, recovering from aborted transactions is very expensive due to DWAL. This is also supported by the facts that (1) the performance of CL is greatly influenced by the length of the transactions and (2) its performance degrades significantly for high multiprogramming levels. This has also been confirmed in a recent evaluation of the same protocols in which deadlock avoidance was replaced by deadlock detection. In this, CL consistently exhibits the second best performance after IYV.

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. When there is a performance difference between 2PC variants, PrC is always the winner, exhibiting the best peak throughput. This is especially the case for short transactions. Further, performance enhancements due to a read-only optimization are more pronounced with short transactions.

Another very interesting result is that the impact on performance of the initiation log record associated with PrC becomes significant in light loaded systems. This can be concluded by the fact that PrA is the winner only in low multipro-

gramming levels. One can observe in all experiments a cross-over point between the performance curves of PrA and PrC even under the assumption that all transactions are to be committed when they reach their commit points. 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. In contrast to the general belief, these results shows that PrC, in general, is better than PrA.

In summary, this study leads to the conclusion that IYV should be the choice for the future gigabit network database systems and only replaced by PrC whenever IYV's assumptions are not applicable.

13.9 CONCLUSIONS

Research in the area of database systems is one of the most active areas in computer science and technology. This is because 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 (e.g., 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 1970s to the mid 1980s. 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 and given recent advances in hardware, software and network technology, recently there has been a renewed interest in the search for efficient atomic commit protocols in the context of traditional database systems as well as in emerging ones such as real-time database systems [57, 25] and multidatabase systems [13, 37, 66, 4]. In this chapter, we have discussed both previous and recent proposals of atomic commit protocols and evaluated the performance of a representative subset of them, both analytically as well as empirically using simulation, providing an insight in the performance trade-off between normal and recovery processing. We hope that it will provide the stimulus for further research and development of efficient atomic commit protocols.

Acknowledgments

We are grateful to Sujata Banerjee, George Kyrou and Susan Lauzac for their comments on previous versions of this chapter, and especially to Rob Conticello who participated in the design of the ACP simulator and helped in the writing of the corresponding section. Support for our database research from our respective institutions (University of Pittsburgh, University of Cyprus and Institute of Public

Administration, Saudi Arabia) and from the National Science Foundation grants IRI-9210588 and IRI-95020091 is gratefully acknowledged.

References

[1] Agrawal, R., M. Carey and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654, Dec. 1987

[2] Al-Houmaily, Y. J. Commit Processing in Distributed Database Systems and in Heterogeneous Multidatabase Systems. Ph.D. Thesis, Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania, April 1997.

[3] Al-Houmaily, Y. and P. Chrysanthis. Two-Phase Commit in Gigabit-Networked Distributed Databases. *Proc. of the 8th Int'l Conference on Parallel and Distributed Computing Systems*, pp. 554–560, Sept. 1995.

[4] Al-Houmaily, Y. J. and P. K. Chrysanthis. Dealing with Incompatible Presumptions of Commit Protocols in Multidatabase Systems. *Proc. of the 11th ACM Annual Symposium on Applied Computing*, pp. 186–195, Feb. 1996.

[5] Al-Houmaily, Y. and P. Chrysanthis. The Implicit Yes-Vote Commit Protocol with Delegation of Commitment. *Proc. of the 9th Int'l Conference on Parallel and Distributed Computing Systems*, pp. 804–810, Sept. 1996.

[6] Al-Houmaily, Y. and P. Chrysanthis. An Atomic Commit Protocol for Gigabit-Networked Distributed Databases. *Journal of Systems Architecture, The EUROMICRO Journal*, (To Appear), 1998.

[7] Al-Houmaily, Y., P. Chrysanthis and S. Levitan. Enhancing the Performance of Presumed Commit Protocol. *Proc. of the 12h ACM Annual Symposium on Applied Computing*, pp. 131–133, Feb. 1997.

[8] Al-Houmaily, Y., P. Chrysanthis and S. Levitan. An Argument in Favor of the Presumed Commit Protocol. *Proc. of the 13th Int'l Conference on Data Engineering*, pp. 255–265, April 1997.

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

[10] Bernstein P. A., V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[11] Bernstein, P., W. Emberton, V. Trehan. DECdta - Digital's Distributed Transaction Processing Architecture. *Digital Technical Journal*, Vol. 3, No. 1, Winter 1991.

[12] Braginski, E. The X/Open DTP Effort. *Proc. of the 4th Int'l Workshop on High Performance Transaction Systems*, Asilomar, California, Sept. 1991.

[13] Breitbart, Y., H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, 1(2):181–239, Oct. 1992.

[14] Carey, M. and M. Livny. Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication. *Proc. of the 14th Int'l Conference on Very Large Data Bases*, pp. 13–25, Aug. 1988.

[15] Carey, M. and M. Livny. Parallelism and Concurrency Control in Distributed Database Machines. *Proc. of the ACM SIGMOD Int'l Conference on the Management of Data*, pp. 122-133, June 1989.

[16] Cooper, E. Analysis of Distributed Commit Protocols. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pp. 175–183, June 1992.

[17] DeWitt, D., S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao and R. Rasmussen. The Gamma Database Machine Project, IEEE Transactions on Knowledge and Data Engineering, 2(1):44–69, June 1990.

[18] DeWitt, D., R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood. Implementation Techniques for Main Memory Database Systems. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pp. 1–8, 1984.

[19] Distributed TP: a) The TX specification P209, b) The XA Specification C193 6/91, c) The XA+ SpecificationS201, *X/Open Consortium*, Nov. 1992, Feb. 1992, April 1993

[20] Eswaran K., J. Gray, R. Lorie and I. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

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

[22] Gray, J. Notes on Data Base Operating Systems. In Bayer R., R.M. Graham, and G. Seegmuller (Eds), *Operating Systems: An Advanced Course*, *Lecture Notes in Computer Science*, Volume 60, pp. 393–481, Springer-Verlag, 1978.

[23] Gray, J. The Transaction Concept: Virtues and Limitations. *Proc. of the 7th Int'l Conference on Very Large Databases*, pp. 144–154, Sept. 1981.

[24] Gray, J. N. and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[25] Gupta, R., J. Haritsa, K. Ramamritham and S. Seshadri. Commit Processing in Distributed Real-Time Database Systems. *Proc. of the 17th IEEE Real-Time Systems Symposium*, Dec. 1996.

[26] Gupta, R., J. Haritsa and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. *Proc. of the ACM SIGMOD Int'l Conference on the Management of Data*, May 1997.

[27] Haerder, T. and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, Dec. 1983.

[28] Hammer M. and D. Shipman. Reliability Mechanisms for SDD–1: A System for Distributed Databases. *ACM Transactions on Database Systems*, 8(4):431–466, Dec. 1980.

[29] Helland, P. The LU6.2 protocol boundary: The 'L' stands for 'Lightweight'. *Proc. 3th Int'l Workshop on High Performance Transaction Systems*, Sept. 1989.

[30] *Information Technology - Open Systems Interconnection - Distributed Transaction Processing - Part 1: OSI TP Model; Part 2: OSI TP Service*, ISO/IEC JTC 1/SC 21 N, April 1992.

[31] Laing, W., Johnson, J. and R. Landau. Transaction Management Support in the VMS Operating System Kernel, *Digital Technical Journal*, Volume 3, No. 1, Winter 1991.

[32] Lampson, B. Atomic Transactions. *Distributed Systems: Architecture and Implementation - An Advanced Course*, B. Lampson (Ed.), *Lecture Notes in Computer Science*, Volume 105, pp. 246-265, Springer-Verlag, 1981.

[33] Lampson, B. and D. Lomet. A New Presumed Commit Optimization for Two Phase Commit. *Proc. of the 19th Conference on Very Large Databases*, pp. 630-640, Aug. 1993.

[34] LeLann, G. Error Recovery. *Distributed Systems: Architecture and Implementation - An Advanced Course, Lecture Notes in Computer Science*, Vol. 105, pp. 371– 376, Springer-Verlag, 1981.

[35] Liu, M. L., D. Agrawal and A. El Abbadi. The Performance of Two-Phase Commit Protocols in the Presence of Site Failures. *Proc. of the 24th Int'l Symposium on Fault-Tolerant Computing*, pp. 234–243, 1994.

[36] Maslak, B., Showalter, J. and T. Szczygielski. Coordinated Resource Recovery in VM/ESA. *IBM Systems Journal*, Vol. 30, No. 1, 1991

[37] Mehrotra, S., R. Rastogi, H. Korth and A. Silberschatz. A Transaction Model for Multidatabase System. *Proc. of the Int'l Conference on Distributed Computing Systems*, pp. 56–63, June 1992.

[38] Mohan, C. and B. Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. *Proc. of the 2nd ACM SIGACT/SICOPS Symposium on Principles of Distributed Computing*, Aug. 1983.

[39] Mohan, C., B. Lindsay and R. Obermarck. Transaction Management in the R^* Distributed Data Base Management System. *ACM Transactions on Database Systems*, 11(4):378–396, Dec. 1986.

[40] Mohan, C., D. Hderle, B. Lindsay, H. Pirahesh and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transaction on Database Systems*, 17(1):94–162, Mar. 1992.

[41] Mohan, C., K. Britton, A. Citron and G. Samaras. Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU6.2 Commit Protocols. *An Int'l Workshop on Advance Transaction Models and Architectures*, India, Sept. 1996.

[42] *Performance of Concurrency Control Mechanisms in Centralized Database Management Systems*, V. Kumar, ed., Prentice Hall, 1995.

[43] Primatesta, F. *TUXEDO, An Open Approach to OLTP.* Prentice Hall, 1995.

[44] Ramamritham, K. and P. K. Chrysanthis. *Advances in Concurrency Control and Transaction Processing*, IEEE Computer Society Press, 1997.

[45] Rosenkrantz D., R. Stearns and P. M. Lewis II. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[46] Rothermel, K. and S. Pappe. Open Commit Protocols for the Tree of Processes Model. *Proc. of the 10th Int'l Conference on Distributed Computer Systems*, pp. 236–244, 1990.

[47] Rothermel, K. and S. Pappe. Open Commit Protocols Tolerating Commission Failures. *ACM Transactions on Database Systems*, 18(2):289–332, June 1993.

[48] Samaras, G., K. Britton, A. Citron and C. Mohan. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. *Proc. of the 9th Int'l Conference on Data Engineering*, pp. 520-529, Feb. 1993.

[49] Samaras, G., Britton, K., Citron, A. and C. Mohan. Enhancing SNA's LU6.2 Sync Point to Include Presumed Abort Protocol, *IBM Technical Report TR29.1751*, IBM Research Triangle Park, Aug. 1993.

[50] Samaras, G., K. Britton, A. Citron and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *Distributed and Parallel Databases*, 3(4):325–360, Oct. 1995.

[51] Samaras, G., S.D. Nikolopoulos. Algorithmic Techniques Incorporating Heuristic Decisions in Commit Protocols. *Proc. of the 25th Euromicro Conference*, Sept. 1995.

[52] Samaras, G. Heuristic Decisions and Commit Protocols. *University of Cyprus Technical Report CS-TR96-17*, Dec. 1996.

[53] Schlighting, R. and F. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computing Systems*, 1(3):222–238, Sept. 1983.

[54] Sherman, M. Architecture of the Encina Distributed Transaction Processing Family. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pp. 460–463, May 1993.

[55] Skeen D. Non-blocking Commit Protocols. *Proc. of the ACM SIGMOD Int'l Conference on the Management of Data.* pp. 133–142, May 1981.

[56] Skeen D., and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, 9(3):219–228, May 1983.

[57] Soparkar N., E. Levy, H. Korth and A. Silberschatz. Adaptive Commitment for Real-time Distributed Transactions. *Dept. of Computer Science TR92-15*, Univ. of Texas-Austin, 1992.

[58] Spector, A. Open, Distributed Transaction Processing with Encina. *Proc. 4th Int'l Workshop on High Performance Transaction Systems*, Sept. 1991.

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

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

[61] Stamos, J. and F. Cristian, Coordinator Log Transaction Execution Protocol. *Distributed and Parallel Databases*, Vol. 1, pp. 383–408, 1993.

[62] Stonebraker, M. Concurrency Control and Consistency of Multiple of Data in Distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, May 1979.

[63] *Systems Network Architecture LU 6.2 Reference: Peer Protocols*, Document Number SC31-6808-1, IBM, Sept. 1990.

[64] *Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2*, Document Number SC30-3084-5, IBM, June 1993.

[65] *Systems Network Architecture. SYNC Point Services Architecture Reference*, Document Number SC31-8134, IBM, Sept. 1994.

[66] Tal, A. and Alonso, R. Commit Protocols for Externalized–Commit Heterogeneous Databases. *Distributed and Parallel Databases*, 2(2):209–234, Apr. 1994.

[67] Upton IV, F. OSI Distributed Transaction Processing, An Overview. *Proc. of the 4th Int'l Workshop on High Performance Transaction Systems*, Sept. 1991.

[68] Zimran, E. The Two-Phase Commit Performance of The DECdtm Services. *Proc. of the 11th Symposium on Reliable Distributed Systems*, pp. 29–38, 1992.