# Commit Processing in Distributed Real-Time Database Systems

Ramesh Gupta [*] Jayant Haritsa [*] Krithi Ramamritham [†] S. Seshadri [‡]

## Abstract

*We investigate here the performance implications of supporting transaction atomicity in a distributed real-time database system. Using a detailed simulation model of a firm-deadline distributed real-time database system, we profile the real-time performance of a representative set of commit protocols. A new commit protocol that is designed for the real-time domain and allows transactions to "optimistically" read uncommitted data is also proposed and evaluated,*

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

## 1. Introduction

Many real-time database applications, especially in the areas of communication systems and military systems, are inherently *distributed* in nature [15]. Incorporating distributed data into the real-time database framework incurs the well-known additional complexities that are associated with transaction concurrency control and database recovery in distributed database systems [3, 11]. While the issue of distributed real-time concurrency control has been considered to some extent(e.g. [13, 16, 18]), comparatively little work has been done with regard to distributed real-time database recovery. In this paper, we investigate the performance implications of supporting transaction atomicity in a distributed real-time database system (RTDBS). To the best of our knowledge, this is the first quantitative evaluation of this issue.

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 by database researchers. These include the classical *two phase commit (2PC)* protocol [6], its variations such as *presumed commit* and *presumed abort* [10], and *three phase commit* [14]. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately. Due to these costs, commit processing can result in a significant increase in transaction execution times, making the choice of commit protocol an important design decision for distributed database systems.

From a performance perspective, commit protocols can be compared on the following two issues: First, the extent to which the commit protocol affects the normal distributed transaction processing performance. That is, how expensive is it to provide atomicity? Second, in the event of a site failure, the ability of the operational sites to correctly terminate the transaction without waiting for recovery of the failed site. That is, is the protocol "blocking" or "non-blocking"?

With respect to the above issues, the performance of a representative set of commit protocols has been analyzed to a limited extent in [4, 10]. These studies were conducted in the context of a conventional DBMS where transaction throughput or response time is the primary performance metric. In a RTDBS, however, performance is usually measured in terms of the *number* of transactions that complete before their deadlines. Due to the difference in objectives, the performance of commit protocols has to be reevaluated for **the real-time environment.**

In the real-time domain, there are two major questions that need to be explored: First, how do we adapt the commit protocols to the real-time domain? Second, how do the real-time variants of the commit protocols compare in their performance? In this paper, we address these questions for the "firm-deadline" application framework [8], wherein transactions that miss their deadlines are considered to be worthless and are

[*] SERC, **Indian Institute of Science, Bangaiore 560012, India**
[†] COINS, **University** of **Massachussetts, Amherst 01003, USA**
[‡] CSE, **Indian Institute of Technology, Bombay 400076, India**

1052-8725/96 $5.00 © 1996 IEEE

immediately "killed", that is, aborted and discarded from the system without being executed to completion. Using a detailed simulation model of a distributed database system, we compare the real-time performance of a representative set of commit protocols. The performance metric is the steady-state percentage of transaction deadlines that are missed.

### 1.1. Related Work

The design of real-time commit protocols has been investigated earlier in [17, 19]. These papers are based on a common theme of allowing individual sites to ***unilaterally*** commit – the idea is that unilateral commitment results in greater timeliness of actions. If it is later found that the decision is not consistent globally, "compensation" transactions are 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: First, the standard notion of transaction atomicity is not supported – instead, a "relaxed" notion of atomicity [9] is provided. Second, the design of a compensating transaction is an application-specific task since it is based on application semantics. Third, compensation transactions need to be designed in advance so that they can be executed as soon as errors are detected – this means that the transaction workload must be fully characterized a ***priori.*** Fourth, "real actions" such as firing a weapon or dispensing cash may not be compensatable at all [9]. Finally, no performance studies are available to evaluate the effectiveness of this approach.

Due to the above limitations, we focus here instead on improving the real-time performance of the ***standard*** mechanisms for maintaining distributed transaction atomicity.

## 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. For this model, a variety of transaction commit protocols have been devised, most of which are based on the classical **two phase commit (2PC)** protocol [6]. In this section, we briefly describe the **2PC** protocol and a few popular variations of this protocol – complete descriptions are available in [10, 14].

### 2.1. Two Phase Commit Protocol

The master initiates the first phase of the 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 a response from the master.

After the master receives the votes from all the cohorts, it initiates the second phase of the protocol. If all the votes 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 acknowledgements from all the prepared cohorts, writes an ***end*** log record and then "forgets" the transaction.

### 2.2. Presumed Abort

A variant of the **2PC** protocol, called **presumed abort (PA)** [10], tries to reduce the message and logging overheads by requiring all participants to follow a "in case of doubt, abort" rule. That is, if after coming up from a failure a site queries the master about the final outcome of a transaction and finds no information available with the master, the transaction is assumed to have been aborted. With this assumption, it is not necessary for cohorts to either send acknowledgments for ***ABORT*** messages from the master or to force-write the ***abort*** record to the log. It is also not necessary for the master to force-write the ***abort*** log record or to write an ***end*** log record after abort.

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

### 2.3. Presumed Commit

**A** variation of the presumed abort protocol is based on the observation that, in general, the number of committed transactions is much more than the number of aborted transactions. In this variation, called **presumed commit (PC)** [10], the overheads are reduced for committing transactions rather than aborted transactions by requiring all participants to follow a "in case of doubt, commit" rule. In this scheme, cohorts do not send acknowledgments for the ***commat*** global decision, and do not force-write the ***commit*** log record. In addition, the master does not write an ***end*** log record. 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 commercial products and are now part of transaction processing standards.

### 2.4. Three Phase Commit

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

To address the blocking problem, a **three phase commit (3PC)** protocol was proposed in [14]. 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 re**garding** the fate **of** the transaction. The information made available to the participating sites **as** a result of this preliminary decision allows a global decision to be made despite a subsequent failure of the master. Note, however, that the nonblocking functionality is obtained at an increase in the communication overhead since there is **an** extra round of message exchange between the master and the cohorts. In addition, both the master and the cohorts have to force-write additional log records in the precommit phase.

## 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*** [12], 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 (due to network delays). To address these issues, we have designed a modified version of the **2PC** protocol, described below.

### 3.1. Optimistic Commit Protocol

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

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

**Borrower Completes Execution First** : Here, the borrowing cohort completes its execution before the lending cohort has 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 ***YES*** vote in response to its master's ***PREPARE*** message. The borrower 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, then the borrower is *"taken* off the shelf" and allowed to respond to its master's messages. However, if the lender aborts, then the borrower is **also** aborted immediately since it has read inconsistent data. In the latter case (borrower's deadline expires while waiting), the borrower is killed in the normal manner.

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

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

### 3.2. Additional Features of OPT

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

**Active Abort** : In the basic 2PC protocol, cohorts are passive in that they inform the master of their status only upon explicit request by the master. In a real-time situation, it may be better for an aborting cohort to immediately inform the master so that the abort at the other sites can be done earlier. Therefore, cohorts in OPT inform the master **as** soon **as** they decide to abort locally.

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

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

An important point to note here is that the policy of using uncommitted data is generally ***not*** recommended in database systems since this can potentially lead to the well-known problem of ***cascading aborts*** **[3]** if the transaction whose dirty data has been accessed is later aborted. However, in our situation, this problem is alleviated due to two reasons: First, the lending transaction is typically expected to commit because (a) the lending cohort is in prepared state and cannot be aborted due to local data conflicts, and (b) the sibling cohorts are also expected to eventually vote to commit since they have survived[2] all their data conflicts that occurred prior to the initiation of the commit protocol (Active Abort policy). Second, even if the lender does eventually abort (e.g. due to deadline expiry), it only results in the abort of the immediate borrower(s) and does not cascade beyond this point (since borrowers are not in the prepared state which is the only situation in which uncommitted data can be accessed). In short, the abort chain is bounded and is of length one.

## 4. Simulation Model

To evaluate the performance of the various commit protocols described in the previous sections, we developed a detailed simulation model of a distributed real-time database system. Our model is based on a loose combination of the distributed database model presented in *[5]* and the real-time processing model of [8]. A summary of the parameters used in the model are given in Table 1.

The database is modeled **as** a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. At each site, transactions arrive in an independent Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline. The deadline is assigned using the formula $D_T = A_T + SF * R_T$, where $D_T$, $A_T$ and $R_T$ are the deadline, arrival time and resource time, respectively, of transaction $T$, while $SF$ is a slack factor. The resource time is the total service time at the resources that the transaction requires for its execution[3]. The $SlackFactor$ parameter is a constant that provides control over the tightness/slackness of transaction deadlines.

[1] A similar, but unrelated, strategy of allowing access to uncommitted data has also been used to improve real-time concurrency control performance [2].

[2] We assume a locking-based concurrency control mechanism.

[3] 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.

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. The cohorts execute one after another in a sequential fashion. At each of the execution sites, the number of pages accessed by the transaction's cohort varies uniformly between **0.5** and **1.5** times *CohortSize*. These pages are chosen randomly from among the database pages located at that site. A page that is read is updated with probability *WriteProb*. **A** transaction that is restarted due to a data conflict makes the same data accesses **as** its original incarnation.

**A** read access involves a concurrency control request to get access permission, followed by a disk **1/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 **1/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** th‘e 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 [7]).

**As** mentioned earlier, transactions in a RTDBS are typically assigned priorities in order to minimize the number of missed deadlines. In our model, all the cohorts of a transaction inherit the master transaction's priority. Further, this priority, which is assigned at arrival time, is maintained throughout the course of the transaction's existence in the system (including the commit processing stage, if any).

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

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

**Table 1. Simulation Model Parameters**

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

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 [7].

The performance metric of our experiments is *Misspercent,* which is the percentage of input transactions that the system is *unable* to complete before their deadlines. Misspercent values in the range of 0 to **20** percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 20 percent to 100 percent represent system performance under "heavy" loads. The transaction priority assignment used in all of the experiments described here is *Earliest Deadline,* 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. ComparativeProtocols

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

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

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

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

### 5.2. Expt. 1: Baseline Experiment

The settings of the workload parameters and system parameters for our baseline experiment are listed in Table 2 . These settings were chosen to ensure significant data and resource contention in the system, thus helping to bring out the performance differences between the various commit protocols, without having to generate very high transaction arrival rates.

**Table 2. Baseline Parameter Settings**

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

For the baseline experiment, Figures 1a and 1b show the Misspercent behavior under normal load and heavy load conditions, respectively. In these graphs, we first observe that there is considerable difference between centralized performance (CENT) and the performance of the standard commit protocols throughout the loading range. For example, at an arrival rate of 2 transactions per second at each site, the centralized system misses virtually no deadlines whereas 2PC and 3PC miss in excess of 30 percent of the deadlines. This difference highlights the extent to which a conventional implementation of distributed processing can affect real-time performance.

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

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

Fig I a: Normal Load (Baseline)

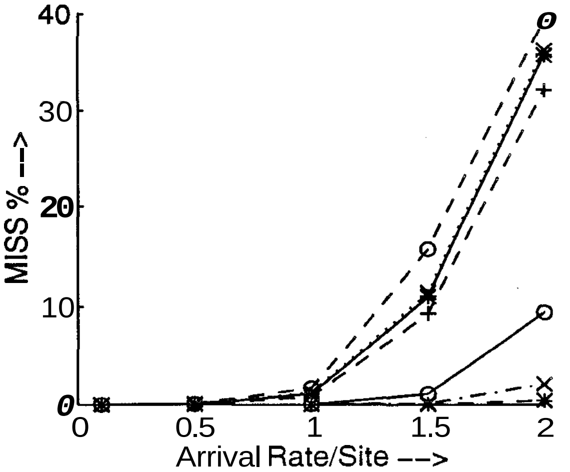
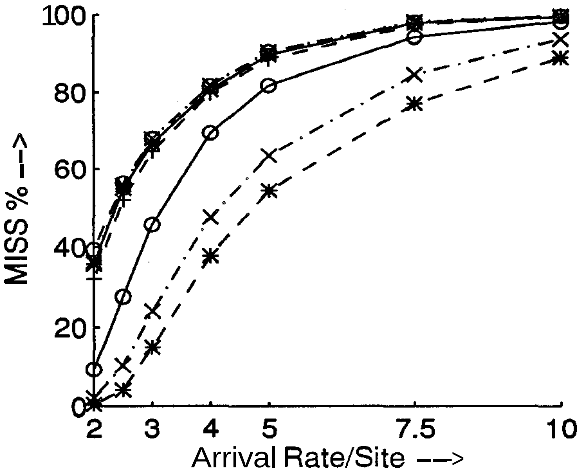


Fig 1b : Heavy Load (Baseline)



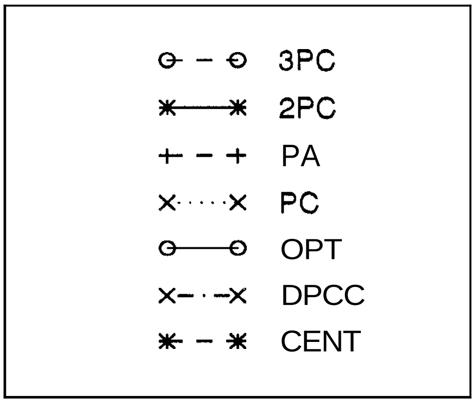


Fig I c : Success Ratio (Baseline)

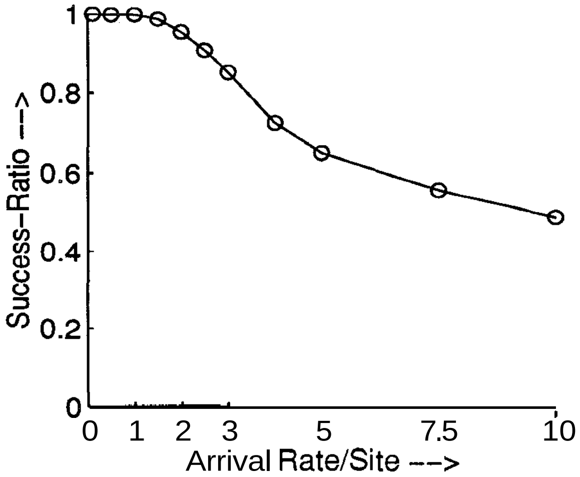


Fig 1d : Forced Writes (Baseline)

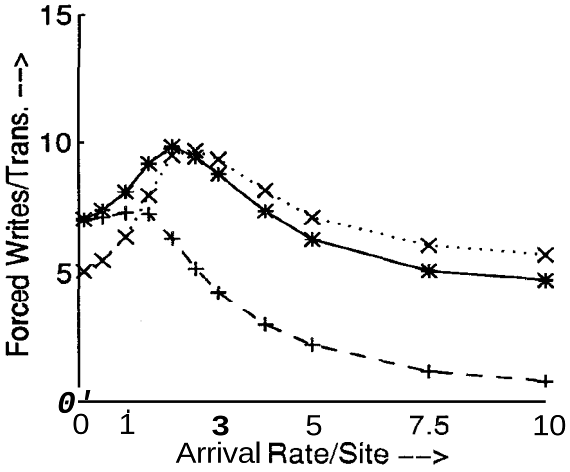
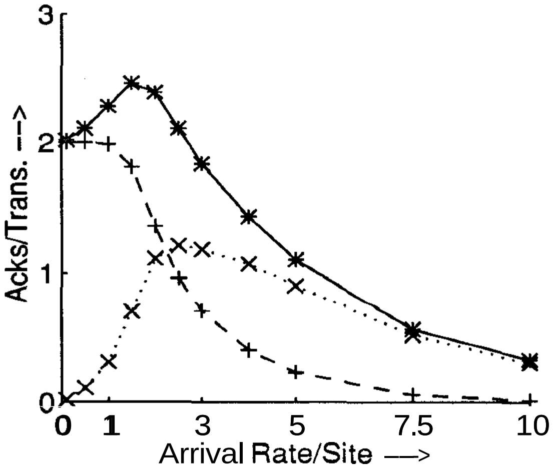


Fig I e : Acks (Baseline)



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

As part of this experiment, we wanted to quantify the degree to which the OPT protocol's optimism about accessing uncommitted data was well-founded – that is, is OPT safe or foolhardy? To evaluate this, we measured the "success ratio", that is, the fraction of times that a borrowing was successful in that the lender committed after loaning the data. This statistic is shown in Figure 1c and clearly shows that under normal loads, optimism is the right choice since the success ratio is almost one. Under heavy loads, however, there is a decrease in the success ratio – the reason for this is that transactions reach their commit phase only close to their deadlines and in this situation, a lending transaction may often abort due to missing its deadline. These results indicate that under heavy loads, the optimistic policy should be modified such that transactions borrow only from "healthy" lenders, that is, lenders who still have considerable time to their deadline – we intend to address this issue in our future work.

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

### 5.3. Expt. 2: Pure Data Contention

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

### 5.4. Expt. 3: OPT-PC and OPT-PA

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

### 5.5. Expt. 4: Non-Blocking OPT

In the previous experiments, we observed that OPT, which is based on 2PC, performed significantly better than the standard protocols. This motivated us to evaluate the effect of incorporating the same optimizations in ***3PC.*** The results showed OPT-3PC's performance to be noticeably but not greatly worse than that of OPT-2PC. Moreover, under infinite resources, the difference between OPT-3PC and OPT-2PC virtually disappears. These results indicate that, in the real-time domain, nonblocking functionality which is extremely useful in case of failures can be purchased at a relatively modest increase in routine processing cost.

Fig 2a : Normal Load (Data Contention)

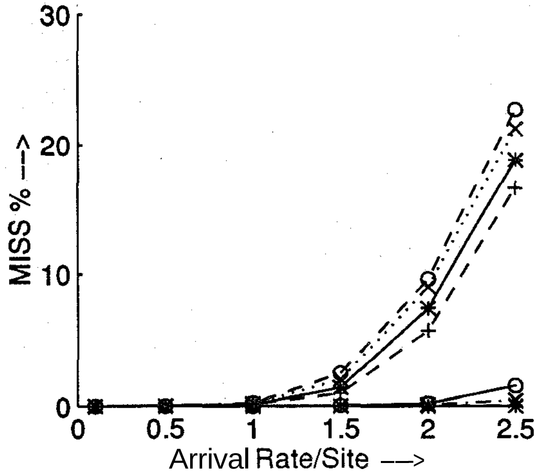
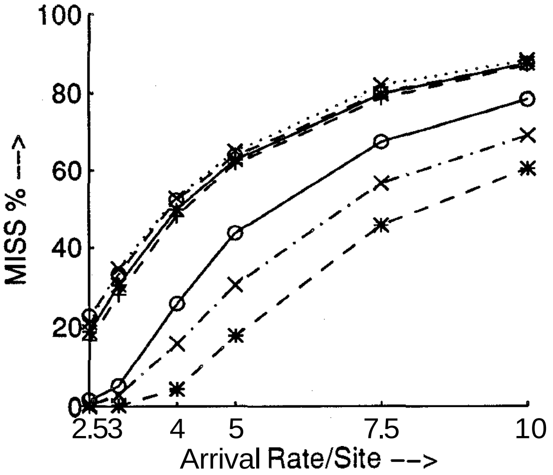


Fig 2b : Heavy Load (Data Contention)



| | |
|---|---|
| ⊖ – ⊖ | 3PC |
| ✳——✳ | 2PC |
| *i* – – + | PA |
| ×·····× | PC |
| ⊖——⊖ | OPT |
| ×– · –× | DPCC |
| ✳ – ✳ | CENT |

Fig 2c : Success Ratio (Data Contention)

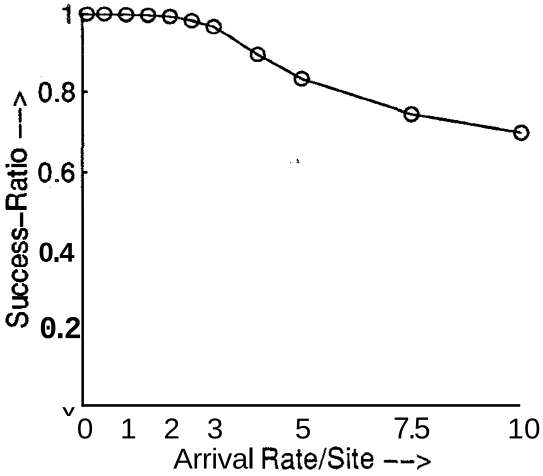


Fig 2d : Forced Writes (Data Contention)

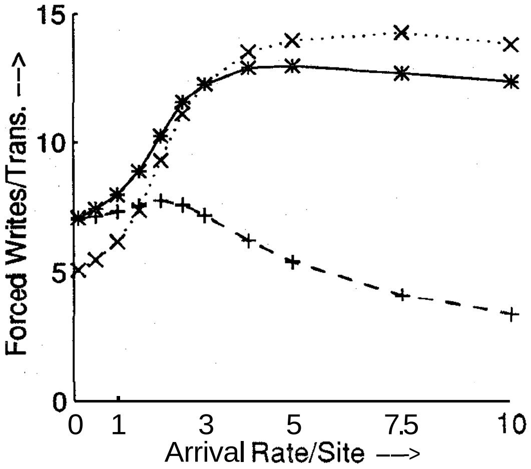
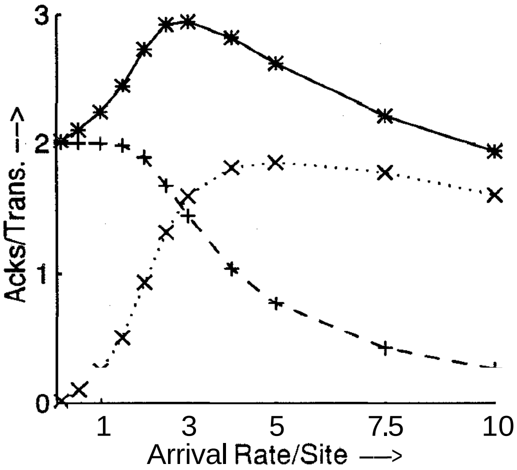


Fig 2e : Acks (Data Contention)



## 6. Conclusions

In this paper, we have proposed and evaluated new mechanisms for designing high performance real-time commit protocols that do not, unlike previous efforts, require transaction atomicity requirements to be weakened. Using a detailed simulation model of a firm-deadline RTDBS, we evaluated the deadline miss performance of a variety of standard commit protocols including 2PC, Presumed Abort, Presumed Commit, and 3PC. We also developed and evaluated a new commit protocol, OPT, that was designed specifically for the real-time environment and included features such as controlled optimistic access to uncommitted data, active abort and silent kill. To the best of our knowledge, these are the first quantitative results in this area.

Our experiments demonstrated the following: First, 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 that are tuned to the real-time domain. Second, the standard 2PC and 3PC algorithms perform poorly in the real-time environment due to their passive nature and due to preventing access to data held by cohorts in the prepared state. Third, the PA and PC variants of 2PC, although reducing protocol overheads, fail to provide tangible benefits in the real-time environment[4]. This is in marked contrast to the conventional DBMS environment where they have been implemented in a number of commercial products and standards. Fourth, the new protocol, OPT, 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 policies. The optimistic access significantly reduces the effect of priority inversion which is inevitable in the prepared state. Sup porting statistics showed that OPT'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. Finally, experiments combining the optimizations of OPT with 3PC indicate that the nonblocking functionality can be obtained in the real-time environment at a relatively modest cost in normal processing performance. This is especially encouraging given the high desirability of the nonblocking feature in a real-time environment.

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

[4] This conclusion is limited to the completely update transaction workloads considered here. PA and PC have additional optimizations for fully or partially read-only transactions [10].

## References

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

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

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

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

[5] M. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication", **Proc. of 14th VLDB Conf.,** August **1988.**

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

[7] R. Gupta, J. Haritsa, K. Ramamritham and S. Seshadri, "Commit Processing in Distributed RTDBS", **TR-96-01,** DSL/SERC, Indian Institute of Science (http://dsl.serc.iisc.ernet .in/reports.html).

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

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

[10] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the $R^*$ Distributed Database Management System", **ACM TODS, 11(4), 1986.**

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

[12] 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.

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

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

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

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

[17] N. Soparkar, E. Levy, H. Korth and A. Silberschatz, "Adaptive Commitment for Real-Time Distributed Transactions", **TR-92-15, *CS, Univ. of Texas* (Austin), 1992.**

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

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