The PROMPT Real-Time Commit Protocol

Jayant R. Haritsa, *Member*, *IEEE*, Krithi Ramamritham, *Fellow*, *IEEE*, and Ramesh Gupta

Abstract—We investigate the performance implications of providing transaction atomicity for firm-deadline real-time applications operating on distributed data. Using a detailed simulation model, the real-time performance of a representative set of classical transaction commit protocols is evaluated. The experimental results show that data distribution has a significant influence on real-time performance and that the choice of commit protocol clearly affects the magnitude of this influence. We also propose and evaluate a new commit protocol, PROMPT (Permits Reading Of Modified Prepared-data for Timeliness), that is specifically designed for the real-time domain. PROMPT allows transactions to "optimistically" borrow, in a controlled manner, the updated data of transactions currently in their commit phase. This controlled borrowing reduces the data inaccessibility and the priority inversion that is inherent in distributed real-time commit processing. A simulation-based evaluation shows PROMPT to be highly successful, as compared to the classical commit protocols, in minimizing the number of missed transaction deadlines. In fact, its performance is close to the best on-line performance that could be achieved using the optimistic lending approach. Further, it is easy to implement and incorporate in current database system software. Finally, PROMPT is compared against an alternative priority inheritance-based approach to addressing priority inversion during commit processing. The results indicate that priority inheritance does not provide tangible performance benefits.

Index Terms—Distributed real-time database, commit protocol, two phase commit, three phase commit, priority inheritance, performance evaluation.

1 Introduction

Many real-time database applications are inherently *distributed* in nature [39], [44]. These include the intelligent network services database described in [10] and the mobile telecommunication system discussed in [46]. More recent applications include the multitude of directory, data-feed, and electronic commerce services that have become available on the World Wide Web. However, although real-time research has been underway for close to a decade now, the focus has been primarily on *centralized* database systems. In comparison, distributed real-time database systems (**DRTDBS**) have received little attention, making it difficult for their designers to make informed choices.

Real-time database systems operating on distributed data have to contend with the well-known complexities of supporting transaction ACID semantics in the distributed environment [3], [35]. While the issue of designing real-time protocols to ensure distributed *transaction serializability* has been considered to some extent (for example, [28], [37], [43], [45]), very little work has been done with regard to the equally important issue of ensuring distributed *transaction atomicity*. We address this lacuna here.

- *J.R. Haritsa is with the Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Sir C.V. Raman Rd., Bangalore 560012, India. E-mail: haritsa@dsl.serc.iisc.ernet.in.*
- *K. Ramamritham is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003, and the Indian Institute of Technology, Bombay. E-mail: krithi@cs.umass.edu.*
- *R. Gupta is with Goldencom Technologies, 392 Acoma Way, Fremont, CA 94539. E-mail: ramesh@goldencom.com.*

Manuscript received 6 July 1998; revised 22 Dec. 1998; accepted 16 Mar. 1999.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107109.

1.1 Commit Protocols

Distributed database systems implement a transaction *commit protocol* to ensure transaction atomicity. Over the last two decades, a variety of commit protocols have been proposed (for non-real-time database systems) by database researchers (see [5], [26], [35] for surveys). These include the classical *Two Phase Commit (2PC)* protocol [16], [30], its variations, such as *Presumed Commit (PC)* and *Presumed Abort (PA)* [29], [34], and *Three Phase Commit (3PC)* [38]. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction was executed. In addition, several log records are generated, some of which have to be "forced," that is, flushed to disk immediately in a synchronous manner. Due to this series of synchronous message and logging costs, commit processing can significantly increase the execution times of transactions [29], [12], [40]. This is especially problematic in the real-time context, since it has a direct adverse effect on the system's ability to meet transaction timing constraints. Therefore, *the choice of commit protocol is an important design decision for DRTDBS*.

The few papers in the literature that have tried to address this issue [8], [15], [46] have required either relaxing the traditional notion of atomicity or strict resource allocation and resource performance guarantees from the system. Instead of resorting to such fundamental alterations of the standard distributed DBMS framework, we take a different approach in this paper—we attempt to design high-performance real-time commit protocols by incorporating novel *protocol features*. The advantage of this approach is that it lends itself to easy integration with current application and system software.

Our study is conducted in the context of real-time applications that impose "firm deadlines" [24] for

1045-9219/00/$10.00 © 2000 IEEE

transaction completion. For such applications, completing a transaction after its deadline has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are "killed"; that is, immediately aborted and discarded from the system without being executed to completion. Accordingly, the performance metric is *KillPercent*, the steady-state percentage of killed transactions.[1]

1.2 Contributions

For the above real-time context, the main contributions of this paper are the following:

1. We precisely define the semantics of firm deadlines in the DRTDBS environment.
2. We investigate the performance implications of supporting transaction atomicity in a DRTDBS. Using a detailed simulation model, we profile the KillPercent performance of a representative set of classical commit protocols, including 2PC, PA, PC, and 3PC. To the best of our knowledge, this is the *first* quantitative evaluation of these protocols in the real-time environment.
3. We propose and evaluate a new commit protocol called **PROMPT** (Permits Reading Of Modified Prepared-data for Timeliness), which is designed specifically for DRTDBS. The main feature of PROMPT is that it allows transactions to "optimistically" borrow the updated data of transactions currently in their commit phase. This borrowing speeds up transaction execution by reducing the data inaccessibility and the priority inversion that is, as explained later, *inherent* in distributed commit processing. At the same time, the borrowing is *controlled* to ensure that cascading aborts, usually associated with the use of dirty (i.e., modified and uncommitted) data, do not occur. PROMPT also incorporates several other features that cater to the special characteristics of the real-time environment. Finally, PROMPT is easy to implement and incorporate in current systems, and can be integrated, often *synergistically*, with many of the optimizations proposed earlier, including industry standard protocols such as PC and PA.

1.3 Organization

The remainder of this paper is organized as follows: The performance framework of our study is outlined in Section 2. A representative set of commit protocols designed for traditional (non-real-time) databases and their drawbacks in DRTDBS environments are described in Section 3. The semantics of firm deadlines in distributed database environments are presented in Section 4. Section 5 introduces PROMPT, our new commit protocol designed specifically for distributed real-time transactions. The performance model is described in Section 6, and the results of the simulation experiments, which compare PROMPT with the traditional commit protocols, are highlighted in Section 7. A few options in PROMPT's design are explored in Section 8. An alternative priority inheritance-based approach for reducing the effect of priority inversion during commit processing is presented and evaluated against PROMPT in Section 9. Related work on real-time commit protocols is reviewed in Section 10. Finally, in Section 11, we present the conclusions of our study.

1. Or, equivalently, the percentage of missed deadlines.

2 Performance Framework

From a performance perspective, commit protocols can be compared with respect to the following issues:

1. **Effect on Normal Processing.** This refers to the extent to which the commit protocol affects the normal (no-failure) distributed transaction processing performance. That is, how expensive is it to provide atomicity using this protocol?
2. **Resilience to Failures.** When a failure occurs in a distributed system, ideally, transaction processing in the rest of the system should not be affected during the recovery of the failed component. With most commit protocols, however, failures can lead to transaction processing grinding to a halt (as explained in Section 3.4), and they are therefore termed as "blocking protocols."

   To ensure that such major disruptions do not occur, efforts have been made to design "nonblocking commit protocols." These protocols, in the event of a site failure, permit transactions that had cohorts executing at the failed site to terminate at the operational sites without waiting for the failed site to recover [35], [38].[2] To achieve their functionality, however, they usually incur *additional* messages and forced-log-writes than their blocking counterparts.

   In general, "two-phase" commit protocols are susceptible to blocking, whereas "three-phase" commit protocols are nonblocking.
3. **Speed of Recovery.** This refers to the time required for the database to be recovered to a consistent state when the failed site comes back up after a crash. That is, how long does it take before transaction processing can commence again in a recovering site?

Of the three issues highlighted above, the design emphasis of most commit protocols has been on the first two (effect on normal processing and resilience to failures) since they directly affect ongoing transaction processing. 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 protocols because of one commit protocol having a faster recovery time than the other. Based on this viewpoint, our focus here also is on the *mechanisms* required during normal (no-failure) operation to provide for recoverability and resilience to failures, and not on the post-failure *recovery process*.

2. It is impossible to design commit protocols that are completely nonblocking to both site and link failures [3]. However, the number of simultaneous failures that can be tolerated before blocking arises depends on the protocol design.

3 Traditional Distributed Commit Protocols

We adopt the common "subtransaction model" [6] of distributed transaction execution in our study. In this model, 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 various sites that are accessed by the transaction.[3] Cohorts are created by the master sending a STARTWORK message to the local transaction manager at that site. This message includes the work to be done at that site and is passed on to the cohort. Each cohort sends a *workdone* message to the master after it has completed its assigned data processing work, and the master initiates the commit protocol (only) after it has received this message from all its cohorts.

For the above transaction execution model, a variety of commit protocols have been devised, most of which are based on the classical 2PC protocol [16]. In our study, we focus on the 2PC, PA, PC, and 3PC protocols since these protocols are well-established and have received the most attention in the literature. We briefly describe these protocols in the remainder of this section—complete descriptions are available in [34], [12], [38]. For ease of exposition, the following notations are used in the sequel—"SMALL CAPS FONT" for messages, "`typewriter font`" for log records, and "sans serif font" for transaction states.

3.1 Two Phase Commit Protocol

The **two-phase commit (2PC)** protocol, as suggested by its name, operates in two phases: In the first phase, called the "voting phase," the master reaches a global decision (commit or abort) based on the local decisions of the cohorts. In the second phase, called the "decision 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 eventually aborted. This is usually implemented by using *logging* mechanisms such as write-ahead-logging (WAL) [16], 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.

After receiving the WORKDONE message from all the cohorts participating in the distributed execution of the transaction, the master initiates the first phase of the commit protocol by sending PREPARE (to commit) messages in parallel to all its 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. At this stage, the cohort has entered a prepared state wherein it cannot unilaterally commit or abort the transaction but has to wait for the final decision from the master. On the other hand, each cohort that decides to abort 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 the decision from the master.

After the master receives votes from all its cohorts, the second phase of the protocol is initiated. If all the votes are YES, the master moves to a committing state by force-writing a `commit` log record and sending COMMIT messages to all its cohorts. Each cohort, upon receiving the COMMIT message, moves to the committing state, force-writes a `commit` log record, and sends an ACK message to the master.

On the other hand, 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 ACKs from all the prepared cohorts, writes an `end` log record and then "forgets" the transaction (by removing from virtual memory all information associated with 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)** [34], tries to reduce these message and logging overheads by requiring all participants to follow, during failure recovery, an "in the no-information case, 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 (correctly) assumed to have been aborted. With this assumption, it is not necessary for cohorts to send ACK for ABORT messages from the master, or to force-write the `abort` record to the log. It is also not necessary for an aborting master to force-write the `abort` log record or to write an `end` log record.

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

Another variant of 2PC, called **presumed commit (PC)** [34], is based on the observation that, in general, the number of committed transactions is much more than the number of aborted transactions. In PC, the overheads are reduced for *committing* transactions, rather than aborted transactions, by requiring all participants to follow, during failure recovery, an "in the no-information case, commit" rule. In this scheme, cohorts do not send ACKs for a commit decision sent from the master, and also do not force-write the `commit` log record. In addition, the master does not write an `end` log record. On the down side, however, the master is required 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 database products and standards [12].

3. In the most general case, each of the cohorts may itself spawn off subtransactions at other sites, leading to the "tree of processes" transaction structure of System R^* [9—for simplicity, we only consider a two-level tree here.

3.4 Three Phase Commit

A fundamental problem with all of 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. These transactions, in turn, become blocked waiting for the resources to be relinquished, resulting in "cascading blocking." So, if the duration of the blocked period is significant, the outcome could be a major disruption of transaction processing activity.

To address the blocking problem, a **three phase commit (3PC)** protocol was proposed in [38]. 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 site. Note, however, that the price of gaining nonblocking functionality is an increase in the communication and logging overheads since: 1) There is an extra round of message exchange between the master and the cohorts, and 2) both the master and the cohorts have to force-write additional log records in the precommit phase.

3.5 Master and Cohort Execution Phases

As described above, commit protocols typically operate in two or three phases. For ease of exposition, we will similarly divide the *overall execution* of masters (which represent the entire transaction), and of individual cohorts, into phases.

A master's execution is composed of two phases: the "data phase" and the "commit phase." The data phase begins with the sending of the first STARTWORK message and ends when all the WORKDONE messages have been received, that is, it captures the data processing period. The commit phase begins with the sending of the PREPARE messages and ends when the transaction is forgotten; that is, it captures the commit processing period.

A cohort's execution is composed of three phases: the "data phase," the "commit phase," and the "wait phase." In the data phase the cohort carries out its locally assigned data processing—it begins with the receipt of the STARTWORK message from the master and ends with the sending of the WORKDONE message to the master. The commit phase begins with the cohort receiving the PREPARE message and ends with the last commit-related action taken by the cohort (this is a function of the commit protocol in use). The wait phase denotes the time period in between the data phase and the commit phase, that is, the time period between sending the WORKDONE message and receiving the PREPARE message.

3.6 Inadequacies in the DRTDBS Environment

The commit protocols described in this section were designed for traditional database systems where transaction throughput or average response time is usually the primary performance metric. With respect to meeting (firm) real-time objectives, however, they fail on two related counts: First, by making prepared data inaccessible, they increase transaction blocking times and therefore have an adverse impact on the number of killed transactions. Second, *prioritized* scheduling policies are typically used in RTDBS to minimize the number of killed transactions. These commit protocols, however, 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 [36]. 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 priorities. At the CPU, for example, a scheduling policy such as Priority Preemptive Resume ensures the absence of priority inversion. 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 *update* 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 DRTDBS 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 (for example, due to network delays). If the inversion period is large, it may have a significantly negative effect on performance.

It is important to note that this "prepared data blocking" is *distinct* from the "decision blocking" (because of failures) that was discussed in Section 3.4. That is, in all the commit protocols, *including 3PC*, transactions can be affected by prepared data blocking. In fact, 3PC's strategy for removing decision blocking *increases* the duration of prepared data blocking. Moreover, such data blocking occurs during normal processing, whereas decision blocking only occurs during failure situations.

To address the above-mentioned drawbacks (prepared data inaccessibility and priority inversion) of the classical commit protocols, we have designed a new commit protocol called **PROMPT**. The PROMPT design is based on a specific semantics of firm deadlines in DRTDBS, defined in the following section—the description of PROMPT itself is deferred to Section 5.

4 Firm Deadline Semantics in DRTDBS

The semantics of firm deadlines is that a transaction should be either committed before its deadline or be killed when

the deadline expires. To implement this notion in a distributed RTDBS, ideally the *master and all the cohorts* of a successfully executed transaction should commit the transaction before the deadline expires or they should abort immediately upon deadline expiry. In practice, however, it is impossible to provide such guarantees because of the arbitrary message delays and the possibility of failures [8]. To avoid inconsistencies in such cases, we define the firm deadline semantics in the distributed environment as follows:

Definition. *A distributed firm-deadline real-time 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 at its site. This definition applies irrespective of whether the cohorts have also received and recorded the commit decision by the deadline.*

To ensure transaction atomicity with the above definition, we require prepared cohorts that receive the final decision *after* the local expiry of the deadline to still implement this decision. Note that this is consistent with the intuitive notion of firm deadlines, since all that happens is that access to prepared data is prevented even beyond the deadline until the decision is received by the cohort; other transactions which would normally expect the data to be released by the deadline only experience a delay. We expect that many real-time database applications, especially those related to electronic commerce (e.g., electronic auctions), will subscribe to these semantics.

Typically, the master is responsible for returning the results of a transaction to the invoker of the transaction. From the above discussion, it is clear that the semantics we prescribe are such that, if a transaction commits, its results will begin to be output *before* the deadline. Further, the problem of delayed access to data, even after the expiry of the deadline of the cohort holding these data items, applies primarily to the *classical protocols*—the effect is considerably reduced with PROMPT, as discussed in the following section.

5 The PROMPT Real-Time Commit Protocol

The main feature of our new **PROMPT** commit protocol is that transactions requesting data items held by other transactions in the prepared state are allowed to access this data.[4] That is, prepared cohorts *lend* their uncommitted data to concurrently executing transactions (without, of course, releasing the update locks). The mechanics of the interactions between such "lenders" and their associated "borrowers" are captured in the following three scenarios, only one of which will occur for each lending:

1. **Lender Receives Decision Before Borrower Completes Data Processing.** Here, the lending cohort receives its global decision before the borrowing cohort has completed its local data processing. If the global decision is to commit, the lending cohort completes in the normal fashion. On the other hand, if the global decision is to abort, the lender is aborted in the normal fashion. In addition, the borrower is also aborted since it has utilized dirty data.
2. **Borrower Completes Data Processing Before Lender Receives Decision.** Here, the borrowing cohort completes its local data processing before the lending cohort has received its global decision. The borrower is now "put on the shelf," that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the (commit-related) processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until either the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, the borrower is "taken off the shelf" (if it has no other "pending" lenders) and allowed to send its WORKDONE message, whereas if the lender aborts, the borrower is also aborted immediately, since it has utilized dirty data (as in Scenario 1 above). In the latter case (deadline expiry), the borrower is killed in the normal manner.
3. **Borrower Aborts During Data Processing Before Lender Receives Decision.** Here, the borrowing cohort aborts in the course of its data processing (due to either a local problem, deadline expiry, or receipt of an ABORT message from its master) before the lending cohort has received its global decision. In this situation, the borrower's updates are undone and the lending is nullified.

In summary, the PROMPT protocol allows transactions to access uncommitted data held by prepared transactions in the "optimistic" belief that this data will eventually be committed.[5] It uses this approach to mitigate the effects of both the data inaccessibility and the priority inversion problems that were identified earlier for traditional commit protocols (Section 3.6).

We wish to clarify here that while the PROMPT design may superficially appear similar to that of optimistic concurrency control [27], it is actually quite different, since updates are made in-place and not to copies or versions of the data; also, data is lent only by transactions that have completed their data processing.

5.1 Additional Real-Time Features of PROMPT

To further improve its real-time performance, three additional features are included in the PROMPT protocol: Active Abort, Silent Kill, and Healthy Lending. These features are described below.

5.1.1 Active Abort

In the basic 2PC protocol, cohorts are "passive" in that they inform the master of their status only upon explicit request

4. While PROMPT is intended for the real-time domain, we have successfully used its basic lending approach to also design efficient commit protocols for *traditional* (non-real-time) distributed database systems [21].

5. A similar, but unrelated, strategy of allowing access to uncommitted data has also been used to improve real-time.

by the master. This is acceptable in conventional distributed DBMS since, after a cohort has completed its data phase, there is *no possibility* of the cohort subsequently being aborted due to serializability considerations (assuming a locking-based concurrency control mechanism).

In a DRTDBS, however, a cohort which is not yet in its commit phase can be aborted due to conflicts with higher priority transactions. Therefore, it may be better for an aborting cohort to immediately inform the master so that the abort of the transaction at the sibling sites can be done earlier. Early restarts are beneficial in two ways: First, they provide more time for the restarted transaction to complete before its deadline. Second, they minimize the wastage of both logical and physical system resources. Accordingly, cohorts in PROMPT follow an "active abort" policy—they inform the master as soon as they decide to abort locally; the subsequent abort process implemented by the master is the same as that followed in the traditional passive environment.

5.1.2 Silent Kill

For a transaction that is killed before the master enters its commit phase, 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).[6] Eliminating this round of messages may help to save system resources. Therefore, in PROMPT, aborts due to deadline misses that occur before the master has initiated the commit protocol are implemented "silently" without requiring any communication between the master and the cohort.

5.1.3 Healthy Lending

A committing transaction that is close to its deadline may be killed due to deadline expiry before its commit processing is finshed. Lendings by such transactions must be avoided, since they are likely to result in the aborts of all the associated borrowers. To address this issue, we have added a feature to PROMPT whereby only "healthy" transactions, that is, transactions whose deadlines are not very close, are allowed to lend their prepared data. This is realized in the following manner: A *health factor*, HF_T, is associated with each transaction T and a transaction is allowed to lend its data only if its health factor is greater than a (system-specified) minimum value $MinHF$. The health factor is computed at the point of time when the master is ready to send the PREPARE messages and is defined to be the ratio *TimeLeft / MinTime*, where *TimeLeft* is the time left until the transaction's deadline, and *MinTime* is the minimum time required for commit processing (recall that a minimum of two messages and one force-write need to be processed before the master can take a decision).

The success of the above scheme is directly dependent on the threshold health factor $MinHF$—set too conservatively (large values), it will turn off the borrowing feature to a large extent, thus effectively reducing PROMPT to standard 2PC; on the other hand, set too aggressively (small values), it will fail to stop several lenders that will eventually abort. In our experiments, we consider a range of values for $MinHF$ to determine the best choices.

An important point to note here is that the health factor is not used to decide the *fate* of the transaction, but merely to decide *whether* the transaction can lend its data. Thus, erroneous estimates about the message processing times and log force-write times only affect the extent to which the optimistic feature of PROMPT is used, as explained above.

5.2 Aborts in PROMPT Do Not Arbitrarily Cascade

An important point to note here is that PROMPT's policy of using uncommitted data is generally *not recommended* in traditional database systems, since this can potentially lead to the well-known problem of *cascading aborts* [3] if the transaction whose dirty data has been accessed is later aborted. However, for the PROMPT protocol, this problem is alleviated due to the following two reasons:

First, the lending transaction is *typically expected to commit* because: 1) The lending cohort is in the prepared state and cannot be aborted due to local data conflicts, and 2) The sibling cohorts are also expected to eventually vote to commit, since they have survived[7] all their data conflicts that occurred prior to the initiation of the commit protocol (given our Active Abort policy).

The only situation where a lending cohort will finally abort is if 1) the deadline expires at the master's node before the master reaches a decision, or 2) a sibling cohort votes NO. The latter case can happen only if the ABORT message sent by the sibling cohort and the PREPARE message sent by the master to the sibling cohort "cross each other" on the network. As the time during which a message is in transit is usually small compared to the transaction execution times, these situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.[8]

Second, even if the lending transaction does eventually abort, it only results in the abort of the immediate borrower and does not cascade beyond this point (since the borrower is not in the prepared state, the only situation in which uncommitted data can be accessed). That is, a *borrower cannot simultaneously be a lender*. Therefore, 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 length of each abort *chain* is limited to one. In short, PROMPT implements a *controlled* lending policy.

5.3 System Integration

We now comment on the *implementation* issues that arise with regard to incorporating the PROMPT protocol in a DRTDBS. The important point to note here is that the required modifications are *local to each site* and do not require intersite communication or coordination.

6. Our firm deadline semantics ensure that skew in clock synchronization, if any, only affects performance, but not atomicity. Further, for minor skews, the performance impact is expected to be marginal.

7. We assume a locking-based concurrency control mechanism.

8. Of course, aborts could also occur after receiving the prepare message due to *non-concurrency-related* issues such as, for example, violation of integrity constraints. Although not described here, our experiments have shown that unless the frequency of such "surprise aborts" is unrealistically high (more than 20 percent), the improvement offered by PROMPT continues to be significant.

- For a borrower cohort that finishes its data processing before its lenders have received their commit/abort decisions from their masters, the *local transaction manager* must not send the WORKDONE message until the fate of all its lenders is determined.
- When a lender is aborted and consequently its borrowers are also aborted, the *local transaction manager* should ensure that the actions of the borrowers are undone *first* and only then are the updates of the associated lender undone—that is, the recovery manager should be invoked in a "borrowers first, lender next" sequence.

  Note that in the event of a system crash, the log records will *naturally* be processed in the above order, since the log records of lenders will always precede those of the borrowers in the sequential log and the log is always scanned backwards during undo processing.
- The *local lock manager* must be modified to permit borrowing of data held by prepared cohorts. The lock mode used by the borrowing cohort should become the current lock mode of the borrowed data item as far as other executing transactions are concerned.
- The *local lock manager* must keep track of the lender-borrower relationships. This information will be needed to handle all possible outcomes of the relationship (for example, if the lender aborts, the associated borrowers must be immediately identified and also aborted), and can be easily maintained using *hash tables*.

The above modifications do not appear difficult to incorporate in current database system software. In fact, some of them are *already provided* in current DBMS—for example, the high-performance industrial-strength ARIES recovery system [11] implements *operation logging* to support semantically rich lock modes that permit updating of uncommitted data. Moreover, as shown later in our experiments, the performance benefits that can be derived from these changes more than compensate for the small amount of run-time overheads entailed by the above modifications and the effort needed to implement them.

5.4 Integrating PROMPT with Other 2PC Optimizations

A particularly attractive feature of PROMPT is that it can be *integrated* with many of the other optimizations suggested for 2PC. For example, Presumed Commit and Presumed Abort (Section 3) can be directly added as a useful supplement to reduce processing overheads. Moreover, the integration may often be *synergistic* in that PROMPT may retain the good features of the added optimization and simultaneously minimize its drawbacks. This is the case, for example, when PROMPT is combined with 3PC: In its attempt to prevent decision blocking, 3PC suffers an *increase* in the prepared data blocking period, but this drawback is reduced by PROMPT's lending feature. The performance improvement that could be obtained from such integrations is evaluated in our experiments (Section 7).

Among additional optimizations [12], PROMPT can be integrated in a straightforward manner with *Read-Only* (one phase commit for read-only transactions), *Long Locks* (cohorts piggyback their commit acknowledgments onto subsequent messages to reduce network traffic), and *Shared Logs* (cohorts that execute at the same site as their master share the same log and therefore do not need to force-write their log records). Further, PROMPT is especially attractive to integrate with protocols such as *Group Commit* [12] (forced writes are batched together to save on disk I/O) and *linear 2PC* [16] (message overheads are reduced by ordering the sites in a linear chain for communication purposes). This is because these optimizations *extend*, like 3PC, the period during which data is held in the prepared state, thereby allowing PROMPT to play a greater role in improving system performance.

Finally, we do not consider here optimizations such as Unsolicited Vote [42], wherein cohorts enter the prepared state at the time of sending the WORKDONE message itself, effectively resulting in "one-phase" protocols.[9] While these protocols reduce the overheads of commit processing due to eliminating an entire phase, they also result in substantially increased priority inversion durations (recall that cohorts in the prepared state cannot be aborted due to conflicts with higher priority transactions). We plan to assess the real-time capabilities of these protocols in our future work.

6 Simulation Model, Metrics, and Baselines

To evaluate the performance of the various commit protocols described in the previous sections, we developed a detailed simulation model of a DRTDBS. Our model is based on a loose combination of the distributed database model presented in [6] and the real-time processing model of [24].

The model consists of a (nonreplicated) database that is distributed over a set of 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 on the completed transactions. The behavior of the communication network is modeled by a *network manager* component.

The following subsections describe the database model, the workload generation process, and the hardware resource configuration. Subsequently, we describe the execution pattern of a typical transaction and the policies adopted for concurrency control and recovery. A summary of the parameters used in the model is given in Table 1.

6.1 Database Model

The database is modeled as a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. Transactions make requests for data pages and concurrency control is implemented at the page level.

9. A detailed survey of such protocols is available in [7].

TABLE 1
Simulation Model Parameters

Parameter	Meaning	Default Setting
$DBSize$	Number of pages in the database	2400
$NumSites$	Number of sites in the database	8
$ArrivalRate$	Transaction arrival rate/site	$0-10$ transactions per second
$TransType$	Transaction Execution Type (Sequential or Parallel)	Sequential
$DistDegree$	Degree of Distribution (number of cohorts)	3
$CohortSize$	Average cohort size	6 pages
$UpdateProb$	Page update probability	1.0
$SlackFactor$	Slack Factor in Deadline Assignment	4.0
$NumCPUs$	Number of processors per site	2
$NumDataDisks$	Number of data disks per site	3
$NumLogDisks$	Number of log disks per site	1
$PageCPU$	CPU page processing time	5 ms
$PageDisk$	Disk page access time	20 ms
$BufHit$	Probability of a buffer hit	0.1
$MsgCPU$	Message send / receive time	5 ms
$MinHF$	Minimum Health Factor (for PROMPT)	0

6.2 Workload Model

At each site, transactions arrive in an independent Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline. All transactions have the "single master—multiple cohort" structure described in Section 3. Transactions in a distributed system can execute in either *sequential* or *parallel* fashion. The distinction is that the data phases of cohorts in a sequential transaction occur one after another, whereas for cohorts in a parallel transaction the data phases occur concurrently. We consider both types of transactions in our study—the parameter $TransType$ specifies whether the transaction execution is sequential or parallel.

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 different sites chosen at random from the remaining $NumSites - 1$ sites. 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 uniformly (without replacement) from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$.[10] A transaction that is aborted due to a data conflict is immediately restarted and makes the same data accesses as its original incarnation.

6.3 Deadline Assignment

Upon arrival, each transaction T is assigned a deadline using the formula $D_T = A_T + SF * R_T$, where D_T, A_T, and R_T are its deadline, arrival time, and resource time, respectively, while SF is a slack factor. The resource time is the total service time at the resources that the transaction requires for its execution. The $SlackFactor$ parameter is a constant that provides control over the tightness/slackness of transaction deadlines.

There are two issues related to the resource time computation: First, 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. Second, while the workload generator utilizes information about transaction resource requirements in assigning deadlines, the RTDBS system itself has *no access* to such information, since this knowledge is usually hard to come by in practical environments.

10. A page write operation is always preceded by a read for the same page; that is, there are no "blind writes" [3].

6.4 System Model

The physical resources at each site consist of $NumCPUs$ processors, $NumDataDisks$ data disks and $NumLogDisks$ log disks. The data disks store the data pages while the log disks store the transaction log records. There is a single common queue for the processors and the service discipline is Preemptive Resume, with preemptions 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 ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. When a transaction makes a request for accessing a data page, the data page may be found in the buffer pool, or it may have to be accessed from the disk. The *BufHit* parameter gives the probability of finding a requested page already resident in the buffer pool.

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 overheads of message transfer, given by the $MsgCPU$ parameter, are taken into account at both the sending and the receiving sites. In our simulations, all requests for the CPU, whether for message processing or data processing, are served in priority order.

Finally, specifically for the PROMPT protocol, the minimum health factor value is determined by the $MinHF$ parameter.

6.5 Transaction Execution

When a transaction is initiated, it is assigned the set of sites where it has to execute and the data pages that it has to access at each of these sites. The master is then started up at

the originating site, which in turn forks off a local cohort and sends messages to initiate each of its cohorts at the remote participating sites.

Based on the transaction type, the cohorts execute either in parallel or in sequence. Each cohort makes a series of read and update accesses. A read access involves a concurrency control request to obtain access, followed by a disk I/O to read the page if not already in the buffer pool, followed by a period of CPU usage for processing the page. Update requests are handled similarly, except for their disk I/O—the writing of the data pages takes place asynchronously after the transaction has committed.[11] We assume sufficient buffer space to allow the retention of data updates until commit time.

If the transaction's deadline expires at any time during its data processing, it is immediately killed. Otherwise, the commit protocol is initiated when the transaction has completed its data processing. If the transaction's deadline expires before the master has written the global decision log record, the transaction is killed (as per the firm deadline semantics defined in Section 4). On the other hand, if the master writes the commit decision log record before the expiry of the deadline at its site, the transaction is eventually committed at all of its execution sites.

6.6 Priority Assignment

For simplicity, we assume here that all transactions have the same "criticality" or "value" [41].[12] Therefore, the goal of the priority assignment is to minimize the *number* of killed transactions. In our model, all cohorts inherit their 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. Messages also retain their sending transaction's priority.

The only exception to the above priority rule is in the PIC commit protocol, described in Section 9. In this protocol, a low priority transaction that blocks a high priority transaction *inherits* the priority of the high priority transaction.

The transaction priority assignment used in all of the experiments described here is the widely used *Earliest Deadline First* (EDF) policy [32], wherein transactions with earlier deadlines have higher priority than transactions with later deadlines.

6.7 Concurrency Control

For transaction concurrency control, we use an extended version of the centralized *2PL High Priority* (2PL-HP) protocol proposed in [1].[13] The basic 2PL-HP protocol, which is based on the classical strict two-phase locking protocol (2PL) [14], operates as follows: When a cohort requests a lock on a data item that is held by one or more higher priority cohorts in a conflicting lock mode, the requesting cohort waits for the item to be released (the wait queue for a data item is managed in priority order). On the other hand, if the data item is held by only lower priority cohorts in a conflicting lock mode, the lower priority cohorts are aborted and the requesting cohort is granted the desired lock. Note that if priorities are assigned uniquely (as is usually the case in RTDBS), 2PL-HP is inherently deadlock-free. Finally, a new reader can join a group of lock-holding readers only if its priority is higher than that of all the writers waiting for the lock.

The extensions that we have made to the above basic protocol for our distributed real-time environment are the following: First, on receipt of the PREPARE message from the master, a cohort releases all its read locks but retains its update locks until it receives and implements the global decision from the master. Second, a cohort that is in the prepared state cannot be aborted, irrespective of its priority. Third, in the PROMPT-based commit protocols, cohorts in the data phase are allowed optimistic access to data held by conflicting prepared cohorts.

6.8 Logging

With regard to logging costs, we explicitly model only *forced* log writes, since they are done synchronously and suspend transaction operation until their completion. The cost of each forced log write is the same as the cost of writing a data page to the disk. The overheads of flushing the transaction log records related to *data processing* (i.e., WAL [16]), however, are not modeled. This is because these records are generated during the cohort's data phase and are therefore *independent* of the choice of commit protocol. We therefore do not expect their processing to affect the *relative* performance behavior of the commit protocols evaluated in our study.

6.9 Default Parameter Settings

The default settings used in our experiments for the workload and system parameters are listed in Table 1. They were chosen to be in accordance with those used in earlier studies (e.g. [6], [24]). While the absolute performance profiles of the commit protocols would, of course, change if alternative parameter settings are used, we expect that the *relative* performance of these protocols will remain qualitatively similar, since the model parameters are not protocol-specific. Further, these settings ensure significant levels of both resource contention (**RC**) and data contention (**DC**) in the system, thus helping to bring out the performance differences between the various commit protocols.[14]

6.10 Performance Metric

The performance metric in all of our experiments is **KillPercent**, which is the steady-state percentage of input transactions that are killed, i.e., the percentage of input transactions that the system is *unable* to complete before their deadlines.[15] KillPercent values in the range of zero to

11. Update-locks are acquired when a data page intended for modification is first read, i.e., lock upgrades are not modeled.

12. For applications with transactions of varying criticalities, the value-cognizant priority assignment mechanisms proposed in the literature (e.g., [25]) can be utilized.

13. The problem of inaccessibility to prepared data does not arise with optimistic CC protocols since they permit unrestricted reads. However, open problems remain with respect to integrating optimistic schemes in practical systems [23], [33].

14. The contention levels are assessed by measuring the CPU and disk utilizations and the data conflict frequencies.

15. Only statistically significant differences are discussed here. All the KillPercent values shown have relative half-widths about the mean of less than 10 percent at the 90 percent confidence level—after elimination of the initial transient, each experiment was run until at least 20,000 transactions were processed by the system.

20 percent are taken to represent system performance under "normal" loads, while values beyond this represent "heavy" load performance.[16] A long-term operating region where the KillPercent value is high is obviously unrealistic for a viable DRTDBS. Exercising the system to high KillPercent levels, however, provides information on the response of the algorithms to brief periods of stress loading.

The simulator was instrumented to generate several other statistics, including resource utilizations, number of transaction restarts, number of messages and force-writes, etc. These secondary measures help to explain the KillPercent behavior of the commit protocols under various workloads and system conditions. For the PROMPT protocol, specifically, we also measure its *borrow factor*, that is, the average number of data items (pages) borrowed per transaction, and the *success ratio*, that is, the fraction of times that a borrowing was successful in that the lender committed after loaning the data.

6.11 Baseline Protocols

To help isolate and understand the effects of distribution and atomicity on KillPercent performance, and to serve as a basis for comparison, we have also simulated the performance for two additional scenarios: CENT and DPCC, below described.

In **CENT** (Centralized), a *centralized* database system that is equivalent (in terms of overall database size and number of physical resources) to the distributed database system is modeled. Messages are obviously not required here and commit processing only requires writing a *single* decision log record (force-write if the decision is to `commit`). Modeling this scenario helps to isolate the overall effect of distribution on KillPercent performance.

In **DPCC** (Distributed Processing, Centralized Commit), data processing is executed in the normal distributed fashion, that is, involving messages. The *commit* processing, however, is like that of a centralized system, requiring only the 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 KillPercent performance (as opposed to CENT, which eliminates the entire effect of distributed processing).

7 Experiments and Results

Using the firm-deadline DRTDBS model described in the previous section, we conducted an extensive set of simulation experiments comparing the real-time performance of the 2PC, PA, PC, 3PC, and PROMPT commit protocols. In this section, we present the results of a representative set of experiments (the complete set is available in [18]).

7.1 Experiment 1: Resource and Data Contention

Our first experiment was conducted using the default settings for all model parameters (Table 1), resulting in significant levels of both resource contention (RC) and data contention (DC). Here, each transaction executes in a *sequential* fashion at three sites, accessing and updating an average of six pages at each site. Each site has two CPUs, three data disks and one log disk. For this environment, Figs. 1a and 1b show the KillPercent 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 a transaction arrival rate of two per second at each site, the centralized system misses less than five percent of the deadlines whereas 2PC and 3PC miss in excess of 25 percent. This difference highlights the extent to which a conventional implementation of distributed commit processing can affect the 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 (i.e., aborted) the performance of both protocols is essentially the same. Overall, it means that, in the real-time domain, the price paid during regular 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. The reason for this is that performance in a firm-deadline RTDBS is measured in *Boolean* terms of meeting or missing the deadline. So, although PC and PA reduce overheads under commit and abort conditions, respectively, all that happens is that the resources made available by this reduction only allow 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, normalized to the number of committed transactions, shown in Figs. 1c and 1d. In these figures, we see that PC has significantly lower overheads at normal loads (when commits are more), while PA has significantly lower overheads at heavy loads (when aborts 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 (the additional `collecting` log record) than 2PC for aborts.

Finally, turning to our new protocol, PROMPT, 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 1) the optimistic access of uncommitted prepared data which allows transactions to progress faster through the system, and 2) the Active Abort policy. The former effect is quantified in Fig. 1e, which plots PROMPT's borrow factor—this graph clearly shows that borrowing is significant, especially in the low to medium loading range. For

16. Heavy load may arise due to a variety of factors: increased transaction arrival rate, more stringent time constraints, etc.

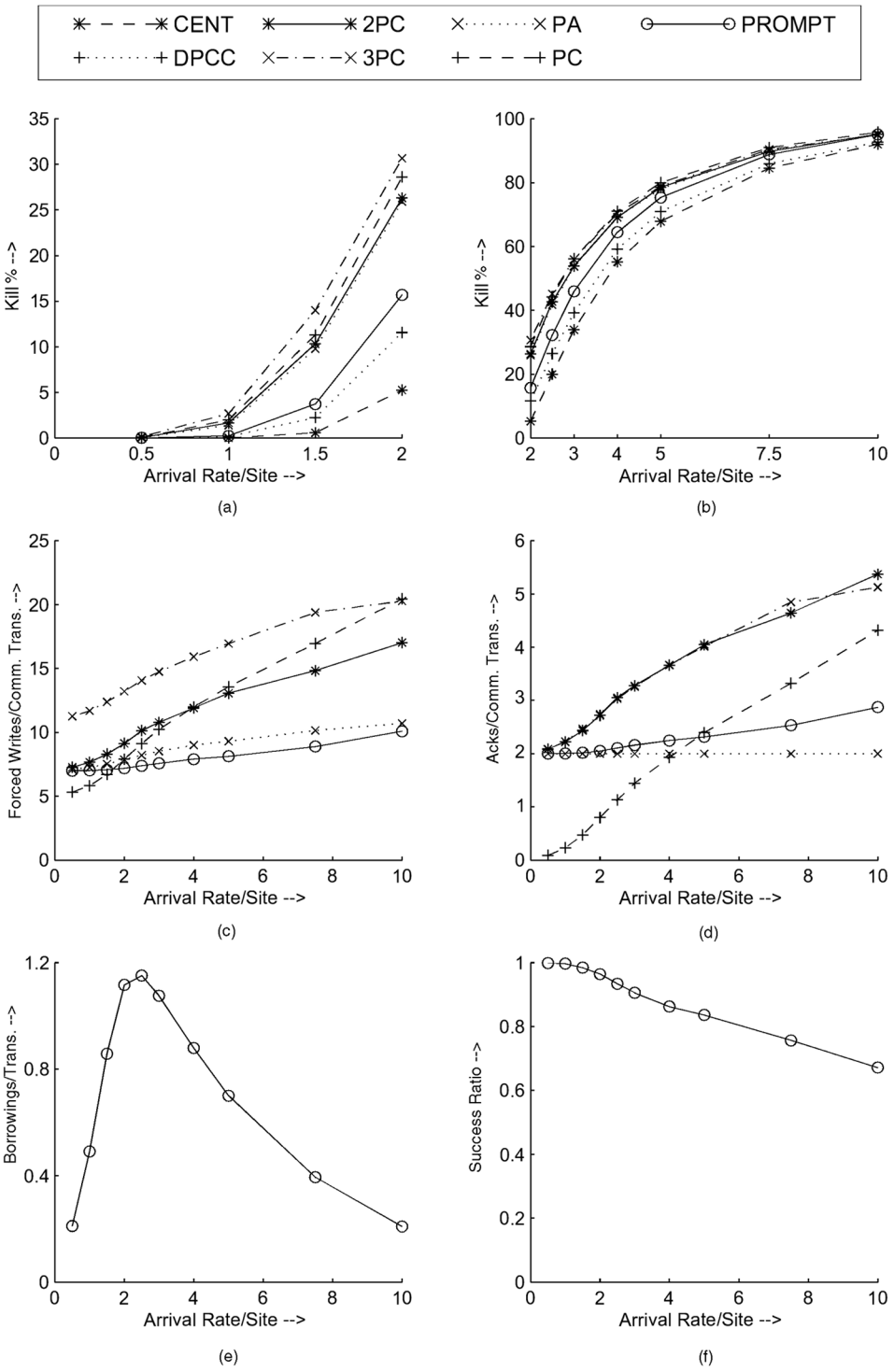


Fig. 1. Sequential Transactions (RC+DC). (a) KillPercent (normal load). (b) KillPercent (heavy load). (c) Forced writes (normalized). (d) Acks (normalized). (e) PROMPT borrow factor. (f) PROMPT success ratio.

example, at an arrival rate of two trans/sec, each transaction on average borrows approximately one page. At high loads, however, only a few transactions are able to make it to the commit processing phase and correspondingly there are very few lenders, leaving little opportunity for the optimistic feature to come into play, as indicated by the dip in the borrow factor.

The Silent Kill optimization (not sending abort messages for kill-induced aborts), on the other hand, gives only a very minor improvement in performance. At low loads this is because transaction kills are few in number 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

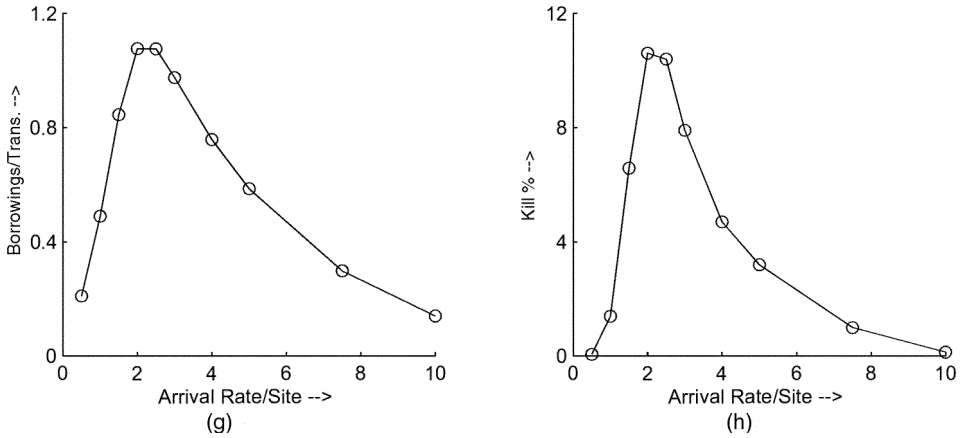


Fig. 1. Sequential Transactions (RC+DC) (continued). (g) PROMPT successful borrow factor. (h) PROMPT Kill reduction.

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 accessed by a transaction before being aborted—it was significantly more when Silent Kill was included.

As part of this experiment, we also wanted to quantify the degree to which the PROMPT protocol's optimism about accessing uncommitted data was well-founded—that is, is PROMPT safe or foolhardy? To evaluate this, we measured the success ratio—this statistic, shown in Fig. 1f, clearly indicates 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. That is, many of the lenders turn out to be "unhealthy." Note that PROMPT's *Healthy Lending* feature, which was intended to address this problem, *did not come into play*, since $MinHF$, the minimum health factor, was set to zero in this experiment—we return to this issue in Experiment 6.

To conclude the discussion of PROMPT's performance, in Fig. 1g we show the *successful* borrow factor, that is, the combination (through multiplication) of Figs. 1e and 1f, and in Fig. 1h we graph the (absolute) *reduction* in Kill Percent achieved by PROMPT as compared to 2PC. Note the close similarity between the shapes of these two graphs, conclusively demonstrating that allowing for borrowing is what results in PROMPT's better performance.

Last, another interesting point to note is the following: In Figs. 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 processing is considerably more than that of distributed data processing for the standard commit protocols, and that the PROMPT protocol helps to significantly reduce this impact. *These results clearly highlight the necessity for designing high-performance real-time commit protocols.*

7.2 Experiment 2: Pure Data Contention

The goal of our next experiment was to *isolate* the influence of *data contention* (DC) on the performance of the commit protocols.[17] Modeling this scenario is important because while resource contention can usually be addressed by purchasing more and/or faster resources, there do not exist any equally simple mechanisms to reduce data contention. In this sense, data contention is a more "fundamental" determinant of database system performance. Further, while abundant resources may not be typical 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.

For this experiment, the physical resources (CPUs and disks) were made "infinite," that is, there is no queueing for these resources [2]. The other parameter values are the same as those used in Experiment 1. The KillPercent performance results for this system configuration are presented in Figs. 2a and 2b for the normal load and heavy load conditions, respectively, and PROMPT's supporting statistics are shown in Figs. 2c and 2d. We observe in these figures that the relative performance of the various protocols remains qualitatively similar to that seen for the RC+DC environment of the previous experiment. Quantitatively, however, the performance of the standard protocols relative to the baselines is markedly worse than before. This is because in the previous experiment, the considerable difference in overheads between CENT and 2PC, for example, was largely submerged due to the resource and data contention in the system having a predominant effect on transaction response times. In the current experiment, however, the commit phase occupies a *bigger proportion* of the overall transaction response time, and therefore, the overheads of 2PC are felt to a greater extent. Similarly, 3PC performs significantly worse than 2PC due to its considerable extra overheads.

Moving on to PROMPT, we observe that it exhibits much better performance as compared to the standard algorithms

17. The corresponding experiment, pure RC, is not considered here since our goal of reducing prepared data inaccessibility ceases to be an issue in the pure RC environment.

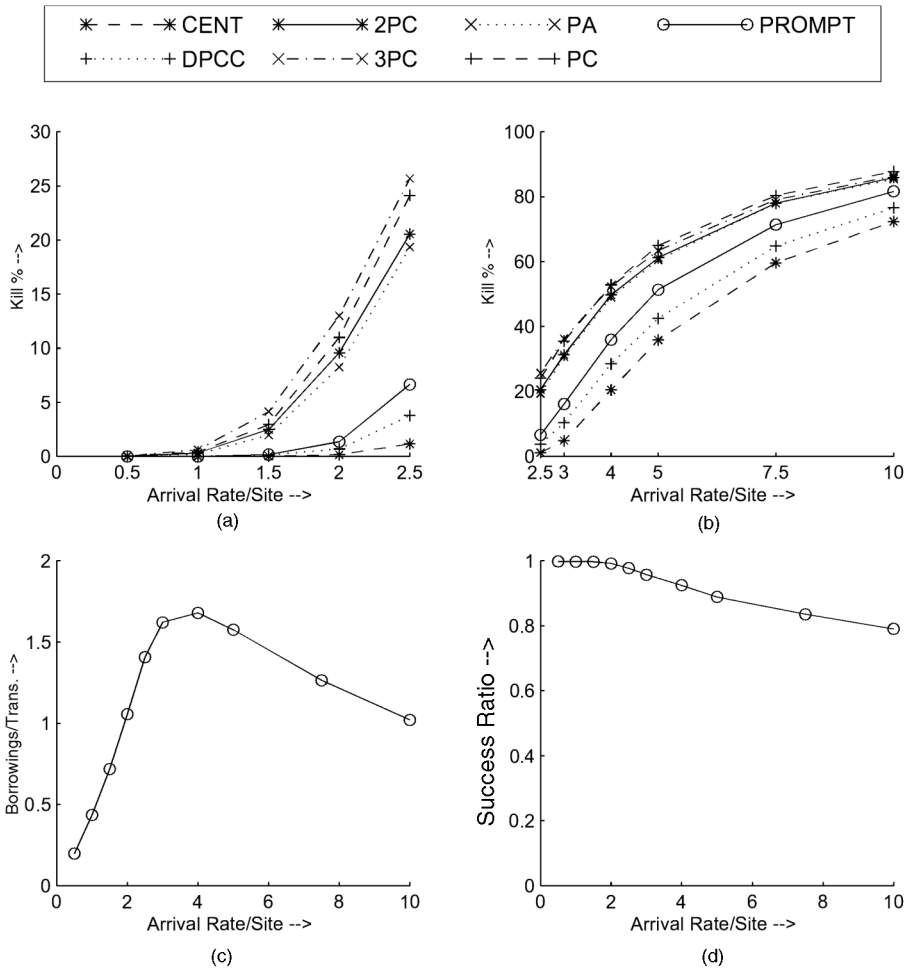


Fig. 2. Sequential Transactions (Pure DC). (a) KillPercent (normal load). (b) KillPercent (heavy load). (c) PROMPT borrow factor. (d) PROMPT success ratio.

over the entire loading range. This is explained in Fig. 2c, which shows the borrow factor being even higher than that of the RC+DC case. Moreover, the success ratio is also better than that of the RC+DC case, not going below 75 percent even at the highest loading levels (Fig. 2d).

The above experiment indicates that when resource contention is reduced by upgrading the physical resources, it is even more important to employ protocols such as PROMPT to mitigate the ill effects of data contention.

7.3 Experiment 3: Fast Network Interface

In the previous experiments, the cost for sending and receiving messages modeled a system with a relatively slow network interface ($MsgCpu = 5\ ms$). We conducted another experiment wherein the network interface was faster by a factor of five, that is, where $MsgCpu = 1\ ms$. The results of this experiment are shown in Figs. 3a and 3b for the RC+DC and Pure DC environments, respectively.[18] In these figures, we see that the performance of all the protocols comes closer together (as compared to that seen in Experiments 1 and 2). This improved behavior of the protocols is only to be expected, since low message costs effectively eliminate the effect of a significant fraction of the overheads involved in each protocol. Their relative performance, however, remains the same. Note also that the PROMPT 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.

In summary, this experiment shows that adopting the PROMPT principle can be of value even with very high-speed network interfaces, because faster message processing does not necessarily eliminate the *data contention* bottleneck.

7.4 Experiment 4: 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 *six* sites. The $CohortSize$ in this experiment was reduced from six pages to three pages in order to keep the average transaction length equal to that of the previous experiments. In addition, a higher value of $SlackFactor = 6$ was used to cater to the increase in response times caused by the increased distribution.

18. The default parameter settings for the RC+DC and Pure DC scenarios in this experiment, as well as in the following experiments, are the same as those used in Experiment 1 and Experiment 2, respectively.

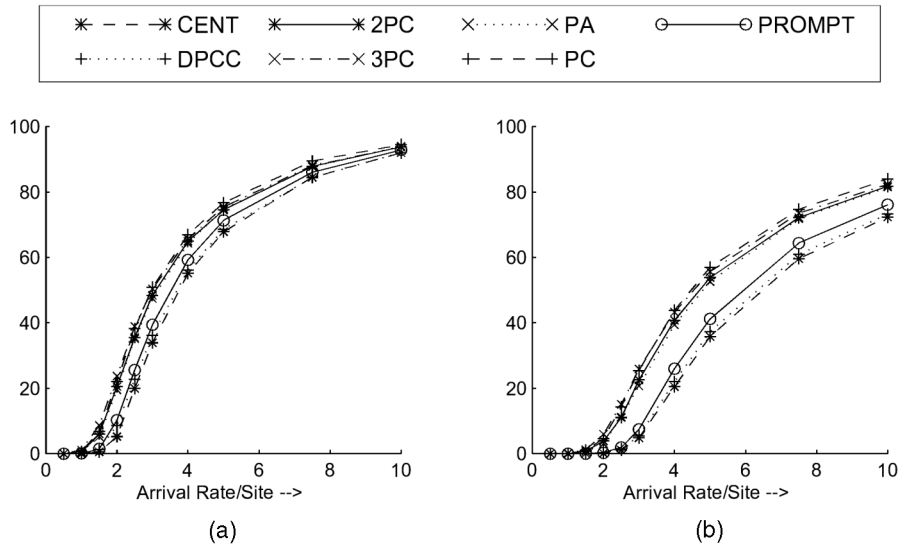


Fig. 3. Fast Network Interface. (a) KillPercent (RC + DC). (b) KillPercent (Pure DC).

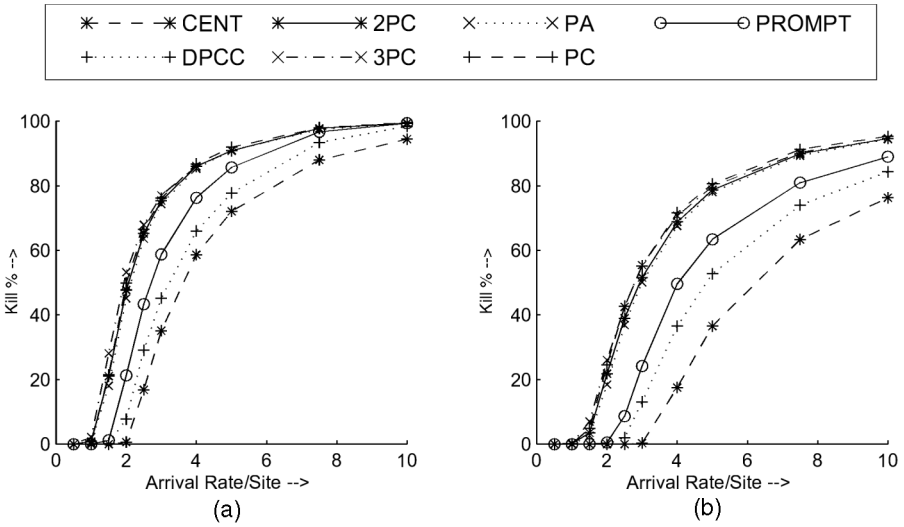


Fig. 4. Higher Degree of Distribution. (a) KillPercent (RC + DC). (b) KillPercent (Pure DC).

The results of this experiment are shown in Figs. 4a and 4b for the RC+DC and Pure DC environments, 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 overheads are larger in this experiment. The relative performance of the protocols, however, remains qualitatively the same with PROMPT continuing to perform better than the standard protocols.

7.5 Experiment 5: Parallel Execution of Cohorts

In all of the previous experiments, the cohorts of each transaction executed in *sequence*. We also conducted similar experiments for transaction workloads with *parallel* cohort execution and we report on those results here. The performance under the RC+DC environment is shown in Figs. 5a, 5b, and 5c, and we observe here that the general trends are like those seen for sequential transactions in Experiment 1. In particular, the effect of distributed commit processing on the KillPercent performance remains considerably more than that of distributed data processing. But, there are also a *few changes*, as below described.

First, we observe that the differences between the performance of CENT and that of 2PC and 3PC have *increased* for parallel transactions as compared to that for sequential transactions. The reason for this is that the parallel execution of the cohorts reduces the transaction response time, *but the time required for the commit processing remains the same*. Therefore, the effect of the commit phase on overall transaction response time is significantly more.

Second, although PROMPT continues to perform the best under normal loads, its effect on the KillPercent performance is partially *reduced* as compared to that for sequential transactions. This is because the Active Abort policy, which had significant impact in the sequential environment, is less useful in the parallel domain. The reason for its reduced utility is that due to cohorts executing in parallel, the duration of the wait phase of the cohorts is shorter, and so

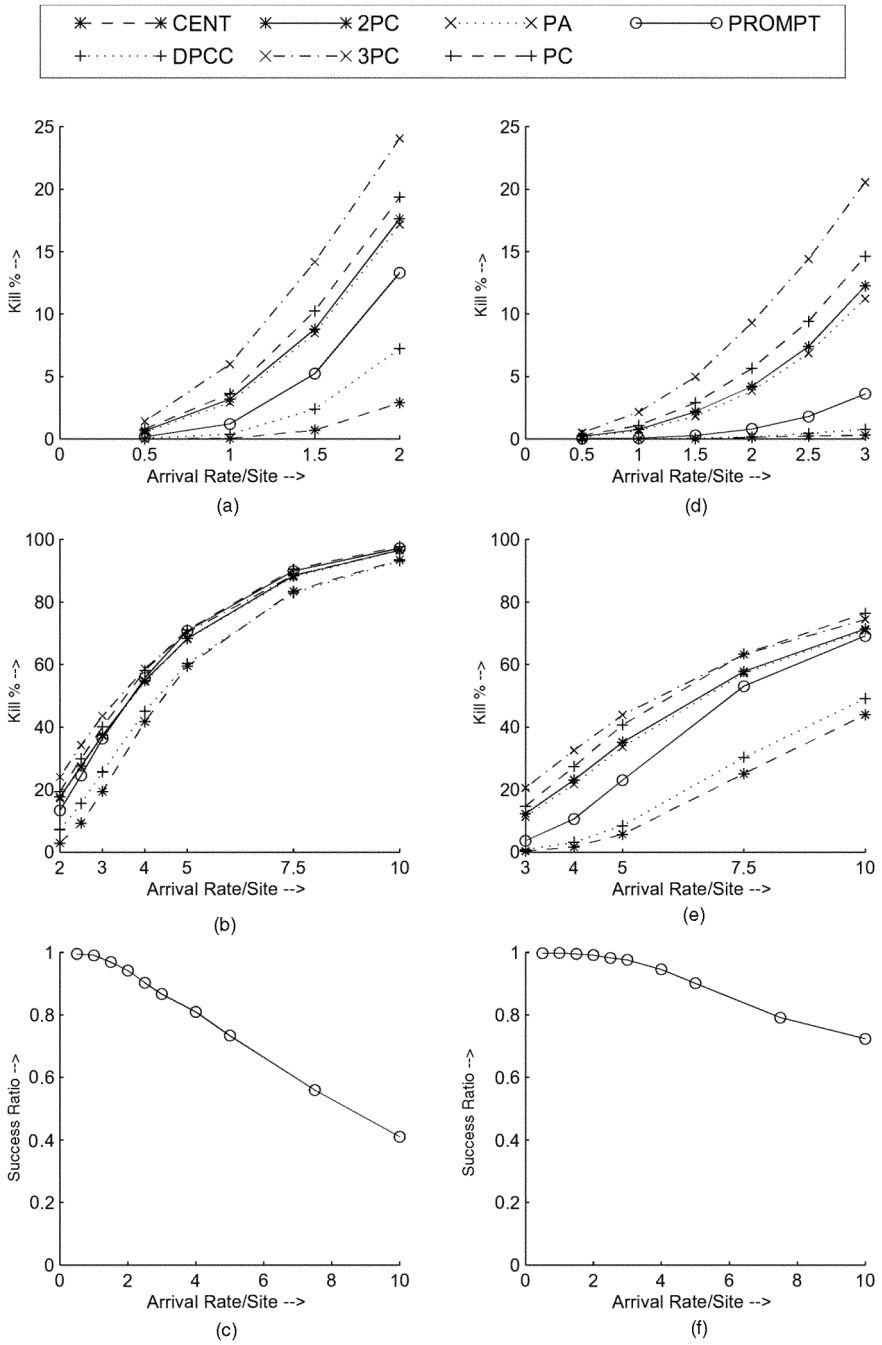


Fig. 5. Parallel Transactions—RC+DC and Pure DC. (a) Normal load (RC + DC). (b) Heavy load (RC + DC). (c) PROMPT success ratio (RC + DC). (d) Normal load (Pure DC). (e) Heavy load (Pure DC). (f) PROMPT success ratio (Pure DC).

there are much fewer chances of a cohort aborting during the wait phase, which is when the Active Abort policy mostly comes into play for the parallel case.

Third, the performance of PROMPT under heavy loads is marginally *worse* than that of 2PC, whereas in the sequential case PROMPT was always better than or matched 2PC. This is explained by comparing PROMPT's success ratios in Figs. 1f and 5c, which clearly indicate that the heavy-load degradation in PROMPT's success ratio is much more under parallel workloads than under sequential workloads. The reason for this is the following: The data contention level is *smaller* with parallel execution than with sequential execution, since locks are held for shorter times on average (this was also confirmed by PROMPT's borrow factor, which was about 30 percent less than in its sequential counterpart). Cohorts are therefore able to obtain the necessary locks sooner than in the sequential case, and hence, those that are aborted due to deadline expiry tend to make further progress than in the sequential case. This leads to a proportionally larger group of cohorts finishing their work closer to the deadline, resulting in a worse success ratio due to more "unhealthy lenders."

When the above experiment was conducted for the Pure DC environment, we obtained Figs. 5d, 5e, and 5f. Interestingly, the performance of PROMPT in these figures

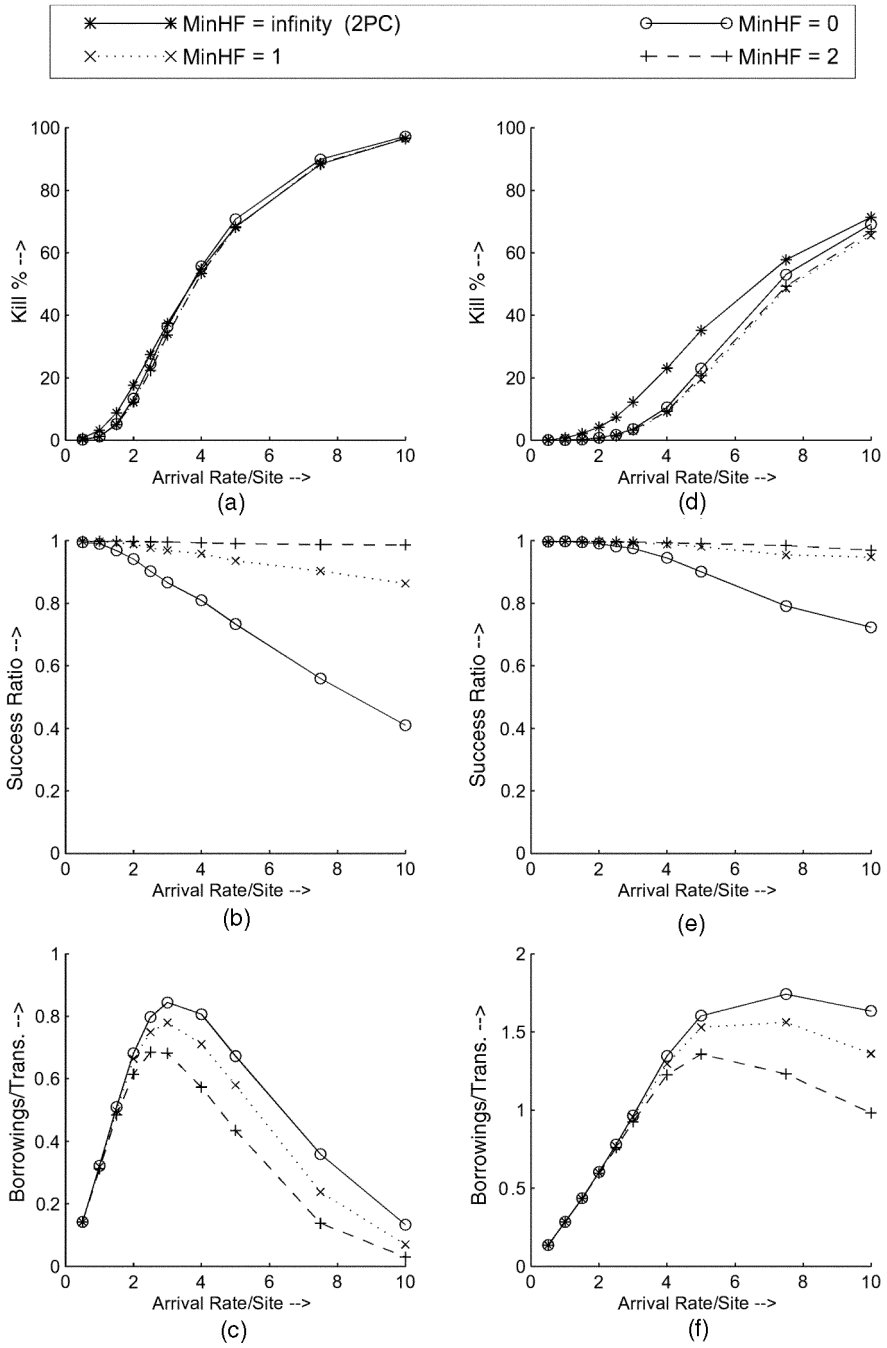


Fig. 6. PROMPT's Healthy Lending Feature—RC+DC and Pure DC. (a) KillPercent (RC + DC). (b) Success ratio (RC + DC). (c) Borrow factor (RC + DC). (d) KillPercent (Pure DC). (e) Success ratio (Pure DC). (f) Borrow factor (Pure DC).

does not show the deterioration observed in the RC+DC environment. The reason is that when DC is the main performance bottleneck, PROMPT, which primarily addresses DC, has the maximum impact. In this case, PROMPT's borrow factor and success ratio (5f) are quite high, resulting in its performance being considerably better than the standard commit protocols.

7.6 Experiment 6: Lending Restricted to Healthy Lenders

Results for the parallel cohort execution workloads of the previous experiment may seem to suggest that PROMPT may not be the best approach for *all* workloads—however, as we will now show, the reduced success ratio of PROMPT can be easily rectified by appropriately setting the $MinHF$ parameter (which was zero in the experiments described thus far). After this optimization is incorporated, PROMPT provides better performance than all the other classical protocols.

A range of $MinHF$ values were considered in our experiments, including $MinHF = 0$, which corresponds to the PROMPT protocol evaluated in the previous experiments, $MinHF = 1$, $MinHF = 2$, and $MinHF = \infty$, which is equivalent to the 2PC protocol. The results for these various settings are shown in Figs. 6a, 6b, and 6c, and

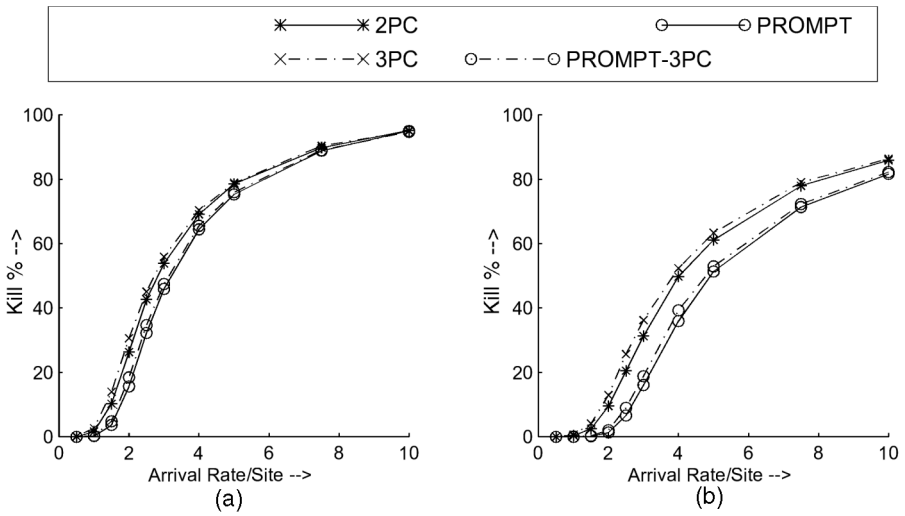


Fig. 7. Nonblocking PROMPT. (a) KillPercent (RC +DC). (b) KillPercent (Pure DC).

in Figs. 6d, 6e, and 6f for the workloads of the previous experiment.

In the KillPercent graphs (Figs. 6a and 6d), we see that $MinHF = 1$ is, in general, slightly better than $MinHF = 0$, and especially so under heavy loads in the Pure DC environment. Further, note that its performance matches that of 2PC in the heavy load region, thereby correcting the problem observed in the previous experiment. The reason for the performance improvement is evident in Figs. 6b and 6e, where the success ratio of $MinHF = 1$ is seen to be considerably higher than that of $MinHF = 0$. Further, from Figs. 6c and 6f, which present the borrow factors, it is clear that $MinHF = 1$ is "efficient" in that it restricts borrowing only in the heavy load region, but not in the normal load region where optimism is almost always a good idea. That is, *healthy lenders are very rarely tagged as unhealthy*.

The last observation deals with the "completeness" and "precision" of borrowing: 1) "Do we always borrow when the borrowing is going to be successful?", and 2) "Is what we borrow always going to be successful?" PROMPT with $MinHF = 0$ is complete by design, since it does not miss out on any successful borrowing opportunities; but because it may also borrow without success, it is not precise. The experimental results show that $MinHF = 1$, in contrast, is quite precise while sacrificing very little completeness in achieving this goal.

Turning our attention to $MinHF = 2$, we observe that the performance of PROMPT for $MinHF = 1$ and for $MinHF = 2$ is almost identical, indicating that a health threshold of unity is sufficient to filter out the transactions vulnerable to deadline kill in the commit phase. The reason for this is that, by virtue of the EDF priority policy used in our experiments, transactions that are close to their deadlines have the highest priority at the physical resources, and therefore the minimum time required to carry out the commit processing is actually sufficient for them to complete this operation.

7.7 PROMPT Combinations

While presenting the PROMPT protocol in Section 5, we had mentioned that one of the attractive features of PROMPT is its ability to combine with other optimizations. We now evaluate the performance benefits possible from such combinations.

7.7.1 Experiment 7: PROMPT-PA and PROMPT-PC

When we analyzed the performance effects of adding the PA and PC optimizations to PROMPT (graphs available in [18]), we observed that just as PA and PC provided little improvement over the performance of standard 2PC, here also they provide no tangible benefits to the performance of the PROMPT protocol, and for the same reasons. While PROMPT-PA is very slightly better than basic PROMPT, PROMPT-PC performs worse than basic PROMPT under heavy loads, especially when data contention is the primary performance bottleneck.

7.7.2 Experiment 8: Nonblocking PROMPT

In our second experiment, we evaluated the effect of combining PROMPT with 3PC, resulting in a *nonblocking* version of PROMPT. The results of this experiment are shown in Figs. 7a and 7b for the RC+DC and Pure DC environments, respectively.

We observe in these figures that the performance of PROMPT-3PC is not only superior to that of 3PC, *but also significantly better than that of 2PC*. In fact, the performance of PROMPT-3PC is close to that of the basic PROMPT itself. The reason for this superior performance of PROMPT-3PC is that the optimistic feature has more effect on 3PC than on 2PC due to the larger commit processing phase of 3PC (as mentioned earlier in Section 5).

These results indicate that, in the real-time domain, the nonblocking functionality, which is extremely useful in the event of failures, can be purchased for a small increase in the KillPercent value.

7.8 Summary of the Results

The set of experiments discussed above, 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 classical commit protocols generally 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. Our new protocol, PROMPT, provides significantly improved performance over the standard algorithms. 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 PROMPT's optimism about uncommitted data is justified, especially under normal loads. The other optimizations of Silent Kill and Presumed Commit/Abort, however, had comparatively little beneficial effect.
4. The results obtained for *parallel* distributed transaction workloads were generally similar to those observed for the sequential-transaction environment. But, performance improvement due to the PROMPT protocol was not as high for two reasons: 1) Its Active Abort policy, which contributed significantly to improved performance in the sequential environment, has reduced effect in the parallel domain. 2) Parallel execution, by virtue of reducing the data contention, results in an increased number of unhealthy lenders. These problems were rectified, however, by appropriately setting the $MinHF$ parameter.
5. We have also found that a health threshold of $MinHF = 1$ provides good performance for a wide range of workloads and system configurations, that is, this setting is *robust*.[19]
6. Experiments combining PROMPT 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, the above results show that the PROMPT protocol may be an attractive candidate for use in DRTDBS commit processing. In the following section, we discuss a variety of options in PROMPT's design.

19. Even otherwise, it should be easy to set $MinHF$ during the inevitable initial system/application tuning process.

8 Alternative Design Choices in PROMPT

We first present and evaluate Shadow-PROMPT, which incorporates a special mechanism to reduce the adverse effects in those cases where lending optimism turns out to be misplaced. Then, we present and evaluate FullLending-PROMPT, which incorporates a mechanism intended to increase the lending opportunities.

8.1 The Shadow-PROMPT Protocol and Its Performance

The **Shadow-PROMPT** protocol combines PROMPT with the "shadow transaction" approach proposed in [4]. In this combined technique, a cohort *forks* off a replica of the transaction, called a *shadow*, whenever it borrows a data page. The original incarnation of the transaction continues execution while the shadow transaction is blocked at the point of borrowing. If the lending transaction finally commits, the (original) borrowing cohort continues its ongoing execution and the shadow is discarded (since the optimistic borrowing proved to be a correct decision). Otherwise, if the lender aborts, the borrowing cohort is aborted and the shadow, which was blocked so far, is activated. Thus, the work done by the borrowing transaction prior to its borrowing is *never wasted*, even if the wrong borrowing choice is made. Therefore, if we *ignore the overheads of the shadow mechanism* (which may be significant in practice[20]), Shadow-PROMPT represents the *best on-line performance* that could be achieved using the optimistic lending approach.

We conducted experiments to evaluate the performance of the Shadow-PROMPT protocol. In these experiments, we modeled a zero-overhead Shadow-PROMPT protocol. In addition, as in [4], at most one shadow (for each cohort) is allowed to exist at any given time. The first shadow is created at the time of the first borrowing—creation of another shadow is allowed only if the original cohort aborts and the current shadow resumes its execution replacing the original cohort.

Our experimental results showed the performance of PROMPT (with $MinHF$ set to 1) and Shadow-PROMPT to be so close that they are difficult to distinguish visually (the graphs are available in [18]). In fact, in all our experiments, the performance difference between Shadow-PROMPT and PROMPT was never more than *two percent*. This means that Healthy Lending can provide performance gains similar to that of the ideal Shadow mechanism without attracting the associated implementation overheads and difficulties.[21] That is, PROMPT is *efficient* in its use of the optimistic premise, since this premise holds most of the time, as expected, especially in conjunction with the Healthy Lending optimization.

8.2 The FL-PROMPT Protocol and Its Performance

A situation may arise in PROMPT wherein the borrower cohort finishes its execution before the lender cohort receives the final decision from its master. In such a case,

20. For example, it is mentioned in [4] that for a single-shadow per transaction environment, a 15 percent increase in number of messages was observed.

21. A significant reworking of the transaction management system is required to support the shadow concept.

the borrower cohort is put "on the shelf" waiting for the lender to complete and is not allowed to immediately send the WORKDONE message to its master (as described in Section 5). This can increase the response time of the transaction, especially for sequential execution, and may therefore result in more missed deadlines. Allowing the borrower to immediately send the WORKDONE message in the normal manner, however, is not a solution because it would lead to the problem of *cascading aborts*, since a borrower at one site could become a lender at other sites.

To address the above issue, we designed another variant of PROMPT, called **Full-Lending PROMPT** (FL-PROMPT), which operates in the following manner: When sending the WORKDONE message to the master, the borrower—which is waiting for its lending to complete—also sends an extra bit, called the "borrower bit," piggybacked on this message. The master, while subsequently sending the PREPARE message to the cohorts, passes on this information to the other sibling cohorts. Cohorts receiving this bit do not lend their data, thereby ensuring that borrower transactions cannot simultaneously be lenders (the condition necessary to ensure the absence of cascading aborts). Note that while this mechanism ensures that the aborts do not cascade, it does not allow the borrower transaction to become a lender *even after the borrowing is over*. This problem is overcome by having such borrowers send a BORROW-OVER message to the master if the lender subsequently commits and the borrower cohort has not yet received a PREPARE message from the master. This message to the master effectively *invalidates* the borrower bit sent earlier.[22] A related issue arising out of the distributed nature of execution is the situation wherein the BORROW-OVER and PREPARE messages "cross" each other on the network—in this situation, the master ignores the BORROW-OVER message.

Our experiments to evaluate the performance of the FL-PROMPT protocol showed that under RC+DC conditions, its performance was virtually identical to that of basic PROMPT (the graphs are available in [18]). This is because the performance gains due to FL-PROMPT's full lending feature are largely offset by the extra CPU message processing overheads arising out of the BORROW-OVER messages.

Even in Pure DC environments, where the CPU overheads are not a factor, the difference between FL-PROMPT and PROMPT turned out to be negligible for most of the loading range. This is because the situation when a "borrower bit" is sent with the WORKDONE message does not occur frequently in this environment. Only a marginal difference at light loads was observed where FL-PROMPT performed slightly better because it allows more transactions to become lenders (by using the BORROW-OVER message).

Thus, even though Full Lending appears to be a useful idea in principle, its effects were not observed in our experiments. However, for priority assignment policies that lead to longer commit phase durations as compared with those occuring with EDF, it may have more impact on performance.

22. If multiple cohorts concurrently send the borrow bit, the master waits till it receives the BORROW-OVER message from all such borrowers before turning lending on again.

9 Priority Inheritance: An Alternative to Prompt—and Its Performance

As discussed earlier, PROMPT addresses the priority inversion problem in the commit phase by allowing transactions to access uncommitted prepared data. A plausible *alternative* approach is the well-known **priority inheritance (PI)** mechanism [36]. 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 positive feature of the PI approach is that it *does not* run the risk of transaction aborts, unlike the lending approach. Further, a study of PI in the context of (centralized) transaction concurrency control [13] sug–gested that priority inheritance is useful only if it occurs towards the end of the low priority 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.

Motivated by these considerations we now describe **Priority Inheritance Commit (PIC)**, a real-time commit protocol based on the PI approach. In the PIC protocol, when a high priority transaction is blocked due to the data locked by a low priority cohort in the prepared state, the latter inherits the priority of the former to expedite its commit processing. To propagate this inherited priority to the master and the sibling cohorts, the priority inheriting cohort sends a PRIORITY-INHERIT message to the master. The master, in turn, sends this message to all other cohorts. After the master or a cohort receives a PRIORITY-INHERIT message, all further processing related to the transaction at that site (processing of the messages, writing log records, etc.) is carried out at the inherited priority.[23] Our experiments to evaluate the performance of the PIC showed that the performance of PIC is *virtually identical* to that of 2PC (the graphs are available in [18]). The reason for this behavior is the following: PI comes into play only when a high priority transaction is blocked by a low priority prepared cohort, which means that this cohort has already sent the YES vote to its master. Since it takes *two* message delays for dissemination of the priority inheritance information to the sibling cohorts, PIC expedites at most the processing of *only the decision message*. Further, even the minor advantage that may be obtained by PIC is partially offset by the extra overheads involved in processing the priority inheritance information messages.

Thus, PIC fails to provide any performance benefits over 2PC due to the delays that are *inherent* in distributed processing. Specifically, it is affected by the delay in the

23. For simplicity, the priority is not reverted to its old value if the high priority waiter is restarted.

dissemination of priority inheritance information to the sibling cohorts at remote sites.

10 Related Work

As mentioned in the Introduction, distributed real-time commit processing has received little attention in the literature and the only papers that we are aware of dealing with this issue are [8], [15], [46]. In this section, we briefly summarize these papers and contrast them with our study.

A *centralized timed* 2PC protocol is described in [8] that guarantees that the fate of a transaction (commit or abort) is known to all the cohorts before the expiry of the 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. Further, the protocol assumes that it is possible for the DRTDBS to guarantee allocation of resources for a duration of time within a given time interval. Finally, the protocol is predicated upon the knowledge of worst-case communication delays.

Our work is different from [8] in the following respects: First, their deadline semantics are different from ours (Section 4) in that 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 cohorts by the deadline, the transaction is killed. Thus, their primary concern is to ensure that all the participants of a transaction reach the decision before the expiry of the deadline, even at the cost of eventually killing more transactions. In our work, however, we have focused instead on increasing the number of transactions that complete before their deadlines expire, which is of primary concern in the "firm-deadline" application framework. Second, we do not assume any guarantees provided by the system for the services offered, whereas such guarantees are fundamental to the design of their protocol. Note that in a dynamic prioritized system, such guarantees are difficult to provide and, further, are generally not recommended, since it requires *preallocation* of resources, thereby running the risk of priority inversion.

A common theme of allowing individual sites to *unilaterally* commit is used in [15], [46]—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 executed to rectify the errors. While the compensation-based approach certainly appears to have the potential to improve timeliness, there are quite a few practical difficulties.

1. The standard notion of transaction atomicity is not supported—instead, a "relaxed" notion of atomicity [31] is provided.
2. The design of a compensating transaction is an application-specific task, since it is based on the application semantics.
3. 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.
4. "Real actions" [17] such as firing a weapon or dispensing money may not be compensatable at all [31].
5. From a performance viewpoint also, there are some difficulties:
   a. The execution of compensation transactions imposes an additional burden on the system;
   b. It is not clear how the database system should schedule compensation transactions relative to normal transactions.
6. 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 transaction atomicity mechanisms *without* relaxing the standard database correctness criterion.

11 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 require fundamental alterations of the standard distributed DBMS framework. We have instead taken a different approach of achieving high-performance by incorporating novel *protocol features*.

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 firm real-time setting. Subsequently, using a detailed simulation model of a firm-deadline DRTDBS, we evaluated the performance of a variety of standard commit protocols including 2PC, PA, PC and 3PC, with respect to the number of killed transactions. We also developed and evaluated a new commit protocol, PROMPT, that is designed specifically for the real-time environment and includes features such as controlled optimistic access to uncommitted data, Active Abort, Silent Kill, and Healthy-Lending. To the best of our knowledge, these are the first quantitative results in this area.

Our performance results showed that distributed commit processing can have considerably more effect than distributed data processing on the real-time performance. This highlights the need for developing commit protocols tuned to the real-time domain. The new PROMPT protocol, which was an attempt in this direction, provided significantly improved performance over the classical commit protocols. By appropriately setting the $MinHF$ parameter, it was possible to eliminate most of the unhealthy lendings, and with this optimization the difference between PROMPT and Shadow-PROMPT, which represents the best on-line usage of the optimistic lending approach, never exceeded *two percent*. Further, this high level of performance was achieved *without* incurring the implementation overheads and integration difficulties associated with the Shadow mechanism. Finally, PROMPT's conservatism in preventing borrowers from continuing to execute if their associated lenders had not yet received their decisions was addressed by incorporating an additional bit and message that informed the master about the borrowing

state and the completion of borrowing by a cohort. However, only minor KillPercent improvements were realized by this optimization.

We also evaluated the priority inheritance approach to addressing the priority inversion problem associated with prepared data. Our experiments showed that this approach provides virtually no performance benefits, primarily due to the intrinsic delays involved in disseminating information in a distributed system. It therefore does not appear to be a viable alternative to PROMPT for enhancing distributed commit processing performance.

In summary, we suggest that DRTDBS designers may find the PROMPT protocol to be a good choice for high-performance real-time distributed commit processing. Viewed in toto, PROMPT is a portable (can be used with many other optimizations), practical (easy to implement), high-performance (substantially improves real-time performance), and efficient (makes good use of the lending approach) distributed commit protocol.

ACKNOWLEDGMENTS

Preliminary versions of the work reported here were presented earlier in [19], [20], [22]. Prof. S. Seshadri of Indian Institute of Technology, Bombay, participated in the initial research efforts.

This work was supported in part by research grants from the Department of Science and Technology, Government of India, and from the U.S. National Science Foundation under grant IRI-9619588.

REFERENCES

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th Int'l Conf. Very Large Databases,* Aug. 1988.

[2] R. Agrawal, M. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. Database Systems,* vol. 12, no. 4, Dec. 1987.

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

[4] A. Bestavros and S. Braoudakis, "Timeliness Via Speculation for Real-Time Databases," *Proc. 15th Real-Time Systems Symp.,* Dec. 1994.

[5] B. Bhargava, ed. *Concurrency and Reliability in Distributed Database Systems.* Van Nostrand Reinhold, 1987.

[6] M. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. 14th Int'l Conf. Very Large Databases,* Aug. 1988.

[7] P. Chrysantis, G. Samaras, and Y. Al-Houmaily, "Recovery and Performance of Atomic Commit Processing in Distributed Database Systems," *Recovery Mechanisms in Database Systems.* V. Kumar and M. Hsu, eds. Prentice Hall, 1998.

[8] S. Davidson, I. Lee, and V. Wolfe, "A Protocol for Timed Atomic Commitment," *Proc. Ninth Int'l Conf. Distributed Computing Systems,* 1989.

[9] B. Lindsay, "Computation and Communication in R^*: A Distributed Database Manager," *ACM Trans. Computer Systems,* vol. 2, no. 1, 1984.

[10] B. Purimetla et al., "A Study of Distributed Real-Time Active Database Applications," *Proc. IEEE Workshop Parallel and Distributed Real-Time Systems,* 1993.

[11] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Trans. Database Systems,* vol. 17, no. 1, 1992.

[12] G. Samaras et al., "Two-Phase Commit Optimizations in a Commercial Distributed Environment," *J. Distributed and Parallel Databases,* vol. 3, no. 4, 1995.

[13] J. Huang et al., "On Using Priority Inheritance in Real-Time Databases," *Int'l J. Real-Time Systems,* vol. 4, no. 3, 1992.

[14] K. Eswaran et al., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM,* Nov. 1976.

[15] N. Soparkar et al., "Adaptive Commitment for Real-Time DistributedTransactions," Technical Report TR-92-15, Dept. of Computer Science, Univ. of Texas, Austin, 1992.

[16] J. Gray, "Notes on Database Operating Systems," *Operating Systems: An Advanced Course.* Springer Verlag, 1978.

[17] J. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. Seventh Int'l Conf. Very Large Databases,* 1981.

[18] R. Gupta, *Commit Processing in Distributed On-line and Real-Time Transaction Processing Systems,* master's thesis, SERC, Indian Inst. of Science, Mar. 1997.

[19] R. Gupta, J. Haritsa, "Commit Processing in Distributed Real-Time Database Systems," *Proc. Nat'l Conf. Software for Real-Time Systems,* Jan. 1996

[20] R. Gupta, J. Haritsa, and K. Ramamritham, "More Optimism about Real-Time Distributed CommitProcessing," *Proc. 18th IEEE Real-Time Systems Symp.,* Dec. 1997.

[21] R. Gupta, J. Haritsa, and K. Ramamritham, "Revisiting Commit Processing Performance in Distributed Database Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* May 1997.

[22] R. Gupta, J. Haritsa, K. Ramamritham, and S. Seshadri, "Commit Processing in Distributed Real-Time Database Systems," *Proc. 17th IEEE Real-Time Systems Symp.,* Dec. 1996.

[23] T. Haerder, "Observations on Optimistic Concurrency Control Schemes," *Information Systems,* vol. 9, no. 2, 1984.

[24] J. Haritsa, M. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *Int'l J. Real-Time Systems,* vol. 4, no. 3, 1992.

[25] J. Haritsa, M. Carey, and M. Livny, "Value-Based Scheduling in Real-Time Database Systems," *Int'l J. Very Large Databases,* vol. 2, no. 2, Apr. 1993.

[26] W. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys,* vol. 13, no. 2, June 1981.

[27] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems,* vol. 6, no. 2, June 1981.

[28] K. Lam, *Concurrency Control in Distributed Real-Time Database Systems,* PhD thesis, City Univ. of Hong Kong, Oct. 1994.

[29] B. Lampson and D. Lomet, "A New Presumed Commit Optimization for Two Phase Commit," *Proc. 19th Int'l Conf. Very Large Databases,* Aug. 1993.

[30] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," technical report, Xerox Palo Alto Research Center, 1976.

[31] E. Levy, H. Korth, and A. Silberschatz, "An Optimistic Commit Protocol for DistributedTransaction Management," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* May 1991.

[32] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a HardReal-Time Environment," *J. ACM,* vol. 20, no. 1, 1973.

[33] C. Mohan, "Less Optimism about Optimistic Concurrency Control," *Proc. Second Int'l Workshop RIDE: Transaction and Query Processing,* Feb. 1992.

[34] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R^* Distributed Database Management System," *ACM Trans. Database Systems,* vol. 11, no. 4, 1986.

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

[36] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Technical Report CMU-CS-87-181, Dept. of Computer Science, Carnegie Mellon Univ., 1987.

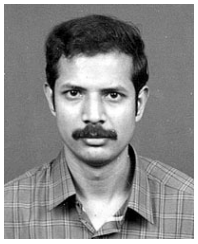
[37] L. Sha, R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record,* vol. 17, no. 1, Mar. 1988.

[38] D. Skeen, "Nonblocking Commit Protocols," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* June 1981.

[39] S. Son, "Real-Time Database Systems: A New Challenge," *Data Eng. Bulletin,* vol. 13, no. 4, Dec. 1990.

[40] P. Spiro, A. Joshi, and T. Rengarajan, "Designing an Optimized Transaction Commit Protocol," *Digital Technical J.,* vol. 3, no. 1, 1991.

[41] J. Stankovic and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record,* Mar. 1988.
[42] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres," *IEEE Trans. Software Eng.,* vol. 5, no. 3, 1979.
[43] O. Ulusoy, "Processing Real-Time Transactions in a Replicated-Database System," *Int'l J. Distributed and Parallel Databases,* vol. 2, no. 4, 1994.
[44] O. Ulusoy, "Research Issues in Real-Time Database Systems," Technial Report BU-CEIS-94-32, Dept. of Computer Eng. and Information Science, Bilkent Univ., Turkey, 1994.
[45] O. Ulusoy and G. Belford, "Real-Time Lock-based Concurrency Control in a Distributed Database System," *Proc. 12th Int'l Conf. Distributed Computing Systems,* 1992.
[46] Y. Yoon, *Transaction Scheduling and Commit Processing forReal-Time Distributed Database Systems,* PhD thesis, Korea Advanced Inst. Science and Technology, May 1994.



Jayant R. Haritsa received the BTech degree in electronics and communications engineering from the Indian Institute of Technology, Madras, and the MS and PhD degrees in computer science from the University of Wisconsin, Madison. Currently, he is on the faculty of the Supercomputer Education and Research Center and the Department of Computer Science and Automation at the Indian Institute of Science, Bangalore. His research interests are in database systems and real-time systems. He is a member of the IEEE and the ACM, and an associate editor of the *International Journal of Real-Time Systems*.

Krithi Ramamritham received the PhD in computer science from the University of Utah. Currently, he is a professor at the University of Massachusetts and holds a visiting position at the Indian Institute of Technology, Bombay. Dr. Ramamritham's interests span the areas of real-time systems, transaction support in advanced database applications, and real-time databases systems. A fellow of the IEEE, he has served as program chair and general chair for the Real-Time Systems Symposium in 1994 and 1995, respectively. He serves on the editorial board of many journals, including the *IEEE Transactions on Parallel and Distributed Systems* and the *Real-Time Systems Journal*. He has coauthored two IEEE tutorial texts on real-time systems, a text on advances in database transaction processing, and a text on scheduling in real-time systems.

Ramesh Gupta received the BS degree in electronics and communications engineering from the Regional Engineering College, Kurukshetra, and the MS degree in computer science from the Indian Institute of Science, Bangalore. Currently, he is a chief technology officer at Goldencom Technologies in California. His research interests are in real-time database systems and network management systems.