An Argument in Favor of the Presumed Commit Protocol*

Yousef J. Al-Houmaily
Dept. of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261
yjast1+@pitt.edu

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
panos@cs.pitt.edu

Steven P. Levitan
Dept. of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261
steve@ee.pitt.edu

Abstract

We argue in favor of the presumed commit protocol by proposing two new presumed commit variants that significantly reduce the cost of logging activities associated with the original presumed commit protocol. Furthermore, for read-only transactions, we apply our unsolicited update-vote optimization and show that the cost associated with this type of transactions is the same in both presumed commit and presumed abort protocols, thus, nullifying the basis for the argument that favors the presumed abort protocol. This is especially important for modern distributed environments which are characterized by high reliability and high probability of transactions being committed rather than aborted.

1. Introduction

In order to ensure consistent termination of distributed transactions despite site and communication failures, all the sites participating in a transaction's execution engage in an *atomic commit protocol* such as the *two-phase commit* protocol (2PC) [8, 10]. Since 2PC consumes a substantial amount of a transaction's execution time during normal processing [19] and is blocking in the case of both communication and site failures [17], a number of 2PC variants have appeared in the literature, e.g., [1, 3, 9, 11, 15, 18], most notably, the *presumed abort* protocol (PrA) and the *presumed commit* protocol (PrC) [12, 11].

PrA has been designed to reduce the cost associated with aborting transactions while, its counterpart, PrC has been designed to reduce the cost associated with committing transactions. To reduce the cost of commit processing further, a number of optimizations have also been proposed, of which the read-only optimization [12] is the most significant, given that read-only transactions are the majority in any general database system. (See [15] for a survey of the most common two-phase commit optimizations.)

Due to the cost of the logging activities associated with PrC even for read-only transactions, the argument usually goes in favor of PrA. However, modern communication networks as well as computing systems are becoming more reliable and distributed transactions tend to commit after all their operations have been successfully executed and acknowledged. Therefore, we were prompted to revisit the debate between PrA and PrC and to investigate techniques that enhance the performance of PrC. Our investigations led us to three techniques which are presented in this paper and which reduce, and under certain circumstances even eliminate, the logging activities from PrC. Whereas PrA has been the current choice of commercial systems and standards [6], the proposed three techniques when combined with PrC form an argument in its favor to become part of the standards. This argument is further strengthened by the fact that the incompatibility between these two variants is not an issue anymore because PrA and PrC can be interoperated in a practical manner [2].

The rest of this paper is structured as follows. In the next section, to establish the stage for our arguments, we briefly overview the basic 2PC, PrA and PrC in the context of two-level as well as multi-level transaction execution models. We also discuss the traditional read-only optimization. In Section 3, we review why the argument usually goes in favor of PrA rather than PrC. In Section 4, given that the multi-level transaction execution model is the one specified in the transaction processing standards and adopted by commercial database systems, we propose two new PrC variants for the multi-level transaction execution model that reduce the logging activities significantly when compared with the original PrC. Then, we show how we can eliminate the logging activities from read-only transactions in PrC as well as in the newly proposed PrC variants by applying our *unsolicited update-vote* optimization. We conclude with a summary in Section 5.

*Supported in part by N.S.F. under grants IRI-9210588 and IRI-95020091 and a Saudi Arabian graduate student scholarship.

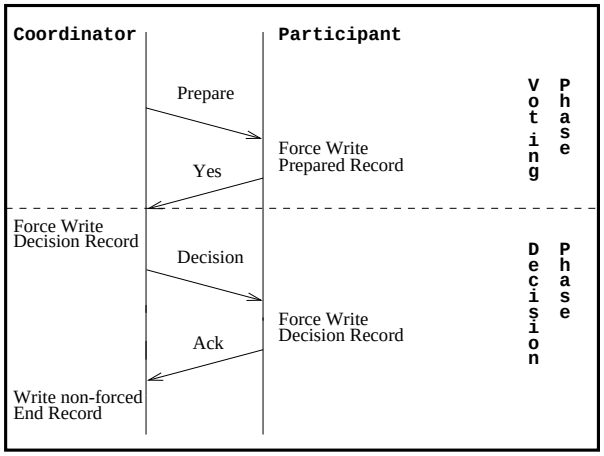


Figure 1. The basic two-phase commit protocol.

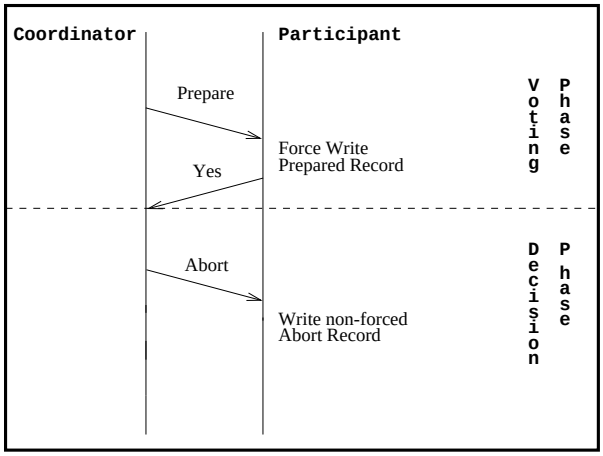


Figure 2. The presumed abort protocol.

2. Background

In a distributed database system, data are typically stored in disjoint partitions at different sites. This data distribution is transparent to a distributed transaction that accesses data by submitting database operations to its *coordinator*. Without loss of generality, we assume that the coordinator of a transaction is the *transaction manager* at the site where the transaction has been initiated. While still adhering to the traditional ACID (i.e., Atomicity, Consistency, Isolation and Durability) properties of transactions [9], a distributed transaction is decomposed into a set of *subtransactions*, each of which executes at a single participant site. When a transaction finishes its execution and submits its commit request, its coordinator initiates an *atomic commit protocol*, such as the two-phase commit protocol.

The basic *two-phase commit* protocol (2PC) [8, 10], as the name implies, consists of two phases, namely a *voting phase* and a *decision phase* (Figure 1). During the voting phase, the coordinator of a distributed transaction requests all the participating sites to *prepare to commit* whereas, during the decision phase, the coordinator either decides to commit the transaction if *all* the participants are *prepared to commit* (voted *Yes*), or to abort if any participant has *decided to abort* (voted *No*). If a participant has voted *Yes*, it can neither commit nor abort the transaction until it receives the final decision. When a participant receives the final decision, it complies and acknowledges the decision. The coordinator discards any information in its *protocol table* in main memory regarding the transaction when it receives acknowledgments from all the participants and forgets the transaction.

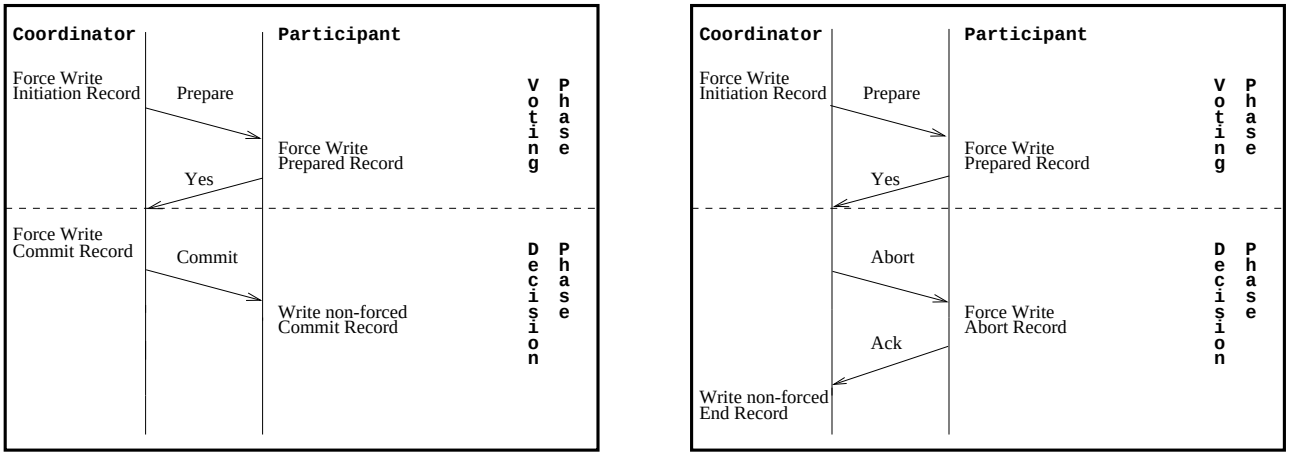
The resilience of 2PC to system and communication failures is achieved by recording the progress of the protocol in the logs of the coordinator and the participants. The coordinator force-writes a **decision** record prior to sending out the final decision. Since a *force-write* ensures that a log record is written into a stable storage that survives system failures, the final decision is not lost if the coordinator fails. Similarly, each participant force-writes a **prepared** record before sending its *Yes* vote and a **decision** record before acknowledging the final decision[1]. When the coordinator completes the protocol, it writes a non-forced **end** record, indicating that the log records pertaining to the transaction can be garbage collected when necessary.

The basic 2PC is also referred to as the *presumed nothing* 2PC protocol (PrN) [11] because it treats all transactions uniformly, whether they are to be committed or aborted, requiring information to be explicitly exchanged and logged at all times. However, in the case of a coordinator's failure, there is a hidden presumption in PrN by which the coordinator considers all active transactions at the time of the failure as aborted ones. The *presumed abort* protocol (PrA) makes this abort presumption explicit [12, 11].

Specifically, in PrA, when a coordinator decides to abort a transaction, it does not force-write the abort decision in its log as in PrN (Figure 2). It just sends abort messages to all the participants that have voted *Yes* and discards all information about the transaction from its protocol table. That is, the coordinator of an aborted transaction does not have to write any log records or wait for acknowledgments. Since the participants do not have to acknowledge abort decisions, they are also not required to force-write such decisions. After a coordinator or a participant failure, if the participant *inquires* about a transaction that has been aborted, the coordinator, not remembering the transaction, will direct the participant to abort it (by presumption).

As opposed to PrA, the *presumed commit* protocol (PrC) is designed to reduce the cost of committing transactions [12, 11]. Instead of interpreting missing information about

[1] Writing the decision at the participants and acknowledging it in a lazy fashion [9] is an optimization that is not considered here.



(a) Commit Case. **(b) Abort Case.**

Figure 3. The presumed commit protocol.

transactions as abort decisions, in PrC, coordinators interpret missing information about transactions as commit decisions. However, in PrC, a coordinator has to force write an **initiation** (which is also called *collecting* in [12]) record for each transaction before sending *prepare to commit* messages to the participants. This record ensures that missing information about a transaction will not be misinterpreted as a commit after a coordinator failure.

To commit a transaction (Figure 3 (a)), the coordinator force writes a **commit** record to logically eliminate the **initiation** record of the transaction and then sends out the *commit* decision. The coordinator also discards all information pertaining to the transaction from its protocol table. When a participant receives the decision, it writes a non-forced **commit** record and commits the transaction without having to acknowledge the decision. After a coordinator or a participant failure, if the participant inquires about a transaction that has been committed, the coordinator, not remembering the transaction, will direct the participant to commit it (by presumption).

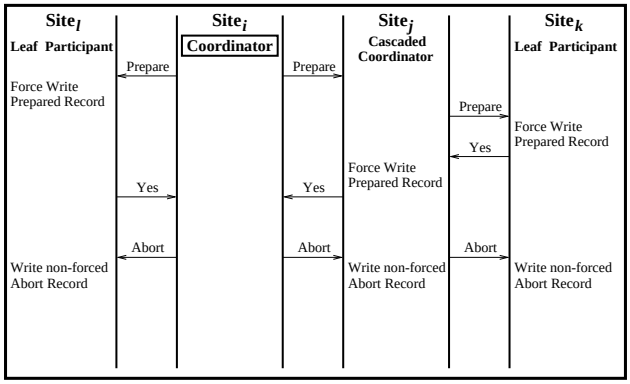
To abort a transaction (Figure 3 (b)), on the other hand, the coordinator does not write the abort decision in its log. Instead, the coordinator, sends out the *abort* decision and waits for the acknowledgments before discarding all information pertaining to the transaction. When a participant receives the decision, it force writes an **abort** record and then *acknowledges* the decision, as in PrN.

2.1. Multi-Level PrA and PrC

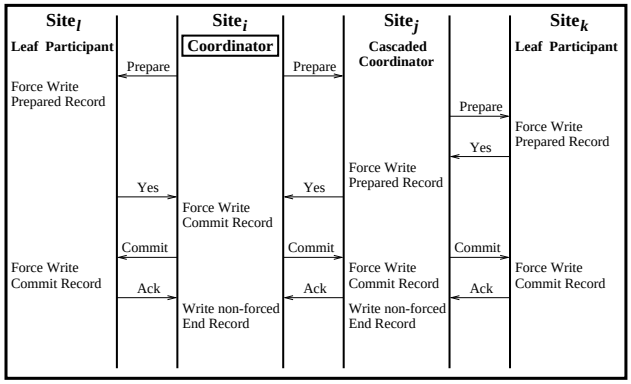
The *multi-level transaction execution* (MLTE) model, the one specified by the standards and adopted by commercial database systems, is similar to the tree of processes model [12]. In this model, a participant is a process that is able to decompose a subtransaction further. Thus, a participant can initiate other participant processes at its site or different sites. Hence, the processes pertaining to a transaction can be represented by a multi-level execution tree where the coordinator process resides at the root of the tree. In this model, the interactions between the coordinator of the transaction and any process have to go through all the intermediate processes that have caused the creation of a process.

In the MLTE model, the behavior of the root coordinator and each leaf participant in the transaction execution tree, in both 2PC variants, remains the same as in two-level transactions. The only difference is the behavior of *cascaded coordinators* (i.e., non-root and non-leaf participants) which behave as leaf participants with respect to their direct ancestors and root coordinators with respect to their direct descendants. Specifically, when a cascaded coordinator receives a *prepare to commit* message, in *multi-level PrA*, it forwards the message to its descendent participants and waits for their votes, as shown in Figure 4. If all descendants have voted *Yes*, the cascaded coordinator force writes a **prepared** log record and then sends a *Yes* vote to its coordinator. If any descendant has voted *No*, the cascaded coordinator sends an *abort* decision to its descendants and a *No* vote to its coordinator. When a cascaded coordinator receives an *abort* decision (Figure 4 (a)), it writes a non-forced **abort** record, forwards the decision to its direct descendants and forgets the transaction. On the other hand, when a cascaded coordinator receives a *commit* decision (Figure 4 (b)), it forwards the decision to its direct descendants and force writes a **commit** record. Afterwards, the cascaded coordinator sends an *acknowledgment* to its coordinator. Once the direct descendants of the cascaded coordinator *acknowledge* the decision, it writes a non-forced **end** record and forgets the transaction.

PrC can be extended in the MLTE model in a manner similar to PrA. However, as shown in Figure 5, each cascaded coordinator in *multi-level PrC* has to force write an **initia-**

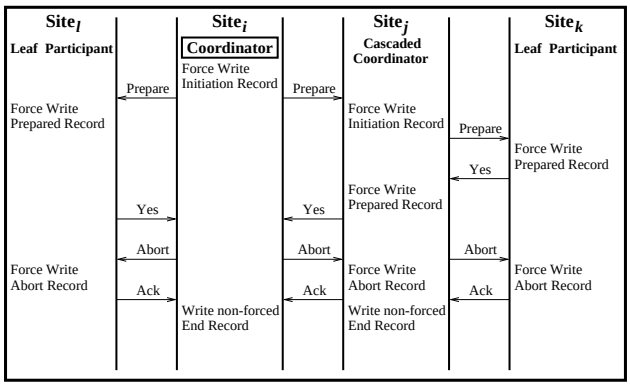


(a) Abort case.

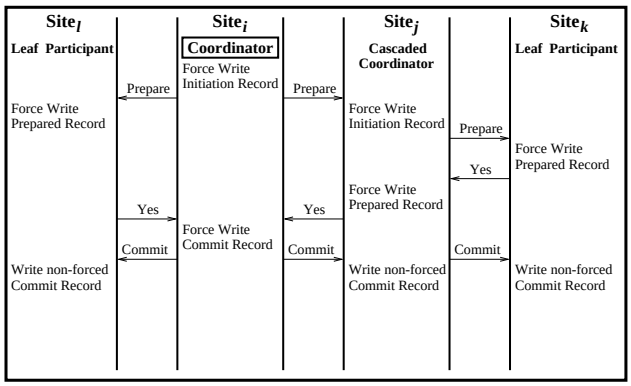


(b) Commit case.

Figure 4. The multi-level presumed abort protocol.



(a) Abort case.



(b) Commit case.

Figure 5. The multi-level presumed commit protocol.

tion record before propagating the *prepare to commit* message to its descendent participants. If the final decision is to abort the transaction (Figure 5 (a)), a cascaded coordinator propagates the decision to its descendants, force writes an **abort** record and, then, *acknowledges* its ancestor. Once the *acknowledgments* arrive from the descendants, a cascaded coordinator writes a non-forced **end** record and forgets the transaction. If the final decision is a commit decision (Figure 5 (b)), a prepared to commit cascaded coordinator propagates the decision to its direct descendants, writes a non-forced **commit** record and, then, forgets the transaction.

2.2. Read-Only Transactions

In the traditional *read-only optimization* [12], when a participant that has executed only read operations on behalf of a transaction receives a *prepare to commit* message from the transaction's coordinator, it either replies with a *No* or *Read-Only* vote instead of a *Yes* and immediately releases all the resources held by the transaction without writing any log records.

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

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

3. The Argument in Favor of PrA

In this section, we illustrate why efficiency arguments usually go in favor of PrA by evaluating the cost associated with commit processing in PrA and showing that it is, in general, less than in PrC. In our evaluation, we also consider read-only transactions.

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

Table 1. The costs for update transactions.

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

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

3.1. Evaluating PrA and PrC

Let us first consider commitment in the two-level transaction execution model. Table 1 summarizes the cost associated with update transactions for the commit as well as the abort case assuming a *Yes* vote from each participant: m is the total number of log records, n is the number of forced log writes, p is the number of messages received from the coordinator and q is the number of messages sent back to the coordinator.

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

Recall that a coordinator, in both PrA and PrC, aborts a read-only transaction since it is cheaper than committing it. As shown in Table 2, both PrA and PrC require the same number of coordination messages to terminate a read-only transaction. However, with respect to the logging activities, a coordinator in PrA does not write any log records whereas in PrC, a coordinator has to write two log records, one of which is forced. Not knowing whether a transaction is going to be read-only, a coordinator in PrC has to force write an **initiation** record. To forget the read-only transaction, the coordinator also writes a non-forced **end** log record when it receives the *Read-Only* votes of the participants.

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

3.2. Evaluating Multi-Level PrA and PrC

In the MLTE model, multi-level PrA and multi-level PrC retain the relative advantages of PrA and PrC. They also retain the relative message complexity of PrA and PrC. However, due to the extra forced **initiation** log records at the cascaded coordinators, the difference between the cost of aborting a transaction in multi-level PrC and multi-level PrA is greater than the difference between PrC and PrA, whereas the difference between the cost of committing a transaction in multi-level PrC and multi-level PrA is less than the difference between PrC and PrA. Let us illustrate this by considering a transaction with N participants of which C are cascaded coordinators and L are leaf participants.

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

In addition to reducing the relative performance advantage of multi-level PrC over multi-level PrA in committing transactions, the fact that these extra forced **initiation** records are written *sequentially* during the voting phase gives rise to another undesirable effect. A coordinator in multi-level PrC experiences more delays to reach a final decision than in multi-level PrA. Consequently, participants in multi-level PrC receive a final decision later than in multi-level PrA, thereby, participants hold the resources longer in multi-level PrC than in multi-level PrA. This means that in the case of transactions with deep trees (e.g., long executing

transactions which potentially access many data items), the tradeoff between reducing conflicts over data items in multi-level PrA and reducing extra forced commit records at every participant (cascaded coordinator or leaf participant) in multi-level PrC goes in favor of multi-level PrA.

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

3.3. Summarizing the Argument

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

4. An Argument in Favor of PrC

The argument, thus far, has been in favor of PrA because of the major drawback of PrC which requires forcing **initiation** records for both read-only and update transactions. Thus, if there is a way to eliminate or reduce the cost associated with the **initiation** records, the argument would go in favor of PrC, especially given the fact that high speed networks and computing systems are becoming highly reliable and distributed transactions will most probably commit after all their operations have been successfully executed and acknowledged. The same intention has been behind the design of the *new presumed commit* protocol for the two-level transaction execution model [11].

In this section, we present two new PrC variants that effectively eliminate *all* the intermediate **initiation** records from cascaded coordinators in the MLTE model. The first PrC variant is called the *rooted PrC* protocol where only the root coordinator force writes an **initiation** record. The second PrC variant is called the *re-structured PrC* protocol which is based on the idea of *flattening* the transaction trees [15]. After we have presented the new PrC variants, we present our *unsolicited update-vote* optimization [4] and apply it to both PrA and PrC (including the two proposed PrC variants) and show that the cost associated with read-only transactions becomes the same in both PrA and PrC, with PrC performing better than PrA for update and partially read-only transactions.

4.1. The Rooted PrC Protocol (RPrC)

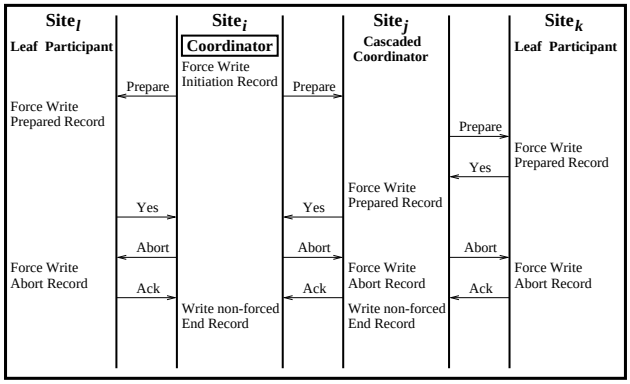
As opposed to multi-level PrC, RPrC does not realize the two-level presumption of PrC on every adjacent level because it structures cascaded coordinators as leaf participants with respect to logging. That is, cascaded coordinators do not force write **initiation** records and, consequently, do not presume commitment in the case that they do not remember transactions.

4.1.1. Description of the Protocol

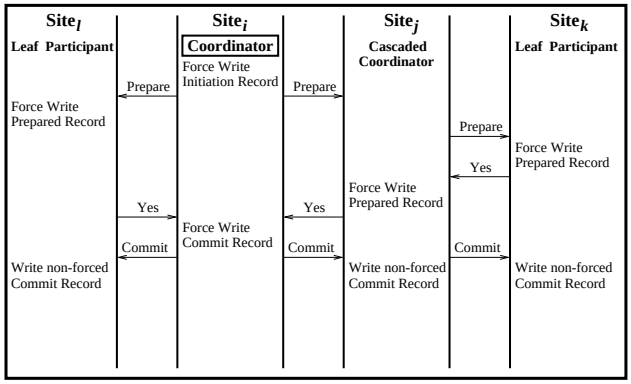
In RPrC, the root coordinator needs to know *all* the participants at all levels in a transaction's execution tree. Similarly, each participant needs to know *all* its ancestors in the transaction's execution tree. The former allows the root coordinator to determine when it can *safely* forget a transaction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

In order for the root coordinator to know the identities of all participants in RPrC, each participant includes its identity in the *acknowledgment* of the *first* operation. When a cascaded coordinator receives an *acknowledgment* of a first operation from a participant, it also includes its identity in the *acknowledgment* message. In this way, the identities of all participants and the chain of their ancestors are propagated to the root coordinator. This technique is similar to the one used with PrA to support *heuristic decisions* [13, 16]. When the transaction submits its commit request, the coordinator, force writes an **initiation** record that includes the identities of all participants in the transaction execution tree. Then, it sends out *prepare to commit* messages to its direct descendants.

The root coordinator sends its identity as part of the *prepare to commit* message (Figure 6). When a cascaded coordinator receives the *prepare to commit* message, it appends its own identity before propagating the message to its direct descendants. When a leaf participant receives a *prepare to commit* message, it copies the identities of its ancestors in the **prepared** log record before sending its *Yes* vote. When a cascaded coordinator receives *Yes* votes from all its direct descendants, the cascaded coordinator also records the identities of its ancestors as well as its descendants in its **prepared** log record before sending its collective *Yes* vote to its direct ancestor.



(a) Abort case.



(b) Commit case.

Figure 6. The rooted presumed commit protocol.

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

If the root coordinator receives a *No* vote, it propagates an *abort* decision to all direct descendants that have voted *Yes* and waits for their *acknowledgments* (Figure 6 (a)), knowing that all the descendants of a direct descendant that has voted *No* have already aborted the transaction. When the coordinator receives the *acknowledgments* of its abort decision, it writes a non-forced **end** record and forgets the transaction. When a cascaded coordinator receives the *abort* message, it behaves as in multi-level PrC. That is, it propagates the message to its direct descendants and writes a forced **abort** record. Then, it *acknowledges* its direct ancestor. Once the cascaded coordinator has received *acknowledgments* from all its direct descendants, it writes a non-forced **end** record and forgets the transaction. When a leaf participant receives the *abort* message, it first force writes an **abort** record and, then, *acknowledges* its direct ancestor.

As in multi-level PrC, when the root coordinator receives *Yes* votes from all its direct descendants, it force writes a **commit** record, propagates its decision to its direct descendants and forgets the transaction. When a cascaded coordinator receives a *commit* message (Figure 6 (b)), it propagates the message to its direct descendants, writes a non-force **commit** record and forgets the transaction. When a leaf participant receives the message, it commits the transaction and writes a non-forced **commit** record.

4.1.2. Failures Considered

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

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

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

In the event that the root coordinator fails, during its recovery process, the root coordinator records in its protocol table each transaction with an **initiation** record without a corresponding **commit** or **end** record. These transactions have not finished their commit processing by the time of the failure and need to be aborted. Thus, for each of these transactions, the coordinator sends an *abort* message to its direct descendants, as recorded in the **initiation** record, along with their lists of descendants in the transaction execution tree.

The recipient of the *abort* message can be either a cascaded coordinator or a leaf participant. In the case of a cascaded coordinator, if it is in a prepared to commit state, the cascaded coordinator behaves as in the case of normal processing discussed above. Otherwise, it responds with a blind acknowledgment, indicating that it has already aborted the transaction. Similarly, if the *abort* message is received by a leaf participant, the participant behaves as in the case of normal processing if it is in a prepared to commit state or replies with a blind acknowledgment.

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

4.2. The Re-Structured PrC Protocol (ReSPrC)

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

The *re-structuring* of a transaction tree has been previously used to enhance the reliability of commit processing by reducing the blocking effects of atomic commit protocols in case of failures [14]. Also, the *flattening* of a distributed transaction's tree has been suggested to reduce the cost of commit processing that is due to the serialization of messages in a transaction's tree [15]. That is, instead of sending the *coordination* messages during commit processing in a sequential fashion from one process at one level to another at the next level in a transaction tree, the flattening technique allows the coordinator of the transaction to send messages directly to the participant processes without having to go through intermediate processes. This technique significantly reduces the cost of commit processing especially in deep trees [15].

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

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

Furthermore, the use of the flattening technique provides a significant performance enhancement in the presence of *loopbacks* [13]. A loopback occurs when a process, for example P_1 at site $Site_1$ creates another process P_2 at $Site_2$, which in turn creates P_3 back at $Site_1$. Assuming P_1 is a coordinator, by using ReSPrC, rather than communicating with P_3 though P_2 located at a different site, the coordinator communicates directly and locally with P_3 without the cost of having to exchange messages with P_3 via an external communication medium.

Although both ReSPrC and RPrC eliminate the **initiation** records of multi-level PrC from cascaded coordinators, ReSPrC is clearly more efficient than RPrC since ReSPrC allows for maximum parallelism during commit processing whereas RPrC suffers from the serialization of messages and forced log writes. However, ReSPrC cannot always be used. ReSPrC cannot be used in an environment where a participant is prohibited to directly communicate with the root coordinator or vice versa for security reasons. In general, ReSPrC also cannot be used when the communication topology does not support direct interaction between a root coordinator and the leaf participants. Similarly, the use of ReSPrC is limited when the establishment of new direct communication channels (i.e., sessions) between the coordinator and the participants are expensive and should be avoided as much as possible. A situation that exists in some commercial systems [9]. On the other hand, RPrC does not suffer from the applicability limitations of ReSPrC even for security reasons. Although RPrC requires the propagation of the

participants' identities through the branches of the trees, by applying some basic encryption techniques to the identities of the participants, RPrC provides sufficient security to prohibit a participant from being able to identify the other participants. For example, if a key-based encryption technique is to be used, each cascaded coordinator in a transaction tree would use a different key to encipher the identity of its direct ancestor before propagating it to its direct descendants. Similarly, a cascaded coordinator enciphers the identities of its direct descendants, using the same key, before propagating them to its direct ancestor.

The flattening technique can also be applied to multi-level PrA resulting in *re-structured PrA* (ReSPrA). When ReSPrC and ReSPrA are applicable, the tradeoff between them is reduced to the tradeoff between PrC and PrA as discussed in Subsection 3.1. Similarly, the relative advantage of RPrC and multi-level PrA is reduced to the relative advantage of PrC and PrA. For instance, to illustrate the latter, consider again N participants of which C are cascaded coordinator and L are leaf participants ($N = L+C$). To commit a transaction in RPrC requires $L+C+2$ (or $N+2$) forced log writes whereas, to abort a transaction requires $2L + 2C + 1$ (or $2N + 1$) forced log writes, which is the same as in PrC. Thus, the decisive factor in selecting one over the other is the cost associated with read-only transactions which, as we show in the next section, can be efficiently handled using the *unsolicited update-vote* optimization.

4.3. The Unsolicited Update-Vote Optimization

Recall that in the traditional read-only optimization (section 2.2), a coordinator determines read-only participants by explicitly polling their votes. To determine which participants are read-only without having to *explicitly* poll their votes, we have proposed the *unsolicited update-vote* optimization (UUV) [4]. In UUV, a coordinator looks at the participants from another perspective. That is, which participants are *update participants*.

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

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

In the case that each participant site employs a *pessimistic* [5] concurrency control protocol that also *avoids cascading abort* [5], such as *strict two-phase locking* [7], the most commercially used protocol, a transaction is guaranteed to be *serializable* and *recoverable* after all its operations have been executed and acknowledged (see [5] for proof). Thus, the coordinator of a transaction is guaranteed that the transaction is serializable and recoverable at each read-only participant after the execution of each read operation. However, in the case that a participant employs an *optimistic* concurrency control protocol, this is not true and the participant has to validate the transaction before acknowledging each read operation as long as it has not already sent an *unsolicited update-vote* as part of a previous operation's *acknowledgment*.

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

4.3.1. UUV with PrC

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

tion is reached faster with fewer coordination messages by using UUV compared to the traditional read-only optimization. Hence, by combining UUV with PrC (similarly with ReSPrC) the cost associated with read-only transactions is cheaper than in PrA combined with the traditional read-only optimization.

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

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

4.3.2. UUV with Multi-Level PrC

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

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

By combining UUV with RPrC, the root coordinator of a read-only transaction also does not force write an **initiation** record. For a partially read-only transaction, since RPrC eliminates intermediate **initiation** records, a read-only participant will suffer from at most a single forced write (i.e., an **initiation** record at the root coordinator). Hence, the cost of commit processing for a committing, partially read-only transaction in RPrC is less than in multi-level PrA considering the saving in the total number of *acknowledgment* messages and the number of forced log writes at the participants. The savings in the number of *acknowledgment* messages and forced log writes are further magnified for update transactions. For example, there are N extra messages and $N-1$ forced log writes in multi-level PrA compared with RPrC for a committing transaction where N is the number participants in the transaction tree excluding the root coordinator[2].

5. Summary

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

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

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

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

In conclusion, this work nullifies the basis for the argument that exclusively favors PrA, i.e., the low cost associated with read-only transactions and transactions in the multi-level transaction execution model, and makes the case that PrC should become part of future protocol standards. In addition, ReSPrC and RPrC as well as the unsolicited update-vote optimization provide sufficiently appealing efficiency characteristics that make them very attractive to be adopted in commercial systems that use a two-phase commit variant that also force writes **initiation** records such as the one's based on IBM SNA LU 6.2 architecture, the de facto standard of the industry [9].

Acknowledgments

We would like to thank George Samaras and the anonymous referees for their helpful comments.

References

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

[2] Y. Al-Houmaily and P. Chrysanthis. Dealing with Incompatible Presumptions of Commit Protocols in Multidatabase Systems. *Proc. of the 11h ACM Annual Symposium on Applied Computing*, pp. 554–560, Feb. 1996.

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

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

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

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

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

[8] J. 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, pp. 393–481, Springer-Verlag, 1978.

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

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

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

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

[13] C. Mohan, K. Britton, A. Citron and G. Samaras. Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols. *Proc. of the 5th Int'l Workshop on High Performance Transaction Systems*, Sept. 1993.

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

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

[16] G. Samaras and S. Nikolopoulos. Algorithmic Techniques Incorporating Heuristic Decisions to Commit Protocols. *Proc. of the 21st Euromicro Conference*, Sept. 1995.

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

[18] J. Stamos and F. Cristian. Coordinator Log Transaction Execution Protocol, *Distributed and Parallel Databases*, 1(4):383–408, 1993.

[19] P. Spiro, A. Joshi and T. Rengarajan. Designing an Optimized Transaction Commit Protocol. *Digital Technical Journal*, 3(1), Winter 1991.