Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment[1]

George Samaras, Kathryn Britton, Andrew Citron, C. Mohan*

IBM Distributed Systems Architecture, IBM Almaden Research Center*
{Samaras, Brittonk, Citron}@ralvm6.vnet.IBM.com
mohan@almaden.ibm.com*

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. One widely used protocol is the two-phase commit (2PC) protocol, which has long appeared in the literature.

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 analyzed in terms of reliability, savings in log writes and network traffic, and reduction in resource lock time. Its unique contributions include the description of some optimizations not described elsewhere and a systematic comparison of the optimizations and the environments where they cause the most benefit.

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 [6, 19].

The performance of a commit protocol substantially affects the transaction volume that a system can support. As pointed out in [28], 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 [28] 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. 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 (see, i.e., [26, 4]) has concentrated on providing a (nearly) non-blocking 2PC variation, i.e., one that adds extra message flows 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 message flows 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 [25, 22] not to wait for recovery processing to discover the outcome because of valuable locks being held. 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. 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

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

1063-6382/93 $03.00 © 1993 IEEE

IBM's Presumed Nothing protocols and several new optimizations, particularly ones that effect peer-to-peer transactions (i.e Leaving Inactive Partners Out, Last Agent, Wait For Outcome, Vote Reliable and Long Locks). Some of these optimizations have been designed on top of IBM's LU6.2 communication protocol[1], however, their presentation here is independent of any communication protocol. LU6.2 implementation specifics for some of these optimizations can be found in [14] and [23].

Section 2 introduces a 2PC protocol that is used as a baseline for comparing the 2PC variations introduced in the rest of the paper. Section 3 presents the Presumed Abort (PA) and IBM's Presumed Nothing (PN) protocols and their usefulness within the commercial sector. Section 4 discusses several optimizations that are refinements of PN or PA or both, along with their advantages and tradeoffs in different environments. Section 5 provides a performance analysis of the presented optimizations. Section 6 concludes the paper.

2. Baseline Two-Phase Commit

As background for later discussion of 2PC variations, this section introduces features of a distributed 2PC protocol that affect performance: network message flows and required log writes [24, 25]. These features are illustrated with a basic 2PC protocol [6, 19] that is used as a comparison baseline for the optimizations that follow.

Two-Phase Commit Concepts

Two types of components participate in 2PC: ***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 coordinate 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.

Participants 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. Two types of log writes are performed by the participants, ***forced log writes*** and ***non-forced writes***. 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.

A widely used commit protocol is the two-phase commit (2PC) protocol. 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, 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 data bases or files, making them available to other transactions.

Figure 1 (and Figure 2) shows the message flows and forced log writes involved in the classic two-phase commit protocol. The first two flows comprise the voting phase, while the next two flows comprise the decision phase.

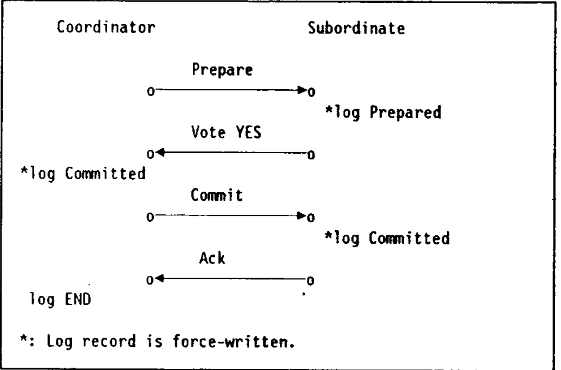

Figure 1. Simple Two-Phase Commit Processing

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 2 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.

Network Traffic

The 2PC protocol involves network traffic to convey the instructions to prepare and later to commit from the coordinator to remote 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 network flows. Sending messages to different participants in parallel also reduces the delay caused by network traffic. In some cases, reducing the number of flows and parallelism are in conflict (see last-agent optimization).

Logging

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 non-volatile 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. For example, the END log record does not need to be forced because the only effect of its absence following a failure is redundant recovery processing, which takes extra recovery time but does no other harm.

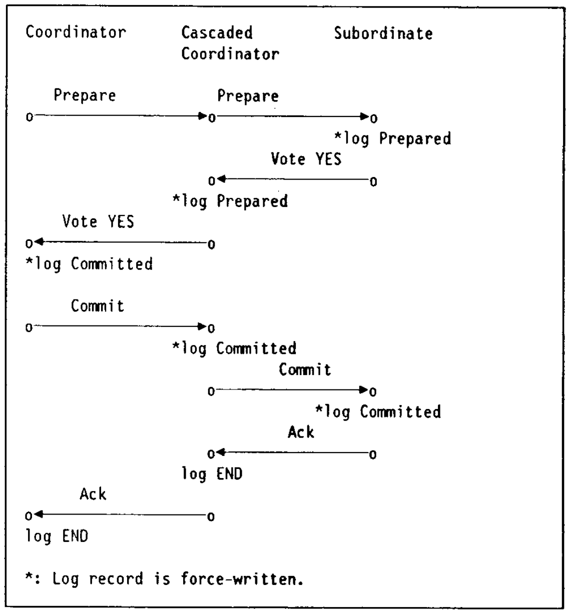

Figure 2. Two-Phase Commit Processing with Intermediate Coordinator

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[2] 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.

[3] IBM, CICS, 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..

3. 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) [14, 17] and initially by LU6.1 [12]. The PN design effort was done independently from the 2PC effort [11]. PN was designed and developed for the commercial environment and, so far, IBM has implemented it in CICS[3] [15], and VM/ESA[3] [22].

The peer-to-peer environment has led to the following unusual features of PN:

- The transaction itself is not inherently a tree. Any program can initiate work; two programs can initiate work independently with or without any communication between them. This is in contrast to a client-server model, assumed by most 2PC protocols, where the client starts the transaction and servers wait until they get requests from clients or other servers.

  The transaction graph (tree) constituting a single distributed computation may cause multiple transactions to begin and terminate in a serial fashion.

- 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. Of course, 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. 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 (see Figure 3). 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.

[2] 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.

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.

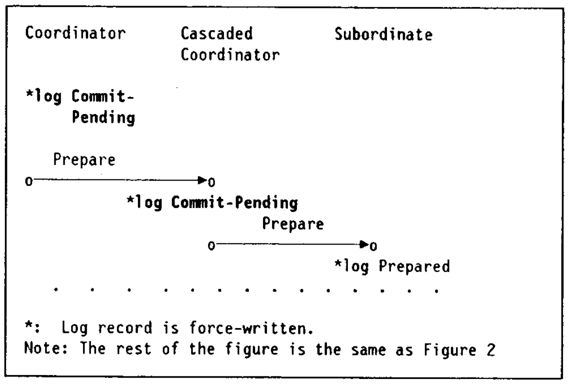

Figure 3. Presumed Nothing Commit Processing with Intermediate Coordinator

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.

Presumed Abort (PA)

Presumed Abort [24, 25] is an extension of the basic 2PC protocol that has been widely studied in academia and industry. It has been implemented by a number of commercial products[3], i.e, Tandem's TMF [30], DEC's VAX/VMS [1, 18], Transarc's Encina Product Suite [27], and Unix System Laboratories' TUXEDO [9], and is now part of the ISO-OSI [31] and X/Open[3] [3] distributed transaction processing standards. PA was developed for the R* distributed database project [20, 21]. 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. 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. If the coordinator has no information about the transaction, it presumes that the transaction aborted and tells the subordinate to abort; hence the name ***presumed abort***.

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

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.

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. Moreover, R* was a research project, and real customers did not have real data involved.

The optimizations developed by PA for the client-server environment have been generalized to be incorporated in the peer-to-peer model [23].

4. 2PC Optimizations

This section describes several optimizations to the PA or PN protocols or both, some of which have been previously published [24, 25, 6, 14]. These optimizations are tuned toward the normal non-failure case. 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 [24, 25].

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.

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. PN still has the coordinator log a commit-pending record, but the subordinate performs no logging.

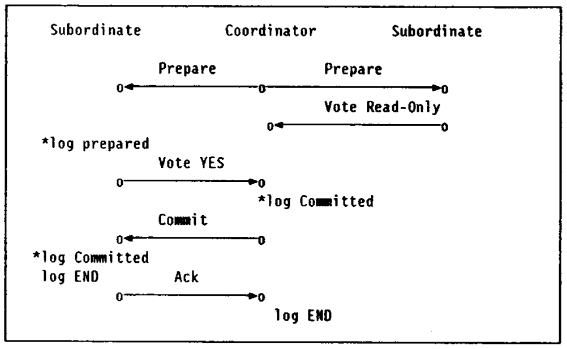

Figure 4. Partial Read-Only Commit Processing.

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 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 requestor/server environment, since the transaction processes cannot start any updates once 2PC has begun.

For a transaction commit tree of **n** members and **m** participants that vote read-only the savings amount to 2m forced-writes and 2m message flows over the basic 2PC protocol.

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* [20]. In this section we briefly described the adaptation of this optimization for PN. Greater detail is presented in [23].

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 Figure 5 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.

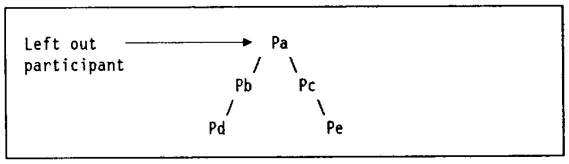

Figure 5. Transaction Tree Partitioned Because of Left Out Partners

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 provides a parameter on the SET_SYNCPT_OPTIONS verb to indicate whether the local transaction program 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 message flows over the basic 2PC protocol.

Last Agent

Experience with CICS/MVS and IBM's DB2 [13] 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 [6].

Unlike the normal 2PC case, the coordinator is not required to send an explicit acknowledgment when it receives a commit message. In the normal case, the coordinator can be blocked waiting for acknowledgments (see late acknowledgment below). 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 its TM to write the End log message and forget the outcome of the transaction. This optimization is illustrated in Figure 6.

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 message flows 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 (i.e connection through satellite) the last-agent optimization provides significant savings. It is, for example, preferable to prepare the closely located partners (fast first phase) and reduce the communication required with the faraway partner to one slow round-trip message exchange.

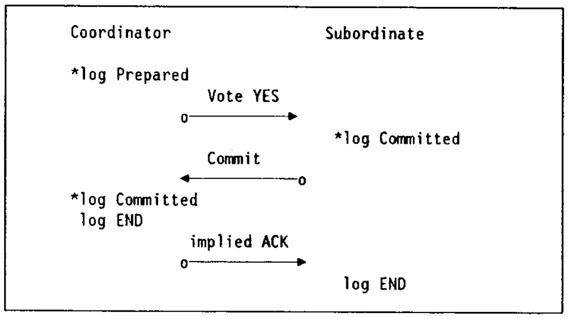

Figure 6. Last-Agent Commit Processing

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 message flows 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 agent is a not-last agent, the coordinator does not force any log record before the Committed record. However, the extra forced write can be avoided if the initiator is read-only. With respect to this case, the initiator can vote read only to the last agent without having to force-write a prepared log record.

With the peer-to-peer nature of PN, 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. In fact, this case looks very much like the unsolicited-vote optimization described next.

For a transaction tree of n members and m last agents this optimization offers savings of 2m message flows 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 conjuction with the last-agent optimization, a bit in the **YES** vote 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 relative 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 [29] and IBM's IMS/VS [12]. For a transaction tree of **n** members and **m** unsolicited-ready participants this optimization saves **m** message flows over the basic 2PC protocol.

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 [25]. 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 occur: 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[3] Fast-Path [5].

For **n** transactions and a group commit of size **m**, this optimization provides an average of **3n/2m** forced-writes savings. 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 [4, 28].

Long Locks

LU 6.2 2PC protocols (PN) allow an application program to trade off network flows 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 flows by one at the cost of keeping the resources at the coordinator locked for a longer period. Note that LU 6.2 allows this 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.

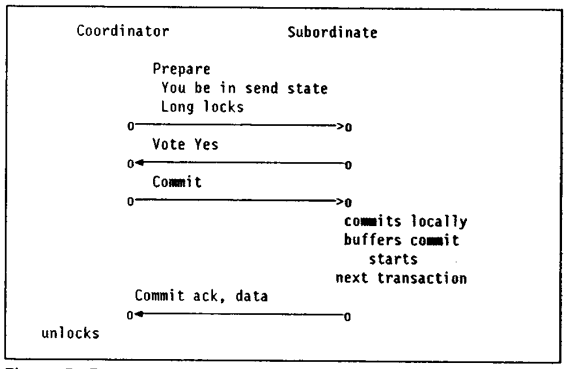

Figure 7. Example of Long Locks committing one transaction.

Figure 7 shows the long-locks variation of the basic LU 6.2 2PC protocol. The LU 6.2 Prepare (to not-last agents) and Vote **YES** (to last agents) messages include instructions about the conversation state the subordinate is expected to be in after a successful commit. It also informs the subordinate whether or not the coordinator wants the long-locks variation. The long-locks variation combined with the last-agent optimization can commit two transactions in three flows.

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 [8]. The application involved banks that needed to reconcile their log accounts at the end of the day. This application is particularly characterized by a large number of short transactions with small delays between them.

Commit Acknowledgment

One of the ways that different 2PC protocols vary is in the timing of the commit acknowledgment. Some have early acknowledgment [11, 25]: an intermediate system acknowledges a commit received as soon as it has logged; others have late acknowledgment [14]: 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 data-base 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-acknowledgment protocols with its coordinator during the commit phase (see Figure 8). 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 is not reliable[4]. However, since the resources involved can vary from transaction to transaction, the intermediates collect the reliability information during every first phase.

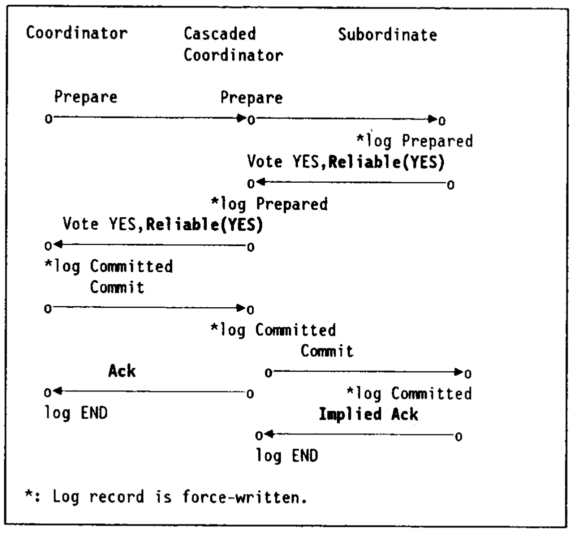

Figure 8. Two-Phase Commit Processing, All Resources Voted Reliable.

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 [22]. 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 [14, 17] 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.

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.

5. Performance Evaluation and Discussion

Two-phase commit optimizations comprise a set of rules applied during 2PC processing that result in network traffic improvements, reduction in the number of forced writes, and decreased resource lock time. However, one other aspect that is important while evaluating those optimizations is reliability: how these optimizations reliably report the outcome of the transaction and whether they increase the chances of heuristic damage.

Not all the optimizations provide improvements across all performance metrics, and often an optimization might trade off one metric for another. However, better performance can be achieved by combining the different optimizations. Interesting configurations can be proposed but because of space limitations we do not discuss them here. We hope to present these intriguing combinations in a future paper.

In the tables that follow, optimizations are analyzed in terms of the absolute number of message exchange with subordinates. Further analysis would break down the messages into those to LRMs and those to remote TMs, which will 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 flows and log writes of the optimization and compares them with the basic two-phase commit, presumed abort, and presume nothing. 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 flows and log writes[2] needed to commit a transaction with n members. Each row in the table describes the benefits gained if m participants use a particular optimization. The example used in this table is a transaction with 11 participants of which 4 followed the same optimization. The intention is not to compare the optimizations with each other but to contrast them with the basic 2PC protocol.

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

[4] 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.

Table 1. Advantages and Disadvantages of 2PC Optimizations

Optimization	Advantages	Disadvantages
Read Only	fewer messages, fewer log writes, early release of locks	no knowledge of the outcome of a transaction, potential serializability problems
Last Agent	fewer messages, early release of locks	one extra forced write possible
Unsolicited Vote	fewer messages, early release of locks	Application specific
Ok To leave Out	no log writes, no messages	N/A
Vote Reliable	fewer message flows	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	Complete outcome of transaction may not be known by coordinator
Long Locks	fewer network flows	commit decision can be delayed and locks held longer if combined with last-agent optimization, and no messages flow for the next transaction (application design problem).
Shared Logs	fewer forced writes	independence of resource manager and transaction manager sacrificed.
Group Commit	fewer forced writes, overall system throughput maximized.	longer lock holding times for individual transactions.

Table 2. Logging and network traffic of 2PC optimizations

2PC Type	Coordinator Flows	Coordinator Logs	Subordinate Flows	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 & Last-Agent	1	3, 2 forced[1]	1[2]	2, 1 forced
PA & Unsolicited Vote	1	2, 1 forced	2	3, 2 forced[1]
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	2	2, 1 forced	2	3, 2 forced[1]
PA & Long Locks (not last-agent)	2	2, 1 forced	1	3, 2 forced[1]
PA & shared logs	2	2, 1 forced	2	3, 0 forced

Note: [1] The 2 forced writes are prepared and committed, the END is not force-logged. 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. The pair (x, y forced) means that x log writes are performed, of which y are forced.

Table 3. Logging and Message Costs for Optimizations. Transaction consists of n partners where m members are following a particular optimization.

2PC TYPE	Flow	Log Write	n = 11, m = 4 /f,w,uf
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 & 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	4(n-1)	3n-1, 2n-1 forced	40, 32, 21
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,uf) 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	Flow	Log Write	r = 12 /f,w,uf
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,uf) refers to (# of messages, # of log writes, # of forced writes)

6. 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 in today's commercial environment it is more advantageous to optimize for the normal, non-failure case. This paper describes eleven 2PC variations 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 message flows, 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 [14, 23].

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.

References

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

3 Braginsky, E. *The X/Open DTP Effort*, **Proc. 4th Int. Workshop on High Performance Transaction Systems**, Asilomar, September 1991.

4 Ching-Liang, Victor O.K. Li *A Quorum-based Commit and Termination Protocol for Distributed Database Systems*, **Fourth Int. Conference on Data Eng.**, Los Angeles, California, February 1-5, 1988

5 Gawlick, D., Kinkade, D. *Varieties of Concurrency Control in IMS/VS Fast Path*, **IEEE Database Eng.**, Vol. 8, No. 2, June 1985.

6 Gray, J.N. *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, Volume 60, Springer-Verlag, 1978. Also Available as **IBM Research Report RJ2188**, IBM Almaden Research Ce nter, February 1978.

8 Helland, P. *The LU6.2 protocol boundary: The 'L' stands for 'Lightweight'*, **Proc. 3th Int. Workshop on High Performance Transaction Systems**, September, 1989.

9 Hesselgrave, M. *Considerations for Building Distributed Transaction Processing Systems on UNIX System V*, **Proc. UNIFORUM**, Washington, January 1990.

10 Helland, P., Sammer, H., Lyon, J., Carr, R., Garrett P., and Reuter, A., Group Commit Timers and High Volume Transaction Processing Systems,, **Proc. 2nd Int. Workshop on High Performance Transaction Systems**, September 1987.

11 *Transaction Processing and Sync Points*, **IBM Document**, **Document Number AWP-0055-6**, IBM/RTP, October,1977.

12 *An Overview of Information Management System/Virtual Storage (IMS/VS) Intersystem Communications (ISC)*, **Document Number G320-5856**, IBM, July, 1980.

13 *IBM Database System DB2* , **Document Number GG24-3400-0, GG24-3202-1**, IBM, 1988.

14 *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.

15 *CICS General Information*, **Document Number GC33-0155-4**, IBM, October 1990.

17 *Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2*, **Document Number GC3C-3084-4**, IBM, September 1991.

18 Laing, W., Johnson, J., Landau, R. *Transaction Management Support in the VMS Operating System Kernel*, **Digital Technical Jour.**, Vol. 3, No. 1, Winter 1991.

19 Lampson, B.W. *Atomic Transactions*, In **Distributed Systems: Architecture and Implementation - An Advanced Course**, B.W. Lampson (Ed.), Lecture Notes in Computer Science, Volume 105, Springer-Verlag, p246-265, 1981.

20 Lindsay, B., Haas, L., Mohan, C., Wilms, P., Yost, R. *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.

21 Lohman, G., Mohan, C., Haas, L., Daniels, D., Lindsay, B., Selinger, P., Wilms, P. *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.

22 Maslak, B., Showalter, J., Szczygielski, T. *Coordinated Resource Recovery in VM/ESA*, **IBM Systems Jour.**, Vol. 30, No. 1, 1991.

23 Mohan, C., Britton, K., Citron, A., Samaras, G. *Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols*, **IBM Research Report**, IBM Almaden Research Center, November, 1991.

24 Mohan, C., Lindsay, B. *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.

25 Mohan, C., Lindsay, B., Obermarck, R. *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.

26 Skeen, D. *Nonblocking Commit Protocols*, **Proc. ACM/SIGMOD Int. Conference on Management of Data**, Ann Arbor, Michigan, 1981, pp. 133-142.

27 Spector, A. *Open, Distributed Transaction Processing with Encina*, **Proc. 4th Int. Workshop on High Performance Transaction Systems**, Asilomar, September 1991.

28 Spiro, P., Joshi, A., and T.K. Rengarajan *Designing an Optimized Transaction Commit Protocol* **Digital Technical Jour.**, Vol. 3, No. 1, Winter 1991.

29 Stonebraker, M. *Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES*, **IEEE Transactions on Software Eng.**, Vol. 5, No. 3, May 1979.

30 The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd Int. Workshop on High Performance Transaction Systems**, Asilomar, September 1987. Also in **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.

31 Upton IV, F. *OSI Distributed Transaction Processing, An Overview*, **Proc. 4th Int. Workshop on High Performance Transaction Systems**, Asilomar, September 1991.