Commit Processing in Distributed Real-Time Database Systems

Ramesh Gupta   Jayant Haritsa

Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, India
{ramesh, haritsa}@serc.iisc.ernet.in

1 Introduction

A **Real-Time Database System (RTDBS)** is a transaction processing system that is designed to handle workloads where transactions have completion deadlines. The objective of the system is to meet these deadlines, that is, to process transactions before their deadlines expire. Real-Time Database Systems are becoming increasingly important in a wide range of applications, such as telecommunications, mobile communication systems, nuclear reactor control, traffic control systems, computer integrated manufacturing, robotics, and military systems [Son90]. Even though many of these systems are inherently *distributed*, research on real-time database systems has almost exclusively focused on centralized systems. The use of distributed database systems poses a new challenge due to the additional complexity involved in maintaining database consistency and transaction recovery. Transaction scheduling and concurrency control in distributed RTDBS have been considered during the past few years [Sha88]. However, comparatively little work has been done to study the impact of *commit* protocols on the performance of distributed RTDBS. In this paper, we examine the behavior of commit protocols and investigate their performance implications with respect to meeting transaction deadlines, using a detailed simulation model of a distributed RTDBS.

2 Commit Protocols

In a distributed database system, the actions of a transaction (an atomic unit of consistency and recovery) may occur at more than one site. Therefore, a distributed transaction commit protocol is required in order to ensure that either *all* the effects of the transaction persist or *none* of the effects persist, despite site or communication link failures. In other words, a commit protocol is needed to guarantee the uniform commitment of distributed transaction execution.

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 the *cohorts*, which execute on behalf of the transaction at the other sites that are accessed by the transaction. Virtually all conventional distributed database systems implement transaction commit protocols based on the classical **two phase commit (2PC)** protocol [Gra78]. This protocol assumes that each cohort of a transaction is able to *provisionally* perform the actions of the transaction in such a way that they can be undone if the transaction is aborted. This is implemented by using *logging* mechanisms [Gra78], which maintain sequential histories of transaction actions in stable storage. The commit protocol writes some of the log records synchronously to stable storage (called *force-writing*). At the time of force-writing, the forced log-record and all the preceding ones are immediately moved from the virtual memory buffers to stable storage. This means that, if the site crashes after the force-write has completed, then the forced record and the preceding ones will have survived the crash and will be available from the stable storage when the site comes back up.

The two-phase commit protocol, as suggested by its name, works in two phases: In the first phase, the master reaches a global decision (*commit* or *abort*) based on the local decisions of the cohorts In the second phase, the master conveys this decision to the cohorts. The log records are written in a manner such that database consistency is maintained even if there is a crash at the master site or at the cohort sites.

The master initiates the first phase of the protocol by sending *PREPARE* messages in parallel to all the cohorts. Each cohort that is ready to commit first force-writes a *prepare* log record to its local stable storage and then sends a *YES VOTE* to the master. It then waits for the final decision from the master. At this stage, the cohort has entered a *prepared* state wherein it cannot unilaterally commit or abort the transaction. On the other hand, each cohort that decides to abort the transaction force-writes an *abort* log record and send 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 votes were *YES*, then it moves to a *committing* state by force-writing a *commit* log record and sending *COMMIT* messages to all the cohorts. The instant at which the force-write completes is when the user can be told that the transaction has committed. Each cohort after receiving a *COMMIT* message moves to the *committing* state, force-writes a *commit* log record, and sends an acknowledgement 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 *ABORTs* 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 acknowledgement to the master.

The master, after receiving acknowledgements from all the cohorts that were sent a message in the second phase, writes an *end* log record and "forgets" the transaction. The acknowledgements in the second phase are required to ensure that all the cohorts are aware of the final decision. If a site, which is in the *prepared* state, crashes before receiving the final decision, the master is forced to retain the required information until the failed site comes back up since it has not received an acknowledgement.

In case of site failures, the recovery manager of the failed site reads the log when the site comes back up and always aborts any cohort that was executing at the time of the failure *unless* the cohort was in the prepared state. In the latter case, the cohort contacts the master to find out the final decision. On the other hand, if the master fails after sending the *PREPARE* message to all the cohorts but before sending the final decision to any of them, then all the cohorts become *blocked* since they need to wait for the master to come back up before proceeding. To address this blocking problem, a non-blocking **three phase commit (3PC)** protocol [Ske82] has been proposed in the literature. However, this protocol has high communication overheads because of its greater message complexity, and is therefore not commonly used.

Real-Time Commit Protocols

In a real-time environment, the above commit protocols may have an adverse effect on the number of transactions completing their deadlines because of the communications overheads involved in the protocols. To address this issue, commit strategies that employ the notion of *compensating transactions* have been recently proposed [Kor90, Yoon94]. A problem with these protocols, however, is that they require prior knowledge of the semantics of the applications, which may not always be available for all applications, and hence are not suited for the general purpose database systems.

Another issue related to the real-time environment is the following: In two-phase commit protocol, the cohorts can not unilaterally *abort* after they have sent the *YES VOTE* to the master, as discussed above. However, the concurrency control mechanisms used in the real-time environment (for example, 2PL-HighPriority [Abb88]) typically abort a transaction if a higher priority transaction requires any of the data items that are currently held by the committing transaction. In this situation, the database system needs to ensure that transactions in the prepared state are not aborted by the concurrency control manager.

3 Our Work

In our research work, we have started investigating the impact of the choice of commit protocol on the performance of distributed real-time database systems. As a first step towards achieving this goal, we have conducted a preliminary simulation study to evaluate the performance impact, in terms of missed transaction deadlines, of the two-phase commit protocol described in the previous section. The simulation model and our experimental results are described in detail in the following sections.

In a database system, conflicts can arise for both *physical* resources (CPU, disk, etc.) and *logical* resources (data). For physical resources, simple priority-based scheduling schemes such as Preemptive-Resume and Head-Of-Line are available and are used in our study. For data access, however, conflicts have to be resolved in a manner that maintains database consistency. The data access policies used in database systems are commonly referred as **concurrency control** techniques. Concurrency control techniques preserve the database integrity by ensuring that transactions operate in a *serializable* schedule [Esw76]. In recent years, several concurrency control mechanisms that are specially tailored for real-time database systems have been developed. In our study, we use the well-known 2PL High Priority (2PL-HP) algorithm [Abb88], which modifies the classical two-phase locking (2PL) protocol [Esw76] by incorporating a high priority conflict resolution scheme for ensuring that the high priority transactions are not delayed by low priority transactions. In 2PL-HP, when a transaction requests a lock on a data item that is held by one or more higher priority transactions in a conflicting lock mode, the requesting transaction waits for the item to be released. On the other hand, if the data item is held by only lower priority transactions in a conflicting lock mode, the lower priority transactions are *restarted* and the requesting transaction is granted the desired lock. Note that if priorities are assigned uniquely (as is usually the case in real-time database systems), 2PL-HP is inherently deadlock-free.

4 Simulation Model

To evaluate the performance impact of commit protocols, we have developed a detailed simulation model of a distributed real-time database system. Our model is loosely based on the simulation models discussed in [Car88, Har92]. The model consists of a database that is distributed, in a non-replicated manner, over n 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 details of the concurrency control algorithms; 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. A *network manager* models the behavior of the communications network.

At each site, transactions arrive in an independent Poisson stream. Each transaction has an associated deadline, and the deadline is "firm" – that is, transactions which miss their deadlines are considered to be worthless and are immediately discarded from the system without being executed to completion [Har91]. Transactions are executed according to the distributed coordinator–participant model discussed earlier. The transaction priority assignment in all of our experiments is *Earliest Deadline* – transactions with earlier deadlines have higher priority than transactions with later deadlines. The 2PL-High Priority scheme is used for concurrency control of conflicting transactions.

We now provide a brief description of each component of the model.

Database Model

The database model parameters are summarized in Table 1. The distributed database is modeled as a collection of $files$, The parameter $NumSites$ is the number of sites, and $NumFiles$ specifies the number of files in the database. Files are assumed to be the unit of distribution across the sites in the distributed database system. The mapping of files to sites is specified via the parameter $FileLocations_{i,j}$, a boolean array which determines the presence of file at a site. Each file is a collection of pages, and $FileSize_i$ determines the number of pages in each file. Transactions make requests for data pages and locking is implemented at the page level.

$NumSites$	Number of sites in the database
$NumFiles$	Number of files in the database
$FileSize_i$	Number of pages in file i
$FileLocations_{i,j}$	Distribution of files at sites

Table 1: Database Model Parameters

$ArrivalRate$	Transaction arrival rate
$SlackFactor$	Slack Factor in Deadline formula
$FileCount$	Number of files accessed
$NumPages$	Average number of pages accessed per file
$WriteProb$	Write probability

Table 2: Workload Parameters for a site

Source

The source generates the transaction workload at each site. Table 2 summarizes the key parameters of the workload model for a site. The workload characterizes transactions in terms of the files that they access and the number of pages that they request in each file. Transactions at each site arrive in an independent Poisson stream at a rate specified by the $ArrivalRate$ parameter. Transactions are assigned deadlines 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 data processing. The $SlackFactor$ parameter is a constant that provides control over the tightness/slackness of transaction deadlines. The parameter $FileCount$ is the number of files the transaction will access and $NumPages$ is the average number of pages the the transaction access from each file. The actual number of pages accessed by a transaction varies uniformly between half and one-and-a-half times the average. The $WriteProb$ parameter determines the probability that a transaction operation is a write.

Transaction Manager

The transaction manager is responsible for accepting transactions from the source and modeling their execution. Based on the data requirements of the transaction, the cohorts are initiated at the sites where the data required by the transaction resides.

Concurrency Control Manager

The concurrency control manager models the concurrency control mechanism. In our study, the 2PL High-Priority real-time concurrency control protocol is implemented and is tailored to integrate with the commit protocol, as discussed earlier.

Resource Manager

The physical resources of the database system consist of multiple CPUs and multiple disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemption 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 $NumCPUs$ and $NumDisks$ parameters specify the hardware resource composition, while the $PageCPU$ and $PageDisk$ parameters capture CPU and disk processing times per data page. Actual disk time is uniformly distributed between half and one-and-a-half times the average. Time required for CPU is exponentially distributed. We assume that preemption costs at the CPU are negligible compared to page processing times.

Recovery Manager

The recovery manager captures the semantics of the commit protocols, and is the only module which is changed for different algorithms. This module controls the behavior of a transaction in its commit phase, and reaches a global decision based on the individual decisions of the participating cohorts. The master process invokes the commit protocol and coordinates with the cohorts through messages sent over the communication network.

Sink

After the transactions are done (committed or aborted due to expiry of deadline), they are sent to Sink, where the statistics about the transactions is collected.

Network Manager

The network manager models the behavior of the communication network. Currently our network model is quite simplistic, and acts as a switch for routing the messages. This is because we assume a local area network where the actual time on the wire for sending the message is negligible. The CPU overheads of the message transfer are captured by the parameter $MsgCPUTime$, which gives the message processing time at both the sending and receiving sites. This cost assumption has become fairly common in the analysis of locally distributed systems. Table 3 specifies the various parameters of the system.

Transaction execution is modeled in the following fashion: When the transaction arrives, it is assigned the set of files and data pages that it has to access. The master is then loaded at the originating site, and it sends a message to initiate each of its cohorts at the participating sites. Each

$NumCPUs$	Number of CPUs per site
$NumDisks$	Number of disks per site
$PageCPU$	CPU time for processing a data page
$PageDisk$	Disk service time for a page
$MsgCPUTime$	Message send or receive time

Table 3: System Parameters

cohort then makes a series of read and write accesses. A read access involves a concurrency control request to get access permission, followed by a disk IO to read the page, followed by a period of CPU usage for processing the page. Write requests are same except for the disk IO; 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. If the transaction is unable to finish the processing within its deadline, it is aborted and discarded. Otherwise, when the transaction has finished its data processing, the commit protocol is initiated, and the transaction is then commited or aborted as per the outcome of the protocol.

5 Experiments and Results

In this section, we present the performance results from our simulation experiments. The simulator used to obtain the results is written in C++SIM. In our experiments, the performance metric is $MissPercent$, which is the percentage of transactions that do not complete before their deadlines. To study the performance impact of the two-phase commit protocol, we have compared it against a baseline "no-phase" commit protocol. In the no-phase commit protocol, the consistency of the database is disregarded and no message exchange between the cohorts and the master is done for the purpose of committing. The cohorts take their individual decisions to *commit* or to *abort*. The transaction is counted as *committed* if all the cohorts have committed, else the transaction is counted as aborted. Aborted transactions are restarted immediately if they have not passed their deadlines. No a priori knowledge about the transactions is assumed, and the transactions are allowed to run till their deadlines. Once the deadline is reached, the transactions are immediately aborted and discarded.

Due to space limitations, we present the performance results for only one representative experiment here. For this experiment, the workload and system parameter settings correspond to those listed in Table 4. These settings ensure that there is both significant data contention and resource contention in the system, thus helping to bring out the performance differences between the commit protocols.

For the above experiment, the MissPercent performance is shown in Figure 1 for both the two-phase commit protocol (2PC) and the no-phase commit protocol (NPC). We observe from this figure that there is an appreciable increase in the number of transactions that miss their deadlines

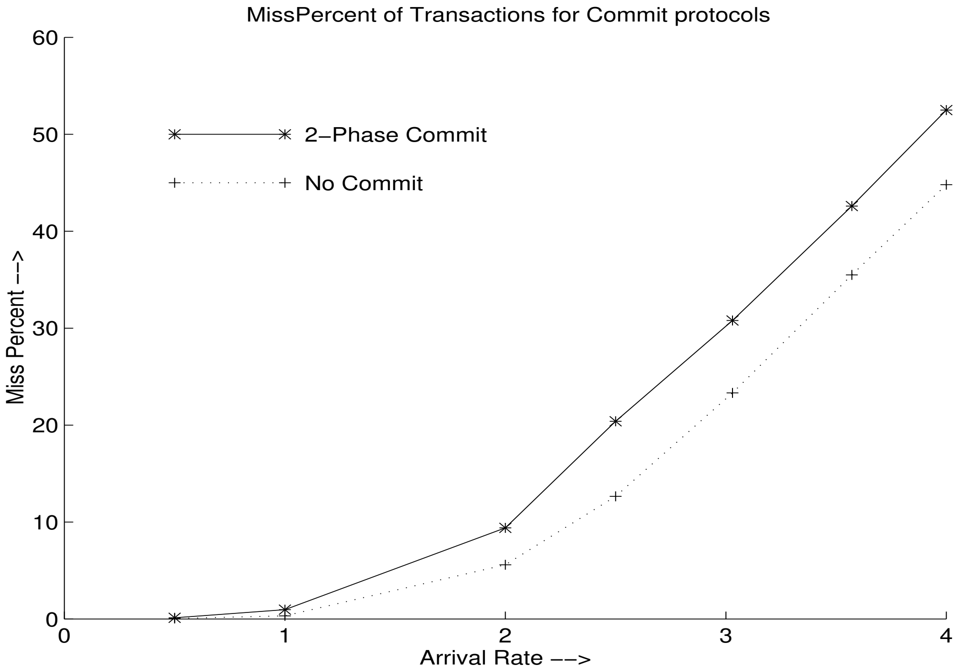

Fig 1 : Miss Percent

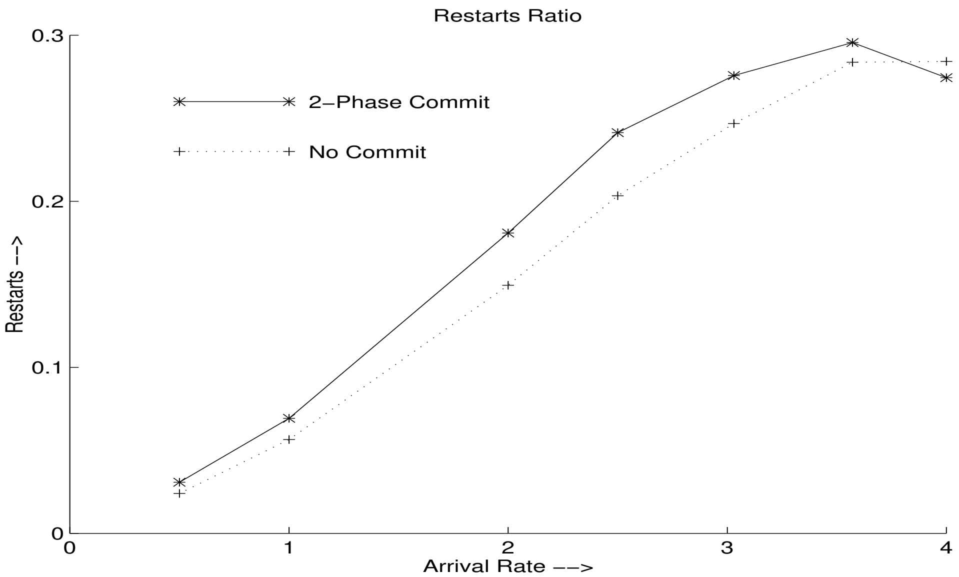

Fig 2 : Restarts Ratio

$NumSites$	8	$NumCPUs$	1
$NumFiles$	24	$NumDisks$	2
$FileSize$	100	$PageCPU$	8 millisecond
$FileCount$	3	$PageDisk$	20 millisecond
$NumPages$	6	$MsgCPUTime$	1 millisecond
$WriteProb$	1	$SlackFactor$	3

Table 4: Parameter Settings

due to commit protocol processing. At low arrival rates, especially, the relative performance of 2PC is more than 100 percent worse than that of NPC.

Figure 2 shows the corresponding *Restart Ratio* for the above experiment. The restart ratio is the average number of restarts suffered by transactions during their sojourn in the system. This figure indicates that in 2PC, transactions suffer a larger number of restarts as compared to NPC. The reason for this is that the transactions hold the locks on the data items for longer periods due to commit protocol, thus causing other transactions to wait, making them vulnerable for aborts. This results in higher number of transactions being restarted, which directly affects the capability of transactions to meet the deadlines.

6 Conclusions

In this paper, we have made a preliminary study of the performance cost of commit protocols in a distributed real-time database system. Using a detailed simulation model of an RTDBS, we studied the performance of the two-phase commit protocol against when no commit protocol is used. Our experiments showed that the two-phase commit protocol results in a significant increase in the number of missed deadlines. In our future work, we intend to investigate ways by which the performance deterioration could be reduced without sacrificing database consistency.

References

[Abb88] Abbott, R., Garcia-Molina, H. Scheduling Real-Time Transactions: a Performance Evaluation, *Proc. of 14th Intl. Conf. on Very Large Databases*, August 1988.

[Car88] Carey, M., Livny, M. Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication, *Proc. of 14th Intl. Conf. on Very Large Databases*, August 1988.

[Esw76] Eswaran, K., et al, The Notions of Consistency and Predicate Locks in a Database Systems, *Communications of ACM*, November 1976.

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

[Har91] Haritsa, J., Carey, M., Livny, M. Earliest Deadline Scheduling for Real-Time Database Systems, *Proc. of 1991 IEEE Real-Time Systems Symp.*, December 1991.

[Har92] Haritsa, J., Carey, M., Livny, M. Data Access Scheduling in Firm Real-Time Database Systems, *The Journal of Real-Time Systems*, 4, 1992.

[Kor90] Korth, H.F., Levy, E., Silberschatz, A. A formal approach to recovery by compensating transactions, *Proc. of 16th Intl. Conf. on Very Large Databases*, August 1990.

[Moh86] Mohan, C., Lindsay, B., Obermarck, R. Transaction Management in the R^* Distributed Database Management System, *ACM Transactions on Database Systems*, 11(4), 1986.

[Sha88] Sha, L., Rajkumar, R., Lehoczky, J.P. Concurrency Control for Distributed Real-Time Databases, *ACM SIGMOD Record*, 17(1), March 1988.

[Ske82] Skeen, D. Nonblocking Commit Protocols, *Proc. of ACM SIGMOD* , June 1982.

[Son90] Son, S.H. Real-Time Database Systems: A New Challenge, *IEEE 6th Intl. Conf. on Data Engineering*, 13(4), December 1990.

[Yoon94] Yoon, Y.I. Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems, *Ph.D. Thesis*, Korea Advanced Institute of Science and Technology, May 1994.