COMMIT PROCESSING IN DISTRIBUTED REAL-TIME DATABASE SYSTEMS

Ramesh Gupta Jayant Haritsa Krithi Ramamritham[1] S. Seshadri[2]

Technical Report

TR-1996-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

`http://dsl.serc.iisc.ernet.in`

Portions of the work presented in this report have appeared in the following publications:

1. Proceedings of the *National Conference on Software for Real-Time Systems*, Cochin, India, January 1996.
2. Proceedings of the *17th IEEE Real-Time Systems Symposium*, Washington, DC, USA, December 1996.

[1]Dept. of Computer Science, University of Massachusetts, Amherst 01003, USA
[2]Dept. of Computer Science and Engineering, Indian Institute of Technology, Bombay 400076, India

Abstract

Incorporating distributed data into the real-time framework incurs the well-known additional complexities that are associated with transaction concurrency control and database recovery in distributed database systems. We investigate here the performance implications of supporting transaction atomicity in a distributed real-time database system. Using a detailed simulation model of a firm-deadline distributed real-time database system, we profile the real-time performance of a representative set of commit protocols. A new commit protocol that is designed for the real-time domain and allows transactions to "optimistically" read uncommitted data is also proposed and evaluated.

The experimental results show that data distribution has a significant influence on the real-time performance and that the choice of commit protocol clearly affects the magnitude of this influence. Among the protocols evaluated, the new optimistic commit protocol provides the best performance for a variety of workloads and system configurations.

1 Introduction

A **Real-Time Database System (RTDBS)** is a transaction processing system that is designed to handle workloads where transactions have completion deadlines. The objective of the system is to meet these deadlines, that is, to process transactions before their deadlines expire. Research on real-time database systems has been underway for close to a decade now (see [Ulu94a] for a recent survey). So far, this research has focused primarily on *centralized* database systems. However, many real-time database applications, especially in the areas of communication systems and military systems, are inherently *distributed* in nature [Son90, Ulu94a]. For example, a mobile telecommunication system that is currently being built is described in [Yoo94] – this system has data distributed across several sites and has real-time transaction processing requirements. Motivated by such applications, efforts have begun to design distributed real-time database systems.

Incorporating distributed data into the real-time database framework incurs the well-known additional complexities that are associated with transaction concurrency control and database recovery in distributed database systems [BHG87, OV91]. While the issue of distributed real-time concurrency control has been considered to some extent [SRL88, SK92, UB92, Ulu94b], comparatively little work has been done with regard to distributed real-time database recovery. We investigate here the performance implications of supporting transaction atomicity in a distributed real-time database system. To the best of our knowledge, this is the first quantitative evaluation of this issue.

Commit Protocols

Distributed database systems implement a transaction *commit protocol* to ensure transaction atomicity. A commit protocol guarantees the uniform commitment of distributed transaction execution, that is, it ensures that all the participating sites agree on the final outcome (commit or abort). Most importantly, this guarantee is valid even in the presence of site or network failures.

Over the last two decades, a variety of commit protocols have been proposed by database researchers (see [Koh81, Bha87, OV91] for surveys). These include the classical *two phase commit (2PC)* protocol [LS76, Gra78], its variations such as *presumed commit* [MLO86, LL93] and *presumed abort* [MLO86], *nested 2PC* [Gra78], *broadcast 2PC* [OV91] and *three phase commit* [Ske81]. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately. Due to these costs, commit processing can result in a significant increase in transaction execution times [SJR91, SBCM93, LL93], and therefore the choice of commit protocol becomes an important decision in the design of a distributed database system.

Performance Issues

A list of desirable features of commit protocols is given in [MLO86]. From a performance perspective, they broadly correspond to evaluating commit protocols based on the following three issues:

Effect on Normal Processing: This refers to the extent to which incorporating commit protocols affects the normal distributed transaction processing performance. That is, how expensive is it to provide atomicity?

Resilience to Failures : When a failure occurs in a distributed system, ideally transaction processing at the remaining sites should not be affected pending recovery of the failed site. With respect to distributed commit specifically, a commit protocol is said to be *nonblocking* if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site [OV91]. Unfortunately, it is impossible to design commit protocols that are completely immune, in the non-blocking sense, to both site and link failures [BHG87]. However, the number of simultaneous failures that can be tolerated before blocking arises varies across different commit protocols.

Speed of Recovery: This refers to the time required for the database to be recovered when the failed component comes back up after a crash.

With respect to the above issues, the relative performance of a representative set of commit protocols have been analyzed to some extent in [Coo82, MLO86, LL93]. These studies were conducted in the context of a conventional DBMS where transaction throughput or response time is the primary performance metric. In a real-time database system, however, performance is usually measured in terms of the *number* of transactions that complete before their deadlines. That is, a transaction that completes just before its deadline is no different, from a performance perspective, to one that finishes much earlier. Due to the difference in objectives, the performance of commit protocols has to be reevaluated for the real-time environment.

Real-Time Commit Processing

In the real-time domain, there are two major questions that need to be explored: First, how do we adapt the commit protocols to the real-time domain? Second, how do the real-time variants of the commit protocols compare in their performance? In this paper, we address these questions for the "firm-deadline" [HCL92] application framework, wherein transactions that miss their deadlines are considered to be worthless and are immediately "killed", that is, discarded from the system without being executed to completion. Using a detailed simulation model of a distributed database system, we compare the real-time performance of a representative set of commit protocols. The performance metric is the steady-state percentage of transaction deadlines that are missed.

Of the three performance issues highlighted above, we believe, from a real-time perspective, that the first two issues (effect on normal processing and resilience to failures) are of primary importance since they directly affect the ability of the database system to meet transaction

deadlines. In comparison, the last issue (speed of recovery) appears less critical for two reasons: First, failure durations are usually orders of magnitude larger than recovery times. Second, failures are usually rare enough that we do not expect to see a difference in *average* performance among the algorithms because of one commit protocol having a faster recovery time than the other. With this viewpoint, we focus here on the *mechanisms* required during normal operation to provide the functionality of recovery, rather than on the recovery *process* itself.

Organization

The remainder of this paper is organized as follows: Related work on real-time commit protocols is reviewed in Section 2. The commit protocols evaluated in our study are described in Section 3, and the real-time aspects of commit processing are discussed in Section 4. The performance model is described in Section 5, and the results of the simulation experiments are highlighted in Section 6. Finally, in Section 7, we present the conclusions of our study and identify future research avenues.

2 Related Work

The design of real-time commit protocols has been investigated earlier in [DLW89, SLKS92, Yoo94]. The paper [DLW89] describes a *centralized timed two-phase commit protocol* where the fate of a transaction (commit or abort) is guaranteed to be known to all the participants of the transaction by a deadline, when there are no processor, communication, or clock faults. In case of faults, however, it is not possible to provide such guarantees, and an *exception state* is allowed which indicates the violation of the deadline. Now, even if the coordinator of a transaction is able to reach the decision by the deadline, but it is not possible to convey the decision to all the participants by the deadline, the transaction is killed. Thus the primary concern in the paper is to ensure that all the participants of a transaction reach the decision before the expiry of the deadline, even at the cost of more transactions eventually being killed. In our work, we focus instead on improving the number of transactions that complete before their deadlines expire, which is of primary concern in the "firm-deadline" application framework (see Section 4.4).

The papers [SLKS92, Yoo94] are based on a common theme of allowing individual sites to *unilaterally* commit – the idea is that unilateral commitment results in greater timeliness of actions. If it is later found that the decision is not consistent globally, "compensation" transactions are used to rectify the errors.

While the compensation-based approach certainly appears to have the potential to improve timeliness, yet there are quite a few practical difficulties. First, the standard notion of transaction atomicity is not supported – instead, a "relaxed" notion of atomicity [KLS90, LKS91a, LKS91b] is provided. Second, the design of a compensating transaction is an application-specific task since it is based on the application semantics. Third, the compensation transactions need to be designed in advance so that they can be executed as soon as errors are detected – this means that the transaction workload must be fully characterized *a priori*. Fourth, "real actions" [Gra81] such as firing a weapon or dispensing cash may not be compensatable at all [LKS91a]. Finally, no performance studies are available to evaluate the effectiveness of this approach.

From a performance viewpoint also, there are some difficulties: First, the execution of compensation transactions is itself an additional burden on the system. Second, it is not clear as to how the database system should schedule compensation transactions relative to normal transactions. Finally, no performance studies are available to evaluate the effectiveness of this approach.

Due to the above limitations of the compensation-based approach, we have in our research focused on improving the real-time performance of the mechanisms for maintaining distributed transaction atomicity *without* relaxing the *standard* database correctness criterion.

3 Distributed Commit Protocols

A common model of a distributed transaction is that there is one process, called the *master*, which is executed at the site where the transaction is submitted, and a set of other processes, called *cohorts*, which execute on behalf of the transaction at the other sites that are accessed by the transaction. For this model, a variety of transaction commit protocols have been devised, most of which are based on the classical **two phase commit (2PC)** protocol [LS76, Gra78]. In this section, we briefly describe the 2PC protocol and a few popular variations of this protocol – complete descriptions are available in [MLO86, Ske81].

3.1 Two Phase Commit Protocol

The two-phase commit protocol, as suggested by its name, operates in two phases: In the first phase, the master reaches a global decision (*commit* or *abort*) based on the local decisions of the cohorts. In the second phase, the master conveys this decision to the cohorts. For its successful execution, the protocol assumes that each cohort of a transaction is able to *provisionally* perform the actions of the transaction in such a way that they can be undone if the transaction is aborted. This is usually implemented by using *logging* mechanisms [Gra78], which maintain sequential histories of transaction actions in stable storage. The protocol also assumes that, if necessary, log records can be *force-written*, that is, written synchronously to stable storage.

The master initiates the first phase of the protocol by sending *PREPARE* messages in parallel to all the cohorts. Each cohort that is ready to commit first force-writes a *prepare* log record to its local stable storage and then sends a *YES VOTE* to the master. It then waits for the final decision from the master. At this stage, the cohort has entered a *prepared* state wherein it cannot unilaterally commit or abort the transaction. On the other hand, each cohort that decides to abort the transaction force-writes an *abort* log record and sends a *NO VOTE* to the master. Since a *NO VOTE* acts like a veto, the cohort is permitted to unilaterally abort the transaction without waiting for a response from the master.

After the master receives the votes from all the cohorts, it initiates the second phase of the protocol. If all the votes were *YES*, then it moves to a *committing* state by force-writing a *commit* log record and sending *COMMIT* messages to all the cohorts. Each cohort after receiving a *COMMIT* message moves to the *committing* state, force-writes a *commit* log record, and sends an *ACK* message to the master. If the master receives even one *NO VOTE*, it moves to the *aborting* state by force-writing an *abort* log record and sends *ABORT* messages to those cohorts

that are in the prepared state. These cohorts, after receiving the *ABORT* message, move to the *aborting* state, force-write an *abort* log record and send an *ACK* message to the master.

Finally, the master, after receiving acknowledgements from all the "prepared cohorts" (that were sent a message in the second phase), writes an *end* log record and then "forgets" the transaction.

3.2 Presumed Abort

As described above, the 2PC protocol requires transmission of several messages and writing or force-writing of several log records. A variant of the 2PC protocol, called **presumed abort (PA)** [MLO86], tries to reduce this overhead by requiring all participants to follow a "in case of doubt, abort" rule. That is, if after coming up from a failure a site queries the master about the final outcome of a transaction and finds no information available with the master, the transaction is assumed to have been aborted. With this assumption, it is not necessary for cohorts to either send acknowledgements for *ABORT* messages from the master or to force-write the *abort* record to the log. It is also not necessary for the master to force-write the *abort* log record or to write an *end* log record after abort.

In short, the PA protocol behaves identically to 2PC for committing transactions, but has reduced message and logging overheads for aborted transactions.

3.3 Presumed Commit

A variation of the presumed abort protocol is based on the observation that, in general, the number of committed transactions is much more than the number of aborted transactions. In this variation, called **presumed commit (PC)** [MLO86], the overhead is reduced for committing transactions rather than aborted transactions by requiring all participants to follow a "in case of doubt, commit" rule. In this scheme, cohorts do not send acknowledgements for the *commit* global decision, and do not force-write the *commit* log record. In addition, the master does not write an *end* log record. However this protocol requires the master to force-write a *collecting* log record before initiating the two-phase protocol. This log record contains the names of all the cohorts involved in executing that transaction.

The above optimizations of 2PC have been implemented in a number of commercial products and are now part of transaction processing standards.

3.4 Three Phase Commit

A fundamental problem with all the above protocols is that cohorts may become *blocked* in the event of a site failure and remain blocked until the failed site recovers. For example, if the master fails after initiating the protocol but before conveying the decision to its cohorts, these cohorts will become blocked and remain so until the master recovers and informs them of the final decision. During the blocked period, the cohorts may continue to hold system resources such as locks on data items, making these unavailable to other transactions, which in turn become blocked waiting for the resources to be relinquished. It is easy to see that, if the blocked period is long, it may result in major disruption of transaction processing activity.

To address the blocking problem, a **three phase commit (3PC)** protocol was proposed in [Ske81]. This protocol achieves a nonblocking capability by inserting an extra phase, called the "precommit phase" in between the two phases of the 2PC protocol. In the precommit phase, a preliminary decision is reached regarding the fate of the transaction. The information made available to the participating sites as a result of this preliminary decision allows a global decision to be made despite a subsequent failure of the master. Note, however, that the nonblocking functionality is obtained at an increase in the communication overhead since there is an extra round of message exchange between the master and the cohorts. In addition, both the master and the cohorts have to force-write additional log records in the precommit phase.

4 Real-Time Commit Processing

The commit protocols described in the previous section were designed for conventional database systems where transaction throughput or response time is the primary performance metric. In real-time database systems, however, satisfaction of transaction timing constraints is the primary goal. Therefore, the scheduling policies at the various resources (both physical and logical) in the system can be reasonably expected to be priority-driven with the priority assignment scheme being tuned to minimize the number of missed deadlines. The commit protocols described above do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [SRL87]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable.

Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priority. At the CPU, for example, a scheduling policy such as Priority Pre-emptive Resume ensures the absence of priority inversion. For a non-preemptable resource such as the disk, although it is not possible to completely eliminate priority inversion due to physical device restrictions, at least it is possible to *bound* the inversion period by ensuring that the wait queue for the device is processed in priority order. Even at the network, it is easy to ensure that messages of high priority transactions are transmitted before those of other lower priority requesters. With respect to data also, priority-based pre-emptive concurrency control algorithms such as 2PL-HP [AG88] and OPT-WAIT [HCL90] have been developed.

Removing priority inversion in the commit protocol, however, is *not* fully feasible. This is because, once a cohort reaches the *PREPARED* state, it has to retain all its data locks until it receives the global decision from the master – this retention is fundamentally necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a "prepared cohort" of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a distributed real-time database system is *inherently* susceptible to priority inversion. More importantly, the priority inversion is *not bounded* since the time duration that a cohort is in the *PREPARED* state can be arbitrarily long (due to network delays). If the inversion period is large, it may have a significant effect on performance. To address this issue, we have designed a modified version of the 2PC protocol, described below.

4.1 Optimistic Commit Protocol

In our modified protocol, transactions waiting for data items held by lower priority transactions in prepared state are allowed to access the data. That is, prepared cohorts *lend* uncommitted data to higher priority transactions. In this context, two situations may arise:

Lender Receives Decision First : Here, the lending cohort receives its global decision before the borrowing cohort has completed its execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. If the global decision is to abort, then the lender is aborted in the normal fashion. In addition, the borrower is also aborted since it has utilized inconsistent data.

Borrower Completes Execution First : Here, the borrowing cohort completes its execution before the lending cohort has reached its global decision. The borrower is now "put on the shelf", that is, it is made to wait and not allowed to send a *YES VOTE* in response to its master's *PREPARE* message. The borrower has to wait until such time as the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, then the borrower is "taken off the shelf" and allowed to respond to its master's messages. However, if the lender aborts, then the borrower is also aborted immediately since it has read inconsistent data. In the latter case (borrower's deadline expires while waiting), the borrower is killed in the normal manner.

In summary, the protocol allows transactions to read uncommitted data held by lower priority prepared transactions in the "optimistic" belief that this data will eventually be committed[1]. In the remainder of this paper, we refer to this protocol as **OPT**.

The primary motivation, as described above, for permitting access to uncommitted data was to reduce priority inversion. However, if we believe that lender transactions will typically commit, then this idea can be carried further to allowing *all* transactions, including low priority transactions, to borrow uncommitted data. This may further help in improving the real-time performance since the waiting period of transactions is reduced, and is therefore incorporated in OPT.

4.2 Additional Features of OPT

The following features have also been included in the OPT protocol since we expect them to improve its real-time performance:

Active Abort : In the basic 2PC protocol, cohorts are passive in that they inform the master of their status only upon explicit request by the master. In a conventional distributed DBMS, after a cohort has finished and acknowledged the work assigned to it by the master, it can not be aborted due to local data conflicts or serializability violations (assuming a locking-based concurrency control mechanism). However, in a distributed RTDBS, even such a cohort can be aborted (provided the cohort is not already in the *PREPARED* state) due

[1] A similar, but unrelated, strategy of allowing access to uncommitted data has also been used to improve real-time concurrency control performance [Bes94].

to higher priority conflicting transactions. Therefore, in a real-time situation, it may be better for an aborting cohort to immediately inform the master so that the abort at the other sites can be done earlier. Hence, cohorts in OPT inform the master as soon as they decide to abort locally.

Silent Kill : For a transaction "kill", that is, an abort that occurs due to missing the deadline, there is no need for the master to invoke the abort protocol since the cohorts of the transaction can independently realize the missing of the deadline (assuming global clock synchronization). Therefore, in OPT, aborts due to deadline misses – before a cohort enters the prepared state – are done "silently" without requiring any communication between the master and the cohorts.

Presumed Abort/Commit : The optimizations of Presumed Commit or Presumed Abort discussed earlier for 2PC can also be used in conjunction with OPT to reduce the protocol overheads. We consider both options in our experiments.

4.3 Cascading Aborts

An important point to note here is that the policy of using uncommitted data is generally *not* recommended in database systems since this can potentially lead to the well-known problem of *cascading aborts* [BHG87] if the transaction whose dirty data has been accessed is later aborted. However, in our situation, this problem is alleviated due to following two reasons:

1. The lending cohort is in prepared state and cannot be aborted due to local data conflicts (we assume a locking-based concurrency control mechanism). Therefore, if the lending transaction finally decides to abort, it does so because of the following reasons (in addition to consistency violations): (a) the master has already sent the *PREPARE* messages, but the deadline expires before it is able to reach a decision (see Section 4.4 for "firm-deadline" semantics), or (b) any of the siblings of the cohort in prepared state votes *NO*. Note that when a cohort is aborted, it immediately sends an *ABORT* message to master (Active Abort policy), and the master, after receiving an *ABORT* message from a cohort never sends the *PREPARE* messages to other cohorts. Therefore, the situations in which a sibling of the prepared cohort could vote *NO* can be the following: (i) the *ABORT* message sent by the sibling cohort was in *transit* when master dispatched the *PREPARE* messages, or (ii) the *PREPARE* message sent by the master to the sibling cohort was in *transit* when it was aborted. As the time during which a message is in *transit* is small, the above situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.

2. Even if the lender does eventually abort, it only results in the abort of the immediate borrower(s) and does not cascade beyond this point (since borrowers are not in the prepared state which is the only situation in which uncommitted data can be accessed).

In short, the abort chain is bounded and is of length one (of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the abort chain for each borrower is bounded as described above).

4.4 Semantics of Firm Deadline

In an ideal distributed RTDBS, the master and all the cohorts of a transaction should agree on commit or abort before the deadline expires. However, in practical distributed RTDBS, it is impossible to provide such guarantees because of the arbitrary message delays and the fact that the system is failure prone [DLW89]. Therefore, to avoid the inconsistencies in such cases, in a firm deadline system, a transaction is said to be committed if the master has reached the commit decision (that is, forced the commit log record to the disk) before the expiry of the deadline irrespective of whether the cohorts can receive the decision by the deadline. Note that once the master has reached the commit decision, this decision eventually will be conveyed to the cohorts even if the deadline has expired in the process, only master must reach the decision before the expiry of the deadline. If a cohort eventually receives the final decision after the expiry of the deadline, all that happens is that access to data in prepared state is prevented even beyond the deadline until the decision is received by the cohort and other transactions which would normally expect the data to be released by the deadline only experience a delay. Using OPT protocol solves this problem also to a considerable extent as the data in the prepared state is made available to other transactions. If the cohort in the prepared state finally receives a commit decision, only those transactions whose deadline expired before the cohort received the decision are affected (who perhaps otherwise also would have been killed, as they were close to their deadline and yet in the execution phase). On the other hand, if the cohort in prepared state finally receives an abort decision, all that happens is that the restarts of the transactions who had accessed uncommitted data is delayed, which otherwise would have occurred at the deadline expiry of the prepared cohort. Typically the master is responsible for returning the results of a transaction to the invoker of the transaction. From this discussion, it is clear that such results will be output before the deadline if a transaction commits.

5 Simulation Model

To evaluate the performance of the various commit protocols described in the previous sections, we developed a detailed simulation model of a distributed real-time database system. In the following subsection, we describe various components of the simulation model. Subsequently, the execution pattern of a typical transaction is described. A summary of the parameters used in the model are given in Table 1.

5.1 The Model

Our model is based on a loose combination of the distributed database model presented in [CL88] and the real-time processing model of [HCL92]. The model consists of a database that is distributed, in a non-replicated manner, over all the sites connected by a network. Each site has six components: a *source* which generates transactions; a *transaction manager* which models the execution behavior of the transaction; a *concurrency control manager* which implements the concurrency control algorithm; a *resource manager* which models the physical resources; a *recovery manager* which implements the details of commit protocols; and a *sink* which collects statistics

Table 1: Simulation Model Parameters

$NumSites$	Number of sites in the database
$DBSize$	Number of pages in the database
$ArrivalRate$	Transaction arrival rate / site
$SlackFactor$	Slack Factor in Deadline formula
$DistDegree$	Degree of Distribution
$CohortSize$	Avg. cohort size (in pages)
$WriteProb$	Page update probability
$NumCPUs$	Number of CPUs per site
$NumDisks$	Number of disks per site
$PageCPU$	CPU page processing time
$PageDisk$	Disk page access time
$MsgCPU$	Message send / receive time

on the completed transactions. A *network manager* models the behavior of the communications network.

The database is modeled as a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. At each site, transactions arrive in an independent Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline. The deadline is assigned using the formula $D_T = A_T + SF * R_T$, where D_T, A_T and R_T are the deadline, arrival time and resource time, respectively, of transaction T, while SF is a slack factor. The resource time is the total service time at the resources that the transaction requires for its execution (since the resource time is a function of the number of messages and the number of forced-writes, which differ from one commit protocol to another, we compute the resource time assuming execution in a *centralized* system). The $SlackFactor$ parameter is a constant that provides control over the tightness/slackness of transaction deadlines.

The physical resources at each site consist of $NumCPUs$ CPUs and $NumDisks$ disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively.

The communication network is simply modeled as a switch that routes messages since we assume a local area network that has high bandwidth. However, the CPU overhead of message transfer is taken into account at both the sending and the receiving sites. This means that there are two classes of CPU requests – local data processing requests and message processing requests. We do not make any distinction, however, between these different types of requests and only ensure that all requests are served in priority order. The CPU overhead for message transfers is captured by the $MsgCPU$ parameter.

5.2 Execution Model

Each transaction in the workload has the "single master – multiple cohort" structure described in Section 3. The number of sites at which each transaction executes is specified by the $DistDegree$ parameter. The master and one cohort reside at the site where the transaction is submitted whereas the remaining $DistDegree - 1$ cohorts are set up at sites chosen at random from the remaining $NumSites - 1$ sites. The cohorts execute one after another in a sequential fashion. At each of the execution sites, the number of pages accessed by the transaction's cohort varies uniformly between 0.5 and 1.5 times $CohortSize$. These pages are chosen randomly from among the database pages located at that site. A page that is read is updated with probability $WriteProb$. A transaction that is restarted due to a data conflict makes the same data accesses as its original incarnation that is we do not use fake restarts.

A read access involves a concurrency control request to get access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. We do not explicitly model concurrency control cost parameters in our model since we assume that the overhead of performing concurrency control is small compared to data processing times. Write requests are handled similar to read requests except for their disk I/O – the writing of the data pages takes place asynchronously after the transaction has committed. We assume sufficient buffer space to allow the retention of updates until commit time.

The commit protocol is initiated when the transaction has completed its data processing. If the transaction's deadline expires either before this point, or before the master has written the global decision log record, the transaction is killed (the precise semantics of firm deadlines in a distributed environment are defined in Section 4.4). If the master has written the commit decision log record before the expiry of the deadline, the transaction is said to be committed.

5.3 Logging

We do not explicitly model the logging associated with transaction execution since the logging overhead during this stage is identical across all the commit protocols. Further, in the commit process, we explicitly model only *forced* log writes since they are done synchronously and suspend transaction operation until their completion.

5.4 Priority Assignment

As mentioned earlier, transactions in a real-time database system are typically assigned priorities in order to minimize the number of missed deadlines. In our model, all the cohorts of a transaction inherit the parent transaction's priority. Further, this priority, which is assigned at arrival time, is maintained throughout the course of the transaction's existence in the system (including the commit processing stage, if any). The processing of messages sent on behalf of a transaction also occurs at the priority of the transaction.

6 Experiments and Results

In this section, we present the performance results from our simulation experiments comparing the various commit protocols in a firm-deadline real-time database system environment. The simulator used to obtain the results was written in C++SIM [LM94], an object-oriented simulation testbed. The transaction priority assignment in all of the experiments described here is the widely-used *Earliest Deadline* – transactions with earlier deadlines have higher priority than transactions with later deadlines. The 2PL High Priority scheme [AG88] is used for concurrency control of conflicting transactions.

6.1 Performance Metric

The performance metric of our experiments is $MissPercent$, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. MissPercent values in the range of 0 to 20 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 20 percent to 100 percent represent system performance under "heavy" loads. A long-term operating region where the miss percentage is high is obviously unrealistic for a viable distributed RTDBS. Exercising the system to high miss levels (as in our experiments), however, provides valuable information on the response of the algorithms to brief periods of stress loading. All the MissPercent graphs of this paper show mean values that have relative half-widths about the mean of less than 10 percent at the 90 percent confidence level, with each experiment having been run until at least 20000 transactions were processed by the system. Only statistically significant differences are discussed here.

The simulator was instrumented to generate a host of other statistical information, including resource utilizations, number of transaction restarts, number of forced log writes per transaction, etc. These secondary measures help to explain the MissPercent behavior of the commit protocols under various workloads and system conditions.

6.2 Parameter Settings

The resource parameter settings are such that the CPU time to process a page is 10 milliseconds while disk access times are 20 milliseconds. For experiments that were intended to factor in the effect of resource contention on the performance of the algorithms, the number of processors and disks (at each site) were set to 2 and 4, respectively. For experiments that were intended to isolate the effects of data contention, an "infinite resource" situation [ACL87], where there is no queueing for resources, was simulated. A point to note here is that while abundant resources are usually not to be expected in conventional database systems, they may be more common in RTDBS environments since many real-time systems are sized to handle transient heavy loading. This directly relates to the application-domain of RTDBSs, where functionality, rather than cost, is usually the driving consideration.

Table 2: Baseline Parameter Settings

$NumSites$	8	$NumCPUs$	2
$DBSize$	2400 pages	$NumDisks$	4
$SlackFactor$	4.0	$PageCPU$	10 ms
$DistDegree$	3	$PageDisk$	20 ms
$CohortSize$	6 pages	$MsgCPU$	10 ms
$WriteProb$	0.5	–	–

6.3 Comparative Protocols

To help isolate and understand the effects of distribution and atomicity on Miss Percent performance, and to serve as a basis for comparison, we have also simulated the performance achievable for two additional scenarios. These scenarios are:

Centralized System : Here, we simulate the performance that would be achieved in a *centralized* database system that is equivalent (in terms of database size and resources) to the distributed database system. In a centralized system, messages are obviously not required and commit processing only requires force-writing a single decision log record. Modeling this scenario helps to isolate the effect of distribution on MissPercent performance.

Distributed Processing, Centralized Commit : Here, data processing is executed in the normal distributed fashion but the commit processing is like that of a centralized system, requiring only the force-writing of the decision log record at the master. While this system is clearly artificial, modeling it helps to isolate the effect of distributed commit processing on MissPercent performance (as opposed to the centralized scenario which isolates the entire effect of distributed processing).

In the following experiments, we will refer to the performance achievable under the above two scenarios as **CENT** and **DPCC**, respectively.

6.4 Expt. 1: Baseline Experiment

We began our performance evaluation by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. The settings of the workload parameters and system parameters for the baseline experiment are listed in Table 2. These settings were chosen to ensure significant data and resource contention in the system, thus helping to bring out the performance differences between the various commit protocols, without having to generate very high transaction arrival rates.

For the baseline experiment, Figures 1a and 1b show the MissPercent behavior under normal load and heavy load conditions, respectively. In these graphs, we first observe that there is considerable difference between centralized performance (CENT) and the performance of the standard commit protocols throughout the loading range. For example, at an arrival rate of 2

transactions per second at each site, the centralized system misses virtually no deadlines whereas 2PC and 3PC miss in excess of 30 percent of the deadlines. This difference highlights the extent to which a conventional implementation of distributed processing can affect real-time performance.

Moving on to the relative performance of 2PC and 3PC, we observe that there is a noticeable but not large difference between their performance at normal loads. The difference arises from the additional message and logging overheads involved in 3PC. Under heavy loads, however, the performance of 2PC and 3PC is virtually identical. This is explained as follows: Although their commit processing is different, the *abort* processing of 3PC is identical to that of 2PC. Therefore, under heavy loads, when a large fraction of the transactions wind up being killed (aborted) the performance of both protocols is essentially the same. Since their performance difference is not really large for normal loads also, it means that, in the real-time domain, the price paid during normal processing to purchase the nonblocking functionality is comparatively modest.

Shifting our focus to the PA and PC variants of the 2PC protocol, we find that their performance is only marginally different to that of 2PC. This means that although these optimizations are expected to perform considerably better than basic 2PC in the conventional DBMS environment, these expectations do not carry over to the RTDBS environment. The reason for this is that performance in an RTDBS is measured in *boolean* terms of meeting or missing the deadline. So, although PA and PC reduce overheads under abort and commit conditions, respectively, all that happens is that the resources released by this reduction only allow executing transactions to execute further before being restarted or killed but is not sufficient to result in many more *completions*. This was confirmed by measuring the number of forced writes and the number of acknowledgements, on a per transaction basis, shown in Figures 1d and 1e. In these figures we see that PA has significantly lower overheads at heavy loads (when aborts are more) and PC has significantly lower overheads at normal loads (when commits are more). Moreover, while PA always does slightly better than 2PC, PC actually does worse than 2PC at heavy loads since PC has higher overheads than 2PC for aborts.

Finally, turning our attention to the new protocol, OPT, we observe that its performance is considerably better than that of the standard algorithms over most of the loading range and especially so at normal loads. An analysis of its improvement showed that it arises primarily from the optimistic access of uncommitted data which allow transactions to progress faster through the system, and from the active abort policy. The silent kill optimization (not sending abort messages for aborts arising out of deadline misses), however, gives only a very minor improvement in performance. At low loads, this is because the number of deadline misses are few and the optimization does not come into play; at high loads, the optimization's effect is like that of PA and PC for the standard 2PC protocol – although there is a significant reduction in the number of messages, the resources released by this reduction only allow transactions to proceed further before being restarted but does not result in many more completions. This was confirmed by measuring the number of pages that were processed at the CPU – it was significantly more when silent kill was included.

As part of this experiment, we wanted to quantify the degree to which the OPT protocol's optimism about accessing uncommitted data was well-founded – that is, is OPT safe or foolhardy? To evaluate this, we measured the "success ratio", that is, the fraction of times that a borrowing was successful in that the lender committed after loaning the data. This statistic is shown in

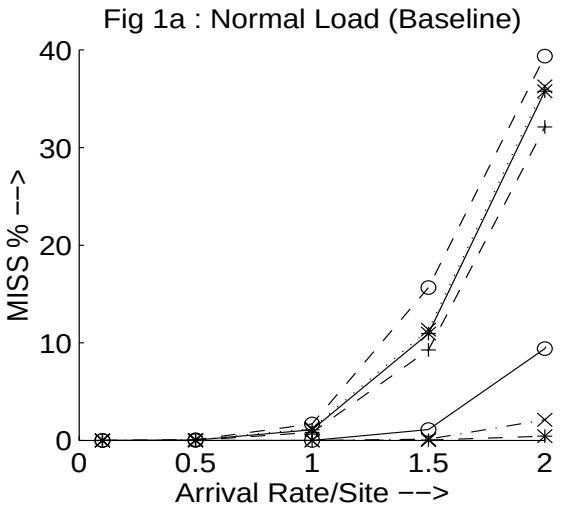


Fig 1a : Normal Load (Baseline)
MISS % −−>
Arrival Rate/Site −−>
0
10
20
30
40
0
0.5
1
1.5
2

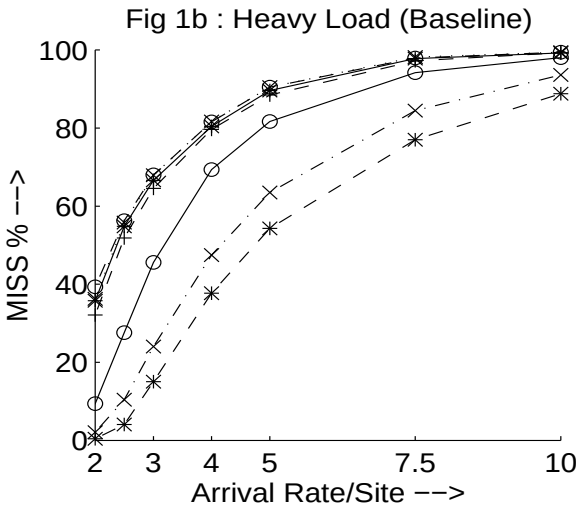
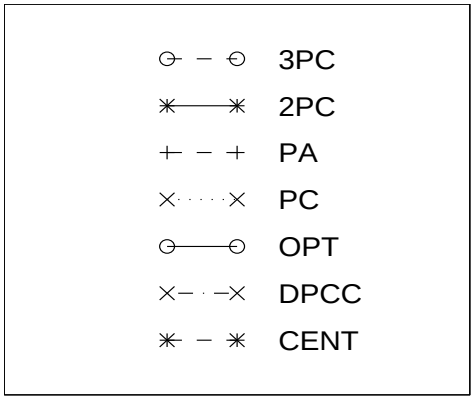


Fig 1b : Heavy Load (Baseline)
MISS % −−>
Arrival Rate/Site −−>
0
20
40
60
80
100
2
3
4
5
7.5
10


3PC
2PC
PA
PC
OPT
DPCC
CENT

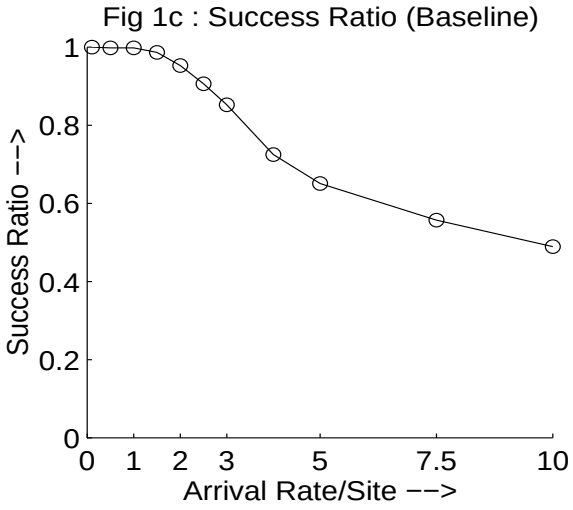


Fig 1c : Success Ratio (Baseline)
Success Ratio −−>
Arrival Rate/Site −−>
0
0.2
0.4
0.6
0.8
1
0
1
2
3
5
7.5
10

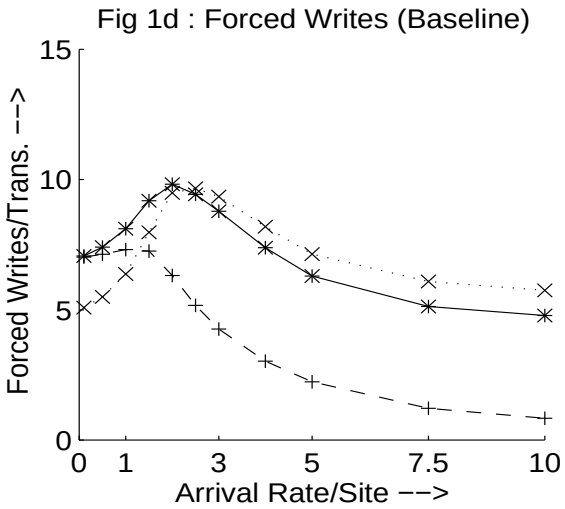


Fig 1d : Forced Writes (Baseline)
Forced Writes/Trans. −−>
Arrival Rate/Site −−>
0
5
10
15
0
1
3
5
7.5
10

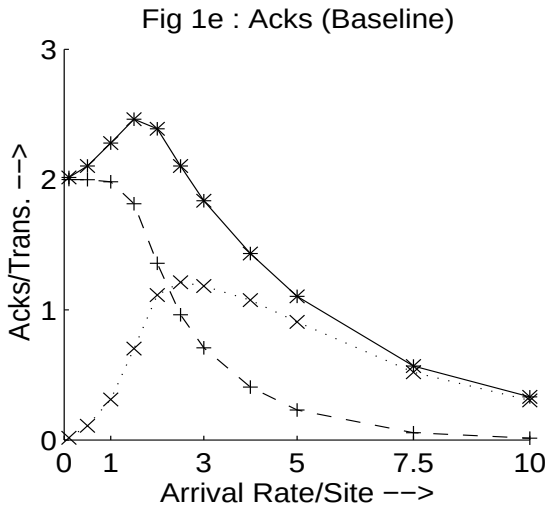


Fig 1e : Acks (Baseline)
Acks/Trans. −−>
Arrival Rate/Site −−>
0
1
2
3
0
1
3
5
7.5
10

Figure 1c and clearly shows that under normal loads, optimism is the right choice since the success ratio is almost one. Under heavy loads, however, there is a decrease in the success ratio – the reason for this is that transactions reach their commit phase only close to their deadlines and in this situation, a lending transaction may often abort due to missing its deadline. These results indicate that under heavy loads, the optimistic policy should be modified such that transactions borrow only from "healthy" lenders, that is, lenders who still have considerable time to their deadline – we intend to address this issue in our future work.

Another interesting point to note is the following: In Figures 1a and 1b the difference between the CENT and DPCC curves shows the effect of distributed *data* processing whereas the difference between the commit protocol curves and the DPCC curve shows the effect of distributed *commit* processing. We see in these figures that the effect of distributed commit is considerably more than that of distributed data processing, even for the OPT protocol. *These results clearly highlight the necessity for designing high-performance real-time commit protocols.*

6.5 Expt. 2: Pure Data Contention

The goal of our next experiment was to isolate the influence of data contention on the real-time performance of the commit protocols. Therefore, for this experiment, the physical resources were made "infinite", that is, there is no queueing for the physical resources. The other parameter values were the same as those used in the baseline experiment. The MissPercent performance results for this system configuration are presented in Figures 2a and 2b, and the supporting statistics are shown in Figures 2c through 2e. We observe in these figures that the relative performance of the various protocols remains qualitatively similar to that seen under finite resources in the previous experiment. The difference in performance between 3PC and 2PC under normal loads is further reduced here since the additional resource overheads present in 3PC have lesser influence as resource contention is not an issue. We also observe that OPT maintains its superior performance as compared to the standard algorithms over the entire loading range. Moreover, its success ratio does not go below 70 percent even at the highest loading levels (Figure 2c).

An important observation here is that while resource contention can be reduced by purchasing more and/or faster resources, there exists no equally simple mechanism to reduce data contention.

6.6 Expt. 3: OPT-PC and OPT-PA

The implementation of OPT used in the previous experiments incorporated only the optimistic access, active abort and silent kill optimizations. We also conducted an experiment to investigate the effect on performance of adding the PA or PC optimizations to OPT. The results of this experiment are shown in Figures 3a and 3b for finite resources and infinite resources, respectively. In these figures, we observe that just as PA and PC provided little improvement on the performance of standard 2PC, here also they provide no tangible benefits to the performance of the OPT protocol and for the same reasons. While OPT-PA is very slightly better than basic OPT, OPT-PC performs worse than OPT under heavy loads, especially with infinite resources.

Fig 2a : Normal Load (Data Contention)

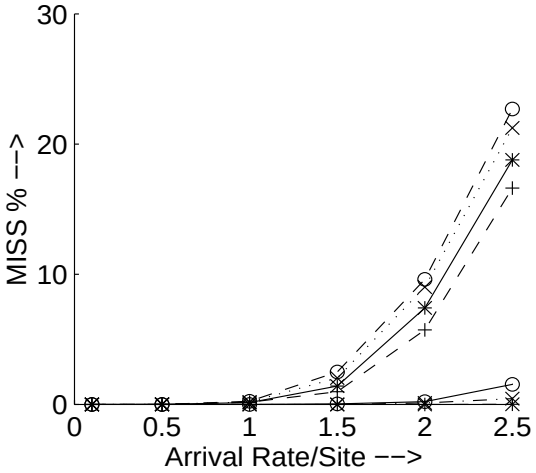
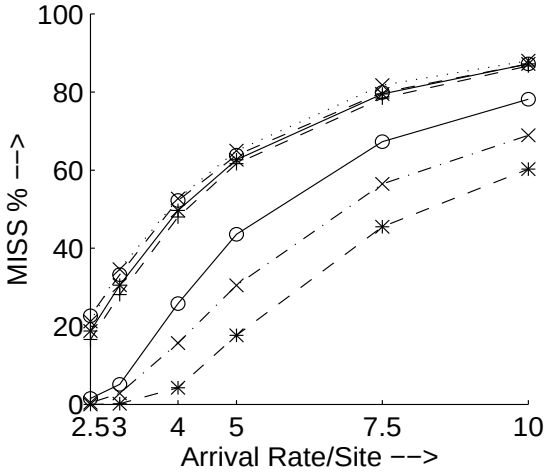


Fig 2b : Heavy Load (Data Contention)



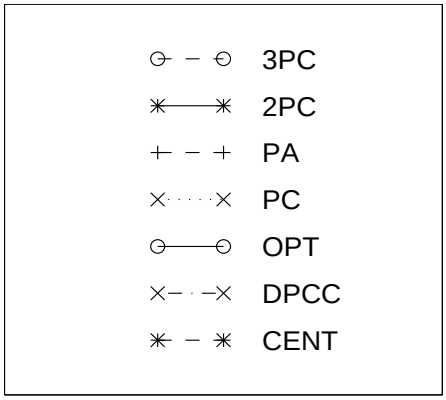


Fig 2c : Success Ratio (Data Contention)

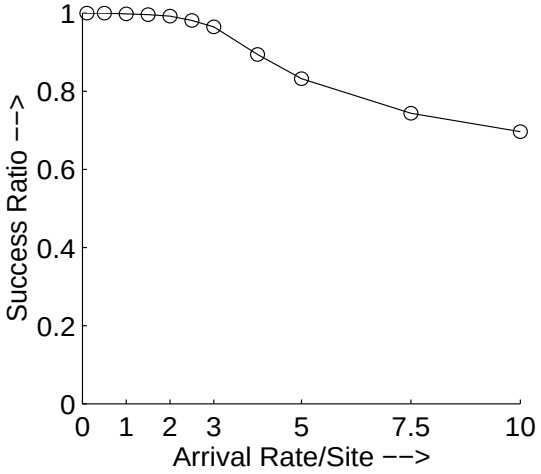


Fig 2d : Forced Writes (Data Contention)

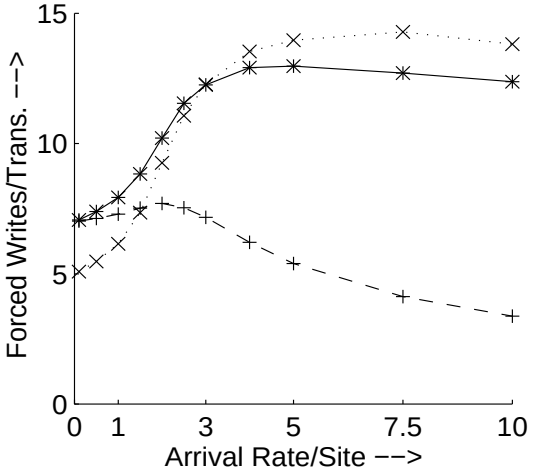
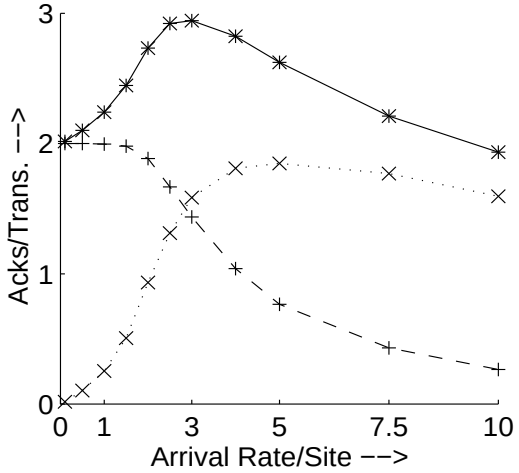


Fig 2e : Acks (Data Contention)



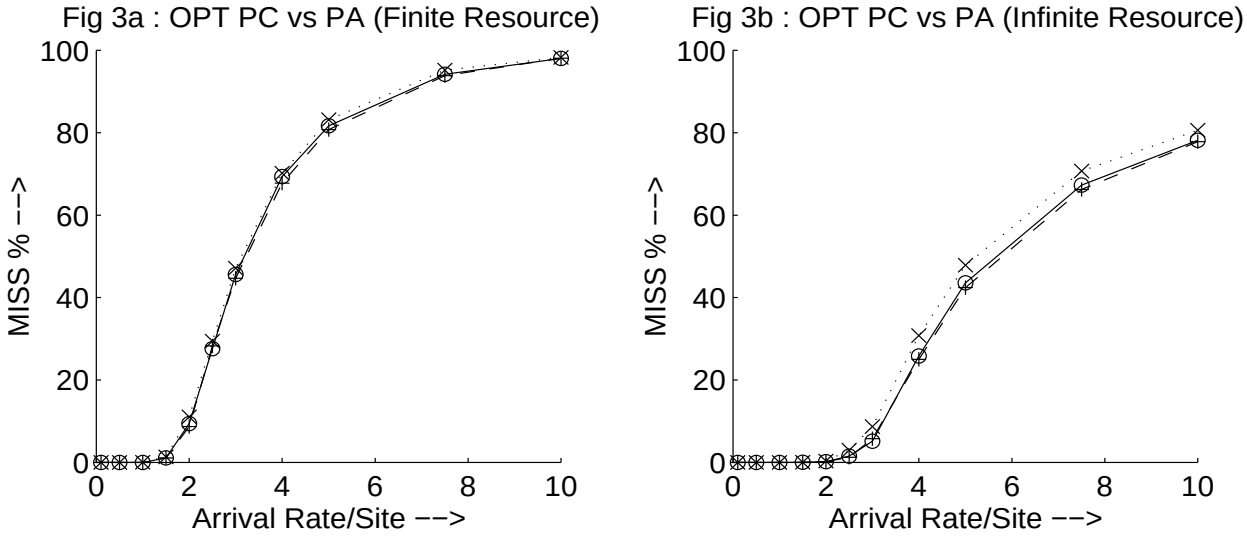
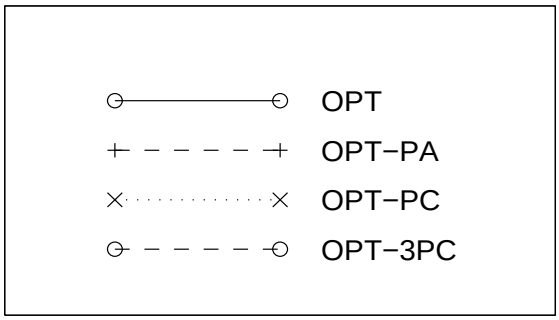


Fig 3a : OPT PC vs PA (Finite Resource)
MISS % −−>
Arrival Rate/Site −−>
0
20
40
60
80
100
0
2
4
6
8
10

Fig 3b : OPT PC vs PA (Infinite Resource)
MISS % −−>
Arrival Rate/Site −−>
0
20
40
60
80
100
0
2
4
6
8
10


OPT
OPT−PA
OPT−PC
OPT−3PC

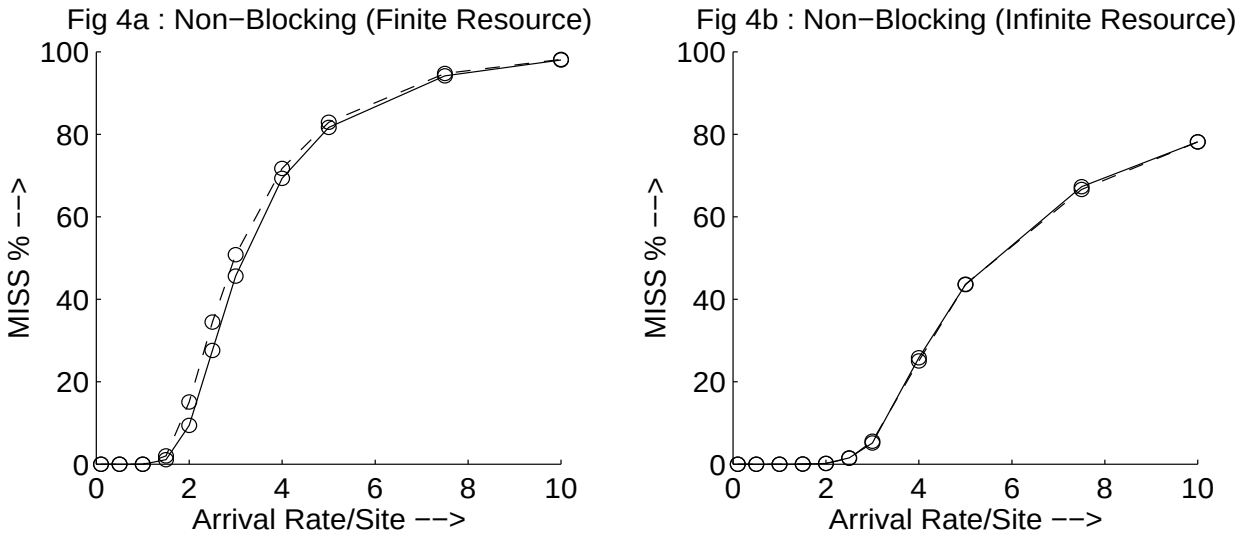


Fig 4a : Non−Blocking (Finite Resource)
MISS % −−>
Arrival Rate/Site −−>
0
20
40
60
80
100
0
2
4
6
8
10

Fig 4b : Non−Blocking (Infinite Resource)
MISS % −−>
Arrival Rate/Site −−>
0
20
40
60
80
100
0
2
4
6
8
10

6.7 Expt. 4: Non-Blocking OPT

In the previous experiments, we observed that OPT, which is based on 2PC, performed significantly better than the standard protocols. This motivated us to evaluate the effect of incorporating the same optimizations in *3PC*. The results of this experiment are shown in Figures 4a and 4b for finite resources and infinite resources, respectively. We see in these figures that for finite resources, OPT-3PC's performance is noticeably but not greatly worse than that of OPT-2PC. Moreover, under infinite resources, the difference between OPT-3PC and OPT-2PC virtually disappears.

The above behavioral patterns of OPT-3PC and OPT-2PC are similar to those observed for basic 3PC and basic 2PC in Experiments 1 and 2 and occur for the same reasons that were outlined there. These results indicate that, in the real-time domain, nonblocking functionality which is extremely useful in case of failures can be purchased at a relatively modest increase in routine processing cost.

6.8 Expt. 5: Fast Network Interface

In all of the previous experiments, the cost for sending and receiving messages modeled a system with a slow network interface ($MsgCpu = 10\,ms$). We conducted another experiment wherein the network interface was faster by a factor of ten, that is, where $MsgCpu = 1\,ms$. The results of this experiment are shown in Figures 5a and 5b for finite resources and infinite resources, respectively. We see here that although all the curves come closer together since distribution has a smaller impact, their relative performance remains the same. Note also that the OPT protocol now provides a performance that is close to that of DPCC, that is, close to the best commit processing performance that could be obtained in a distributed RTDBS, especially with infinite resources.

6.9 Expt. 6: Higher Degree of Distribution

In the experiments described so far, each transaction executed on *three* sites. To investigate the impact of having a higher degree of distribution, we performed an experiment wherein each transaction executed on *five* sites. The $CohortSize$ in this experiment was set to 4 pages as compared to the $CohortSize$ of 6 pages used in the previous experiments. This was required in order to keep the transaction length in this experiment comparable to that in the previous experiments. The results of this experiment are shown in Figures 6a and 6b for finite resources and infinite resources, respectively. In these figures, we observe that increased distribution results in an increase in the magnitudes of the performance differences among the commit protocols. This is to be expected since the commit processing overhead is larger in this experiment. The relative performance of the protocols, however, remains qualitatively the same with OPT continuing to perform much better than the standard protocols.

Fig 5a : Fast Network (Finite Resource)

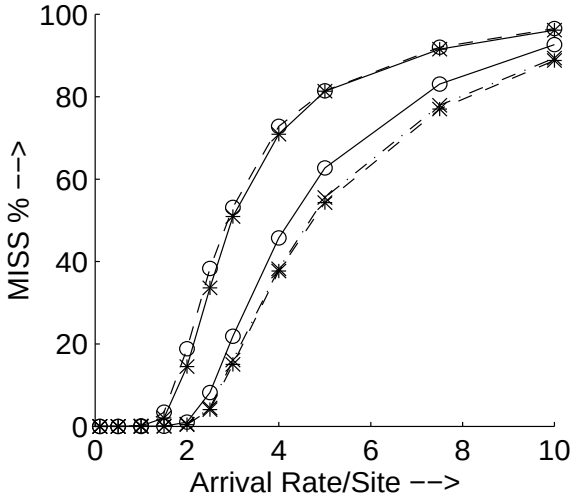
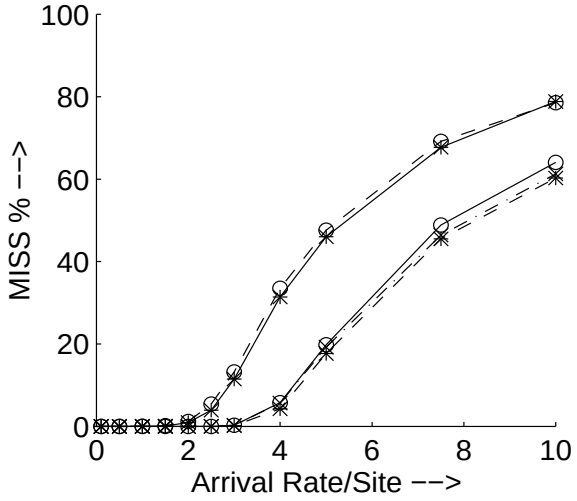


Fig 5b : Fast Network (Infinite Resource)



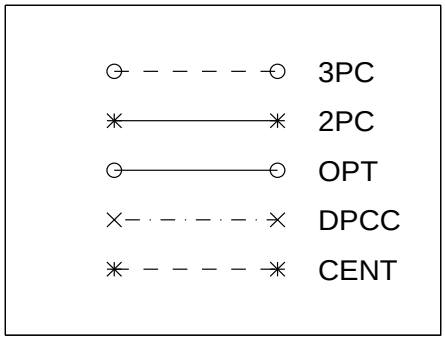


Fig 6a : More Distribution (Finite Resource)

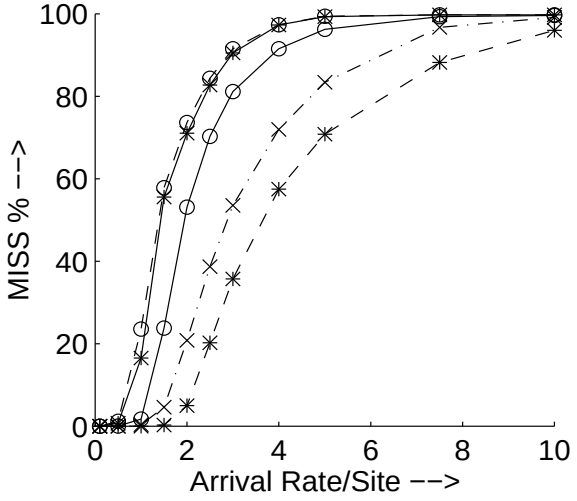
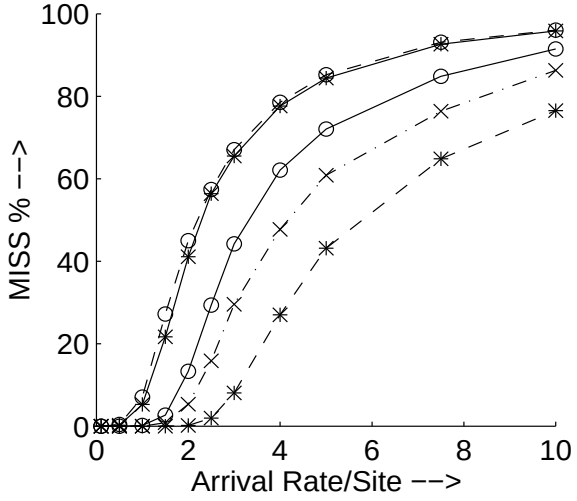


Fig 6b : More Distribution (Infinite Resource)



7 Conclusions

Although a significant body of research literature exists for centralized real-time database systems, comparatively little work has been done on distributed RTDBS. In particular, the problem of commit processing in a distributed environment has not yet been addressed in detail. The few papers on this topic assume that it is necessary to relax the traditional notion of atomicity when commit protocols are designed for distributed real-time databases. In this paper, we have proposed and evaluated new mechanisms for designing high performance real-time commit protocols that do not require transaction atomicity requirements to be weakened.

We first precisely defined the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a distributed setting. Subsequently, we made a detailed study of the relative performance of different commit protocols in a distributed real-time database system. Using a detailed simulation model of a firm-deadline RTDBS, we evaluated the deadline miss performance of a variety of standard commit protocols including 2PC, Presumed Abort, Presumed Commit, and 3PC. We also developed and evaluated a new commit protocol, OPT, that was designed specifically for the real-time environment and included features such as controlled optimistic access to uncommitted data, active abort and silent kill. To the best of our knowledge, these are the first quantitative results in this area.

Our experiments, which covered a variety of transaction workloads and system configurations, demonstrated the following:

1. Distributed commit processing can have considerably more effect than distributed data processing on the real-time performance, especially on systems with slow network interfaces. This highlights the need for developing commit protocols tuned to the real-time domain.

2. The standard 2PC and 3PC algorithms perform poorly in the real-time environment due to their passive nature and due to preventing access to data held by cohorts in the prepared state.

3. The PA and PC variants of 2PC, although reducing protocol overheads, fail to provide tangible benefits in the real-time environment[2]. This is in marked contrast to the conventional DBMS environment where they have been implemented in a number of commercial products including Tandem's TMF, DEC's VAX/VMS, Transarc's Encina and USL's TUXEDO, and are now part of the ISO-OSI and X/OPEN distributed transaction processing standards [SBCM93].

4. The new protocol, OPT, provides significantly improved performance over the standard algorithms for all the workloads and system configurations considered in this study. Its good performance is attained primarily due to its optimistic borrowing of uncommitted data and active abort policy. The optimistic access significantly reduces the effect of priority inversion which is inevitable in the prepared state. Supporting statistics showed that OPT's optimism about uncommitted data is justified, especially under normal loads. The other

[2]This conclusion is limited to the completely update transaction workloads considered here. PA and PC have additional optimizations for fully or partially read-only transactions [MLO86].

optimizations of silent kill and presumed commit/abort, however, had comparatively little beneficial effect.

5. Experiments combining the optimizations of OPT with 3PC indicate that the nonblocking functionality can be obtained in the real-time environment at a relatively modest cost in normal processing performance. This is especially encouraging given the high desirability of the nonblocking feature in the real-time environment.

In summary, our results have shown that in the firm-deadline real-time domain, the performance recommendations for distributed commit processing can be considerably different from those for the corresponding conventional database system.

Future Work

An alternative to the optimistic access approach followed in the OPT protocol would be to use *priority inheritance* [SRL87] – in this scheme, a low priority transaction that blocks a high priority transaction inherits the priority of the high priority transaction. The expectation is that the blocking time of the high priority transaction will be reduced since the low priority transaction will now execute faster and release its resources earlier. A study of priority inheritance in the context of transaction concurrency control was made in [HSRTP92] and their results indicate that priority inheritance is useful only if it occurs towards the end of a transaction's lifetime. This seems to fit well with handling priority inversion during commit processing since this stage occurs at the end of transaction execution. We intend to explore this issue in our future work. We also intend to investigate other ways by which the performance deterioration could be reduced without sacrificing standard notions of database correctness, for example, by restricting borrowers to borrow only from "healthy" lenders in the OPT protocol. Finally, the concurrency control protocols discussed in literature include both the locking based and the optimistic ones, but the commit protocols discussed so for are the locking based ones only. With the "optimistic commit" protocol proposed in this work, the following four combinations are possible: Locking CC + Locking Commit, Locking CC + Optimistic Commit, Optimistic CC + Locking Commit, and Optimistic CC + Optimistic Commit. In this study we have focussed on the combination "Locking CC + Optimistic Commit". It will be interesting to study the effects of these various combinations of concurrency control and commit protocols, and find out the integrated concurrency control and commit protocol that provides the best performance.

Acknowledgements

The work of Jayant Haritsa was supported in part by a research grant from the Dept. of Science and Technology, Govt. of India.

References

[ACL87] R. Agrawal, M. Carey and M. Livny, "Concurrency Control performance modeling: alternatives and implications", *ACM Trans. on Database Systems*, 12(4), December 1987.

[AG88] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: a Performance Evaluation", *Proc. of 14th Intl. Conf. on Very Large Databases*, August 1988.

[Bes94] A. Bestavros, "Multi-version Speculative Concurrency Control with Delayed Commit", *Proc. of Intl. Conf. on Computers and their Applications*, March 1994.

[Bha87] B. Bhargava, (editor), *Concurrency and Reliability in Distributed Database Systems*, Van Nostrand Reinhold, New York, 1987.

[BHG87] P. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Coo82] E. Cooper, "Analysis of Distributed Commit Protocols", *Proc. of ACM Sigmod Conf.*, June 1982.

[CL88] M. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication", *Proc. of 14th Intl. Conf. on Very Large Databases*, August 1988.

[DLW89] S. Davidson, I. Lee and V. Wolfe, "A Protocol for Timed Atomic Commitment" *Proc. of 9th Intl. Conf. on Distributed Computing Systems*, 1989.

[Gra78] J. Gray, "Notes on Database Operating Systems", *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, 60, 1978

[Gra81] J. Gray, "The Transaction Concept: Virtues and Limitations", *Proc. of 7th Intl. Conf. on Very Large Databases*, 1981.

[HCL90] J. Haritsa, M. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control", *Proc. of 11th IEEE Real-Time Systems Symp.*, December 1990.

[HCL92] J. Haritsa, M. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Real-Time Systems Journal*, 4 (3), 1992.

[HSRTP92] J. Huang, J. Stankovic, K. Ramamritham, D. Towsley and B. Purimetla, "On Using Priority Inheritance in Real-Time Databases", *Real-Time Systems Journal*, 4 (3), 1992.

[Koh81] W. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer System", *ACM Computing Surveys*, 13(2), June 1981.

[KLS90] H. Korth, E. Levy, and A. Silberschatz, "A formal approach to recovery by compensating transactions", *Proc. of 16th Intl. Conf. on Very Large Databases*, August 1990.

[LKS91a] E. Levy, H. Korth and A. Silberschatz, "An optimistic commit protocol for distributed transaction management", *Proc. of ACM SIGMOD Conf.*, May 1991.

[LKS91b] E. Levy, H. Korth and A. Silberschatz, "A theory of relaxed atomicity", *Proc. of ACM Symp. on Principles of Distributed Computing*, August 1991.

[LL93] B. Lampson and D. Lomet, "A New Presumed Commit Optimization for Two Phase Commit", *Proc. of 19th VLDB Conference*, 1993.

[LM94] M. C. Little and D. L. McCue, *C++SIM User's Guide Public Release 1.5*, Dept. of Computing Science, Univ. of Newcastle upon Tyne, U.K., 1994.

[LS76] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System", *Tech. Report, Xerox Palo Alto Research Center*, 1976.

[MLO86] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R^* Distributed Database Management System", *ACM Transactions on Database Systems*, 11(4), 1986.

[OV91] M. Oszu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, 1991.

[Ske81] D. Skeen, "Nonblocking Commit Protocols", *Proc. of ACM SIGMOD Conf.*, June 1981.

[Son90] S. Son, "Real-Time Database Systems: A New Challenge", *Data Engineering*, 13(4), December 1990.

[SBCM93] G. Samaras, K. Britton, A. Citron and C. Mohan, "Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment", *Proc. of IEEE Data Engineering Conf.*, February 1993.

[SJR91] P. Spiro, A. Joshi and T. Rengarajan, "Designing an Optimized Transaction Commit Protocol", *Digital Technical Journal*, 3(1), 1991.

[SK92] S. Son and S. Kouloumbis, "Replication Control for Distributed Real-Time Database Systems", *Proc. of 12th Intl. Conf. on Distributed Computing Systems*, 1992.

[SLKS92] N. Soparkar, E. Levy, H. Korth and A. Silberschatz, "Adaptive Commitment for Real-Time Distributed Transactions", *TR-92-15, Dept. of Computer Science, Univ. of Texas-Austin*, 1992.

[SRL87] L. Sha, R. Rajkumar and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization", *Tech. Report CMU-CS-87-181*, Carnegie Mellon University.

[SRL88] L. Sha, R. Rajkumar and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases", *ACM SIGMOD Record*, 17(1), March 1988.

[Ulu94a] O. Ulusoy, "Research Issues in Real-Time Database Systems", *Tech. Report BU-CEIS-94-32*, Dept. of Computer Engg. and Information Science, Bilkent University, Turkey, 1994.

[Ulu94b] O. Ulusoy, "Processing Real-Time Transactions in a Replicated Database System", *Intl. Journal of Distributed and Parallel Databases*, 2(4), 1994.

[UB92] O. Ulusoy and G. Belford, "Real-Time Lock Based Concurrency Control in a Distributed Database System", *Proc. of 12th Intl. Conf. on Distributed Computing Systems*, 1992.

[Yoo94] Y. Yoon, "Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems", *Ph.D. Thesis*, Korea Advanced Institute of Science and Technology, May 1994.