Performance of Five Atomic Commit Protocols in Gigabit-Networked Database Systems

Y. J. Al-Houmaily
Dept. of Computer Programs
Institute of Public Administration
Riyadh 11141, Saudi Arabia

R. Conticello, J. Pike, P. K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

In this paper, based on an empirical simulation study, we report on the performance implications of five atomic commit protocols on transaction throughput in future, wide-area gigabit-networked database systems. In our study, in addition to the three well known two-phase commit (2PC) variants, we considered two single-phase commit protocols, namely, the implicit yes-vote (IYV) and coordinator log. Salient results of our study show that IYV is, in general, better than all the other evaluated protocols during normal processing. Performance enhancements due to a read-only optimization are more pronounced with short transactions. The choice of a 2PC variant has very little impact on performance in the case of long transactions as opposed to short ones, with presumed commit being better than presumed abort in most cases.

1 Introduction

In order to study the performance implications of atomic commit protocols (ACPs) on *transaction throughput* in both the absence and presence of site failures, we have implemented a comprehensive simulator of a distributed database system. With the current advances in network technologies, in the future, database servers will be interconnected via high speed wide-area networks with data transfer rates in the order of gigabits per second. We used our simulator to evaluate five atomic commit protocols with respect to their applicability in these future, gigabit-networked distributed database systems. Specifically, we considered the three well known two-phase commit (2PC) variants, namely, *basic 2PC*, *presumed abort* (PrA) and *presumed commit* (PrC) [9], and two new, single-phase commit protocols, namely, the implicit yes-vote and *coordinator log* protocols which were designed to utilize the bandwidths available in high speed networks to enhance transaction throughput.

We considered the basic 2PC as a baseline in our experimentations together with PrA which is currently part of the standards. The selection of PrC to be part of the evaluation was motivated by the fact that PrC was designed to reduce the cost of committing transactions and the expectation that gigabit-networked database systems will be characterized by high reliability and high probability of transactions being committed rather than aborted. Also, motivated by the fact that the majority of transactions in any general database system are read-only, we considered two read-only optimizations in our study, specifically, the traditional *read-only* [9] and *unsolicited update-vote* [3] optimizations.

In contrast to the other comparative performance evaluations of 2PC variants in local area networks [8, 7], we explicitly model (a) the propagation latency of the communication network, (b) the overhead of the management of the database buffer and of flushing the transaction and protocol execution log records in accordance to the type of *write-ahead logging* (WAL) used (i.e., centralized, distributed or replicated WAL) and (c) the overhead of recovery from site failures.

In the rest of the paper, we first briefly describe the evaluated protocols and optimizations, present our simulation system model and its associated parameters and discuss the results of our study in the absence of failures. The results of the failure experiments are not presented in this paper due to space limitations.

2 Commit Protocols Under Evaluation

In a distributed database system, a distributed transaction accesses data by submitting database operations to its *coordinator* which in turn, forwards them to the appropriate site for execution. Typically, the coordinator of a transaction is the *transaction manager* at the site where the transaction has been initiated. When a transaction finishes its execu-

tion and submits its commit request, its coordinator engages all the participant sites in an ACP, such as the protocols in our evaluation and which we briefly review in this section. (See [11, 4] for a survey of ACPs and 2PC optimizations).

2.1 The Two-Phase Commit Protocol

The basic *two-phase commit* protocol (2PC) [6], as the name implies, consists of two phases, namely a *voting phase* and a *decision phase*. During the voting phase, the coordinator of a distributed transaction requests all the participating sites in the transaction's execution to *prepare to commit* whereas, during the decision phase, the coordinator either decides to commit the transaction if *all* the participants are *prepared to commit* (voted *Yes*), or to abort if any participant has *decided to abort* (voted *No*). If a participant has voted *Yes*, it can neither commit nor abort the transaction until it receives the final decision. When a participant receives the final decision, it complies, releasing the resources held by the transaction, 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.

To cope with failures, the coordinator force-writes a **decision** log record prior to sending out the final decision to the participants. Since a *force-write* ensures that a log record is written into a stable storage that survives system failures, the final decision is not lost if the coordinator fails. Similarly, each participant force-writes a **prepared** record before sending its *Yes* vote and a **decision** record before acknowledging the final decision. When the coordinator completes the protocol, it writes a non-forced **end** record, indicating that the log records pertaining to the transaction can be garbage collected when necessary. When a participant is in doubt about the outcome of a transaction, it sends an *Inquiry* to the coordinator.

2.2 The Presumed Abort Protocol

The *presumed abort* protocol (PrA) [9] is designed to reduce the cost of aborting transactions by interpreting no knowledge about a transaction's outcome as an abort decision. Specifically, in PrA, when a coordinator decides to abort a transaction, it does not force-write the abort decision in its log as in 2PC. Instead, it just sends out *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).

On commit decision, PrA behaves like 2PC.

2.3 The Presumed Commit Protocol

The *presumed commit* protocol (PrC) is designed to reduce the cost of committing transactions [9]. As opposed to PrA, in PrC, coordinators interpret missing information about transactions as commit decisions. To prevent missing information about a transaction to be misinterpreted as a commit after a coordinator failure, a coordinator has to force-write an **initiation** record before sending *prepare to commit* messages.

On a commit decision, the coordinator force writes a **commit** record to logically eliminate the **initiation** record of the transaction, sends out the *commit* decision and discards all information about the transaction. When a participant receives the decision, it writes a non-forced **commit** record and commits the transaction without sending an acknowledgment. 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).

On an abort decision, 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 acknowledgments as in 2PC.

2.4 One-Phase Commit Protocols

The two one-phase commit protocols in our study, *coordinator log* protocol (CL) [12] and *implicit yes-vote* protocol (IYV) [2], share the same basic idea. That is, the (explicit) *voting* phase of 2PC is eliminated by overlapping it with the execution of operations. Assuming *strict two-phase locking* [5], coordinators in both protocols interpret an acknowledgment received from a participant in response to an operation request to mean that the transaction is in a prepared to commit state at the participant. When a participant receives a new operation for execution, the transaction becomes active again at the participant. When a transaction is aborted by a participant, the participant responds to an operation with a *negative*

acknowledgment message that forces the coordinator to abort the transaction.

CL eliminates the need for logging at the participants, and hence forced log writes, by having the coordinators maintain the logs and using *distributed write-ahead logging* (DWAL) with cache management. When a participant executes an operation, the participant propagates any redo and undo log records generated during the execution of the operation to the transaction's coordinator to write them in its log.

To commit a transaction, in CL, the coordinator first force-writes a **commit** log record and then sends out *commit* messages to all participants. When a participant receives a *commit* message, it commits the transaction without sending an acknowledgment as in PrC. On the other hand, when the coordinator decides to abort the transaction, it writes a non-forced abort record and sends out abort messages to all participants. Each *abort* message includes the undo records needed to rollback the transaction. When a participant rolls back the transaction, it sends an acknowledgment that includes the log records generated during the transaction rollback. CL assumes ARIES for recovery [10] in which an operation is undone by executing its inverse operation. When the coordinator receives the acknowledgments, it writes both the attached log records and an **end** record in its log in a non-forced manner and forgets the transaction.

As opposed to CL, IYV supports participant logs but it eliminates the forced **prepared** records at the participants by using *replicated write-ahead logging* (RWAL), i.e., replicating the redo part of its log pertaining to a transaction at the transaction's coordinator site. As in CL, this is achieved by including the redo records generated during the execution of an operation in its acknowledgment.

When all the operations of a transaction are executed and acknowledged, the coordinator commits the transaction. On a commit decision, the coordinator force-writes a **commit** record, sends out the *commit* decision to all the participants and waits for their acknowledgments. When a participant receives a *commit* message, it commits the transaction releasing all its resources, and writes a non-forced **commit** log record. When the **commit** log record is flushed into a stable storage, due to a periodic flushing of the log, the participant acknowledges the commit decision. Once the coordinator receives acknowledgments from all participants, it writes a non-forced **end** log record and forgets the transaction.

On an abort decision, IYV behaves like PrA in which the participants undo the transaction using their local log.

3 Read-Only Optimizations

In this section, we discuss the two read-only optimizations used in our study.

3.1 Traditional Read-Only (TRO)

In TRO [9], when a participant that has executed only read operations on behalf of a transaction is prepared to commit the transaction, it replies with a *Read-Only* vote instead of a *Yes* and forgets about the transaction, releases its resources and writes no log records. A read-only participant is not involved in the decision phase because it does not matter whether the transaction is finally committed or aborted to ensure its atomicity. The *Read-Only* vote is enough to let the coordinator know that the transaction has read consistent data.

If a transaction is *read-only* (i.e., all its participants performed only 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 transaction with respect to logging.

3.2 Unsolicited Update-Vote (UUV)

In UUV [3], a coordinator determines read-only participants without having to *explicitly* poll their votes. When a transaction starts executing, its coordinator marks the transaction as a read-only one as well as all its participants. 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.

When a transaction submits its commit request, its coordinator checks the protocol table to determine which participants have sent an *unsolicited update-vote*. For each such participant, the coordinator knows that it is an update participant and sends to it a *prepare to commit* message during the voting phase. The coordinator excludes the read-only participants from voting by sending a *read-only* message indicating to the participant that the transaction has been terminated. When a read-only participant receives a *read-only* message, it releases all the resources held by the transaction without writing any log records.

A special case of UUV is used with IYV and CL. Since a coordinator in both protocols can determine read-only participants based on whether it has received any log records from them, participants do not have to send *unsolicited update-votes*. Thus, in this

special case, which we call *RO* in this paper, the coordinator sends *read-only* messages to read-only participants before force writing the **commit** record, thereby allowing read-only participants to release resources earlier than their update counterparts.

4 Simulation System

We have implemented our simulator in C/C++ using the CSIM simulation library (by Mesquite Software Inc.) on Linux running on Pentium workstations.

4.1 Simulation System Model

We model our system in a manner similar to other database simulation models (e.g., [1, 7]). Table 1 contains our simulation model parameters.

In our model, a database is a collection of objects that are uniformly distributed across a number of sites without data replication. A data objects is uniquely identified by the tuple $< Site_{id}, O_{id} >$. The number of sites ($NumSites$) and objects ($NumObjs$) are specified as database parameters.

The propagation latency ($PropLatency$) of the network is specified as a resource parameter. Wide-area networks implemented in current technology have a propagation delay 300–500ms, while the WANs of the future are expected to have delays of 150–250ms. We selected a propagation delay of 200ms.

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

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

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

A LM at a site follows the *strict two-phase locking* protocol [5] in granting and releasing of locks at its site. If the LM cannot satisfy a lock request, the requesting transaction is blocked, and deadlock detection is performed. If a deadlock is found, the youngest transaction involved in the deadlock is aborted.

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

At a site, there are one or more disks dedicated to storing data, and separate disks dedicated to storing the logs. The RSM maintains a separate queue for each disk at its site. We assume that a database page is equivalent to a disk block. For log disks, the log buffer may be limited to $LogSize$. When the log buffer reaches $LogSize$, the log buffer must be flushed to disk. We assume that a log record has a size of 1 KB; therefore our parameter of 10,000 pages would require a 10 MB log buffer. Sensitivity analysis has shown us that once the log buffer size becomes greater than 640 KB, there is no longer an impact on throughput performance. The cost of flushing to or reading from disk is represented by the access time ($DiskTime$), and a transfer rate ($DiskTransfTime$) for each page moved to/from the disk. Thus, the cost associated with disk services can be summarized as follows:

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

A DCM at a site is responsible for the management of the data transfer between database cache and data disk(s). A DCM determines if a page needs to be fetched from the data disks based on a *HitRate* parameter. Similarly, a DCM is responsible for locating a slot in the cache to swap the requested database page in the case of a miss. If the page to be replaced in the cache is dirty, the page must first be flushed to disk before it is replaced. However, before flushing the replaced page to disk, the DCM determines if WAL logging needs to be performed based on the $LogFlushRate$.

Database Parameters			
1.	*NumSites*	The number of database sites	8
2.	*NumObjs*	The number of data items per database site	1000
Transaction Parameters			
3.	*ExecPattern*	Execution Model	Sequential
4.	*DistDegree*	Number of participants	3
5.	*ParticipantSize*	Transaction's average access per participant	6 (long), 2 (short)
6.	*ThinkTime*	Think time between database operations	0 msec
7.	*PercRead-OnlyTrx*	percentage of read-only transactions	0 (update), 70%
Site Parameters			
8.	*NumCPUs*	Number of CPUs	1
9.	*NumDisks*	Number of disks	4
10.	*MPL*	Degree of multiprogramming per site	2-14 (long); 5-50, 10-80 (short)
11.	*HitRate*	Buffer pool hit probability	80%
12.	*LogFlushRate*	Log pool flush probability due to WAL	50%
13.	*LogSize*	Maximum log buffer size in pages	10000 pages
Resource Parameters			
14.	*CPU(MESG)*	CPU Time for processing a message	1 msec
15.	*CPU(READ)*	CPU Time for processing a read operation	5 msec
16.	*CPU(WRITE)*	CPU Time for processing a write operation	5 msec
17.	*DiskTime*	Disk access time	20 msec
18.	*DiskTransfTime*	Page transfer time	0.1 msec
19.	*PropLatency*	Propagation time for a message	200 msec
20.	*Timeout*	Message timeout	0 msec

Table 1: Simulation parameters.

4.2 Transaction Execution and Workload

In our system, the execution model of distributed transactions (*ExecPattern*) can be either *sequential*, *participant-sequential* or *parallel*. The sequential execution model is more general than the other two and for this reason we consider only this in this paper. In this model, before a transaction submits an operation, it waits until the previous submitted operation has been executed and acknowledged by the corresponding participant. When a transaction receives the results of an operation, it spends some *ThinkTime* which represents the processing time of the received results before it sends the next operation for execution.

Each site is associated with a multiprogramming level (*MPL*) that is specified as a parameter to the system. MPL is used to limit the number of active transactions at a site at any given time. The simulator is run at full capacity (i.e., peak load). That is, when a transaction terminates, a new transaction enters the system and starts executing at the site where the previous transaction has terminated. For aborted transactions, we use *fake* restarts where an aborted transaction is restarted as an independent transaction after a delay time that is equal to the mean transaction response time.

The trace used with all protocols in a run is generated based on the *ExecPattern* of transactions, the percentage of read-only transactions (*PercRead-OnlyTrx*), the number of sites participating in a transaction's execution, which is specified by the *DistDegree* parameter, the number of data operations that a transaction performs at each participant site, which is uniformly distributed between 0.5 and 1.5 of the *ParticipantSize* parameter. For each run, the simulator executes until 30,000 transactions are committed. The performance curves in all our experiments represent the statistical mean of three independent runs with a confidence half-length interval of no more than 2.7 at the 90% confidence level and no more than 3.5% relative precision (i.e., relative error).

5 Performance of ACPs

In this section, we evaluate the performance of ACPs in the absence of failures, assuming that when a transaction submits its commit request, the transaction will be committed. The parameter settings for these experiments are shown in Table 1. Since 2PC and PrA behave the same in the absence of failures for committing transactions, we show only the performance curves of PrA in our figures.

To isolate the impact of ACPs on the overall system performance, the study also simulates the behavior of the system when *distributed-execution centralized-*

commit (DECC) is used as in [7]. Though artificial (no ACP is used), DECC shows the *highest attainable* system performance allowing us to better relate the performance enhancement of the evaluated ACPs.

We conducted four sets of experiments. The first set E1 focuses on the impact of ACPs on the system's performance in the case of relatively long transactions, while the second set E2 on the impact of relatively short transactions. Given the size of the simulated database, long transactions execute, on average, 6 operations at each participant, while short transactions execute, on average, 2 operations at each participant. Two types of traces were simulated, one composed of update transactions, and the second one of 70% read-only transactions. An update transaction invokes at least one write operation. E1 and E2 included no optimizations. The last two sets of experiments, E3 and E4, focus on the effects of read-only optimizations on the performance of ACPs for long and short, 70% read-only transactions, respectively.

In all experiments we measure system throughput, which is the total number of committed transactions per second, while varying the MPL. As indicated in other studies that use a closed-queuing system model (e.g., [7]), the performance curves of the response time of transactions is the inverse of the system throughput.

E1. Long Transactions: As shown in Fig 1(a), for update transactions, the performance curves of all ACPs start to increase at the beginning, peak at MPL 6 and then start to decline. This *thrashing* behavior of the system is due to the contention of transactions over the data objects as well as system resources and appears in all our experiments as well as other simulation studies [1, 8, 7]. Due to this contention, at high MPLs, transactions tend to abort because of deadlocks, reducing the overall system performance.

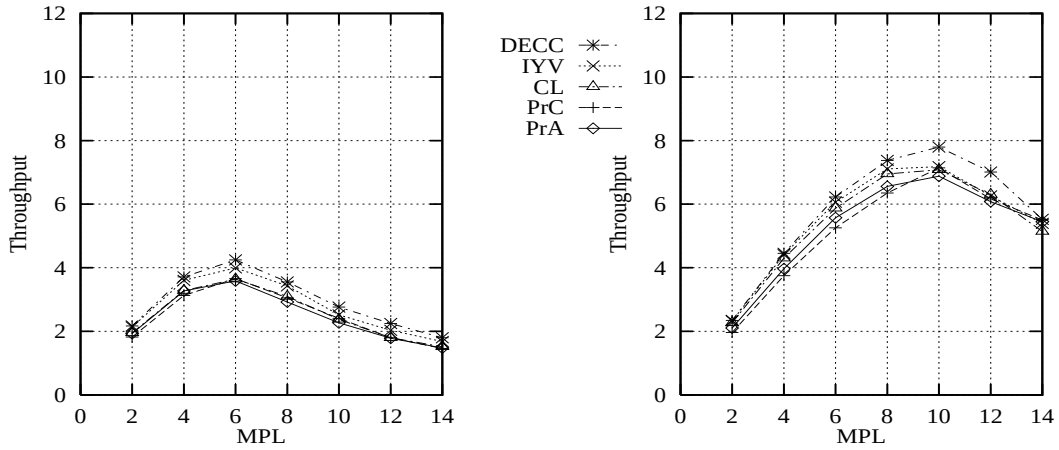
In the case of long transactions, IYV outperforms all other protocols. For update transactions, at the peak MPL 6, DECC outperforms IYV by about 6% whereas the performance difference between DECC, the ideal case, and the worst case (PrA), is 0.7 transactions per second which translates to about 16% performance difference. IYV is better than CL by 8% at the peak MPL whereas it outperforms all the 2PC variants at the peak MPL by about 10%. An interesting observation is that all three 2PC variants support about the same throughput in case of long transactions.

By comparing Figures 1(a) and 1(b), we notice that, when read-only transactions are introduced, the performance of all the evaluated ACPs was enhanced by at least 10%, across all MPLs (which is the case of PrC protocol). and the peak performance point of all protocols shifted from MPL 6 to 10. This is consistent with the fact that transactions do not conflict at the same rate as in the case of update transactions.

Another interesting observation in this and the other experiments is the existence of a cross-over point between the performance curves of PrA and PrC even though all transactions are committed once they reach their commit point. Based on the simple counting of messages and log writes, this point should not exist since PrC always have the least number of coordination messages and forced log writes. Clearly, our experiments reveal that under low system loads, the initiation records of PrC affect its performance and makes it worse than PrA. After a certain point (at higher MPLs), the effects of the forced log writes at the participants in PrA as well as the acknowledgments overshadow the cost of the initiation records of PrC, making PrC performing better than PrA. Our experiments also indicate that the location of this cross-over point depends on the transaction mix, the length of transactions and whether or not a read-only optimization is used.

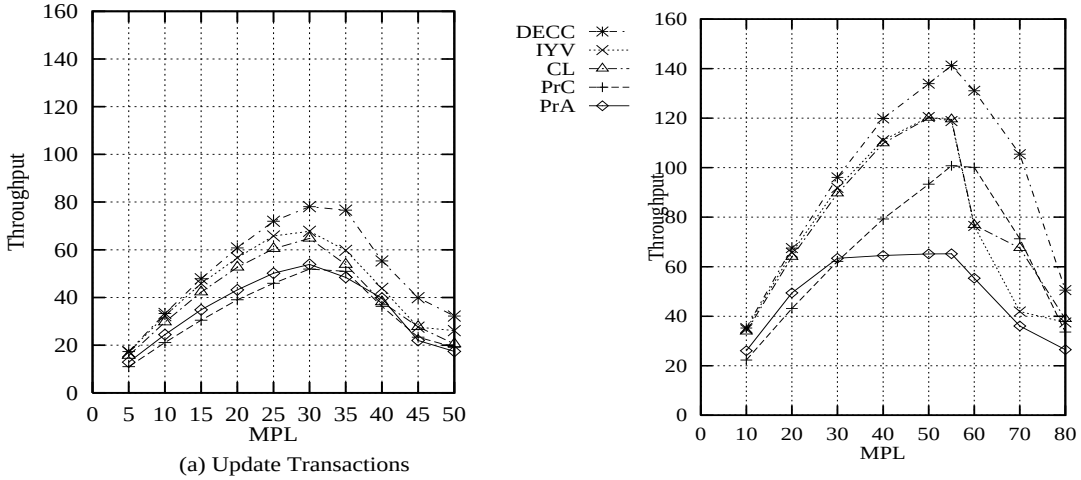
E2. Short Transactions: Compared to long transactions, in the case of short transactions, the overall system performance has significantly enhanced. At the same time, although the relative performance of the different ACPs, with the exception of PrC, has not changed, the actual performance gaps between one-phase and two-phase commit variants have become wider. For example, in Fig. 2(a), DECC outperforms CL by about 20% while it outperforms IYV by about 15%. With respect to PrA and PrC, DECC outperforms PrC by about 50% whereas DECC outperforms PrA by about 45%, a situation which is reversed in the case of predominately short, read-only transactions (Fig. 2(b)) with PrC outperforming PrA by 35%.

CL and IYV are clear winners compared to the three 2PC variants. This result clearly supports the motivation behind the design of CL which assumes short transactions with high probability of being committed once they reach their commit point, and IYV with its reduced message count (compared to CL). However, we note that the performance of CL starts to degrade very quickly after the peak MPLs in the two figures in Fig. 2. This is due to CL's DWAL which forces a participant that aborts a transaction to wait until it receives the undo log records pertaining to the transaction from the coordinator before it can release the locks held by the transaction. In contrast, the other protocols do not suffer from such an overhead since the undo records of an aborting transaction at a



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

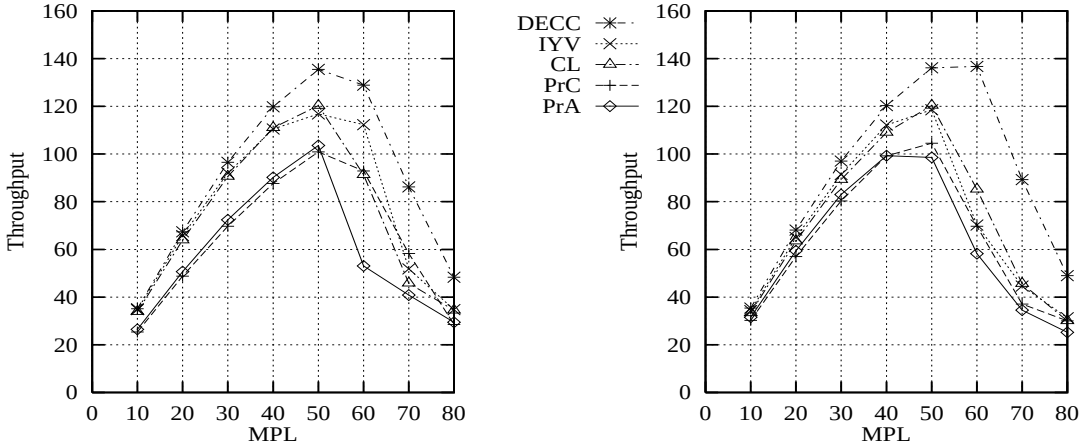
Figure 1: The performance of ACPs for long transactions.



(a) Update Transactions

(b) 70% Read-Only Transactions

Figure 2: The performance of ACPs for short transactions.



(a) Read-Only with TRO Optimizations (b) Read-Only with UUV Optimizations

Figure 3: The performance of ACPs for short transactions with read-only optimizations.

participant are available locally in its own log.

E3. Long Transactions with Read-Only Optimizations: In this experiment, we factored in the effects of TRO, UUV and RO (special case of UUV) optimizations, discussed in Section 3, on the behavior of the evaluated ACPs. IYV and CL have only gained a small overall performance enhancement, yet this is very significant because it has yielded a performance which is 70% closer to the optimal performance possible represented by the DECC protocol. Clearly, the 2PC variants have greatly benefited from TRO and UUV as well reducing the performance gap with IYV at high MPLs from about 10% before to 5%.

E4. Short Transactions with Read-Only Optimizations: As above, IYV and CL combined with UUV exhibit the best performance for short transactions. As we have seen in E2, any extra messages or forced log writes in the case of short transactions, have a significant impact on performance of an ACP, compared to long transactions. Conversely, any reduction in the messages or forced log writes greatly enhances the performance of an ACP in the case of short transactions. Thus, unlike the results of E3, the performance of all protocols has been enhanced with PrA gaining the most and CL the least, as shown in Fig. 3. PrA has gained about 65% performance enhancement using TRO, bringing its performance comparable to PrC, which clearly indicates why PrA combined with TRO is the choice of the standards.

6 Conclusion

In this paper, we evaluated five ACPs for their applicability in gigabit-networked distributed database systems, expressing it in terms of transaction throughput in the absence of failures. Our results confirmed that one-phase commit protocols are a better choice, when applicable, with IYV exhibiting the best overall performance. The exception was CL combined with the read-only optimization performing better than IYV in the case of short, predominately read-only transactions. Our results also showed that, as opposed to IYV, CL is greatly influenced by the transactions' length and the degree of multiprogramming. CL performance degrades rapidly for long transactions or high MPLs. Another interesting result is that, when there is a performance difference between 2PC variants, PrC is generally the winner at peak MPLs. This is especially the case for short transactions. This result is in contrast with the general belief that PrA is better than PrC.

Acknowledgments: This was supported in part by N.S.F. under grants IRI-9210588 and IRI-9502091.

References

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

[2] Al-Houmaily, Y., 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, 1995.

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

[4] Chrysanthis P., G. Samaras and Y. J. Al-Houmaily. Recovery and Performance of Atomic Commit Processing in Distributed Database Systems. *Performance of Database Recovery Mechanism*, V. Kumar and M. Hsu, eds., pp. 370-416, Prentice Hall, 1998.

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

[6] Gray, J. Notes on Data Base Operating Systems. In Bayer R. et al. (Eds), *Operating Systems: An Advanced Course*, *Lecture Notes in Computer Science*, Vol. 60 pp. 393–481, Springer-Verlag, 1978.

[7] Gupta, R., J. Haritsa, and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. *Proc. of the ACM SIGMOD Int'l Conf. on the Management of Data*, pp. 486-496, 1997.

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

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

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

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

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