Distributed and Parallel Databases, 3, 325–360 (1995)

# Two-Phase Commit Optimizations in a Commercial Distributed Environment*

GEORGE SAMARAS cssamara@zeus.cc.ucy.ac.cy
*IBM Distributed Systems Architecture, IBM Corporation, P.O. Box 12195, Research Triangle Park, NC 27709, USA and Department of Computer Science, University of Cyprus, Nicosia, Cyprus*

KATHRYN BRITTON AND ANDREW CITRON {brittonk, citron}@vnet.ibm.com
*IBM Distributed Systems Architecture, IBM Corporation, P.O. Box 12195, Research Triangle Park, NC 27709, USA*

C. MOHAN mohan@almaden.ibm.com
*Database Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA*

*Received February 23, 1993; Revised November 30, 1994*

**Recommended by:** Amit Sheth

**Abstract.** An atomic commit protocol can ensure that all participants in a distributed transaction reach consistent states, whether or not system or network failures occur. The atomic commit protocol used in industry and academia is the well-known two-phase commit (2PC) protocol, which has been the subject of considerable work and technical literature for some years.

Much of the literature focuses on improving performance in failure cases by providing a non-blocking 2PC that streamlines recovery processing at the expense of extra processing in the normal case. We focus on improving performance in the normal case based on two assumptions: first, that networks and systems are becoming increasingly reliable, and second, that the need to support high-volume transactions requires a streamlined protocol for the normal case.

In this paper, various optimizations are presented and analyzed in terms of reliability, savings in log writes and network traffic, and reduction in resource lock time. The paper's unique contributions include the description of some optimizations not described elsewhere in the literature and a systematic comparison of the optimizations and the environments where they cause the most benefit. Furthermore, it analyzes the feasibility and performance of several optimization combinations, identifying situations where they are effective.

**Keywords:** agreement protocols, distributed systems, transaction management, SNA LU 6.2, communication protocols, commit protocols, recovery, fault tolerance

## 1. Introduction

A *distributed transaction* is the execution of one or more statements that access data distributed on different systems. A distributed *commit protocol* is required to ensure that the effects of a distributed transaction are *atomic*, that is, either all the effects of the transaction persist or none persist, whether or not failures occur.

A well-known commit protocol is the two-phase commit (2PC) protocol [9, 17]. This protocol ensures that all participants commit if and only if all can commit successfully. The two phases are the *voting* phase and the *decision* phase. During the voting phase, one

*Disclaimer: Some of the optimizations described in this paper may never be shipped in an IBM product. Others may change before they are shipped.

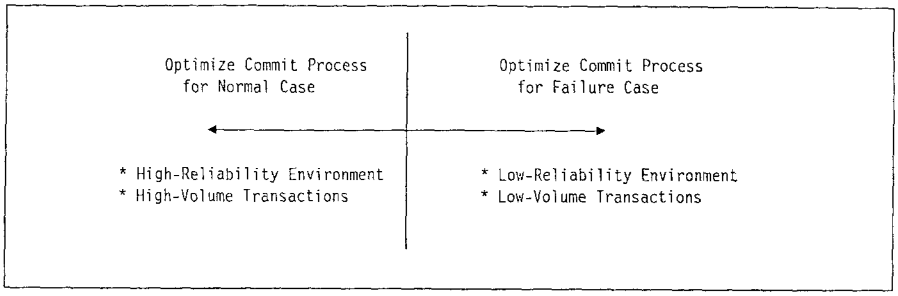
Optimize Commit Process for Normal Case
Optimize Commit Process for Failure Case
* High-Reliability Environment
* High-Volume Transactions
* Low-Reliability Environment
* Low-Volume Transactions

*Figure 1.* Commit performance optimization for different environments.

participant in the transaction, known as the coordinator of the commit protocol, asks all the other participants to prepare to commit. A participant votes YES if it can guarantee that it can perform the outcome requested by the coordinator, either commit or abort, whether or not system or network failures occur. If a participant is unable to prepare to commit for any reason, it votes NO. During the decision phase the coordinator propagates the outcome of the transaction to all participants: if all participants voted YES, the commit outcome is propagated; if any participant voted NO, the abort outcome is propagated. Each participant in the transaction commits or aborts the effects of the transaction based on the outcome. It can then release locks on local resources, such as databases or files, making them available to other transactions.

The performance of a commit protocol substantially affects the transaction volume that a system can support. As pointed out in [29], for transaction processing applications such as hotel reservations, airline reservations, stock market transactions, banking applications, or credit card systems, the commit processing takes up a substantial portion of the transaction time. For example, it was shown in [29] that the commit processing part of a transaction updating one record of a general-purpose database typically represents about a third of the transaction duration. For distributed systems where network messages and delays are involved, the relative commit cost is, on average, much higher.

A faster commit protocol can improve transaction throughput in two ways: first, by reducing the commit duration for each transaction, and second, by causing locks to be released sooner, reducing the wait time of other transactions.

The problem of improving 2PC performance can be met using two different approaches (see Fig. 1). The first approach concentrates on reducing recovery time, and therefore lock time, for failure cases. In an environment prone to failures, transactions can be blocked indefinitely waiting for the recovery of a failed site. Since it is unknown whether the transaction will commit or abort, resource locks cannot be released. Thus, other transactions can also be blocked waiting for the locked resources to become available. Much research [5, 27] has concentrated on providing a (nearly) non-blocking 2PC variation, i.e., one that adds extra messages to the basic 2PC protocol in order to reduce the blocking delay required to resolve the transaction outcome following a failure. Thus, the normal non-failure case is slowed down to prevent intolerable delays following failures.

The tradeoff of reducing recovery time at the expense of increasing the duration of normal commit operations may not be acceptable in a highly reliable environment characterized by high-volume transactions. The second approach focuses on optimizing the basic 2PC protocol for this environment. The rest of this paper describes several optimizations that reduce the number of messages and/or local processing required for the non-failure case, sometimes at the expense of greater recovery processing and delay for the failure case. These optimizations take advantage of properties that are common in real-world distributed transactions.

For the failure cases (hopefully, rare) where the protocol outcome is blocked, certain participants might choose not to wait for recovery processing to discover the outcome because of valuable locks being held [23, 20]. Rather than waiting, these participants unilaterally commit or abort the transaction. This *heuristic decision* may damage the consistency of the transaction. Heuristic decisions and their effect on 2PC reliability have been, to our knowledge, little addressed in the literature, but they are considered a practical necessity in the commercial environment. Heuristic decisions are discussed in Section 3. A commit protocol and its optimizations should be able to cope with these heuristic decisions: recognize them and report them reliably. The need for heuristic decisions cannot be entirely avoided even with a "so-called" non-blocking 2PC protocol, although the window in which they might occur is reduced.

This paper presents several 2PC optimizations, and analyzes them in terms of reliability (potential for heuristic decisions), number of log writes, network traffic, resource lock time, and other tradeoffs. Its unique contributions include a description of IBM's Presumed Nothing protocols and several new optimizations, particularly ones that affect peer-to-peer transactions (i.e., Leaving Inactive Partners Out, Last Agent), and ones dealing with heuristic decisions (i.e., Wait For Outcome, Vote Reliable). It also shows how resource managers can use their specific characteristics to further improve the performance of the commit processing (i.e., Vote Reliable optimization). An interesting optimization (Long Locks) that uses network capabilities to further improve the 2PC performance is also described. Finally, the paper presents how certain combinations affect the performance, correctness, and reliability of the 2PC processing. Some of these optimizations have been designed on top of IBM's LU6.2 communication protocol. However, their presentation here is independent of any communication protocol. LU6.2 implementation specifics for some of these optimizations can be found in [31] and [21].

Section 2 presents the distributed transaction model used in this paper to describe transactions and commit processing. Section 3 discusses the aspects of commit processing that most affect 2PC performance. Section 4 introduces a 2PC protocol that is used as a baseline for comparing the 2PC variations introduced in the rest of the paper. Section 5 presents the Presumed Abort (PA) and IBM's Presumed Nothing (PN) protocols and their usefulness within the commercial sector. Section 6 discusses several optimizations that are refinements of PN or PA or both, along with their advantages and tradeoffs in different environments. Section 7 describes the effects of combining these optimizations. Section 8 provides a performance analysis of the presented optimizations. Section 9 reviews related work, and Section 10 concludes the paper.

## 2. Distributed transaction execution

A distributed system consists of a set of computing nodes linked by a communications network. The nodes of the system cooperate with each other in order to process distributed computations. For the purpose of cooperation, the nodes communicate by exchanging messages via the communications network.

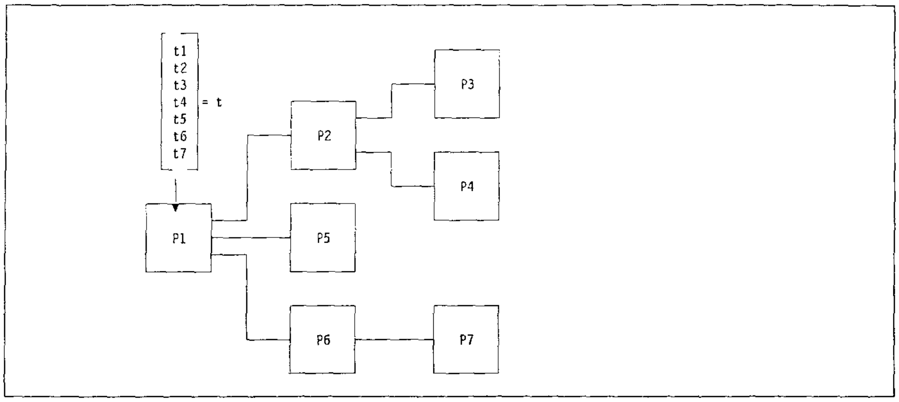

*Figure 2.* A process tree within the client/server model.

A user's *application program* initiates or participates in a distributed computation which consists of a set of transactions executed in a serial fashion. A transaction (or logical unit of work, LUW) consists of a set of operations that are executed to perform a particular logical task, generally making changes to data resources such as databases or files. The changes to these resources must be committed or aborted before the next transaction in the series can be initiated.

A distributed computation is associated with a tree of processes[1] that is created as the application executes. The process tree links the processes that perform the transactions of the distributed computation. Processes may be created at remote nodes (and even locally) in response to the data access requirements imposed by the application program. Consequently, there exists a creator-createe relationship between the processes. The tree may grow as new sites are accessed by the transactions. Subtrees may disappear either in response to application logic or because of site and communication link failures.

Figure 2 shows a process tree together with the associated distributed computation $t = \{t1, t2, t3, t4, t5, t6, t7\}$ as it is executed within a hierarchical model, such as that usually associated with client/server computing. In this model all the transactions $t1, \ldots, t6, t7$ constituting the computation $t$ are initiated by the root process representing the client. The server processes are participating in the computation by executing requests from the client. They neither initiate work independently nor issue requests to the client. Servers can issue requests to additional servers on behalf of the client; the subordinate servers therefore treat them as client processes. All requests flow in one direction, from client to server to subordinate servers. Thus, in this model the process tree has a fixed hierarchical structure that grows in only one direction (downstream). In addition, the client process at the root is the overall initiator of the commit protocol (2PC). Consequently the commit protocol tree is exactly the same as the process tree so that creator-createe relationship implies the coordinator-subordinate relationship for the purpose of executing the commit protocol.

Figure 3 shows the distributed computation $t$ within an alternative, peer-to-peer model [31]. In the peer environment each process has the same privileges and rights as any other process in the process tree. Any program can initiate a transaction. Two programs can initiate work independently with or without any communication between them. This is in contrast to the hierarchical model, where the client starts the transaction and the servers wait

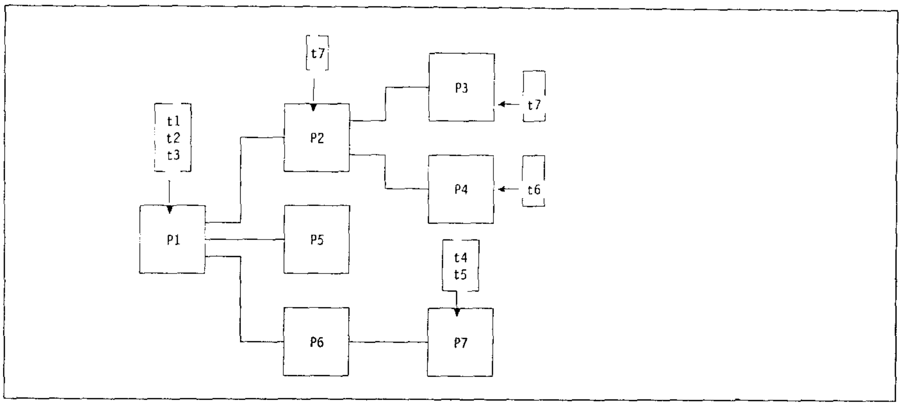

*Figure 3.* A process tree within the peer-peer model.

until they get requests from clients or other servers. For example, in Fig. 3, transactions *t*1, *t*2, *t*3 are initiated by the root process *p*1, transaction *t*4, *t*5 by process *p*7, transaction *t*6 by process *p*4, and transaction *t*7 by both *p*3 and *p*2. Any participant in the transaction can initiate the commit protocol and thus become the root of the transaction commit tree. Therefore, the member of the process tree that serves as the coordinator can change from one transaction to another. The coordinator-subordinate relationship is established at the beginning of commit processing and endures only for the current transaction. This ability to allow any participant to coordinate the commit procedure can be particularly useful if the request that starts a particular distributed transaction comes from an unreliable node, such as a workstation that is frequently turned on and off. In this case, it may be advantageous to have more reliable hosts coordinate the commit procedure [4, 25], since they are more likely to continue to be available when failures and recoveries cause substantial delays in a commit procedure.

As shown in Fig. 4, a process participating in a transaction accesses local resources such as databases and files. A remote request is sent via the communication network[2] to a remote process, which can access either local resources or additional remote resources.

Once the computations of a transaction are completed, the application instructs the transaction manager (TM) of its site to initiate and coordinate the commit protocol. Two types of components participate in 2PC protocol: *local resource managers* (*LRMs*), such as database and file managers, which have responsibility for the state of their resources only, and *transaction managers* (*TMs*), which manage multiple participants, including both local resource managers and other remote transaction managers.

The TMs and LRMs that participate in 2PC include one *coordinator* and one or more *subordinates*. The coordinator is the TM acting on behalf of the process that initiates a commit operation; a subordinate is either an LRM or a remote TM that is acting on behalf of another process in the distributed transaction. Remote TMs may also have subordinate LRMs and TMs. The coordinator is the one that coordinates the final outcome of the commit processing. The coordinator must arrive at a COMMIT or ABORT decision and propagate that decision to all subordinates. Subordinate TMs propagate the decision to their subordinate TMs or LRMs. Thus, the subordinates defer to the coordinator for the result of the commit decision.

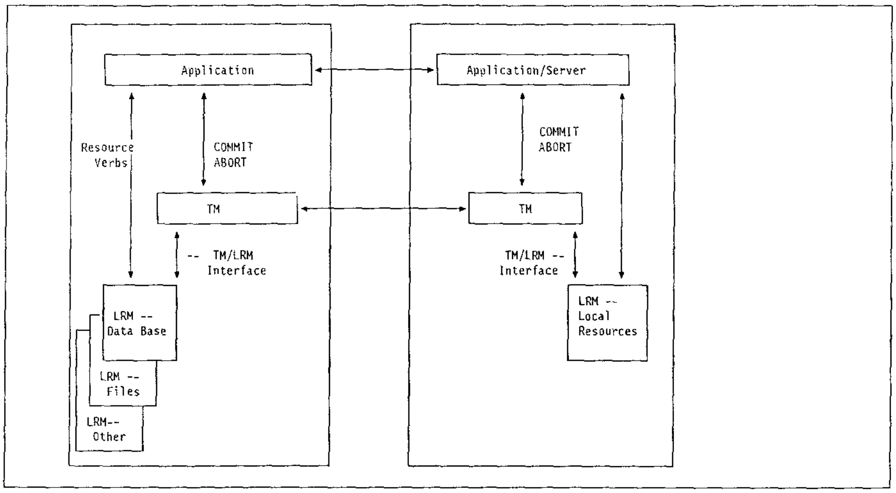

*Figure 4.* Components involved in a transaction and transaction commit tree.

## 3. Two-phase commit performance considerations

This section describes aspects of a distributed 2PC protocol that have the greatest impact on performance and reliability: network traffic, logging, and heuristic decisions.

### *Network traffic*

The 2PC protocol involves network traffic to convey instructions from the coordinator TM to subordinate TMs and to convey the responses from the subordinates back to the coordinator.

Any message that is sent over the network slows down the commit protocols since it adds network transit delays. Several of the 2PC optimizations described later in this paper reduce commit time by reducing the number of messages sent. Sending messages to different participants in parallel also reduces the delay caused by network traffic. In some cases, reducing the number of messages and parallelism are in conflict (see last-agent optimization).

### *Logging*

Participant TMs and LRMs log information about intermediate states of a commit operation in order to be able to recreate the state of the transaction after a system failure. Logged information is data written in non-volatile storage that can be used to figure out how to return distributed resources to consistent states following the loss of working memory of the transaction state.

During *forced log writes*, the 2PC operation is suspended; the TM does nothing until the record is guaranteed to be in stable storage. *Non-forced log writes* do not suspend the 2PC operation but are not guaranteed to survive a system failure. A non-forced log write is written to nonvolatile storage when the next forced log write occurs, or when some other log manager event occurs, such as log buffer overflow. Since non-forced log writes are not guaranteed, information that is vital for correct processing after a system failure must be forced. However, forced writes are not required when the logged information can be recreated after a failure by recovery processing.

A 2PC performance goal is to minimize the number of times a log write is forced. A forced log entry slows down commit protocols because the system waits until the entry is written to nonvolatile storage. Minimizing forced log writes and conducting extra recovery processing to regain the lost information is one way to optimize the normal, non-failure case rather than the failure case.

### *Heuristic decisions*

If one or more of the systems involved in a transaction fails during a two-phase commit operation, there can be substantial delays before the operation completes and the affected resources are available for use by other transactions. Because these delays can cause business to be lost, most commercial systems give an operator a way to force a blocked transaction to complete. In the process, the operator must decide whether to commit or abort the changes to affected data resources. Once a two-phase commit operation has started, either choice runs the risk of causing heuristic damage, that is, of making the local resources abort when the rest of the transaction commits, or vice versa.

Consider an airline reservation database with the records for a particular set of planes locked waiting for a transaction to complete. The operator starts getting calls from irate travel agents, who want to sell tickets on those airplanes. On investigation, the system operator learns that the system coordinating the transaction has failed, with an expected fix time of two hours. To free the data records for use by other transactions, the operator forces the transaction to complete locally, making a heuristic decision to commit the local changes. Later it turns out the operator made the wrong choice, since the rest of the transaction aborted. Finding and fixing inconsistencies can be time-consuming and expensive. A business may find it necessary to risk heuristic damage and database inconsistency for one transaction in order to make the database available for other transactions. Whether to allow heuristic decisions involves business tradeoffs between the cost of fixing database inconsistencies and the cost of missed opportunities.

A heuristic decision is usually taken by a system operator (or programmed operator) in the absence of a direct command from the commit coordinator. If a heuristic decision is required, it should be done in consultation with the system operators of the other systems that were part of the distributed transaction. A two-phase commit protocol that detects and reports damage at least simplifies the task of identifying problems that must be fixed.

A simple case of heuristic damage reporting is shown in Fig. 8. More complex cases are shown in Figs. 23 and 24.

## 4. Baseline two-phase commit

This section illustrates the effects of network messages and required log writes [22, 23] on the performance of a basic distributed 2PC protocol [9, 17] that is used as a comparison baseline for the optimizations that follow.

In the first, or *voting* phase of two-phase commit, the coordinator issues *prepare* messages in parallel to all subordinates to determine whether they are willing to commit. Subordinates may be LRMs or remote TMs. Each subordinate votes YES or NO indicating its willingness to commit or abort the transaction. Before voting YES, a subordinate force-writes a *prepared* log record that ensures that it can successfully commit or abort the transaction, even if a system failure causes it to lose working memory of the transaction. Thus, a database manager acting as a subordinate forces enough information so that it can either recreate or undo the changes made during the transaction. A TM force-writes enough information so that it can initiate recovery processing following a failure, information including the identity of the coordinator, the identities of subordinates, and the state of the 2PC operation.

A YES vote places the subordinate in an *in-doubt* state, implying that it will neither commit nor abort the transaction without an explicit order from the coordinator. If a subordinate decides to abort the transaction, it force-writes an *abort* log record and sends a NO vote to the coordinator. Since a NO vote defines the outcome of the transaction, the subordinate does not need to wait for the coordinator decision any more. Therefore, the subordinate aborts the transaction, releases all its locks, and then forgets the transaction.

The second, or *decision*, phase begins after the coordinator receives all expected votes. If all subordinates voted YES, the coordinator decides to commit; otherwise it decides to abort. The coordinator propagates the decision to all subordinates as either an order to commit or an order to abort. Subordinates that voted to abort during phase one are not included in the second phase since they already know the outcome.

Because the coordinator's decision needs to survive failures, a *commit* or *abort* log record is force logged before the decision is propagated to all its subordinates. The completion of the force-write takes the transaction to its *committing* or *aborting* state. Each subordinate, after receiving the commit/abort order from the coordinator, moves into the committing/aborting state, force-writes a commit/abort log record to ensure that the transaction will be committed/aborted, and then sends an *acknowledgment (Ack)* message back to the coordinator indicating that the subordinate will commit/abort as the coordinator requested. The subordinate then commits/aborts, and forgets about the transaction. The coordinator, after collecting acknowledgment messages from all subordinates that voted YES, writes a non-forced END log record and forgets the transaction. The END log record indicates that all subordinates have successfully completed the commit processing and thus, no recovery processing is required if a failure occurs.

Figure 5 shows a time sequence of the 2PC protocol for a coordinator with one subordinate.

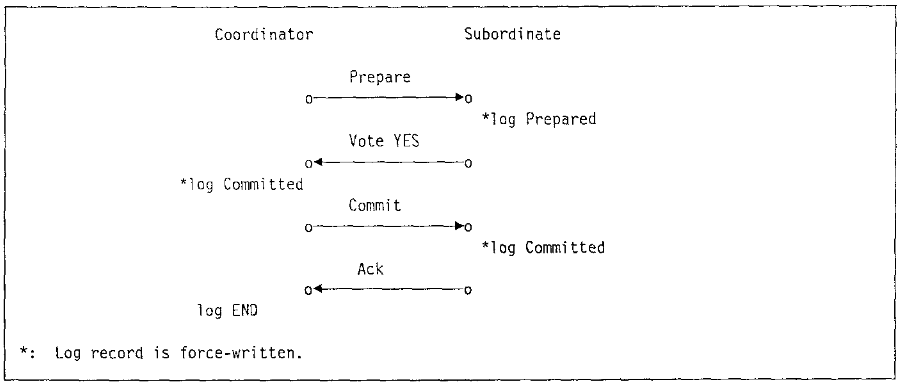

*Figure 5.* Simple two-phase commit processing.

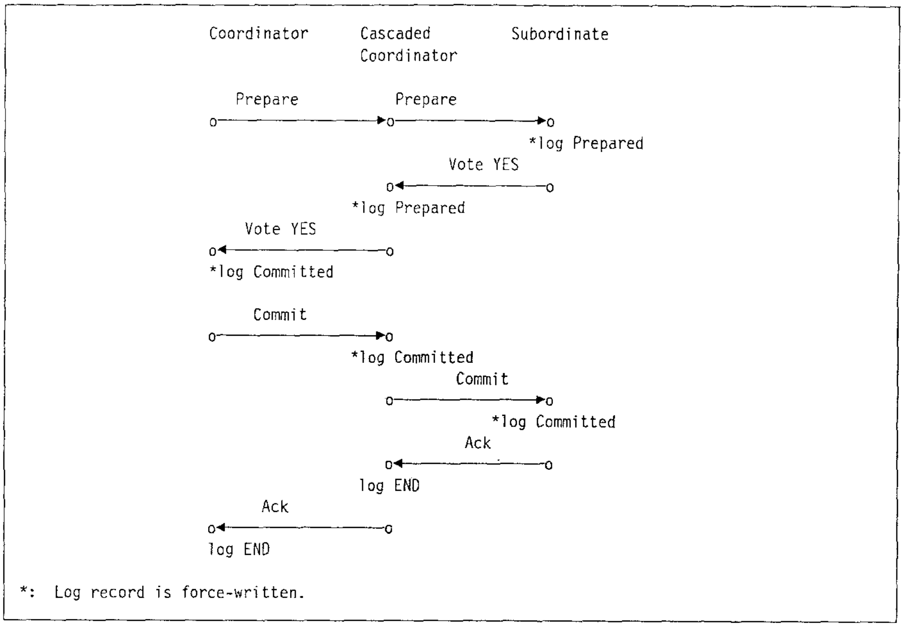

*Figure 6.* Two-phase commit processing with intermediate coordinator.

A subordinate agent may also function as a cascaded (intermediate) coordinator to downstream subordinates. The coordinator, cascaded coordinators, and remaining subordinates form a transaction commit tree. The cascaded coordinator propagates messages from the coordinator downstream and collects responses from its subordinates to send back upstream to the coordinator. Figure 6 shows a time sequence of the 2PC protocol with a cascaded coordinator. A participant in the tree does not generally know whether its coordinator is the root of the commit tree or a cascaded coordinator, just as a coordinator does not know whether its subordinates are cascaded coordinators or leaf subordinates.

## *Baseline summary*

The overall cost of the baseline 2PC protocol for the commit case is: each subordinate writes three log records (one prepared record, one committed/abort record and one END[3] record—the prepared and the committed records are forced) and sends two messages. The coordinator sends two messages to each immediate subordinate and writes two log records (one committed record and one END record—the committed record is forced). For a transaction commit tree with *n* participants the cost is **4(n-1)** messages, **2n-1** forced writes and **n** non-forced writes.

The basic 2PC protocol survives failures and derives a consistent single outcome for a transaction. However, many commercial products minimize the number of message exchanges and forced writes to optimize for high-volume, performance-sensitive distributed transactions. The next section describes two variants of the basic 2PC protocol and discusses the impact of heuristic decisions and heuristic damage notification.

## 5. Two-phase commit variations

*Presumed Nothing (PN)*

Presumed Nothing was developed in the mid 1970's for the peer-to-peer environment that is supported by LU 6.2 (also known as APPC) [31, 32] and initially by LU6.1 [24]. The PN design effort was done independently from the 2PC effort [34]. PN was designed and developed for the commercial environment and, so far, IBM has implemented it in CICS/MVS[4] [6], and VM/ESA [20].

The peer-to-peer environment has led to the following unusual feature of PN. Any participant in the transaction can decide to initiate a commit operation and thus become the root of the transaction commit tree (the coordinator). Thus, the member of a collection of cooperating processes that serves as the coordinator can change from one transaction to the next. Since the communicating processes are considered peers, there is no hierarchical relationship among them that determines the best place to initiate commit processing; therefore it is left to application design to determine which process should be the commit coordinator for a particular transaction. It is an error for two participants to initiate commit processing independently for the same transaction, since that would mean two TMs owning the commit decision; if this occurs, the transaction aborts.

As a result, the coordinator of a particular commit operation is not known in advance; it is only known once 2PC processing starts.

Since it was designed for a real-world environment with intense demands on data resources, the PN protocol explicitly accommodates *heuristic decisions* resulting from intolerable delays. Since there are situations where heuristic decisions need to be made, the PN designers felt it was important for the root coordinator to be informed of any *heuristic damage* that occurred, i.e., any heuristic decision inconsistent with the outcome of the transaction.

The primary impact of these design decisions on the PN protocols is that the coordinator (or cascaded-coordinator) must log a *commit-pending* record before sending the prepare message to subordinates. This is necessary because the coordinator must remember that there are subordinates. The subordinates may be waiting for the outcome or may have made heuristic decisions. The coordinator is responsible for initiating recovery processing both to allow the subordinates to complete commit processing and to find out whether they made heuristic decisions.

In Fig. 7, the changes from Fig. 6 are highlighted. The need for accurate reporting causes the application at the root of the transaction commit tree to be kept in suspense about the outcome of the 2PC operation until all acknowledgments are collected. If the application were informed earlier, it could proceed on the assumption that the entire transaction were committed or aborted, when actually heuristic damage might have occurred. Figure 8 illustrates PN heuristic damage reporting.

Thus, PN protocols provide reliable reporting of damage at the expense of an extra log force and collecting acknowledgments from all subordinates. However, to offset these performance penalties, PN, as implemented in LU 6.2, includes a number of other optimizations described in the next section: last agent, long locks, vote read only, and wait for outcome.

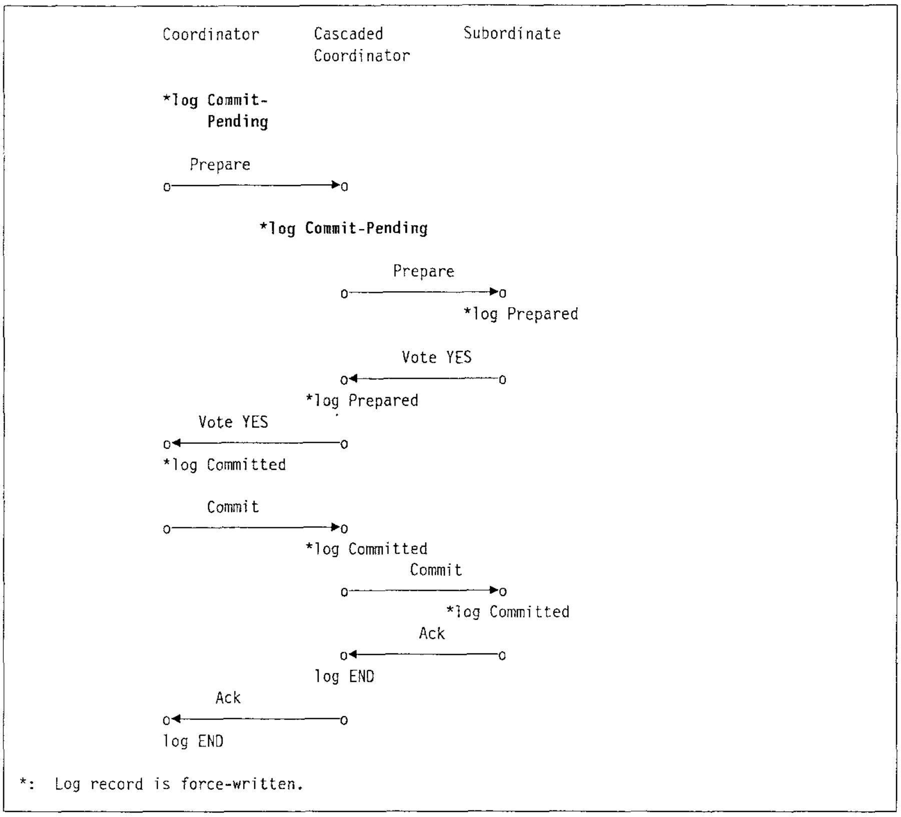

*Figure 7.* Presumed nothing commit processing with intermediate coordinator.

## *Presumed Abort (PA)*

Presumed Abort [22, 23] is an extension of the basic 2PC protocol that has been widely studied in academia and industry.[5] It has been implemented by a number of commercial products,[4] i.e, Tandem's TMF [33], DEC's VAX/VMS [1, 16], Transarc's Encina Product Suite [28], and Unix System Laboratories' TUXEDO [14], and is now part of the ISO-OSI [35] and X/Open[4] [3] distributed transaction processing standards. PA was developed for the $R*$ distributed database project [18, 19]. In the $R*$ client-server model, the participants have fixed requester-server roles. Servers initiate no work unless the requester asks for it. Servers never ask their clients to act in the role of server. The coordinator is the TM of the client, and the subordinates are the servers.

Like the baseline 2PC, PA does not log before sending the Prepare message. Since the PA processing involved in successfully committing a transaction is the same as that shown for basic 2PC in Figs. 5 and 6, no flow diagram is shown here for the commit case.

Unlike the baseline 2PC, a subordinate does not have to force write an abort record before acknowledging an abort command. If a prepared record is found on its log after a crash, the subordinate initiates recovery processing with its coordinator. Similarly, the coordinator does not have to force write the abort record. If the coordinator has no information about

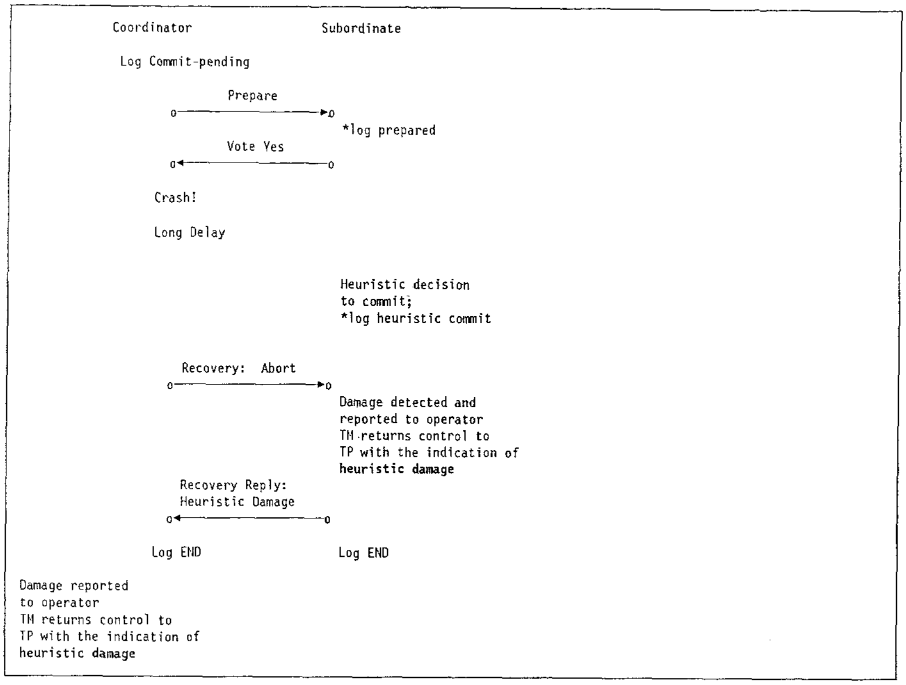
Coordinator
Subordinate
Log Commit-pending
Prepare
*log prepared
Vote Yes
Crash!
Long Delay
Heuristic decision to commit;
*log heuristic commit
Recovery: Abort
Damage detected and reported to operator
TM returns control to TP with the indication of heuristic damage
Recovery Reply: Heuristic Damage
Log END
Log END
Damage reported to operator
TM returns control to TP with the indication of heuristic damage

*Figure 8.* Heuristic damage example.

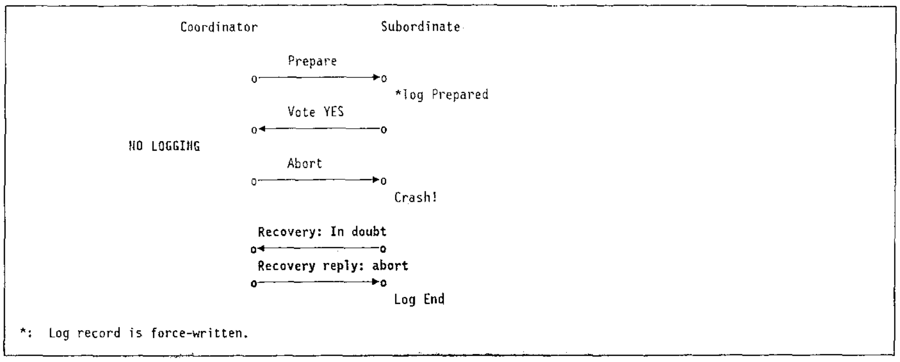

*Figure 9.* Presumed abort with an aborted transaction.

the transaction on its log, it presumes that the transaction aborted and tells the subordinate to abort; hence the name *presumed abort.*

The subordinate (server) initiates recovery processing when it finds itself in doubt after a failure. This is necessary since the coordinator may have no memory of the transaction if it also failed.

Differences between presumed abort and the baseline 2PC protocol are highlighted in Fig. 9 for a transaction that aborts, followed by a subordinate failure. This in contrast to the baseline 2PC coordinator, which is responsible for initiating recovery and therefore must force an abort log record before sending the abort message to the subordinate. The presumed abort coordinator performs no logging at all in this case, since the subordinate can initiate recovery.

The PA protocol incorporates the read-only and leave-inactive-partners-out optimizations described in the next section.

In $R*$, heuristic decisions that caused database inconsistencies were only reported to the immediate coordinator, which is not necessarily the root of the tree, and to the subordinate system's operator. This meant that the root coordinator might be told the transaction committed successfully when it had not. This was considered acceptable because heuristic decisions did not happen frequently.

The optimizations developed by PA for the client-server environment have been generalized to be incorporated in the peer-to-peer model [21].

## 6. 2PC optimizations

This section describes several optimizations to the PA or PN protocols or both, some of which have been previously published [22, 23, 9, 31]. These optimizations are tuned toward the normal non-failure case. See [21] for a description of the way some of these optimizations fit with LU 6.2's half-duplex conversational model.

Our analysis assumes that we are dealing with a transaction tree with **n** participants unless otherwise noted.

### *Read only*

A partner that has participated in a transaction, but has not performed any updates, is allowed to *vote read-only*. This vote implies that the effects of commit and abort outcomes would be identical for that subordinate. That partner is left out of the second phase of the commit processing and avoids any log writes [22, 23].

A cascaded coordinator is allowed to vote "read-only" if and only if all its subordinates have voted read-only; otherwise it needs to learn the outcome in order to propagate it to the subordinates that did not vote read-only.

For an environment that is dominated by read-only transactions this optimization provides enormous savings, since it reduces the commit operation to a one-phase commit operation.

This optimization is used in both the PA and PN protocols. The PA protocol is especially optimized for this type of transaction: PA performs no logging at all if all subordinates vote read-only. Figure 10 illustrates the read-only optimization with the PA protocol. PN still has the coordinator log a *Commit-pending* record, but the subordinate performs no logging.

However, this optimization has some drawbacks. First, the read-only partners are not informed of the final outcome of the transaction, which could cause undesirable side effects if the applications are written to use this information in any way. Second, the read-only optimization can cause serialization problems. A subordinate can receive a prepare message before it is finished with its part of the transaction. In the peer-to-peer environment it is allowed to finish before it votes. Consider the case where participants Pa and Pb are subordinates to a common coordinator. Both receive prepare messages. Pa votes read-only and releases locks before Pb has finished with the transaction. Pb needs to access a resource that Pa unlocked, but another unrelated transaction has locked the resource and

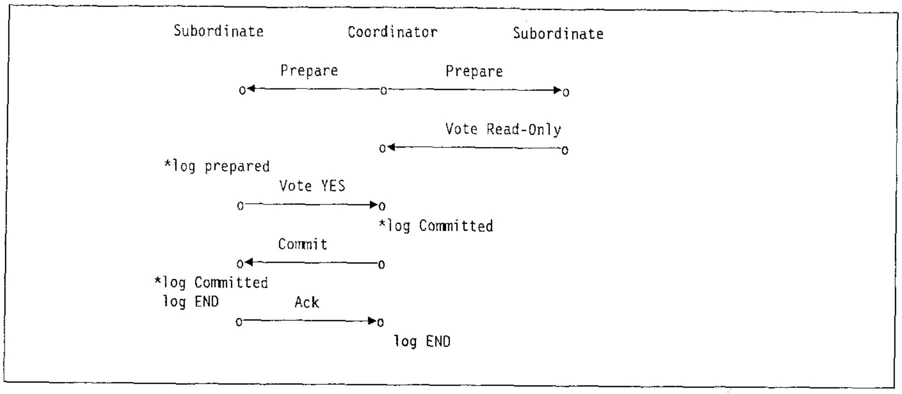

*Figure 10.* Partial read-only commit processing.

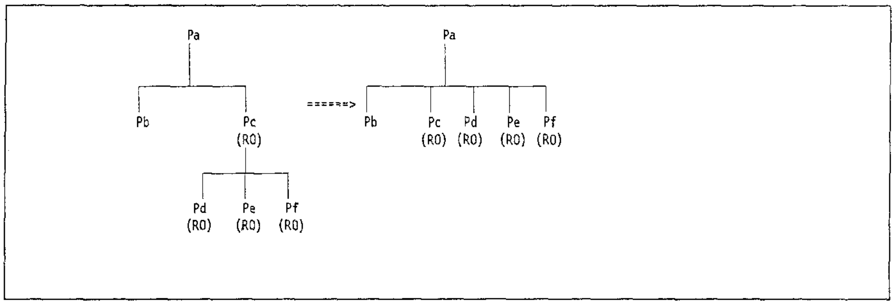

*Figure 11.* Equivalent read-only trees in terms of messages and log writes. The participants that vote read-only (or vote ok-to-leave-out) are marked "(RO)."

changed it. When Pb gains access to the resource, the resource is not the same as it was when Pa unlocked it. Thus, use of the read-only optimization prior to global termination of a transaction may violate two-phase locking and serialization rules, and may cause the transaction to behave incorrectly.

However, these serialization problems do not occur in a requester/server environment, since the servers do not initiate independent work and the requester does not initiate commit processing until the transaction work is complete.

The tree topology can affect whether or not a participant in the tree can vote read-only because a participant cannot vote read-only if any of its subordinates voted YES or NO. Figure 11 shows two different tree topologies. In the right tree, all participants make the vote read-only decision independently. In the left tree, Pc's ability to vote read-only is affected by the votes of Pd, Pe, and Pf. However, if the two trees have the same set of participants voting read-only, the savings in messages and log writes are identical.

This equivalence simplifies the performance analysis for complex transaction commit trees with read-only partners, since the savings are associated only with the partners voting read-only. For a transaction commit tree of **n** members and **m** participants that vote read-only, the savings amount to $2m$ forced-writes and $2m$ messages over the basic 2PC protocol. Thus the performance of the optimization is affected only by the number of participants that exercise the optimization, not by the total number of commit tree members.

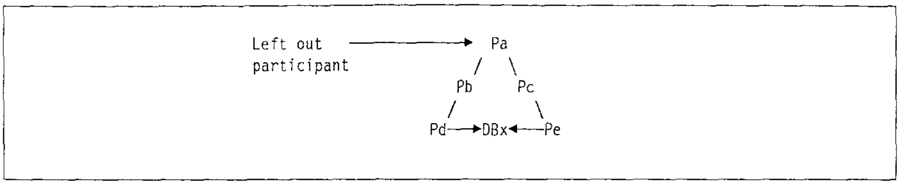

*Figure 12.* Transaction tree partitioned because of left our partners. Both programs Pd and Pe are accessing the same database, DBx.

## *Leaving inactive partners out (OK-TO-LEAVE-OUT)*

In a model where servers respond only to requests and do not initiate any work of their own, commit processing may be optimized if subordinates that have not participated in a transaction are left out of the 2PC protocol. A form of this optimization was originally implemented in $R^*$ [18]. In this section we briefly described the adaptation of this optimization for PN. Greater detail is presented in [21].

During a normal 2PC operation, the coordinator includes all partners as subordinates, whether or not it has exchanged data with them during this transaction. With the OK-TO-LEAVE-OUT optimization, a coordinator leaves out any partners with which it has exchanged no data during the transaction, based on the assumption that they have therefore not participated in the transaction. When a partner is left out, the coordinator does not send it the Prepare or Commit messages; nor does it have to wait for the Vote and Ack replies. Any intermediate coordinator in the tree can leave out its inactive subordinates as well.

This optimization is easy to include in PA, since PA is based on a requester-server model. It is more difficult to include in PN, since PN assumes a model of independent peers. In PN, the more general case where any partner can be left out if it has not exchanged data with the commit coordinator does not work, since the partner may have started work independently. The configuration in Fig. 12 illustrates this situation: assume programs Pd and Pe both initiate a commit operation, and Pa has been left out of the current transaction by both Pb and Pc. The two commit operations would occur independently, and might come to different results. If a program from one subtree touched the same resources as a program from the disjoint subtree, damage could occur, since the changes from both programs would appear to belong to the same transaction. This can be illustrated with programs Pd and Pe in Fig. 12 both writing records in the same database, DBx. Both Pd and Pe belong to the same transaction tree, and therefore use the same transaction identifier in their interactions with DBx. If Pd and Pe participate in independent commit operations that achieve different outcomes, damage has been caused, since the first commit operation to complete involving DBx caused all of the changes associated with the transaction identifier to either commit or abort. This makes the state of changes in DBx inconsistent with the state of changes made by either Pb or Pc.

Further analysis indicated that the full generality is not required. Most of the advantage of leaving partners out can be gained by leaving out server subtrees that only operate in response to requests from the coordinator.

The PN model includes a way for a subordinate to indicate that it operates only in response to requests from the coordinator. A subordinate may vote "OK to leave out" only if it will be suspended until its services are needed again. No member in the left-out subtree can initiate another commit operation or perform any independent work, since it is suspended until the coordinator process includes it in another transaction.

Whether a subordinate process is a server that only responds to requests is known by the application developer. LU 6.2, for example, provides a parameter on the SET_SYNCPT _OPTIONS verb for the local transaction program to indicate whether it may be suspended until it receives a request from its coordinator. If so, the subordinate communicates this information to its coordinator on the YES vote. The value returned on the YES vote is considered a protected variable, i.e., it takes effect only if the transaction commits. The LU 6.2 default is "not OK to leave out."

The following requirements must be met before a coordinator can leave another partner out of the 2PC for the next transaction:

- No data has been exchanged with that partner during the current transaction.
- The partner indicated in the previous successful commit operation that it would be okay to leave it out of subsequent transactions. For this to occur, three things must have happened:
  - All resources subordinate to the subordinate indicated that they may be left out.
  - The subordinate is suspended in the commit operation. Control will be returned to its program only when it has been sent data for a subsequent transaction.
  - All resources subordinate to the partner are similarly suspended waiting for the beginning of a new transaction.

Just because a subordinate indicates that it **can** be left out does not mean that it **will** be left out. The decision to leave a subordinate out is based on the work that is carried out during the next transaction. If there is reason for a requester to include its server in the next transaction, it will do so regardless of the OK_TO_LEAVE_OUT value specified.

For a transaction tree of **n** members out of which **m** voted **OK-TO-LEAVE-OUT**, this optimization saves $2m$ forced-writes and $4m$ messages over the basic 2PC protocol. Again, the number of messages does not depend on the position of the OK-TO-LEAVE-OUT participants in the transaction tree, nor on the total size of the tree. The explanation is similar to the one given in the read-only section (see Fig. 11).

### *Last agent*

Experience with CICS/MVS and IBM's DB2 [15] has shown that a transaction often contains a single remote partner. This particular situation allows a highly optimized commit path. The coordinator prepares itself to commit and gives the subordinate the commit decision, i.e., it is up to the subordinate to decide the outcome of the transaction. The coordinator that uses this optimization prepares all of its other subordinates and itself to go either way, force-writes a prepared record and sends a **YES** vote to the *last agent*, so called because it is the last subordinate contacted during the voting phase.

Unlike the normal 2PC case, the coordinator is not required to send an explicit acknowledgment when it receives a commit message. The last agent is not blocked waiting for acknowledgment; as soon as it sends the Commit message, it can proceed with the next transaction. The next data sent to the subordinate serves as an *implied* acknowledgment, since it implies that the coordinator received the earlier Commit message. Receipt of the implied acknowledgment allows the last agent TM to write the End log message and forget the outcome of the transaction. This optimization is illustrated in Fig. 13.

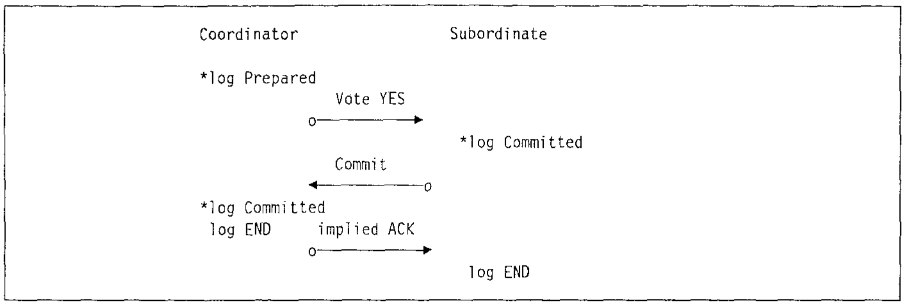

*Figure 13.* Last-agent commit processing.

This optimization yields the greatest benefit when the coordinator has no other remote subordinates. If it has other subordinates, they must all vote YES before the coordinator can send its YES vote to the last agent. The prepare message can be sent in parallel to multiple subordinates so that their phase-one processing can occur concurrently. Communication with a last agent cannot overlap any other commit processing. Thus, the last-agent optimization that reduces messages to one agent conflicts with the optimization inherent in preparing multiple agents concurrently. However, if messages to one of the remote partners involve long network delays (e.g., connection through satellite), the last-agent optimization provides significant savings. It is, for example, preferable to prepare the close-by partners (fast first phase) and reduce the communication required with the faraway partner to one slow round-trip message exchange.

The last-agent optimization is most useful with PN, since the coordinator always logs before it sends a message to any subordinate. With PA, the savings in messages conflicts with the need for a possibly extra log force. Thus, the last-agent optimization requires that the initiator force-write a prepared record before it sends its YES vote to the last agent. If the subordinate is not a last agent, the coordinator does not force any log record before the Committed record.

For a transaction tree of $n$ members and $m$ last agents, this optimization offers savings of $2m$ messages over the basic 2PC protocol, but no savings in forced-writes. It is possible to have multiple last agents, since each last agent may choose one of its subordinates to be a last agent.

### *Unsolicited vote*

If a participant is a server that is designed to know when it has finished its part of a transaction, it can prepare itself to commit and vote YES without waiting for the prepare request from the coordinator. Thus, the server can remove the need for the first message flow of 2PC by preparing itself on its own initiative, force-writing a prepared record, and sending an unsolicited YES vote to its coordinator. If used in conjunction with the last-agent optimization, a bit in the **YES** vote can distinguish this optimization from the last-agent one. An unsolicited YES vote does not initiate any commit processing in the receiver, but does indicate that the subordinate is already prepared.

For servers associated with relatively high network delays, the unsolicited-vote optimization provides significant performance improvement. A form of this optimization was originally proposed in the context of distributed INGRES [30] and IBM's IMS/VS [24].

For a transaction tree of **n** members and **m** unsolicited-vote participants, this optimization saves **m** messages over the basic 2PC protocol.

*Flattening the transaction tree*

The typical 2PC protocol treats the distributed transaction as a tree. Each participant cascades the 2PC protocol to its own descendents. This is illustrated in Fig. 14 and in Fig. 6. The cascading of the protocol means that TPb must receive and process the Prepare message before TPc can be sent the cascaded Prepare from TPb. This serialization of the 2PC messages increases the duration of the 2PC processing as the tree depth grows.

An alternative to this is feasible in communication protocols where a round trip is required before commit processing. Remote Procedure Calls (RPC) or message-based protocols where each request must receive a reply, are examples of protocols where round trips must occur before commit processing is initiated.

With these protocols, the identity of all cascaded subordinate TPs can be returned to the transaction coordinator when the child replies to its parent. This is illustrated in Fig. 15 part *a*.

In Fig. 15 part *b* the TM for TPa sends the 2PC messages directly to TPb, TPc, and TPd. These messages are sent in parallel. This avoids the propagation delays and can be a big performance winner in distributed transactions that contain deep trees.

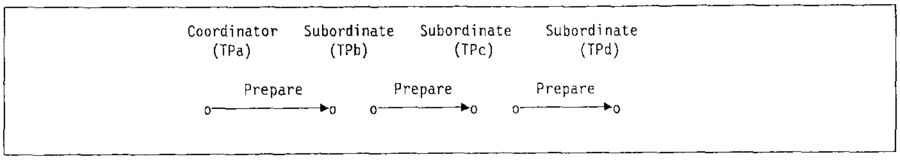

*Figure 14.* A commit tree of depth 3.

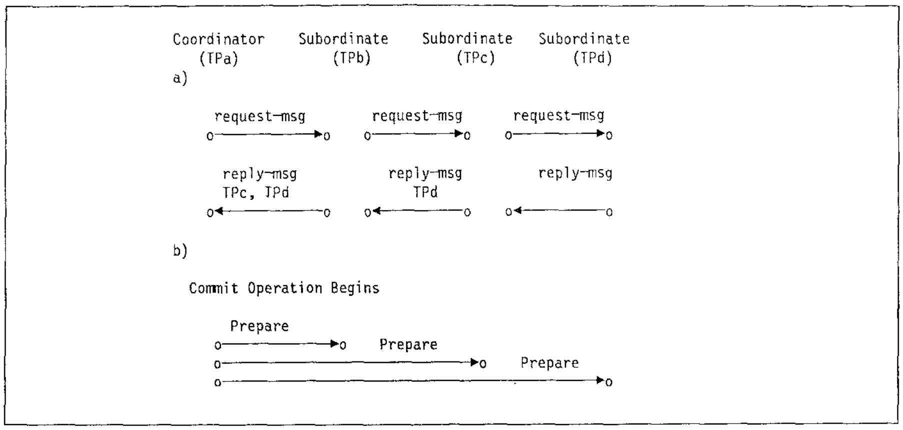

*Figure 15.* Commit tree of depth 3 flattened down to depth 1.

A limitation of this optimization is that in some distributed transactions, security policies of the nodes running the TPs may not permit the computer that is running TPa to communicate directly to TPc or TPd. Protocols that support security features that prohibit any-to-any connectivity cannot use this optimization without additional protocols to handle the case where a partner cannot connect directly to the commit coordinator.

Another limitation is that a reply message is required so the identity of all the partners is known to the coordinator prior to phase 1 of the 2PC protocols. Protocols that do not require replies, such as conversational protocols, may not know the identities of all the agents prior to phase one. These protocols save time by not requiring a reply to every request. For those protocols it is possible to flatten the tree during phase 2, if the identity of each subordinate is returned to the coordinator during the reply to the Prepare message.

### *Sharing the log*

A local resource manager uses a log to keep track of updates so that it can either abort or commit a transaction. Before an LRM votes YES, it ensures that this information has been forced to non-volatile storage. When it learns of a commit outcome, it also force-writes a commit record.

The LRM can share the same log as the coordinator transaction manager [23]. With this optimization, the LRM takes advantage of the knowledge that the TM will force-write a commit record. The LRM does not force-write the prepared record because the TM's force-write of the commit record causes the local LRM's earlier non-forced write to be written to the log. If the transaction successfully commits, the TM's commit record and the LRM'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 LRM 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 LRM's prepared record are both on the log, the recovery process will successfully commit the transaction.

This optimization saves two forced-writes per LRM that shared the log. The more LRMs that share the log with the TM, the more savings per transaction.

### *Group commits*

There are certain points during 2PC where logging must complete before the commit processing can continue. This blocks the commit processing until the log I/O completes.

In systems where there are many disk I/Os, I/O requests can queue up waiting for a previous I/O request to complete. This queueing can decrease the overall throughput of the transaction processing system.

Where transaction rates are high, the *group commit* optimization is practical. With this optimization the log manager delays performing a force-write request until one of two things occurs: either a defined number of force-write requests arrive, or a timer expires indicating that the force-write request(s) should be processed even though the expected number of requests has not arrived. This optimization was originally proposed and implemented in IMS/VS[5] Fast-Path [8].

By processing a group of force-write requests at once, the logger can do all the requests with one large I/O operation instead of many small ones. Since there is overhead involved with starting I/O requests, the overall system throughput is maximized at the expense of delaying individual commit procedures.

Group commits are a form of log sharing. However, the log sharing is done among different transactions as opposed to the previous case where the sharing was done among the different components of the same transaction.

For **n** transactions and a group commit of size **m**, this optimization provides an average of **3n/2m** savings in force-writes. In this simple analysis we assumed that only one member of each transaction resides at each node.

A detailed analysis of the group commit optimization is quite complicated since several parameters are involved: I/O rate, group size, number of participants, response time, and time to allow the commit group to build up. Such analysis can be found in [13, 29].

*Long locks*

LU 6.2 2PC protocols allow an application program to trade off packets sent against duration of the commit operation, and therefore the length of time that resource locks are held (long locks). In the usual case, the subordinate sends the commit acknowledgment to the coordinator as soon as it has ensured that it has finished committing the transaction. If the coordinator enables the long-locks variation, the subordinate delays sending the commit acknowledgment until it sends the message beginning the next transaction. Since the commit acknowledgment can be packaged in the same packet as the next transaction data, this reduces the network traffic by one at the cost of keeping the resources at the coordinator locked for a longer period.

LU 6.2 half duplex protocols ensure that only side of a conversation can send at a time. The other side is in RECEIVE state, meaning that it can only receive data. The sending partner can relinquish the permission to send, causing the direction of data flow on the conversation to turn around. LU 6.2 allows the long-locks variation only if the coordinator will be in RECEIVE state at the end of the commit operation, waiting for the subordinate to begin the next transaction. The coordinator controls the state of a conversation at the end of a 2PC operation, informing the subordinate in the Prepare message, as shown in Fig. 16.

Figure 16 shows the long-locks variation of the basic LU 6.2 2PC protocol. The LU 6.2 Prepare message to a subordinate agent includes instructions about the conversation state expected after a successful commit. It also informs the subordinate whether or not the coordinator wants the long-locks variation. The Commit acknowledgment message is placed in the outgoing send buffer, but is not actually sent until data for the next transaction is sent.

Long locks are advantageous where network resources are expensive and delays between transactions are small. A good application of this particular optimization was presented in [12]. The application involved banks that needed to reconcile their log accounts at the end of the day. In the simplest form of this application, one bank (bank A), takes the money out of an account and deposits it in the account of a second bank (bank B). The withdrawal/deposit must be done as an atomic action in order to make sure that no money is lost. Banks typically batch these sorts of transactions until the end of the day. At that time, all the withdrawals and deposits that occurred during banking hours are reconciled.

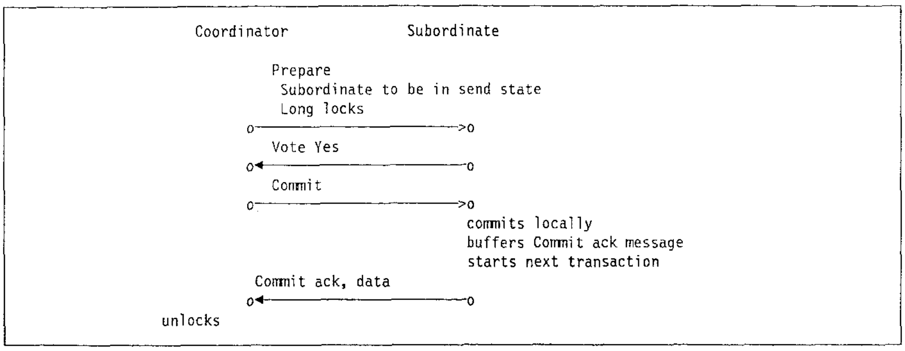

*Figure 16.* Example of long locks committing one transaction.

One alternative is to do all deposit/withdrawal requests as one large transaction. This amortizes the cost of the 2PC protocol, with its four messages, over all the deposit/withdrawal requests. The drawback is the amount of work that must be redone in the event of a failure during the one large transaction.

An alternative is to perform and commit several requests at a time, repeating the process until there is no further reconciliation work to be done. To take advantage of the long-locks optimization, two banks take turns initiating a transaction. This reduces the average number of individually transmitted 2PC messages per transaction to three, since the commit acknowledgment is piggybacked on the request that starts the next transaction, as shown in Fig. 16.

## *Commit acknowledgment*

One of the ways that different 2PC protocols vary is in the timing of the commit acknowledgment. Some have early acknowledgment [34, 23]: an intermediate system acknowledges a commit received as soon as it has logged; others have late acknowledgment [31]: an intermediate system waits to acknowledge the commit received from its coordinator until it has collected acknowledgments from all its subordinates. Early acknowledgment means "I have committed and am in the middle of propagation;" late acknowledgment means "I and all members of my subordinate subtree have committed successfully." Early acknowledgment has the advantage that the commit operation completes earlier for the root and intermediate systems, allowing them to begin useful work earlier. Late acknowledgment has the advantage that there is no uncertainty at the root of the commit tree when it starts the next transaction that it is building on the solid basis of a previously committed transaction; if any heuristic damage has occurred, it has heard about it. Thus, there is a tradeoff between wait time and confidence in the outcome of the transaction. Of course, any intermediate only knows about the commit outcome in its own subtree, so this confidence is limited in a true peer-to-peer environment where any program in the tree can start further work.

One acknowledgment pattern may not make sense for all applications and resource types. Thus, if the chance of a heuristic decision is vanishingly small for all resources involved in a transaction, late acknowledgment does not add any value. Similarly, interactive programs may choose to reduce wait time, even if doing so involves a reduction in confidence, in order not to keep a human at a terminal waiting longer than absolutely necessary. Some variations to the late acknowledgment pattern based on these considerations are described below.

### *Voting reliable*

Late acknowledgment is based on the assumption that any node in a transaction tree may make a heuristic decision that disagrees with the decision taken by the rest of the tree, and that the root of the commit tree should be informed if damage of this nature occurs. It is possible however to have nodes in the tree that make heuristic decisions only in drastic circumstances. For example, a database system may be built on the assumption that correcting heuristic damage is so difficult that heuristic decisions should be utterly discouraged. The probability of heuristic decisions can be made so small that early acknowledgment is acceptable, even for applications that rely on the semantics of late acknowledgment.

The **vote reliable** optimization uses information gathered from LRMs to gain the early completion advantages of early-acknowledgment protocols while maintaining the semantics of late-acknowledgment protocols. When a LRM votes YES, it indicates whether it is a reliable resource, i.e., one for which heuristic decisions are very unlikely. An intermediate TM collects the reliability indicators from all its subordinates. If all vote reliable, then it can use early (or implied)-acknowledgment protocols with its coordinator during the commit phase (see Fig. 17). If any LRM votes "not reliable," the intermediate uses late-acknowledgment protocols. Generally speaking, the "reliability" characteristic is a static one that will not vary from transaction to transaction. Thus, a database system either is or

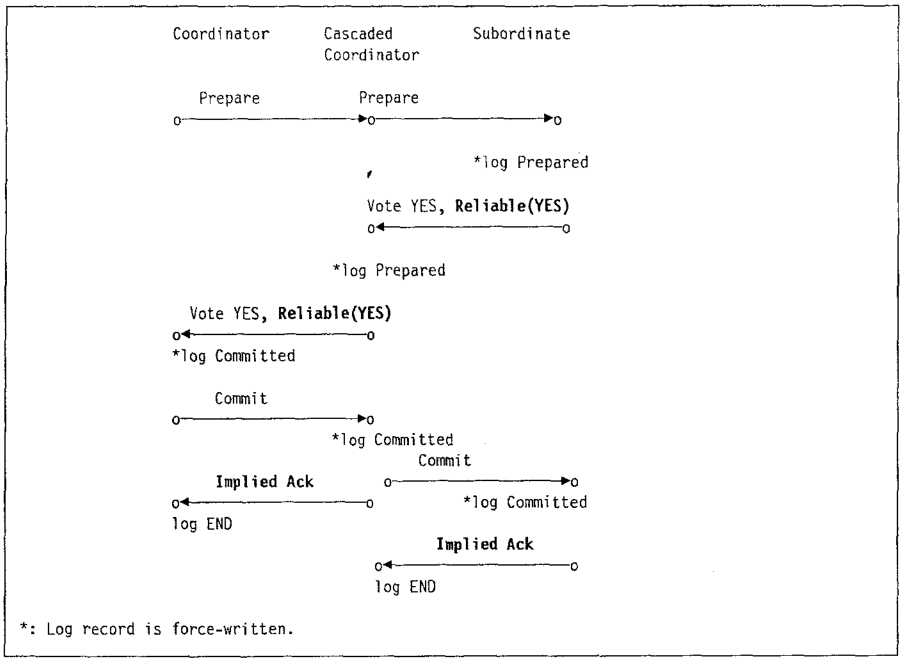

*Figure 17.* Two-phase commit processing, all resources voted reliable.

is not reliable.[6] However, since the resources involved can vary from transaction to transaction, the intermediates collect the reliability information during every first phase.

The default value of this characteristic is "not reliable." Thus, LRMs that can provide this information may achieve a performance advantage in overall commit processing, but LRMs that are either not reliable or do not understand the parameter still receive full 2PC coverage.

With implied acknowledgment, a transaction tree of **n** members and **m** vote-reliable participants, this optimization saves **m** messages over the basic 2PC protocol.

### *Wait for Outcome*

Late acknowledgment implies that the intermediate does not respond to its coordinator until it has collected acknowledgments from its subordinates, even if failures occur that require recovery processing. For major system failures, waiting for recovery processing may involve considerable delay. An intermediate may make multiple attempts to contact a subordinate before it succeeds.

When implementing the PN protocols for APPC in VM/ESA, usability evaluations uncovered a problem with this aspect of late acknowledgment: a human waiting for the outcome of a transaction gets very impatient waiting for recovery processing to complete. Some people would rather get control back earlier, even if they could not be guaranteed certainty that the transaction completed without heuristic damage.

A feature was added to the IBM PN protocols and the APPC interface [31, 32] to allow the application program to specify whether it requires all recovery processing to complete before it is told the outcome of the commit operation. If yes, then late acknowledgment occurs as usual; the coordinator application is blocked, awaiting all acknowledgments and recovery processing to occur. If no, one attempt to contact a failed partner is attempted. If the first attempt fails, the system attempts to complete the recovery processing in the background, but allows the commit or abort operation to complete with an indication to the application program that the outcome of the entire transaction is not yet known. Similarly, an intermediate system will attempt to contact a failed subordinate only once before sending an acknowledgment to its coordinator indicating that "recovery is in progress." The commit or abort operation completes at the coordinator with the "outcome pending" indication (see Fig. 18).

One recovery attempt is always attempted so that the program only hears "outcome pending" for long-term failures. It is considered preferable for a program to wait a short time for one attempted recovery than to get an "outcome pending" indication for every failure.

In the original versions of this feature, the decision was made independently at each node in the transaction tree. In a later version we decided to allow the coordinator to inform its subordinates whether it wants to wait for the outcome during phase one, allowing the root of the commit tree to control the way the rest of the tree responds to failures [26]. An additional benefit of this decision is described in the section titled "Wait for Outcome and Presumed Abort."

This feature allows the application developer to decide the relative merits of shorter wait time and confidence in outcome. Unlike early acknowledgment protocols, the normal case is complete confidence in outcome, and the application program is informed when that cannot be achieved.

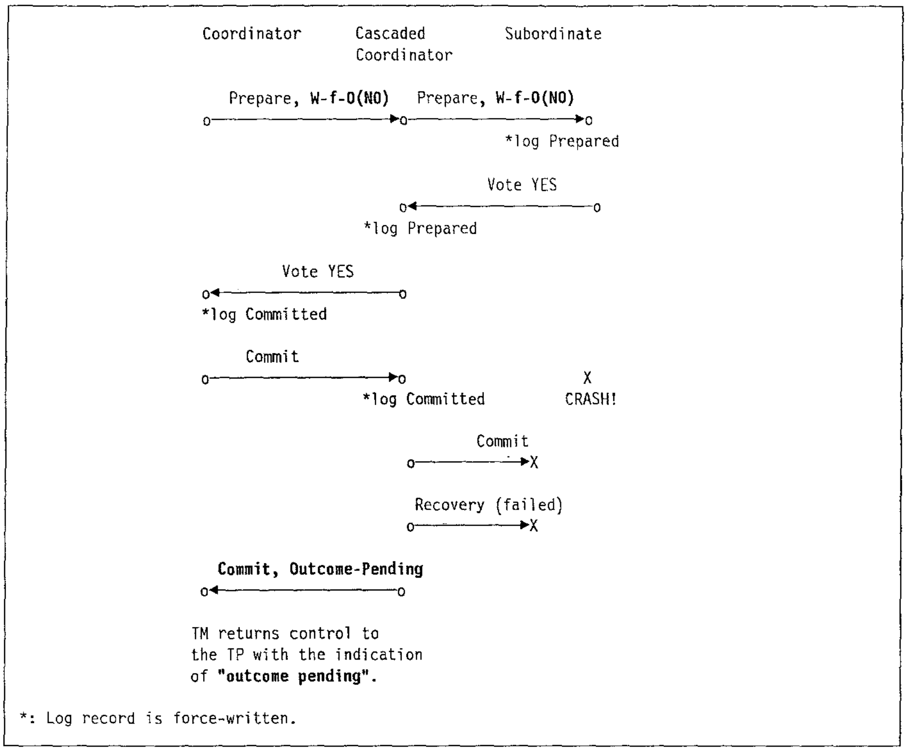

*Figure 18.* Two-phase commit processing, all participants choose Wait-for-Outcome (NO).

## 7. Combining 2PC optimizations

We have so far described each 2PC optimization independently of other optimizations. It is possible to use more than one optimization in the same 2PC operation. This section describes certain combinations of 2PC optimizations and how they affect the performance, correctness, and reliability of 2PC processing. Unless otherwise stated, these combinations are described in the context of Presumed Abort. The list is not exhaustive; the combinations presented here are the ones with the most interesting effects.

### *Last agent and read only*

The last-agent optimization allows the commit coordinator to transfer the commit decision to a remote partner at the expense of force-writing a Prepared log record that is superfluous if the coordinator votes read-only. This is true because the behavior of a read-only coordinator is not changed by the outcome of transaction. Therefore the coordinator does not need any recovery processing if it fails after giving the decision to the last agent.

In order to maintain the advantages of both the last-agent and read-only optimizations, the combined optimization gives the coordinator the ability to vote read-only to a last agent without forcing any log records at all. If the last agent also votes read only, the commit operation can complete without any log records being forced anywhere.

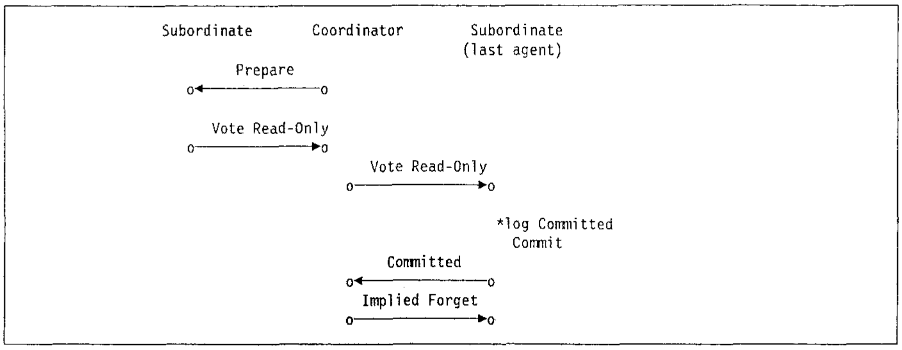

*Figure 19.* Read-only coordinator with committing last agent.

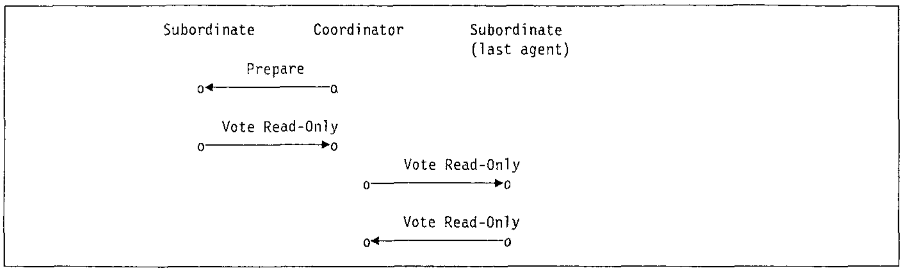

*Figure 20.* Read-only coordinator with read-only last agent.

This combination of optimizations is illustrated in Fig. 19, where the coordinator votes read-only and the last agent commits, and Fig. 20, where both vote read-only. The last agent does not owe any recovery processing to the coordinator if the coordinator fails during the commit processing.

Both figures show that a coordinator must collect read-only votes from all other subordinates before it can vote read-only to a last agent. Thus it cannot vote read-only if any of its other subordinates needs to know the outcome of the transaction.

Since 80% of distributed transactions are read only with one or two remote partners, this combined optimization provides tremendous savings. For a transaction tree with **n** members and **m** cascaded last agents that vote read-only, this combination of optimizations saves **2m** messages and **2m** forced log writes.

There is, however, an interesting interaction of this combination with the Wait-for-Outcome optimization: If the TP that is executing at the coordinator side indicates that it wants to learn the outcome of the transaction then the last agents will have to force write the Prepared log record. This is so, because the remote partner's lack of logging could cause undetected damage of the transaction. For example, a read-only transaction will appear to have backed out upon recovery and heuristic damage could be lost because of the lack of logging at a subordinate. For more details see the "Wait for Outcome and Presumed Abort" section.

Such a bizarre combination is very unlikely to happen since the semantics of the read-only optimization (don't care about the final outcome) conflict with those of the Wait-for-Outcome (YES) optimization. Thus, if the initiator is interested in the final outcome of the transaction, it should vote YES instead of Read-Only.

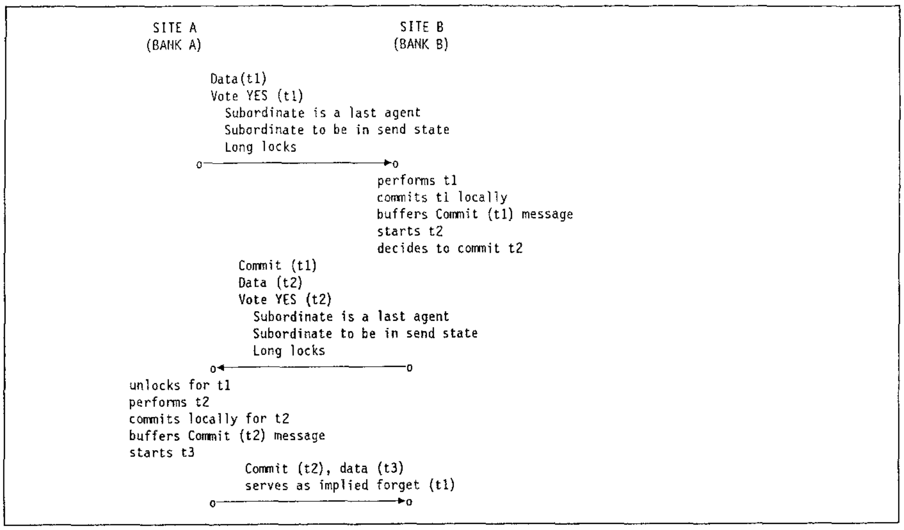

*Figure 21.* Long locks (last agent commits).

## *Long locks and last agent*

Both the long-locks and the last-agent optimizations reduce network traffic. The long-locks optimization does so at the expense of longer resource lock time. If used together, these two optimizations can make the amount of extra network traffic for 2PC vanishingly small for an alternating application such as the bank reconciliation application described in the early section on Long Locks.

Figure 21 shows the long-locks variation combined with the last-agent optimization to perform and commit two transactions (and start on a third) in three separate packets. The LU 6.2 Vote **YES** to a last agent, like the prepare message to a not-last agent, includes instructions about the conversation state the subordinate is expected to be in after a successful commit and whether the coordinator wants the long-locks variation.

In Fig. 21, three sequential transactions are referred to as $t1$, $t2$, and $t3$. Thus Data(t1), Vote YES ($t1$), and Commit($t1$) in the figure refer respectively to the transaction data, Vote message, and Commit message for the first transaction.

Taking this approach, applying 2 optimizations, long locks and last-agent and using the conversational model, the banking application described previously can commit frequently, minimizing the number of requests that have to be re-run if there is a failure, without adding any extra packets exchanged just for 2PC.

See Fig. 21 for an example of these two optimizations working together. Bank A starts a conversation with bank B. Bank A then requests that '*n*' updates be made, invokes the Long-Locks optimization, and initiates a commit operation. This sends all '*n*' transactions, the vote YES message, and the command that indicates "Bank B is in SEND state after the transaction, and please buffer the commit message until Bank B sends application data back." Only one message flow has occurred so far. Bank B, after receiving the incoming request, making the updates, committing the updates, is now in SEND state. It does the same thing that Bank A did: it requests '*n*' updates, invokes the Long-Locks optimization, and initiates the commit processing. Since the last-agent optimization uses implied forget, the message that begins one transaction acts as an implied forget for the previous transaction. Bank A and B can repeat this alternating process until there is no further reconciliation work to do. This results in 2 committed transactions for 3 messages.

For $r$ transactions that overlap data transfer and commit processing in this way, this optimization combination saves $5r/2$ messages.

### *Last agent and unsolicited vote*

With the peer-to-peer model, a subordinate can be selected as the last agent by multiple coordinators. Unlike the normal case, where the existence of multiple coordinators causes an abort, this can occur legally, since there is still only one participant responsible for making the commit decision. As shown in Fig. 22, this case looks very much like the unsolicited-vote optimization.

This optimization provides extra savings over the unsolicited-vote optimization since the coordinators can use implied acknowledgment instead of sending explicit acknowledgments. For a transaction tree with **n** members and **m** participants that send unsolicited ready messages to the same last agent, this optimization combination saves **2m** messages.

This optimization is relatively easy to implement for a system that implemented the last-agent optimization; CICS [6] implemented the unsolicited-vote optimization by disabling the error check for multiple coordinators in last agents.

However, the commercial requirement to report heuristic damage becomes more difficult when a subordinate does not have a single coordinator. With protocols such as LU 6.2 sync point that allow only one answer to the "Vote YES" message, a last-agent subordinate that detects heuristic damage among its subordinates sends the "Heuristic Damage" message to the coordinator in place of the "Commit" or "Abort" message, leaving the coordinator to make a heuristic decision for its local resources. As illustrated in Fig. 23, if this is done with multiple coordinators, the damage may be extended unless appropriate administrative controls are in place to make sure both sites make the same heuristic decision.

A simpler alternative is illustrated in Fig. 24. Only one partner is identified as the coordinator that must be informed of heuristic damage. For example, LU 6.2 sync point has the following rules for determining which partner is the official coordinator:

- If only one partner sends a YES vote, it is considered the coordinator and the local server is a last agent.

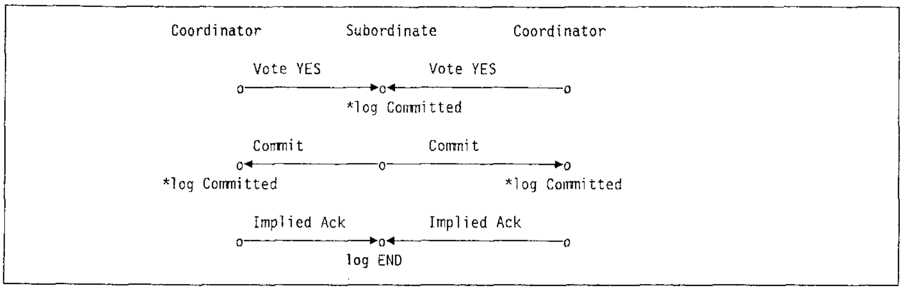

*Figure 22.* Last agent and unsolicited vote: successful.

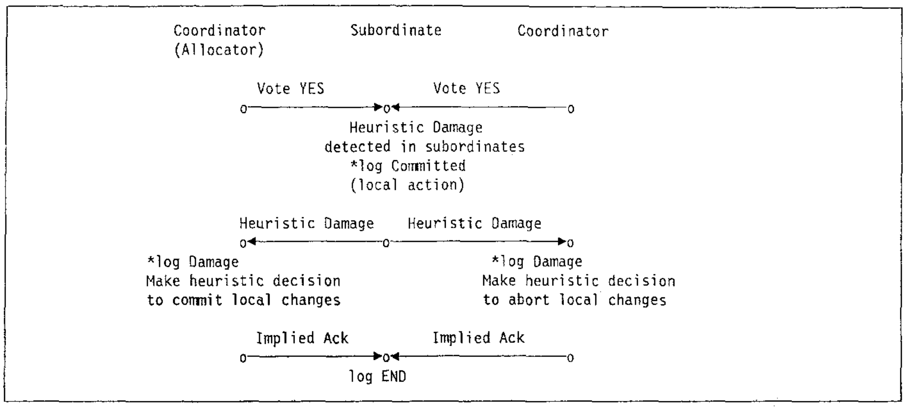
Coordinator (Allocator)
Subordinate
Coordinator
Vote YES
Vote YES
Heuristic Damage detected in subordinates
*log Committed (local action)
Heuristic Damage
Heuristic Damage
*log Damage
Make heuristic decision to commit local changes
*log Damage
Make heuristic decision to abort local changes
Implied Ack
Implied Ack
log END

*Figure 23.* Last agent and unsolicited vote: damage problem.

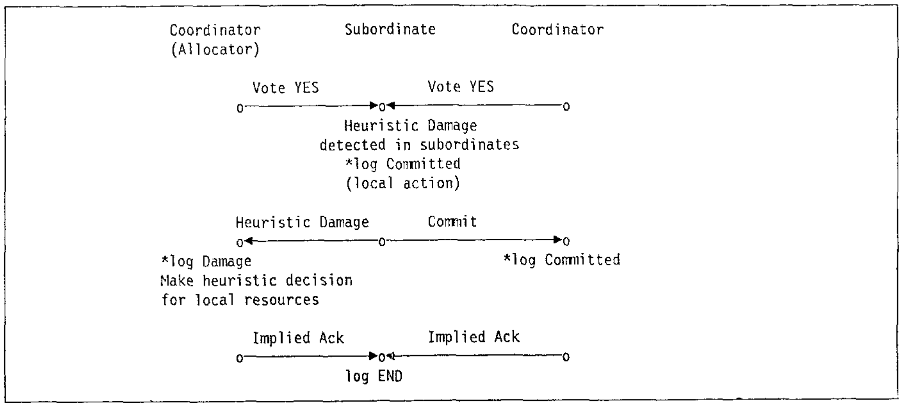

*Figure 24.* Last agent and unsolicited vote: damage reduced.

- If multiple partners send YES votes, only the program that started the local program (allocated the conversation to it) can be treated as a coordinator; all others are treated as unsolicited-ready subordinates. This may mean that no coordinator is selected for the purposes of reporting heuristic damage.

Although these rules may not yield the answer that matches a specific application, they yield the right answer in most cases. Where they do not, damage reports will be misrouted, but the outcome of the transaction should not be changed.

## *Wait for Outcome and Presumed Abort*

The Wait-for-Outcome optimization was originally designed for the Presumed Nothing protocol. When used with Presumed Abort, Wait-for-Outcome(YES) causes extra logging at intermediates.

The reason for this can be shown with the following configuration:

$$\text{TPa} \longrightarrow \text{TPb} \longrightarrow \text{TPc}$$

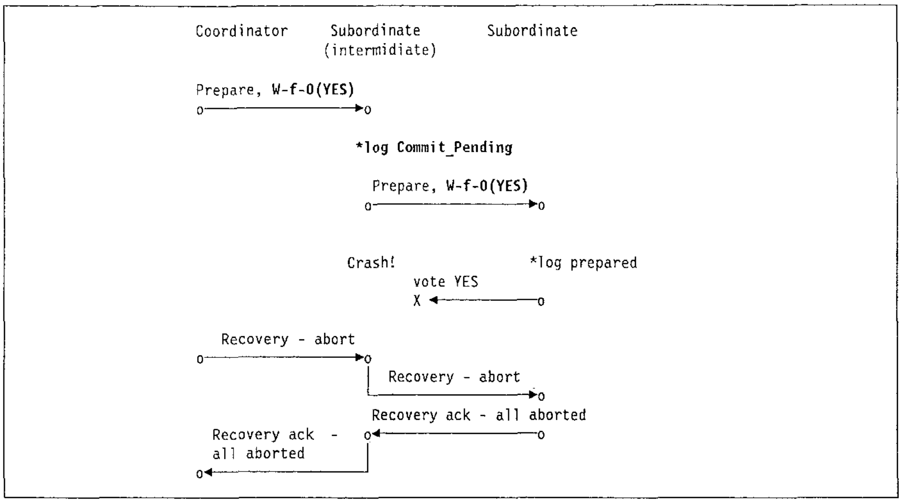

*Figure 25.* Wait-for-Outcome (YES) and presumed abort.

Participants TPa and TPb both indicate they want to wait for the outcome. Participant TPa sends prepare to TPb. TPb sends prepare to TPc. With presumed abort, the TMs at TPa and TPb have not logged anything. Thus if the TPb site crashes, its TM has no memory of the 2PC operation in progress.

Since TPa specified Wait-for-Outcome(YES), its TM will expect to learn the outcome of the 2PC in the entire subtree. It will therefore initiate recovery processing. The TPb TM, having no memory of the transaction, assumes it aborted successfully. Since it passes this information to its coordinator, TPa will think that the entire transaction aborted. Unfortunately this may not be true. Since TPc is in doubt, there is potential for a heuristic commit decision to occur there. Therefore TPa has not gotten what it requested with Wait-for-Outcome (YES).

The solution to this problem is for the coordinator to indicate in the Prepare message (not-last agent) or YES vote (last agent) whether any program between it and the root of the tree has specified Wait-for-Outcome(YES). If so, the intermediate coordinators must force write Commit-Pending (or Prepared if last agents) log records before propagating Prepare (or YES vote) to their subordinates. Therefore they cannot lose memory of subordinate participants when they owe complete information to their coordinators. This is not necessary at root coordinators since a crash that destroys the TM's memory also destroys the TP that requested Wait-for-Outcome(YES), removing the obligation to give it complete outcome information. This solution trades off extra logging against reliable reporting.

Figure 25 shows the Wait-for-Outcome optimization used with presumed abort.

The full benefit of PA logging is received only if all coordinators specify that they don't want to wait for the outcome of the transaction.

Even if all subordinates in the tree vote reliable indicating they do not make heuristic decisions, the extra Commit-Pending log writes cannot be avoided for Wait-for-Outcome(YES). The reason for this is that the intermediate coordinators do not know that the subordinates vote reliable until the votes return, after the log write has already been done.

For a transaction tree with **n** members and **m** intermediate coordinators, reliable reporting is achieved at the cost of **m** extra forced log writes.

## 8. Performance evaluation and discussion

Two-phase commit optimizations can be evaluated in terms of reduction in network traffic, reduction in the number of forced writes, and decreased resource lock time. Since these optimizations can also affect reliability, an evaluation is affected by whether these optimizations reliably report the outcome of the transaction and whether they increase the chances of heuristic damage.

A single optimization does not provide improvements across all performance metrics, and often an optimization might trade off one metric for another. In some cases, performance can be improved by combining different optimizations. In other cases, combined optimizations reduce the reliability of the commit processing.

In the tables that follow, optimizations are analyzed in terms of the absolute number of messages exchanged with subordinates. Further analysis would break down the messages

*Table 1.* Advantages and disadvantages of 2PC optimizations.

| Optimization | Advantages | Disadvantages |
|---|---|---|
| *Read Only* | Fewer messages<br>Fewer log writes<br>Early release of locks | No knowledge of the outcome of a transaction<br>Potential serializability problems |
| *Last Agent* | Fewer messages<br>Early release of locks | One extra forced write possible |
| *Unsolicited Vote* | Fewer messages<br>Earlier release of locks | Application specific |
| *OK to leave Out* | No log writes<br>No messages | N/A |
| *Vote Reliable* | Fewer messages<br>Next transaction can begin earlier | Damage reporting to root coordinator lost if reliable resource *does* take a heuristic decision |
| *Wait for Outcome* | 2PC doesn't block for most network partitions<br>Next transaction can begin earlier | Complete outcome of transaction may not be known by coordinator |
| *Long Locks* | Fewer packets per transaction | Locks held longer<br>Commit decision may be delayed if combined with last-agent optimization |
| *Shared Logs* | Fewer forced writes | Interdependence of resource manager and transaction manager |
| *Group Commit* | Fewer forced writes<br>System throughput maximized | Longer lock times for individual transactions |
| *Flattening the Transaction Tree* | Reduced commit processing time<br>Earlier release of locks<br>System throughput maximized | Extra overhead in reply messages<br>Intermediate systems do not learn outcome at subordinates |
| *Vote Reliable & Wait for Outcome (Yes)* | No extra advantage | May have unnecessary forced log writes |

*Table 2.* Logging and network traffic of 2PC optimizations.

| 2PC type | Coordinator messages | Coordinator logs | Subordinate messages | Subordinate logs |
|---|---|---|---|---|
| *Basic 2PC* | 2 | 2, 1 forced | 2 | 3, 2 forced[1] |
| *PN* | 2 | 3, 2 forced | 2 | 4, 3 forced[1] |
| *PA, Commit case* | 2 | 2, 1 forced | 2 | 3, 2 forced[1] |
| *PA, Abort case* | 2 | 0, 0 forced | 1 | 0, 0 forced |
| *PA, Read-Only case* | 1 | 0 | 1 | 0 |
| *PA & Unsolicited Vote* | 1 | 2, 1 forced | 2 | 3, 2 forced[1] |
| *PA & Last-Agent* | 1 | 3, 2 forced[1] | 1[2] | 2, 1 forced |
| *PA & Last-Agent & Coordinator Read-Only*[3] | 1 | 0 | 1 | 2, 1 forced[1] |
| *PA & Last-Agent & All Read-Only*[3] | 1 | 0 | 1 | 0 |
| *PA & Last-Agent as Unsolicited Vote* | 1 | 3, 2 forced | 1 | 2, 1 forced |
| *PA & ok-to-leave-out* | 0 | 0 | 0 | 0 |
| *PA & Vote Reliable* | 2 | 2, 1 forced | 1[2] | 2, 2 forced |
| *PA & Wait-for-Outcome (No)* | 2 | 2, 1 forced | 2 | 3, 2 forced[1] |
| *PA & Wait-for-Outcome (Yes)*[4] | 2 | 2, 1 forced<br>3, 2 forced[5] | 2 | 3, 2 forced[1] |
| *PA & Long Locks (not last-agent)* | 2 | 2, 1 forced | 1 | 3, 2 forced[1] |
| *PA & Long Locks & Last-Agent* | 1 | 3, 2 forced | 1 | 2, 1 forced[1] |
| *PA & shared logs* | 2 | 2, 1 forced | 2 | 3, 0 forced |

Note: [1]The *Prepared* and *Committed* records are force-logged; the *END* record is not forced. It is possible to combine the *Committed* and *END* into one forced log for leaf subordinates. [2]In this optimization an implied-Ack is used, saving a link flow. [3]In this combination only the coordinator is read only. The pair ($x$, $y$ forced) means that $x$ log writes are performed, of which $y$ are forced. [4]To learn the outcome in the entire tree, a PA coordinator must behave as a PN coordinator. [5]This is the value for intermediate coordinators.

into those to LRMs and those to remote TMs, which in general involve greater delays. Since there are no exact weights that can be associated with those two type of messages we did not carry the analysis this far.

Table 1 summarizes the advantages and disadvantages of the various optimizations.

Table 2 describes number of messages and log writes of the optimization and compares them with the basic two-phase commit, presumed abort, and presume nothing protocols. For comparison purposes, each optimization is evaluated within presumed abort. The calculations are done within the context of a transaction with 2 participants.

Table 3 provides a higher level of comparison by describing the number of messages and log writes needed to commit a transaction with $n$ members. Each row in the table describes the benefits gained if $m$ participants use a particular optimization. Each entry in the table is illustrated with a real case shown in Fig. 26, consisting of 11 participants, 4 of which follow the same optimization. While the numbers 11 and 4 have no special significance, a specific practical example makes the relative benefits of the different optimizations easier to perceive.

The intention of Table 3 is to contrast these optimizations with the basic 2PC protocol, rather than to compare them to each other. Comparing the different optimizations does

*Table 3.* Logging and message costs for optimizations. Each transaction consists of $n$ partners where $m$ members are following a particular optimization.

| 2PC type | Messages | Log writes | $n = 11, m = 4$ $/f, w, wf$ |
|---|---|---|---|
| *Basic 2PC (no optimizations present)* | $4(n-1)$ | $3n-1, 2n-1$ forced | 40, 32, 21 |
| *PA & Read Only* | $4(n-1)-2m$ | $3(n-m)-1, 2(n-m)-1$ forced | 32, 20, 13 |
| *PA & Last Agent* | $4(n-1)-2m$ | $3n-1, 2n-1$ forced | 32, 32, 21 |
| *PA & Last Agent & Read-Only* | $4(n-1)-2m$ | $3(n-m)-1, 2(n-m)-1$ forced | 32, 20, 13 |
| *PA & Unsolicited Vote* | $4(n-1)-m$ | $3n-1, 2n-1$ forced | 36, 32, 21 |
| *PA & Last-Agent as Unsolicited Vote* | $4(n-1)-2m$ | $3n-1, 2n-1$ forced | 32, 32, 21 |
| *PA & Ok-To-Leave-Out* | $4(n-1)-4m$ | $3(n-m)-1, 2(n-m)-1$ forced | 24, 20, 13 |
| *PA & Vote Reliable* | $4(n-1)-m$ | $3n-1, 2n-1$ forced | 36, 32, 21 |
| *PA & Wait-For-Outcome (No)* | $4(n-1)$ | $3n-1, 2n-1$ forced | 40, 32, 21 |
| *PA & Wait-For-Outcome (Yes)* | $4(n-1)$ | $3n-1+m-1, 2n-1+m-1$ forced | 40, 35, 24 |
| *PA & Share Logs* | $4(n-1)$ | $3n-1, 2(n-m)-1$ forced | 40, 32, 13 |
| *PA & Long Locks* | $4(n-1)-m$ | $3n-1, 2n-1$ forced | 36, 32, 21 |

Note: The triplet $(f, w, wf)$ refers to (# of messages, # of log writes, # of forced writes.)

*Table 4.* Logging and message costs for long-locks optimization. $r$ transactions occur, each consisting of 2 members.

| 2PC type | Messages | Log writes | $r = 12/f, w, wf$ |
|---|---|---|---|
| *Basic 2PC* | $4r$ | $5r$, $3r$ forced | 48, 60, 36 |
| *PA & Long Locks (not last agent)* | $3r$ | $5r$, $3r$ forced | 36, 60, 36 |
| *PA & Long Locks (last agents)* | $3r/2$ | $5r$, $3r$ forced | 18, 60, 36 |

Note: The triplet $(f, w, wf)$ refers to (# of messages, # of long writes, # of forced writes).

not make sense since they are used within different contexts, and therefore cannot be used interchangeably.

Table 4 shows the benefits of the long-locks optimization when it is used by **r** transactions with small delays between them.

## 9. Related work

This section discusses additional 2PC optimizations that were not analyzed earlier either because they are variations of the ones already presented or because they take advantage of specific machine architectures.

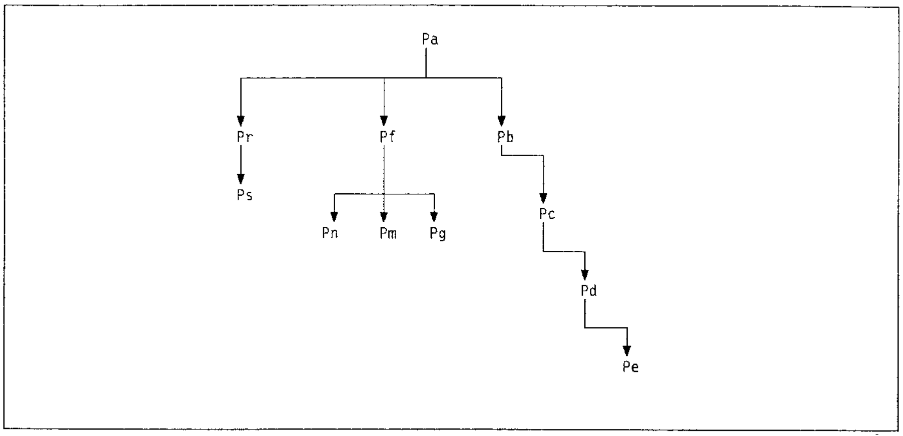

*Figure 26.* Transaction tree used for analysis. 11 participants, where 4 of them (Pb, Pc, Pd, Pe) follow a particular optimization.

DEC's DECdtm services [16] takes advantage of the VAXcluster architecture to reduce the number of required forced log writes and to achieve a nonblocking 2PC protocol within VAXcluster transactions without the extra messages described in [27]. Any subordinates within the coordinator's VAXcluster can access the coordinator's log. Therefore the subordinates are not blocked if the coordinator fails in the middle of a 2PC operation since they can access the log to determine the transaction outcome. The resulting log reductions are similar to those of the sharing-the-log optimization described in Section 6. Based on the assumption that most transactions commit, the DEC optimization further reduces the commit latency by allowing intermediate coordinators to force write the prepared log record while waiting for votes from subordinates to arrive. This allows the intermediate coordinator to respond immediately to its coordinator as soon as the last subordinate vote arrives, without having to wait for a log write to complete.

Transarc's Encina [7] uses a variation of the unsolicited vote optimization. Each server prepares itself before responding to each and every remote procedure call (RPC), indicating its prepared status on the return message. If the client decides to initiate the commit protocol, phase one can be skipped. The main differences from the unsolicited-vote optimization are that the subordinate is always prepared and that it can still accept new work. In the case where a single transaction involves multiple RPCs to a server, the Transarc optimization results in more force writes in the server.

An optimization known as coordinator migration [7, 11] improves reliability by transferring the commit decision to more reliable partners. With the last-agent optimization, the original commit coordinator can unilaterally transfer the commit decision to an immediate subordinate, which can further transfer the decision to one of its subordinates, and so on. Coordinator migration, however, provides a formal way to negotiate which member of the transaction tree will serve as the coordinator. The current coordinator on the prepare message indicates to the subordinates the potential coordinator. If all subordinates agree, the potential coordinator becomes the official coordinator. Unlike the last-agent optimization, this flexibility is at the expense of always including the first phase of the 2PC protocol.

## 10. Conclusions

Two-phase commit protocols have been studied extensively by the research community. While some of the research has concentrated on improving performance in the failure case, we find it is more advantageous to optimize for the normal, non-failure case in today's commercial environment. This paper describes several 2PC variations and their combinations that optimize towards the normal case, comparing them to a baseline 2PC protocol and describing environments where they are most effective. The variations are compared and contrasted in terms of number of messages, number of log writes (both forced and non-forced), probability of heuristic damage, how damage is reported, and other tradeoffs.

Although most of these optimizations have been incorporated in IBM's LU 6.2 sync point protocols, they were presented in this paper independently of the underlying communications protocol to avoid implementation details. A description of some of these optimization as they might be incorporated in IBM's LU6.2 is presented in [31, 21].

## Acknowledgments

We would like to acknowledge contributions of our colleague at Almaden Research Center, Bruce Lindsay, and valuable feedback from Ajay Kshemkalyani, Gary Schultz, and James P. Gray.

## Notes

1. Whenever we refer to a process, we are not necessarily referring to a process as defined by the operating system. A process may in fact be a lightweight *thread*, a piece of *context*, or a thread of control (XOPEN model).
2. The communication network is actually accessed through appropriate communication resource managers (CRM). Such CRMs can be conversational (LU 6.2 or OSI/TP), RPC based, or message based. These CRMs do not effect the performance of the 2PC processing but merely utilize the underlying communication network. Thus, they are not shown in Fig. 4.
3. The END log record at a leaf subordinate (LRM) is not strictly needed. Since it is included in some 2PC implementations, we included it here to simplify the analysis.
4. IBM, CICS/MVS, DB2, IMS/VS and VM/ESA are trademarks of International Business Machines Corp. TMF and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX and VMS are trademarks of Digital Equipment Corp. Transarc is a registered trademark of Transarc Corp. Encina is a trademark of Transarc Corp. Tuxedo and Unix are registered trademarks of Unix System Laboratories, Inc. X/Open is a trademark of X/OPEN Company Ltd.
5. Presumed Commit protocols [22] are not described in this paper because they have not been implemented in commercial products.
6. There can be specific resources within an overall DB system (e.g. a specific set of tables, or a specific set of IMS/ESA DL1 databases) that are not allowed to be heuristically changed. For example, in CICS/MVS, protected transient data can sometimes have this property, while all other resources are subject to heuristic damage.

## References

1. P. Bernstein, W. Emberton, and V. Trehan, "DECdta—Digital's Distributed Transaction Processing Architecture," Digital Technical Journal, Vol. 3, No. 1, Winter 1991.
2. P. Bernstein, "Transaction Processing Monitors," Communications of the ACM, Vol. 33, No. 11, November 1990.
3. E. Braginsky, "The X/Open DTP Effort," Proc. 4th International Workshop on High Performance Transaction Systems, Asilomar, September 1991.
4. U. Buerger, "A Flexible Two-Phase Commit Protocol," Computer Networks and ISDN Systems, Vol. 17, No. 3, September 1989.
5. C.-L. Huang and V.O.K. Li, "A Quorum-Based Commit and Termination Protocol for Distributed Database Systems," Fourth International Conference on Data Engineering, Los Angeles, California, February 1–5, 1988.
6. CICS General Information, Document Number GC33-0155-4, IBM, October 1990.
7. J. Eppinger and M. Yung, Transarc's Encina Environment, USENIX, 1990.
8. D. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," IEEE Database Engineering, Vol. 8, No. 2, June 1985.
9. J.N. Gray, "Notes on Data Base Operating Systems," in Operating Systems—An Advanced Course, R. Bayer, R. Graham, and G. Seegmuller (Eds.), Lecture Notes in Computer Science, Vol. 60, Springer-Verlag, 1978. Also available as IBM Research Report RJ2188, IBM Almaden Research Center, February 1978.
10. J.N. Gray, "The Transaction Concept: Virtues and Limitations," Proc. 7th International Conference on Very Large Data Bases, October 1981.
11. J.N. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann series, 1993.
12. P. Helland, "The LU 6.2 Protocol Boundary: The 'L' Stands for 'Lightweight'," Proc. 3rd International Workshop on High Performance Transaction Systems, September 1989.
13. P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group Commit Timers and High Volume Transaction Processing Systems," Proc. 2nd International Workshop on High Performance Transaction Systems, September 1987.
14. M. Hesselgrave, "Considerations for Building Distributed Transaction Processing Systems on UNIX System V," Proc. UNIFORUM, Washington, January 1990.
15. IBM Database System DB2, Document Number GG24-3400-0, IBM, 1988.
16. W. Laing, J. Johnson, and R. Landau, "Transaction Management Support in the VMS Operating System Kernel," Digital Technical Journal, Vol. 3, No. 1, Winter 1991.
17. B.W. Lampson, "Atomic Transactions," in Distributed Systems: Architecture and Implementation—An Advanced Course, B.W. Lampson (Ed.), Lecture Notes in Computer Science, Vol. 105, Springer-Verlag, 1981, pp. 246–265.
18. B. Lindsay, L. Haas, C. Mohan, P. Wilms, and R. Yost, "Computation and Communication in $R*$: A Distributed Database Manager," ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984. Also available as IBM Research Report RJ3740, IBM Almaden Research Center, January 1983.
19. G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, "Query Processing in $R*$," in Query Processing in Database Systems, W. Kim, D. Reiner, and D. Batory (Eds.), Springer-Verlag, 1985. Also available as IBM Research Report RJ4272, IBM Almaden Research Center, April 1984.
20. B. Maslak, J. Showalter, and T. Szczygielski, "Coordinated Resource Recovery in VM/ESA," IBM Systems Journal, Vol. 30, No. 1, 1991.
21. C. Mohan, K. Britton, A. Citron, and G. Samaras, Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols, IBM Research Report RJ8684, IBM Almaden Research Center, November 1991.
22. C. Mohan and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983. Also available as IBM Research Report RJ3881, IBM Almaden Research Center, June 1983.
23. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the $R*$ Distributed Data Base Management System," ACM Transactions on Database Systems, Vol. 11, No. 4, December 1986. Also available as IBM Research Report RJ5037, IBM Almaden Research Center, February 1986.
24. An Overview of Information Management System/Virtual Storage (IMS/VS) Intersystem Communications (ISC), Document Number G320-5856, IBM, July 1980.

25. K. Rothermel and S. Pappe, "Open Commit Protocols for the Tree of Processes Model," Proc. 10th International Conference on Distributed Computing Systems, Paris, May–June 1990.
26. G. Samaras, K. Britton, A. Citron, and C. Mohan, Enhancing SNA's LU6.2 Sync Point to Include Presumed Abort Protocol, IBM Techical Report TR29.1751, IBM Research Triangle Park, December 1992.
27. D. Skeen, "Nonblocking Commit Protocols," Proc. ACM/SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, 1981, pp. 133–142.
28. A. Spector, "Open Distributed Transaction Processing with Encina," Proc. 4th International Workshop on High Performance Transaction Systems, Asilomar, September 1991.
29. P. Spiro, A. Joshi, and T.K. Rengarajan, "Designing an Optimized Transaction Commit Protocol," Digital Technical Journal, Vol. 3, No. 1, Winter 1991.
30. M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Transactions on Software Engineering, Vol. 5, No. 3, May 1979.
31. Systems Network Architecture LU 6.2 Reference: Peer Protocols, Document Number SC31-6808-1, IBM, September 1990. Chapter 8 is the one that introduces and describes in detail the Presumed Nothing commit protocol.
32. Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2, Document Number GC30-3084-4, IBM, September 1991.
33. The Tandem Database Group, "NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL," Proc. 2nd International Workshop on High Performance Transaction Systems, Asilomar, September 1987. Also in Lecture Notes in Computer Science, Vol. 359, D. Gawlick, M. Haynie and A. Reuter (Eds.), Springer-Verlag, 1989.
34. Transaction Processing and Sync Points, IBM Document, Document Number AWP-0055-6, IBM/RTP, October, 1977.
35. F. Upton IV, "OSI Distributed Transaction Processing, An Overview," Proc. 4th International Workshop on High Performance Transaction Systems, Asilomar, September 1991.