More Optimism about Real-Time Distributed Commit Processing

Ramesh Gupta * Jayant Haritsa * Krithi Ramamritham †

Abstract

In [6], we proposed a new commit protocol, OPT, specially designed for use in distributed firm-deadline real-time database systems. OPT allows transactions to "optimistically" borrow uncommitted prepared data in a controlled manner. This controlled borrowing reduces the data inaccessibility and the priority inversion that is inherent in real-time commit processing. Experimental evaluations showed the new OPT protocol to be highly successful, as compared to the classical distributed commit protocols, in minimizing the number of missed transaction deadlines.

In this paper, we extend and improve upon this prior work in the following ways: First, we consider parallel distributed transactions whereas the previous study was restricted to sequential transactions. Second, we evaluate the extent to which OPT's real-time performance is adversely affected by those cases where its optimism turns out to be misplaced. This is achieved by comparing OPT's performance with that of Shadow-OPT, a protocol that augments OPT with the "shadow transaction" approach of [3] and ensures that the right decision about access to uncommitted data is always eventually made. In all of our experiments, which considered a wide range of workloads and system configurations, the difference between OPT and Shadow-OPT never exceeded ten percent. Moreover, the difference was reduced to less than two percent when OPT was enhanced with a simple "healthy lenders" heuristic. Finally, we compare the performance of OPT to that of an alternative priority inheritance-based approach to addressing priority inversion during commit processing. Our results show that the benefits that priority inheritance provides are much smaller than those obtained with the OPT approach.

1. Introduction

Distributed database systems implement a transaction *commit protocol* to ensure transaction atomicity. A variety of protocols have been proposed in the literature including the classical *Two Phase Commit (2PC)*, its *Presumed Abort (PA)* and *Presumed Commit (PC)* variations which have become industry standards, and the non-blocking *Three Phase Commit (3PC)*. These protocols require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately. Due to these costs, commit processing can result in a significant increase in transaction execution times [10, 14]. Consequently, the choice of commit protocol is an especially important design decision for distributed *real-time* database systems (RTDBS).

In a recent paper [6], using a detailed simulation model of a distributed RTDBS, we profiled the performance of the above-mentioned commit protocols for real-time applications with *firm deadlines* [7], wherein transactions that miss their deadlines are considered to be worthless and are immediately "killed", that is, aborted and discarded from the system without being executed to completion. We also developed and evaluated a new commit protocol called **OPT** that incorporates modifications to the 2PC protocol. The OPT protocol allows executing transactions to *borrow* data held by transactions that are in the commit processing stage, unlike the standard protocols which make this so-called "prepared data" inaccessible. If the lender commits, the borrowing is successful in that it provides a "head start" to the borrower, whereas if the lender aborts, the borrower also has to be aborted. The OPT protocol is based on the "optimistic" premise that lender transactions will *typically commit*, thereby helping the borrowers and improving overall performance. The ability to borrow helps to reduce the blocking arising out of prepared data and also to reduce the impact of the *priority inversion* to which the commit phase in a distributed RTDBS is *inherently* susceptible [6]. OPT also incorporates novel features such as "Active Abort" and "Silent Kill" that are specifically designed to improve its performance in a real-time environment. A special feature of OPT is that it does not, unlike previous efforts in the area [13, 15], require transaction atomicity requirements to be weakened.

Our experimental results in [6] showed that, with respect

* SERC, Indian Institute of Science, Bangalore 560012, India
† Dept. of Computer Science, Univ. of Massachusetts, Amherst 01003, USA

to the metric of the steady-state percentage of missed deadlines, OPT provided by far the best performance, primarily due to its optimistic borrowing and active abort policies. In fact, a *non-blocking* version of OPT proved to be superior to the standard blocking protocols for most of the workloads considered in our study. This is especially encouraging given the high desirability of the nonblocking feature in a real-time environment.

In this paper, we extend and improve upon this prior research in the following ways:

First, only *sequential* distributed transactions were modeled in our previous study. However, for real-time database applications, given their time-critical nature, it may be more common to have *parallel* distributed execution. Therefore, we have conducted again all the experiments of the previous study for parallel distributed transaction workloads and report their results here. In the process our simulation model has been made more realistic – it now includes separate log and data disks, and the effects of having a buffer pool are modeled.

Second, the OPT protocol is an "indiscriminate" (an optimist would call it "fully-optimistic") lender in that data is *always* lent whenever requested. Although our experiments showed OPT to perform well under this premise, it was not clear to what extent its real-time performance was adversely affected by those cases where its optimism turned out to be misplaced. We quantitatively evaluate the "efficiency" of OPT here by comparing its performance to that of **Shadow-OPT**, a protocol that combines OPT with the "shadow transaction" approach suggested in [3]. As explained later, if we *ignore the overheads of the shadow mechanism* (which may be significant in practice), Shadow-OPT represents the best *on-line* performance that could be achieved using the optimistic approach.

Third, OPT does not take into account the possibility that a commit-phase transaction that is close to its deadline may be killed due to deadline expiry before the commit processing is over. Lendings by such transactions are obviously harmful to system performance and therefore should be avoided. To address this issue, we have designed and evaluated the **Healthy-OPT** protocol, which augments the basic OPT protocol with a simple heuristic called "healthy lenders" wherein a transaction is allowed to lend its data only if its estimated ability to meet its deadline is greater than a (system-specified) threshold value.

Finally, OPT addresses the priority inversion problem in the commit phase by allowing high priority transactions to access the uncommitted prepared data of low priority transactions. A plausible alternative approach is the well-known **priority inheritance (PI)** mechanism [11]. 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. We evaluate here the performance of a real-time commit protocol based on the PI approach.

2. Distributed Commit Protocols

A common model of a distributed transaction is that there is one process, called the *master*, which is executed at the site where the transaction is submitted, and a set of other processes, called *cohorts*, which execute on behalf of the transaction at the various sites that are accessed by the transaction. Each cohort sends a WORKDONE message to the master after it has completed its assigned work, and the master initiates the commit protocol after it has received this message from all its cohorts. A variety of transaction commit protocols have been devised for this model, most of which are based on the classical **two phase commit (2PC)** protocol [4]. In this section, we briefly describe the 2PC protocol and a few popular variations of this protocol – complete descriptions are available in [9, 10, 12].

In the *two phase commit protocol*, the master initiates the first phase of the commit protocol by sending PREPARE (to commit) messages in parallel to all the cohorts. Each cohort that is ready to commit first force-writes a `prepare` log record to its local stable storage and then sends a YES vote to the master. 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 final decision from the master.

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

If the master receives even one NO vote, it moves to the *aborting* state by force-writing an `abort` log record and sends ABORT messages to those cohorts that are in the *prepared* state. These cohorts, after receiving the ABORT message, move to the *aborting* state, force-write an `abort` log record, and send an ACK message to the master.

Finally, the master, after receiving acknowledgments from all the prepared cohorts, writes an `end` log record and then "forgets" the transaction.

Two variants of the 2PC protocol called **presumed abort (PA)** and **presumed commit (PC)** were presented in [9].

These protocols try to reduce the message and logging overheads by requiring all participating cohorts to follow certain rules at failure recovery time. The protocols have been implemented in a number of database products and PA is, in fact, now part of the ISO-OSI and X/OPEN distributed transaction processing standards [10].

A fundamental problem with all of the above protocols is that cohorts may become *blocked* waiting for a decision in the event of a failure at the master site and remain blocked until the failed site recovers [6]. To address the blocking problem, a **three phase commit (3PC)** protocol was proposed in [12]. This protocol achieves a non-blocking 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 non-blocking functionality is an increase in the communication overheads since there is an extra round of message exchange between the master and the cohorts. In addition, both the master and the cohorts have to force-write additional log records in the precommit phase.

3. Real-Time Commit Processing

The commit protocols described above were designed for conventional database systems and do not take transaction priorities into account. In a real-time environment, this is clearly undesirable since it may result in *priority inversion* [11], wherein high priority transactions are made to wait by low priority transactions. Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priority. Removing priority inversion in the commit protocol, however, is *not* fully feasible. This is because, once a cohort reaches the prepared state, it has to retain all its data locks until it receives the global decision from the master – this retention is fundamentally necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a "prepared cohort" of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a distributed RTDBS is *inherently* susceptible to priority inversion. More importantly, the priority inversion interval 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).

It is important to note that the *prepared data blocking* described above is *orthogonal* to the *decision blocking* (because of failures) that was discussed under 3PC. That is, in all the commit protocols, including 3PC, transactions can be affected by prepared data blocking. Moreover, such data blocking occurs during normal processing whereas decision blocking occurs only during failure situations.

3.1. The OPT Protocol

The OPT protocol [6] was designed to address the above-mentioned issue of prepared data blocking. The main feature of OPT (the complete description is available in [6]) is that transactions requesting data items held by other transactions in the prepared state are allowed to access this data. That is, prepared cohorts *lend* uncommitted data to concurrently executing transactions in the "optimistic" belief that this data will eventually be committed.

The mechanics of the interactions between lenders and borrowers are captured in the following two scenarios:

Lender Receives Decision First: Here, the lending cohort receives its global decision before the borrowing cohort has completed its execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. 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 inconsistent data.

Borrower Completes Execution First: Here, the borrowing cohort completes its execution and receives its PREPARE message before the lending cohort receives its global decision. The borrower is then "put on the shelf", that is, it is made to wait and not allowed to enter the prepared state (and hence, to send a YES vote). The borrower waits 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" and allowed to respond to its master's messages, whereas if the lender aborts, the borrower is also aborted immediately since it has read inconsistent data. In the latter case, the borrower is killed in the normal manner.

OPT also features an optimization called "Active Abort" to enhance its real-time performance which operates as follows: In the basic 2PC protocol, cohorts are passive in that they inform the master of their status only upon explicit request by the master. However, in a real-time situation, 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. Therefore, cohorts in OPT inform the master as soon as they decide to abort locally.

Finally, as explained in detail in [6], although OPT permits use of uncommitted data, because only transactions in the prepared state are allowed to lend, the borrowing *does not* result in the well-known problem of cascading aborts [2].

3.2. Shadow-OPT

As mentioned in the Introduction, we wished to evaluate the efficiency of OPT with respect to the extent to which its real-time performance was adversely affected by those cases where its optimism turned out to be misplaced. This was achieved by comparing its performance with that of the **Shadow-OPT** protocol, described below.

The Shadow-OPT protocol combines the OPT protocol with the "shadow transaction" approach suggested in [3]. 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 the 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. 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), Shadow-OPT represents the best *on-line* performance that could be achieved using the optimistic approach. We model such a zero-overhead Shadow-OPT protocol in our experiments.

For correctness, a shadow cohort can resume execution only if the original cohort had not exchanged any message with the master after the creation of the shadow. Otherwise, there can be dependencies among the original cohort and the master of which the shadow cohort is unaware of, and these dependencies need to be handled before the shadow cohort can resume the execution. In our experiments, such dependencies can arise only if the original cohort has sent the WORKDONE message to the master, in which case we discard the shadow cohort.

In addition, for the sake of simplicity, we allow in our experiments at most one shadow (for each cohort) 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 shadow resumes its execution replacing the original cohort.

3.3. Healthy-OPT

As mentioned in the Introduction, the OPT protocol does not take into account the possibility that a transaction entering its commit phase close to its deadline may be killed due to deadline expiry before the commit processing is over. Lendings by such transactions are obviously harmful to system performance since they result in the aborts of all the associated borrowers and therefore should be avoided. To address this issue, we have designed the **Healthy-OPT** protocol, described below.

The Healthy-OPT protocol augments the basic OPT protocol with a simple heuristic called "healthy lenders" that ensures only transactions whose deadlines are not very close (i.e., healthy transactions) are allowed to lend their prepared data. This is implemented in the following manner: A *health factor* H_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 M. 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 (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 M – set too conservatively, it will turn off the borrowing feature to a large extent, thus effectively reducing Healthy-OPT to standard 2PC; on the other hand, set too aggressively, it will fail to stop several lenders that will eventually abort, effectively reducing Healthy-OPT to basic OPT. In our experiments, we consider a range of values for M 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 OPT is used, as explained above.

3.4. Shadow-OPT versus Healthy-OPT

At this point, it may be asked as to why there is a need for the Healthy-OPT protocol when Shadow-OPT with its guarantee of eventually making the right borrowing decision can itself be implemented as discussed above. The point to note here is that Shadow-OPT, in contrast to Healthy-OPT, requires significant reworking of the transaction management system to support the shadow concept. Further, its realization may incur non-negligible overheads in a real system, for example, for creating shadows and for managing updates to buffers from the multiple versions of a transaction. This may result in significant differences between its *actual* performance and that seen in the artificial zero-overhead model used in our experiments. In addition, Shadow-OPT has restrictions on its applicability (for example, the dependency constraints discussed in Section 3.2).

Keeping the above points in mind, Healthy-OPT provides, as will be quantitatively demonstrated in our experiments, a simple and efficient alternative that performs almost as well as Shadow-OPT and can at the same time be easily integrated into current systems.

3.5. The PIC Protocol

As discussed above, OPT 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 [11]. 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 optimistic approach. Further, a study of PI in the context of (centralized) transaction concurrency control was made in [8] and the results suggest 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.

We evaluate in our experiments the performance of **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 inherited 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.[1]

4. Simulation Model

To evaluate the performance of the various commit protocols described in the previous sections, we used a detailed simulator of a distributed RTDBS. The simulator implements a more realistic version of the model used in our previous study [6]. Due to space limitations, we only highlight the main features here – the complete details are in [5]. A summary of the parameters used in the model are given in Table 1.

The database is a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. At each site, transactions arrive in an independent Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline. The deadline is assigned using the formula $D_T = A_T + SF * R_T$, where D_T, A_T and R_T are the deadline, arrival time and resource time, respectively, of transaction T, while SF is a slack factor. The resource time is the total service time at the resources that the transaction requires for its execution.[2] The $SlackFactor$ parameter is a constant that provides control over the tightness/slackness of transaction deadlines.

Each transaction in the workload has the "single master – multiple cohort" structure described in Section 2. The number of sites at which each transaction executes is specified by the $DistDegree$ parameter. The master and one cohort reside at the site where the transaction is submitted whereas the remaining $DistDegree - 1$ cohorts are set up at sites chosen at random from the remaining $NumSites - 1$ sites. All these cohorts execute in parallel at their respective 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 randomly from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$. A transaction that is restarted due to a data conflict makes the same data accesses as its original incarnation.

A read access involves a concurrency control request to obtain access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. Write requests are handled similarly except for their disk I/O – the writing of the data pages takes place asynchronously after the transaction has committed. We assume sufficient buffer space to allow the retention of updates until commit time.

The commit protocol is initiated when the transaction has completed its data processing. If the transaction's deadline expires either before this point, or before the master has written the global decision log record, the transaction is killed (the precise semantics of firm deadlines in a distributed environment are defined in [5]).

As mentioned earlier, transactions in an RTDBS are typically assigned priorities so as to minimize the number of missed deadlines. 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.

The physical resources at each site include $NumCPUs$ CPUs, $NumDataDisks$ data disks and $NumLogDisks$ log disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ pa-

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

[2] 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.

Table 1. Simulation Model Parameters

NumSites	Number of sites in the database
DBSize	Number of pages in the database
ArrivalRate	Transaction arrival rate / site
SlackFactor	Slack Factor in Deadline formula
TransType	Trans. Type (Sequential or Parallel)
DistDegree	Degree of Distribution
CohortSize	Average cohort size (in pages)
UpdateProb	Page update probability
NumCPUs	Number of processors per site
NumDataDisks	Number of data disks per site
NumLogDisks	Number of log disks per site
PageCPU	CPU page processing time
PageDisk	Disk page access time
MsgCPU	Message send / receive time
BufHit	Probability of buffer hit

rameters capture the CPU and disk processing times per data page, respectively. The *BufHit* parameter gives the probability of finding a page that is requested 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 overhead of message transfer is taken into account at both the sending and the receiving sites, and these overheads are captured by the $MsgCPU$ parameter.

With regard to logging costs, we explicitly model only *forced* log writes since they are done synchronously and suspend transaction operation until their completion.

5. Experiments and Results

Using the distributed firm-deadline RTDBS model described in the previous section, we conducted an extensive set of simulation experiments comparing the performance of the various commit protocols presented earlier. Due to space limitations, we discuss only a representative set of results here – the complete details are available in [5].

The performance metric in all of our experiments is $MissPercent$, which is the percentage of input transactions that the system is *unable* to complete before their deadlines.[3] MissPercent values in the range of 0 to 20 percent are taken to represent system performance under "normal" loads, while values beyond this represent "heavy" load performance. The transaction priority assignment used in all of the experiments described here is *Earliest Deadline*,

[3] The MissPercent values shown here have relative half-widths about the mean of less than 10% at the 90% confidence level – each experiment was run until at least 20000 transactions were processed by the system.

Table 2. Baseline Parameter Settings

NumSites	8	*NumCPUs*	2
DBSize	2400 pages	*NumDataDisks*	3
SlackFactor	4.0	*NumLogDisks*	1
TransType	Parallel	*PageCPU*	5 ms
DistDegree	3	*PageDisk*	20 ms
CohortSize	6 pages	*MsgCPU*	5 ms
UpdateProb	1.0	*BufHit*	0.1

wherein transactions with earlier deadlines have higher priority than transactions with later deadlines. For concurrency control, the 2PL High Priority scheme [1] is employed.

5.1. Comparative Protocols

To help isolate and understand the performance effects of distribution and atomicity, we have also simulated, just as in [6], the performance behavior for two additional scenarios, **CENT** and **DPCC**, described below:

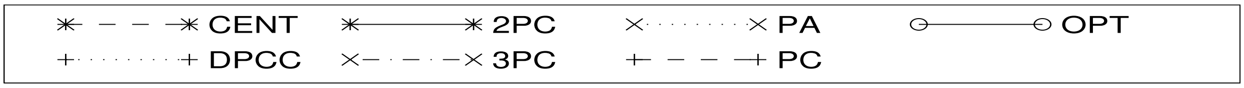
In CENT (Centralized), a *centralized* database system that is equivalent (in terms of database size and physical resources) to the distributed database system is modeled. Messages are obviously not required here and commit processing only requires force-writing a *single* decision log record. Modeling this scenario helps to isolate the overall effect of distribution on MissPercent 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 force-writing of the decision log record at the master. While this system is clearly artificial, modeling it helps to isolate the effect of distributed commit processing on MissPercent performance (as opposed to CENT which eliminates the entire effect of distributed processing).

5.2. Expt. 1: Parallel (RC+DC)

In our first experiment, the performance of the various commit protocols was evaluated for a *parallel* transaction workload. The settings of the workload and system parameters for this experiment are listed in Table 2. These values were chosen to 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 protocols. With the given settings, each transaction executes in a parallel fashion at three sites, accessing and updating an average of six pages at each site.

For this experiment, Figures 1a and 1b show the MissPercent behavior under normal load and heavy load conditions, respectively. In these graphs, we first observe that there is a



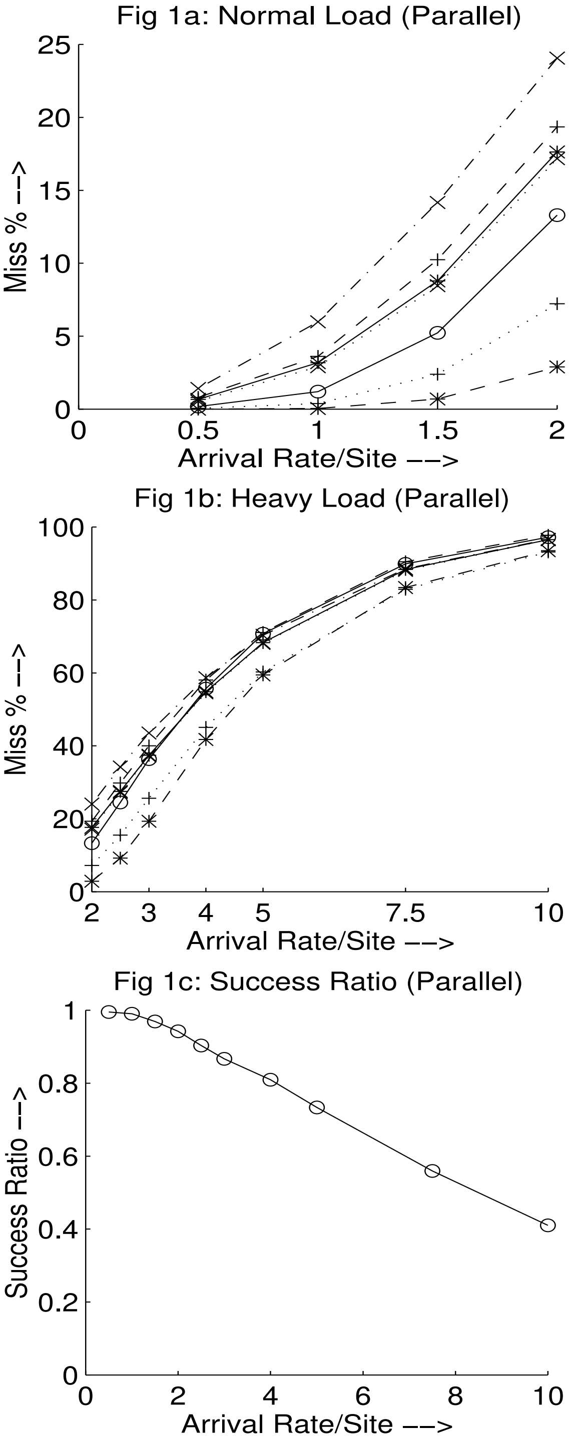


Figure 1: Parallel Transactions (RC+DC)

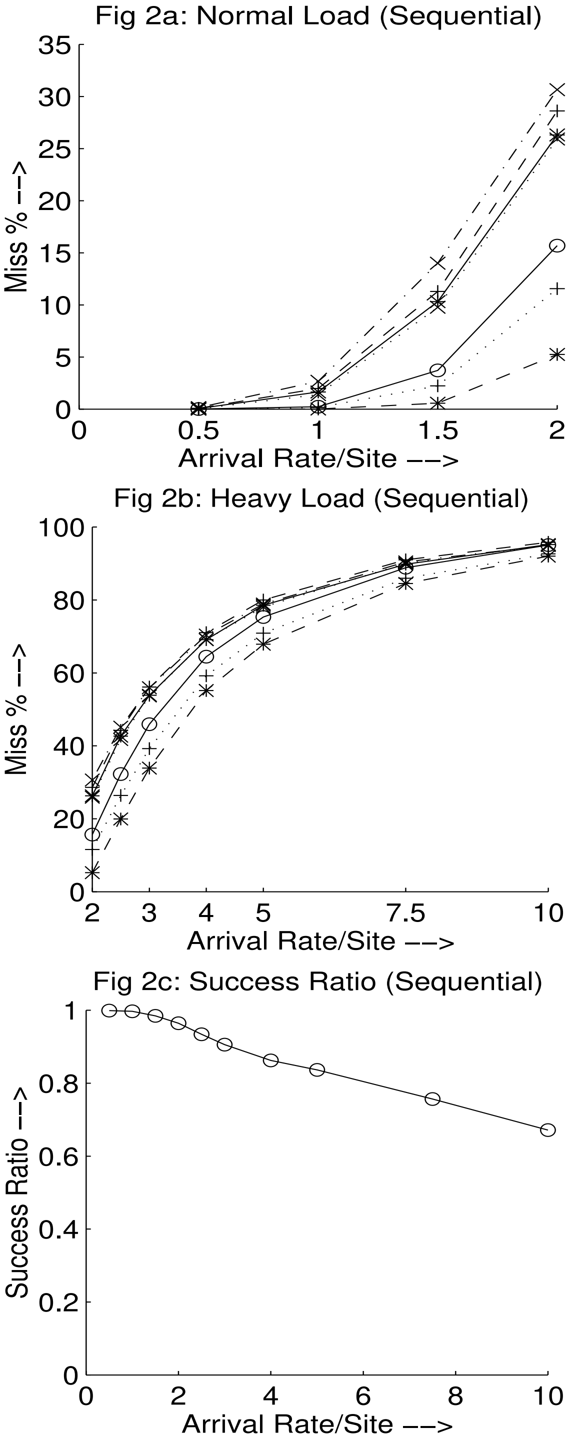


Figure 2: Sequential Transactions (RC+DC)

noticeable difference between the performance of the baseline systems (CENT and DPCC) and the performance of the classical protocols (2PC, PA, PC, 3PC) throughout the loading range. This demonstrates that distributed *commit* processing can have considerably more effect (difference between DPCC and 2PC) than distributed *data* processing (difference between CENT and DPCC) on the MissPercent performance. This highlights the need for designing high-performance commit protocols.

Moving on to the relative performance of 2PC and 3PC, we observe a noticeable difference which arises from the additional message and logging overheads involved in 3PC. The performance of PA and PC, however, is only marginally different from that of 2PC.

Finally, OPT provides a performance that is significantly better than that of 2PC under normal loads. In Figure 1c, OPT's "success ratio", that is, the fraction of times that a borrowing was successful, is shown. This statistic clearly shows that under normal loads, optimism is the right choice since the success ratio is almost one. Under heavy loads, however, there is a decrease in the success ratio – the reason for this is that transactions reach their commit phase only close to their deadlines and in this situation, a lending transaction may often abort due to missing its deadline. That is, many lenders turn out to be "not healthy" – we address this issue in more detail in Experiment 3 (Section 5.4).

The observations made above were also seen, to a large extent, for *sequential* transactions in our previous study – for ease of comparison, Figures 2a through 2c present these corresponding results for the sequential transactions.[4] There are a few changes, however, with respect to OPT's performance:

First, although OPT continues to perform the best under normal loads, its effect on the MissPercent 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 environment. The reason for its reduced utility is that due to cohorts executing in parallel, there are much fewer chances of a cohort aborting after sending the WORKDONE message, but before receiving the PREPARE message, which is when the active abort policy mostly comes into play for the parallel case.

Second, the performance of OPT under heavy loads is marginally *worse* than that of 2PC, whereas in the sequential case OPT was always better or matched 2PC. This is explained by comparing OPT's success ratios in Figures 1c and 2c, which clearly indicate that the heavy-load degradation in OPT'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. Therefore in general, cohorts are 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 and hence becoming unhealthy lenders thereby resulting in a worse success ratio.

The above experiment was repeated for a pure data contention (pure DC) environment, wherein there is no queuing for the physical resources, in order to isolate the influence of data contention on the real-time performance. The performance (not shown here but available in [5]) of OPT in this experiment was, unlike the RC+DC environment, comparable to that for the corresponding sequential transaction experiment. The reason for this is that in the pure DC scenario the OPT approach, due to its increased concurrency, reduces the data contention significantly. This results in a high success ratio for OPT and consequently significantly better performance than that of the standard commit protocols.

5.3. Expt. 2: Efficiency of OPT

In our next experiment, we compared the performance of OPT to that of the (zero-overhead) Shadow-OPT protocol described in Section 3.2. The results of this experiment are depicted in Figures 3 and 4 under RC+DC and pure DC conditions, respectively. We see from these figures that the performance of Shadow-OPT is better than that of OPT only by a marginal amount under RC+DC and by a modest amount in the pure DC environment. In fact, in all our experiments, maximum improvement for Shadow-OPT over OPT was never more than *ten percent*. Moreover, this improvement was restricted to the heavily loaded region and never occurred in the normal load region. The above results indicate that OPT is *highly successful in its utilization of the optimistic premise*.

5.4. Expt. 3: Healthy Lenders

In our next experiment, we evaluated the performance of the Healthy-OPT protocol. The results for this experiment are shown in Figures 5a through 5c for the RC+DC scenario, and in Figures 6a through 6c for the pure DC scenario. The results for two different values (1.0 & 2.0) of the health threshold M are shown in these figures (the Healthy-OPT curves are very close to Shadow-OPT in Figures 5a and 6a and may therefore be difficult to distinguish visually).

From the MissPercent graphs (Figures 5a and 6a), we see that Healthy-OPT (for $M = 1$) is in general superior

[4] The sequential results shown here were obtained using the *refined* system model described in Section 4 and are therefore quantitatively different from those shown in [6].

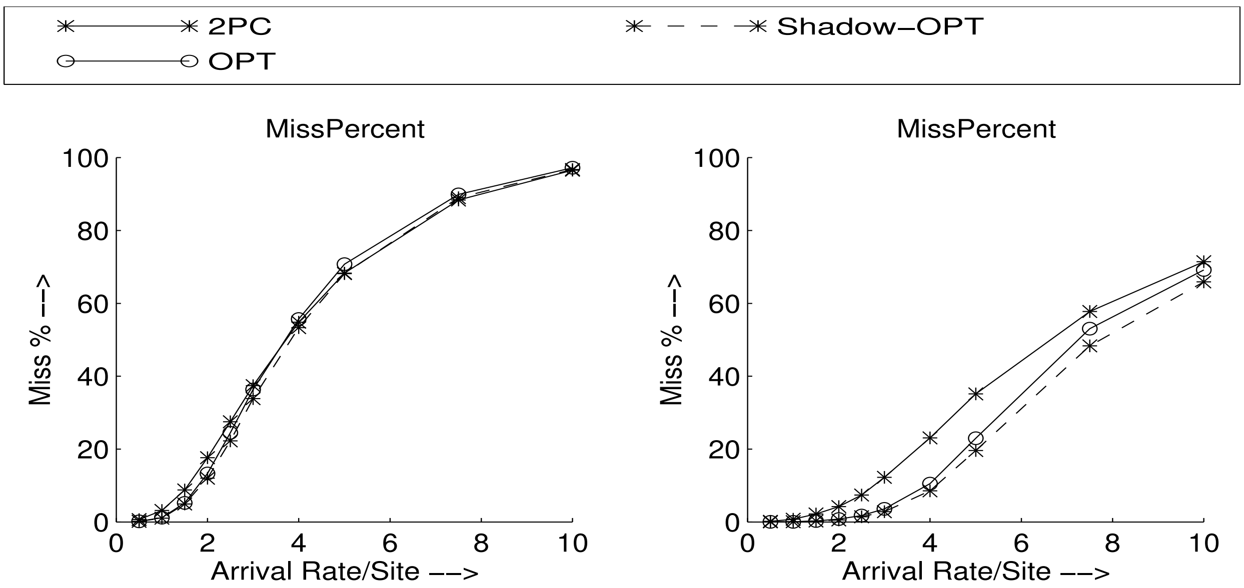


Figure 3: Shadow-OPT (RC+DC)

Figure 4: Shadow-OPT (pure DC)

to OPT and especially so under heavy loads in the pure DC environment. The reason for the improvement is evident in Figures 5b and 6b where the success ratio of Healthy-OPT ($M = 1$) is seen to be considerably higher than that of basic OPT. In Experiment 1, we had observed that OPT's heavy load performance in the RC+DC environment was worse than that of 2PC (see Figure 1a) – notice now that with the Healthy Lenders optimization, OPT's performance matches that of 2PC in this region (Figure 5a).

Figures 5c and 6c present the "borrow ratio" (average number of data items borrowed per transaction). It is clear from these figures that Healthy-OPT ($M = 1$) 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*.

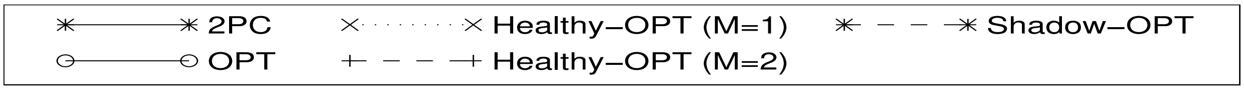
The last observation deals with the "completeness" and "precision" of borrowing, that is: (1) Do we always borrow when the borrowing is going to be successful, and (2) Is what we borrow always going to be successful. OPT is complete by design since it does not miss out on any successful borrowing opportunities, but because it may also borrow without success, it is not precise. The experimental results for Healthy-OPT show that it is far more precise in comparison but sacrifices very little completeness in achieving this goal.

Turning our attention to Healthy-OPT ($M = 2$), we observe that the performance of Healthy-OPT for $M = 1$ and for $M = 2$ are 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 *Earliest Deadline* 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 do the commit processing is actually sufficient for them to complete this operation.

Finally, we observe that the performance improvement of Healthy-OPT over basic OPT is very similar to that obtained by Shadow-OPT. In fact, in all our experiments, the performance difference between Shadow-OPT and Healthy-OPT was never more than *two percent*. This means that Healthy-OPT can provide performance gains similar to that of Shadow-OPT *without attracting the implementation problems and overheads associated with Shadow-OPT*. We have also found that a health threshold of $M = 1$ provides good performance for a wide range of workloads and system configurations, that is, this setting is *robust* . In short, *Healthy-OPT provides extremely efficient use of the optimistic premise in terms of both performance and simplicity*.

5.5. Expt. 4: Priority Inheritance

In our final experiment, we evaluated the performance of the PIC protocol described in Section 3.5, which incorporates the priority-inheritance approach to addressing the problem of commit-phase priority inversion. The results showed that the performance of PI is virtually identical to



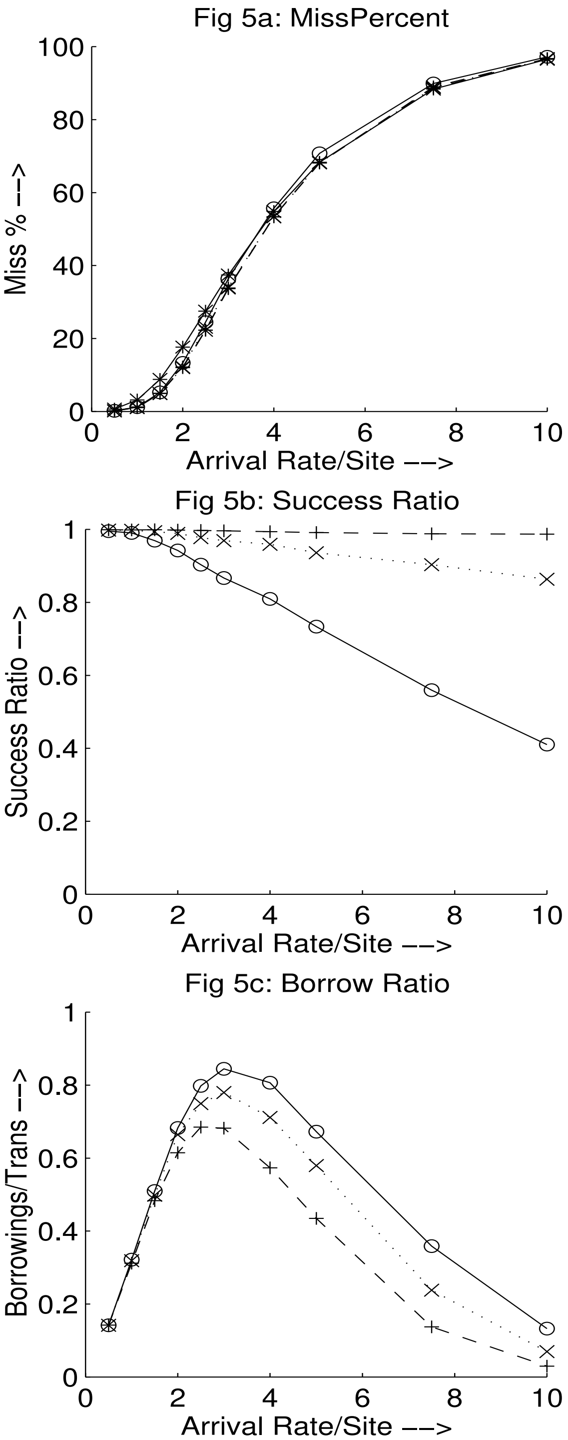


Figure 5: Healthy-OPT (RC+DC)

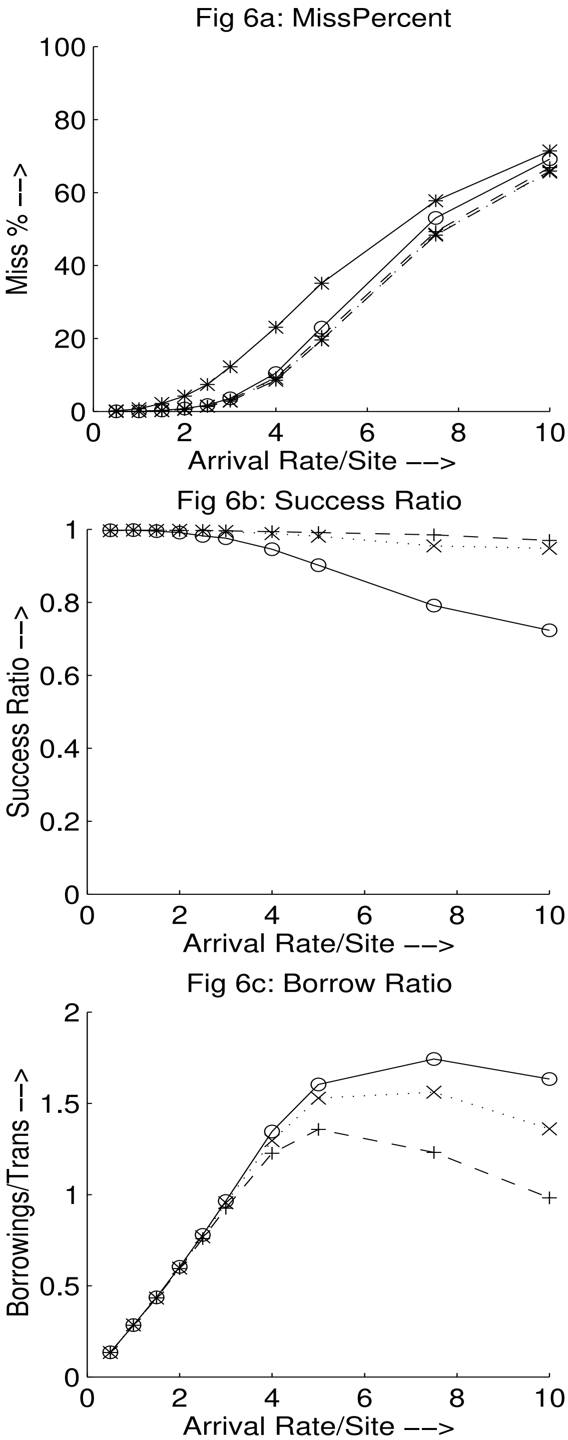


Figure 6: Healthy-OPT (pure DC)

that of 2PC. The reason for this 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, PI expedites at most the processing of only the decision message. Further, even the minor advantage that may be obtained by PI is partially offset by the extra overheads involved in processing the priority inheritance information messages.

In summary, PI fails to provide any performance benefits over 2PC due to a *fundamental* problem of distributed processing – delay in the dissemination of information (in this case, priority inheritance information) to remote sites (in this case, the sibling cohorts).

6. Conclusions

In this paper, we have significantly extended and improved on our earlier simulation-based study [6] of the performance implications of supporting transaction atomicity in a distributed firm-deadline RTDBS environment. These improvements included enhancements in the model, the workloads, the protocols and the paradigms considered for real-time commit processing. The main conclusions of our current study are the following:

First, the results obtained for *parallel* distributed transaction workloads were generally similar to those observed for the sequential-transaction environment in [6]. But, performance improvement due to the basic OPT 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.

Second, the basic OPT protocol was shown to make good use of the optimistic premise – the difference between OPT and Shadow-OPT, which represents the best on-line usage of the optimistic approach, never exceeded *ten percent* and occurred only under significantly overloaded conditions.

Third, the Healthy-OPT protocol, which augments basic OPT with a simple "healthy lenders" heuristic was found to provide performance very close to that of Shadow-OPT. It is especially important to note that Healthy-OPT provides this high level of performance *without* incurring the potentially significant overheads associated with implementing the Shadow mechanism in a real system.

Finally, we evaluated the priority inheritance approach to addressing the priority inversion problem. 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 OPT for enhancing distributed commit processing performance.

In summary, we suggest that designers of distributed real-time database systems may find the Healthy-OPT protocol to be a good choice for achieving high-performance real-time distributed commit processing.

Acknowledgments

This work was supported in part by research grants from the Dept. of Science and Technology, Govt. of India, and from the National Science Foundation of the United States under grant IRI-9619588.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proc. of 14th VLDB Conf.*, August 1988.

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

[3] A. Bestavros and S. Braoudakis, "Timeliness via Speculation for Real-time Databases", *Proc. of 15th Real-Time Systems Symp.*, December 1994.

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

[5] R. Gupta, J. Haritsa and K. Ramamritham, "More Optimism about Real-Time Distributed Commit Processing", TR-97-04, DSL/SERC, Indian Institute of Science.

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

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

[8] J. Huang, J.A. Stankovic, K. Ramamritham, D. Towsley and B. Purimetla, "Priority Inheritance In Soft Real-Time Databases", *Real-Time Systems Journal*, 4 (3), 1992.

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

[10] G. Samaras, K. Britton, A. Citron and C. Mohan, "Two-Phase Commit Optimizations in a Commercial Distributed Environment", *Journal of Distributed and Parallel Databases*, 3(4), 1995 (also in *Proc. of 9th IEEE Intl. Conf. on Data Engineering*, April 1993).

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

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

[13] N. Soparkar et al, "Adaptive Commitment for Real-Time Distributed Transactions", *TR-92-15, CS, Univ. of Texas (Austin)*, 1992.

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

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