©2010 International Journal of Computer Applications (0975 – 8887)
Volume 1 – No. 3

SPEEDITY-A Real Time Commit Protocol

S. Agrawal
Dept of CSE
M. M. M. E. College
Gorakhpur, India

Udai Shanker
Dept of CSE
M. M. M. E. College
Gorakhpur, India

Abhay N. Singh
Dept of CSE
M. M. M. E. College
Gorakhpur, India

A. Anand
Dept of CSE
M. M. M. E. College
Gorakhpur, India

ABSTRACT
This paper presents Shadow, Piggy bag, Elemental External Dependency Inversion and in Time Yielding (SPEEDITY) commit protocol for distributed real time database systems (DRTDBS). Here, only abort dependent cohort having deadline greater than a specific value ($T_{shadow_creation_time}$) needs to forks off a replica of itself called a shadow, whenever it borrows dirty value of a data item. Commit-on-Termination external dependency between final commit of lender and shadow of its borrower and Begin-on-Abort internal dependency between shadow of borrower and borrower itself are defined. Due to heavy delay in commitment of lender in the case of update-read conflict, execution of borrower is started with its shadow by sending YES-VOTE message piggy bagged with the before value [11] to its coordinator after aborting it and abort dependency created between lender and borrower is reversed to commit dependency between shadow and lender with read-update conflict and commit operation governed by Commit-on-Termination dependency. The performance of SPEEDITY is compared with shadow PROMPT, SWIFT and DSS-SWIFT commit protocols [6, 22, 23] for both main memory resident and disk resident databases with and without communication delay. Simulation results show that the proposed protocol improves the system performance up to 5% as transaction miss percentage.

Categories and Subject Descriptors
H.2.4 [Database Management System]: System---Transaction Processing

General Terms
Algorithms, Performance, Design, Theory, Verification

Keywords
Distributed Real Time Database System, Commit Protocol, Conflict Resolution, Dependency Inversion, Lender, Borrower.

1. INTRODUCTION
Database systems are currently being used as backbone to thousands of applications. Some of these have very high demands for high availability and fast real-time responses. Typically, these systems generate a very large transaction workload against the distributed real time database, and a large part of the workload consists of read, write and update transactions. Unavailability of real time or slow response in processing these transactions used by business applications could, however, be financially devastating and, in worst case, cause injuries or deaths. Examples include telecommunication systems, trading systems, online gaming, sensor networks etc. Typically, a sensor network consists of a number of sensors (both wired and wireless) which report on the status of some real-world conditions. The conditions include sound, motion, temperature, pressure & moisture, velocity etc. The sensors send their data to a central system that makes decisions based on both present and past inputs. To enable the networks to make better decisions, both the number of sensors and the frequency of updates should be increased. Thus, sensor networks must be able to tolerate an increasing load. For applications such as health care in a hospital, automatic car driving systems, space shuttle control, etc., data is needed in real-time and must be extremely reliable as any unavailability or extra delay could result in loss of human lives [7]. Recent years have seen increasing interest in providing support for warehouse-like systems that support fine-granularity insertions of new data and even occasional updates of incorrect or missing historical data; these modifications need to be supported concurrently with traditional updates. Such systems are useful for providing flexible load support in traditional warehouse settings, for reducing the delay for real-time data visibility, and for supporting other specialized domains such as customer relationship management (CRM) and data mining where there is a large quantity of data that is frequently added to the database in addition to a substantial number of read-only analytical queries to generate reports and to mine relationships. These “updatable warehouses” have the same requirements of high availability and disaster recovery as traditional warehouses but also require some form of concurrency control, commit protocol and recovery to ensure transactional semantics. Many applications listed above using DRTDBS require distributed transaction executed at more than one site. Traditional log-based systems require sites force-write log records to disk at various stages of commit processing in order to ensure atomicity. A commit protocol ensures that either all the effects of the transaction persist or none of them persist despite the failure of site or communication link and loss of messages. The Commit processing should add as little overhead as possible to transaction processing. Therefore, the design of a better commit protocol is very important for DRTDBS.

2. BACKGROUND
The two phase commit protocol (2PC) referred to as the Presumed Nothing 2PC protocol (PrN) is the most commonly used protocol in the study of DDBS [1, 2, 3]. It ensures that sufficient information is force-written on the stable storage to

reach a consistent global decision about the transaction. A number of 2PC variants [12] have been proposed and can be classified into following four groups [8].

- Presumed Abort/Presumed Commit Protocols
- One Phase Commit Protocols
- Group Commit Protocols
- Pre Commit/Optimistic Commit Protocols

Presumed commit (PC) and presumed abort (PA) [13] are based on 2PC. Soparkar et al. [24] have proposed a protocol that allows individual site to unilaterally commit. Gupta et al. proposed optimistic commit protocol and its variant in [4, 5]. Enhancement has been made in PROMPT commit protocol, which allows executing transactions to borrow data in a controlled manner only from the healthy transactions in their commit phase. However, it does not consider the type of dependencies between two transactions. The impact of buffer space and admission control is also not studied. In case of sequential transaction execution model, the borrower is blocked for sending the WORKDONE message and the next cohort can not be activated at other site for its execution. It will be held up till the lender completes. If its sibling is activated at another site anyway, the cohort at this new site will not get the result of previous site because previous cohort has been blocked for sending of WORKDONE message due to being borrower. In shadow PROMPT, a cohort forks off a replica of the transaction, called a shadow, without considering the type of dependency whenever it borrows a data page.

Lam et al. proposed deadline-driven conflict resolution (DDCR) protocol which integrates concurrency control and transaction commitment protocol for firm real time transactions [11]. DDCR resolves different transaction conflicts by maintaining three copies of each modified data item (before, after and further) according to the dependency relationship between the lock requester and the lock holder. This not only creates additional workload on the systems but also has priority inversion problem. The serializability of the schedule is ensured by checking the before set and the after sets when a transaction wants to enter the decision phase. The protocol aims to reduce the impact of a committing transaction on the executing transaction which depends on it. The conflict resolution in DDCR is divided into two parts (a) resolving conflicts at the conflict time; and (b) reversing the commit dependency when a transaction, which depends on a committing transaction, wants to enter in the decision phase and its deadline is approaching.

If data conflict occurs between the executing and committing transactions, system's performance will be affected. Pang Chung-leung and Lam K. Y. proposed an enhancement in DDCR called the DDCR with similarity (DDCR-S) to resolve the executing-committing conflicts in DRTDBS with mixed requirements of criticality and consistency in transactions [14]. In DDCR-S, conflicts involving transactions with looser consistency requirement and the notion of similarity are adopted so that a higher degree of concurrency can be achieved and at the same time the consistency requirements of the transactions can still be met. The simulation results show that the use of DDCR-S can significantly improve the overall system performance as compared with the original DDCR approach.

Based on PROMPT and DDCR protocols, Qin B. and Liu Y. proposed double space commit (2SC) protocol [15]. They analyzed and categorized all kind of dependencies that may occur due to data access conflicts between the transactions into two types commit dependency and abort dependency. The 2SC protocol allows a non-healthy transaction to lend its held data to the transactions in its commit dependency set. When the prepared transaction aborts, only the transactions in its abort dependency set are aborted and the transactions in its commit dependency set execute as normal. These two properties of the 2SC reduce the data inaccessibility and the priority inversion that is inherent in distributed real-time commit processing. 2SC protocol uses blind write model. Extensive simulation experiments have been performed to compare the performance of 2SC with that of other protocols such as PROMPT and DDCR. The simulation results show that 2SC has the best performance. Furthermore, it is easy to incorporate it in any current concurrency control protocol.

Ramamritham et al. [17] have given three common types of constraints for the execution history of concurrent transactions. The paper [16] extends the constraints and gives a fourth type of constraint. Then the weak commit dependency and abort dependency between transactions, because of data access conflicts, are analyzed. Based on the analysis, an optimistic commit protocol Two-Level Commit (2LC) is proposed, which is specially designed for the distributed real time domain. It allows transactions to optimistically access the locked data in a controlled manner, which reduces the data inaccessibility and priority inversion inherent and undesirable in DRTDBS. Furthermore, if the prepared transaction is aborted, the transactions in its weak commit dependency set will execute as normal according to 2LC. Extensive simulation experiments have been performed to compare the performance of 2LC with that of the base protocols PROMPT and DDCR. The simulation results show that 2LC is effective in reducing the number of missed transaction deadlines. Furthermore, it is easy to be incorporated with the existing concurrency control protocols.

The SWIFT commit protocol is beneficial only if the database is main memory resident. The unnecessary creation of shadow by Shadow PROMPT is solved to some extent in DSS-SWIFT commit protocol. However, DSS-SWIFT still creates the non beneficial shadows in some cases.

The remainder of this paper is organized as follows. Section 3 introduces the distributed real time database system model with assumptions. Section 4 presents SPEEDITY commit protocol and its pseudo code. Section 5 discusses the simulation results. Section 6 presents an outlook on future work. Section 7 finally concludes the paper.

3. Distributed Real Time Database System Model

In distributed database system model, the global database is partitioned into a collection of local databases stored at different sites. A communication network interconnects the sites. There is no global shared memory in the system, and all sites communicate via message exchange over the communication

network. We assume that the transactions are firm real time type. Each transaction in this model exists in the form of a coordinator that executes at the originating site of the transaction and a collection of cohorts that execute at various sites, where the required data items reside. If there is any local data in the access list of the transaction, one cohort is executed locally also. Before accessing a data item, the cohort needs to obtain lock on data items. Sharing of the data items in conflicting modes creates dependencies among the conflicting local transactions/cohorts, and constraints their commit order. We also assume that:

- The processing of a transaction requires the use of CPU and data items located at local site or remote site.
- Arrival of transactions at a site is independent of the arrivals at other sites and uses Poisson distribution.
- Each cohort makes read and update accesses.
- Each transaction pre-declares its read-set (set of data items that the transaction will only read) and write-set (set of data items that the transaction will write).
- S2PL-HP is used for locking the data items.
- Cohorts are executed in parallel.
- A lending transaction cannot lend the same data item in read/update mode to more than one cohort.
- The cohort already in the dependency set of another cohort cannot permit another incoming cohort to read or update.
- A distributed real time transaction is said to commit, if the coordinator has reached to the commit decision before the expiry of the deadline at its site. This definition applies irrespective of whether cohorts have also received and recorded the commit decision by the deadlines or not.
- Studies have been made for both main memory resident and disk resident database.
- Communication delay considered is either 0 or 100 ms.
- In case of disk resident database, buffer space is sufficiently large to allow the retention of all data updates until commit time.
- The updating of data items is made in transaction own memory rather than in place updating.

4. SPEEDITY

In this sub section, we introduce our protocol which is combined with SWIFT. The conflict resolution in DDCR is solved by resolving the conflicts at the conflict time and reversing the commit dependency when a transaction, which depends on a committing transaction, wants to enter in its decision phase and its deadline is approaching. Here, in case of Write-Read conflict, the dependency can not be reversed if the lender has entered in decision phase. In the following sub section, we will discuss how the problem has been solved in SPEEDITY with help of concept of shadowing and deferred commitment of the transaction.

4.1 Tshadow_creation_time Computation

The deadline of a transaction is controlled by the runtime estimate of a transaction and the parameter slack factor, which is the mean of an exponential distribution of slack time. We allocate deadlines to arriving transactions using the method given below. The deadlines of transactions (both global and local) are calculated based on their expected execution times [22, 9, 11]. The deadline (*Di*) of transaction (*Ti*) is defined as:

$$Di = Ai + SF * Ri$$

where, *Ai* is the arrival time of transaction (*Ti*) at a site; SF is the slack factor; *Ri* is the minimum transaction response time. As cohorts are executing in parallel, the *Ri* can be calculated as:

$$Ri = Rp + Rc$$

where, *Rp*, the time for execution phase and *Rc*, the time for commitment phase are given as below. For global transaction

$$Rp = \max.\ ((2 * T\text{lock} + T\text{process}) * N\text{oper local}, (2 * T\text{lock} + T\text{process}) * N\text{oper remote})$$

$$Rc = N\text{comm} * T\text{com}$$

For local transaction

$$Rp = (2 * T\text{lock} + T\text{process}) * N\text{oper local}$$

$$Rc = 0$$

Where, *T*lock is the time required to lock/unlock a data item; *T*process is the time to process a data item (assuming read operation takes same amount of time as write operation); *N*comm is no. of messages; *T*com is communication delay i.e. the constant time estimated for a message going from one site to another; *N*oper local is the number of local operations; *N*oper remote is maximum number of remote operations taken over by all cohorts. If T_2 is abort dependent on T_1

$$T_{shadow_creation_time} = R_1 + R_2$$

Where, R_1=Deadline Time of T_1 and R_2 is minimum Time required for T_2 from sending Yes Vote response to finally committing.

4.2 Types of Dependencies

Sharing of data items in conflicting mode creates dependencies among conflicting transactions and constraints their commit order. We assume that a cohort requests an update lock if it wants to update a data item *x*. The prepared cohorts, called as lenders. lend uncommitted data to concurrently executing transactions known as borrower. If a cohort fork off a replica of the transaction, it is called as shadow. The original incarnation of the transaction continues its execution, while the shadow is blocked after finishing its execution. If the lender finally commits, the borrower continues its on-going execution and the shadow is discarded; otherwise, borrower is aborted due to abort of lender and shadow is activated. Two new dependencies are

defined apart from commit and abort dependencies. The modified definitions of dependencies used in this paper are given below.

Commit dependency (CD). If a transaction T_2 updates a data items read by another transaction T_1, a commit dependency is created from T_2 to T_1. Here, T_2 is called as commit dependent and is not allowed to commit until T_1 commits.

Abort dependency (AD). If T_2 reads/updates an uncommitted data item updated by T_1, an abort dependency is created from T_2 to T_1. Here, T_2 is called as abort dependent. T_2 aborts, if T_1 aborts and T_2 is not allowed to commit before T_1.

Begin-on-Abort Dependency (BAD). This dependency is created between shadow and borrower who created it. Here, shadow of cohort will be activated, only if borrower aborts due to abort of its lender.

Commit-on-Termination Dependency (CTD). This dependency is created between final commit operations of lender and shadow of its borrower. The final commit operation by a lender is deferred and resumed only after termination (i.e. commit or aborts) of shadow.

Here, BAD is internal dependency while others are external dependencies [27]. Each site maintains shadow set (DDDS) also, which is the set of shadows of those cohorts that are abort dependent on lender and deadline beyond $T_{shadow_creation_time}$.

DDDS (S_i): Shadow of those cohorts which are in abort dependency set of T_1

Hence, if T_2 is abort dependent on T_1 and fulfills the shadow creation criteria, then Shadow of T_2, S_2 is created in DDDS (T_1). Each transaction/cohort T_i, that lends its data while in prepared state to an executing transaction/cohort T_j, maintains three/four sets.

- Commit Dependency Set CDS (T_i): set of commit dependent borrower T_j, that has borrowed dirty data from lender T_i.
- Abort Dependency Set ADS (T_i): set of abort dependent borrower T_j that has borrowed dirty data from lender T_i.
- Begin-on-Abort Dependency Set BAD (T_j): set of shadow of borrower T_j who has created it. T_j has borrowed dirty data from lender T_i.
- Commit-on-Termination Dependency Set CTD (T_j): set of shadow of a borrower attached with its lender whose dependency has been reversed.

4.3 Type of Dependency in Different Cases of Data Conflicts

Case 1: Read-Update Conflict
The lock manager processes the data item accesses in conflicting mode as follows.

```
If (T2 CD T1)
{
        CDS (T1) =CDS (T1) {T2};
        T2 is granted Update lock;
}
else
{
        if ((T2 AD T1) AND (HF(T1) ≥ MinHF))
        {
            ADS (T1) =ADS (T1) {T2};
            T2 is granted the requested lock;
            If (deadline (T2)>Tshadow_creation_time)
            {
                Add shadow of T2 in DDDS (Si);
                BAD (T2) =BAD (T2) {T2's shadow};
            }
        }
      else if (T2 has read dirty value of T1 and received
      VOTE-REQ message from its coordinator)
        {
            CDS (shadow (T2)) =CDS (shadow (T2)){T1};
            CTD (T1) = CTD (T1) {T2};
            Shadow of T2 is activated and granted the
            requested lock after aborting T2 and deleting
            T2 from ADS (T1);
        }
}
```

4.4 Mechanics of Interaction between Lender and Borrower Cohorts

If T_2 accesses a data item already locked by T_1, one of the following four scenarios may arise.

Scenario 1: T_1 receives decision before T_2 has completed its local data processing:
If the global decision is to commit,
T_1 commits. All cohorts in ADS (T_1) & CDS (T_1) will execute as usual.
T_2 completes its commit operation, if it has been deferred.
Sets of ADS (T_1), BAD (T_2), DDDS (S_i) & CDS (T_1) will be deleted.
If the global decision is to abort,
T_1 aborts. Cohorts in the dependency set of T_1 will execute as follows:
T_2 completes its commit operation, if it has been deferred.
Shadows of all cohorts Begin-on-Abort Dependent on T_2 in DDDS (S_i) will be activated and sends YES-VOTE to their coordinator, only if they can complete execution; otherwise, discarded;
Transactions in CDS (T_1) will execute normally.
Delete Set ADS (T_1), BAD (T_2), DDDS (S_i) and CDS (T_1).

Scenario 2: T_2 is about to start processing phase after getting all its locks before T_1 receives global decision.
T_2 sends WORKSTARTED message to its master.

Scenario 3: T_2 aborts before, T_1 receives decision

In this situation, all works carried out by T_2 are undone and T_2 is removed from the dependency set of T_1.

Scenario 4: T_2 has read dirty value of T_1 and received VOTE-REQ message from its coordinator before T_1 receives commit decision message.
Abort T_2 and Delete T_2 from ADS (T_1).
Activate execution of T_2 with its shadow.
Add T_2 in CTD (T_1).
Shadow sends YES-VOTE message piggy bagged with the new result to its coordinator.

4.5 Algorithm

On the basis of above discussions, the complete pseudo code of the protocol is given below.

```
if (T1 receives global decision before, T2 ends execution)
 {if (T1 is in Commit-on-Termination Dependency Set)
     Defer commit processing till termination of shadow.
 else
 {
     One: if (T1's global decision is to commit)
        {
          T1 enters in the decision phase;
          T2 completes its commit operation, if it has been
          deferred.
          Delete sets of ADS (T1), BAD (T2), DDDS (Si) and
          CDS (T1);
        }
      else //T1's global decision is to abort
        {
          T1 aborts;
          T2 completes its commit operation, if it has been
          deferred.
          Transactions in CDS (T1) will execute as usual.
           For all the abort dependent cohorts
          {
             if (shadow)
               {
                IF (Shadow of cohort can complete its execution)
                  Execute Cohorts Shadow in DDDS (Si) and
                  send YES-VOTE;
              else
                Discard shadow;
               }
           }
             Delete sets of ADS (T1), BAD (T2), DDDS (Si) and
             CDS (T1);
        }
 }
}
else
{
    if (T2 aborted by higher transaction before T1 receives
    decision OR T2 expires its deadline)
    {
      Undo the computation of T2;
      Abort T2;
      Delete T2 from CDS (T1) & ADS (T1);
     if (shadow)
      Delete T2 from DDDS (Si);
    }
      else if (T2 ends executing phase  before T1 receives global
             decision )
          T2 sends WORKSTARTED message;

             else if (T2 has read dirty value of T1 and received
             VOTE-REQ message from its coordinator)
               {
                    Shadow of T2 is activated and granted the
                    requested lock after aborting T2 and deleting
                    T2 from ADS (T1);
                    Shadow sends YES-VOTE message piggy
                    bagged with the new result;
               }
}
```

4.6 Main Contributions

1. Abort dependent cohort having deadline beyond a specific value ($T_{shadow_creation_time}$) can only forks off a replica of itself called a shadow
2. Two new dependencies Begin-on-Abort and Commit-on-Termination are defined.
3. Reversing of abort dependency created between lender and borrower due to Update-Read conflict to commit dependency between shadow of borrower and lender. The final commit operations of lender and shadow is governed by Commit-on-Termination Dependency.
4. On activation of borrower transaction with help of shadow, it sends YES-VOTE message piggy bagged with the new result to its coordinator in case of abort of lender & borrower

To maintain consistency of database, cohort sends the YES-VOTE in response to its coordinator's VOTE-REQ message only when its dependencies are removed & it has finished its processing, and, in case of reversal of dependency, final commit operation by a lender is deferred and resumed only after termination (i.e. commit or aborts) of shadow [18].

5. PERFORMANCE EVALUATION

The default values of different parameters for simulation experiments are given below and same as taken in [9, 19]. The concurrency control scheme used is static two phase locking with higher priority [10]. Miss Percentage (MP) is the primary performance measure used in the experiments and is defined as the percentage of input transactions that the system is unable to complete on or before their deadlines [26]. Since, there were no practical benchmark programs available in the market or with research communities to evaluate the performance of protocols and algorithms, an event driven based simulator was written in C language [25]. In our simulation, a small database (200 data items per site) is used to create high data contention

environment. For each set of experiment, the final results are calculated as an average of 10 independents runs. In each run, 100000 transactions are initiated.

5.1 Simulation Results

Simulation was done for both the main memory resident and the disk resident databases at communication delay of 0ms and 100ms. We compared SPEEDITY with shadow PROMPT, SWIFT and DSS-SWIFT in this experiment. Figure 1 to Figure 5 show the Miss Percent behaviour under normal and heavy load conditions with/without communication delay. In these graphs, we first observe that there is a noticeable difference between the performances of the various commit protocols throughout loading range. Let us consider the case of update-read conflict. If there is serious problem in commitment of lender, the final commit decision can be delayed for an indefinite time. Meanwhile, it is possible that borrower has received VOTE-REQ message from its coordinator. Now, the execution of transaction can be started with borrower's shadow after aborting borrower itself and abort dependency created between lender and borrower due to update-read conflict is reversed to commit dependency between shadow and lender with read-update conflict. Now, the shadow sends YES-VOTE message piggy bagged with the new result (initial value of data item) to its coordinator In this way, shadow and lender can proceed for their execution without any further much delay and interferences. Only, the final commit operations of lender will be deferred till termination of shadow. Again, let us take the case of update-update or update-read conflicts. If the lender and, in turn, its borrower have been aborted, the execution of borrower transaction can be started with its shadow in case of any chance of completion. Here, the shadow sends YES-VOTE message piggy bagged with the new result (initial value of data item) to its coordinator because it has completed all the activities parallel with borrower's execution. In both the above cases, the survival of transaction with borrower's shadow utilizes the concept of single phase commit protocol but sends YES-VOTE message as compared to WORKDONE Message. Due to this reason, it is free from disadvantage of single phase commit protocol of long duration data item locking. Here the work done by borrower is never wasted in most of the cases even if a wrong borrowing decision is made. Due to aforementioned reason, SPEEDITY minimizes the number of messages needed for execution and commit of cohort, and is also free from long duration locking of data items. Hence, the SPEEDITY commit protocol provides a performance that is significantly better than other commit protocols.

Main Memory Resident Database

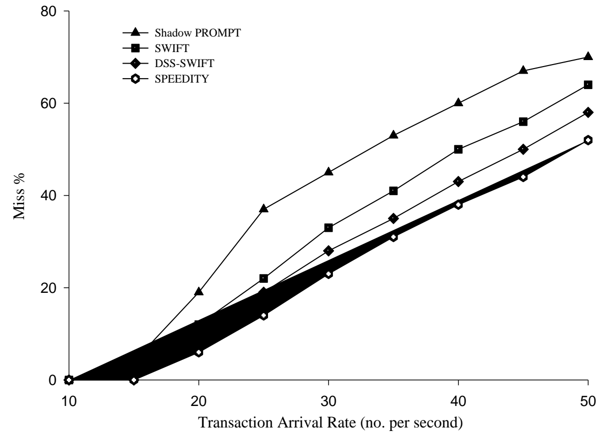

Figure 1: Miss % with (RC+DC) at Communication delay=0 ms Normal and Heavy Load

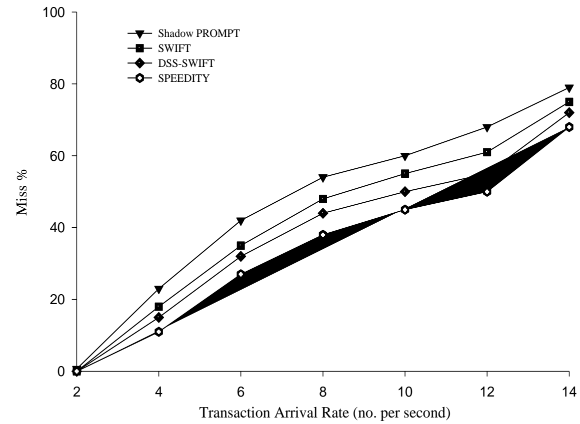

Figure 2 : Miss % with (RC+DC) at Communication delay=100 ms Normal and Heavy Load

Disk Resident Database

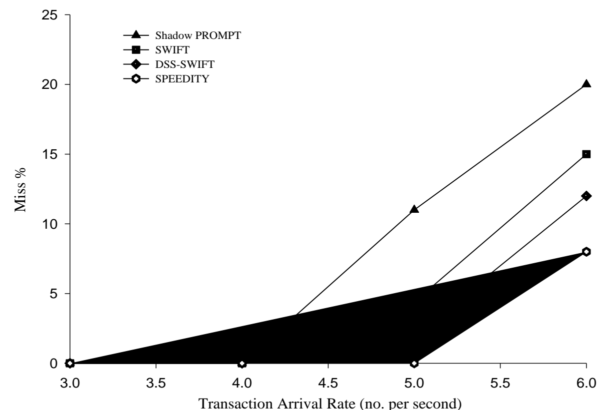

Figure 3: Miss % with (RC+DC) at Communication delay=0 ms Normal Load

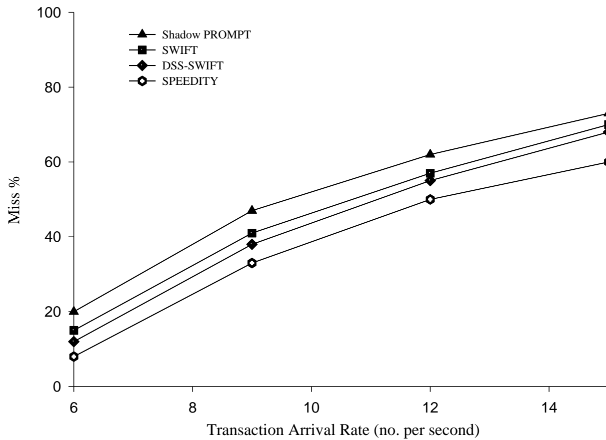

Figure 4: Miss % with (RC+DC) at Communication delay=0 ms Heavy Load

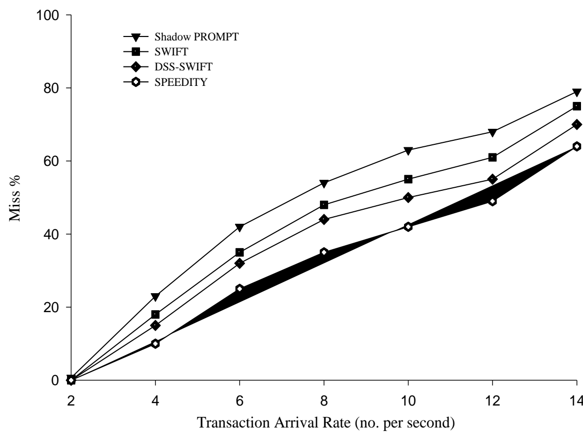

Figure 5: Miss % with (RC+DC) at Communication delay=100 ms Normal and Heavy Load

6. FUTURE RESEARCH DIRECTIONS

Following are some suggestions to extend this work [20, 21].

- Our performance studies are based on the assumption that there is no replication. Hence, a study of relative performance of the topic discussed here deserves a further look under assumption of replicated data.
- The work can be extended for Mobile DRTDBS, pear-to-pear database systems, grid database systems etc. where memory space, power and communication bandwidth are bottleneck. There is a need to design various protocols for different purposes that may suit to the specific need of hand held devices.
- The integration and the performance evaluation of proposed commit protocol with 1PC and 3PC protocols.
- Although tremendous research efforts have been reported in the hard real time systems in dealing with hard real time constraints, very little work has been reported in hard real time database systems. So, the performance of SPEEDITY can be evaluated for hard real time constrained transactions.
- Biomedical Informatics is quickly evolving into a research field that encompasses the use of all kinds of biomedical information, from genetic and proteomic data to image data associated with particular patients in clinical settings. Biomedical Informatics comprises the fields of Bioinformatics (e.g., genomics and proteomics) and Medical Informatics (e.g., medical image analysis), and deals with issues related to the access to information in medicine, the analysis of genomics data, security, interoperability and integration of data-intensive biomedical applications. Main issues in this field is provision of large computing power such that researchers have access to high performance distributed computational resources for computationally demanding data analysis, e.g., medical image processing and simulation of medical treatment or surgery and large storage capacity and distributed databases for efficient retrieval, annotation and archiving of biomedical data. What is missing today is full integration of methods and technologies to enhance all phases of biomedical informatics and health care, including research, diagnosis, prognosis, etc. and dissemination of such methods in the clinical practice, whenever they are developed, deployed and maintained. Hence it is another topic of research interest.

7. CONCLUSION

This paper presented a new commit protocol SPEEDITY with help of Commit-on-Termination dependency between final commit operations of lender and shadow of its borrower, and Begin-on-Abort dependency between shadow of borrower and borrower itself. In case of delay in commitment of lender due to some serious problem, the execution of transaction is started via reversing the abort dependency with commit dependency in between borrower's shadow and lender with read-update conflict. Only, the final commit operation of lender has been permitted to resume only after the termination of shadow. Also, the shadow has been allowed to send YES-VOTE message piggy bagged with the new result to its coordinator in case of abort of lender & borrower and activation of execution of transaction with help of borrower's shadow. In this way, the S3 improves the system performance up to 5% by minimizes long duration locking of data items and reducing the number of messages needed for commit of cohort. It is very much beneficial in abort oriented systems. It ensures the survival of transactions with shadowing approach.

8. REFERENCES

[1] Attaluri, Gopi K., and Salem, K. 2002. The Presumed-Either Two-Phase Commit Protocol. IEEE Transactions on Knowledge and Data Engineering, 14, 5(Sep. 2000), 1190-1196.

[2] Gray, J., and Reuter, A. 1993 Transaction Processing: Concepts and Technique. San Mateo, CA, USA: Morgan Kaufman.

[3] Gray, J. 1978. Notes on Database Operating Systems. Operating Systems: an Advanced Course, Lecture Notes in Computer Science, Springer Verlag, 60, 393-481.

[4] Gupta, R., Haritsa, J. R., and Ramamritham, K. 1997 More Optimism About Real-Time Distributed Commit

Processing. Technical Report. Database System Lab, Supercomputer Education and Research Centre, I.I.Sc. Bangalore, India

[5] Gupta, R., Haritsa, J. R., Ramamritham, K., and Seshadri, S. 1996. Commit processing in distributed real time database systems. In Proceedings of the Real-time Systems Symposium, Washington DC, San Francisco.

[6] Haritsa, J. R., Ramamritham, K., and Gupta, R. 2000. The PROMPT real time commit protocol. IEEE Transaction on Parallel and Distributed Systems, 11, 2(Feb. 2000), 160-181.

[7] Huang, J. 1991 Real Time Transaction Processing: Design, Implementation and Performance Evaluation. Doctoral Thesis, University of Massachusetts, USA.

[8] Inseon, L., and Yeom H. Y. 2002. A Single Phase Distributed Commit Protocol for Main Memory Database Systems. In Proceedings of the 16th International Parallel & Distributed Processing Symposium (IPDPS 2002), Ft. Lauderdale, Florida, USA.

[9] Lam, K. Y. 1994 Concurrency Control in Distributed Real-Time Database Systems. Doctoral Thesis, City University of Hong Kong, Hong Kong.

[10] Lam, K. Y., Hung, S. L., and Son, S. H. 1997. On Using Real-Time Static Locking Protocols for Distributed Real-Time Databases. Real - Time Systems, 13, 2(Sep. 1997), 141-166.

[11] Lam, K. Y., Pang, C., Son, S. H., and Cao, J. 1999. Resolving Executing-Committing Conflicts in Distributed Real - time Database Systems. Journals of Computer, 42, 8, 674-692.

[12] Misikangas, P. 1997. 2PL and Its Variants. Seminar on Real - Time Systems, Department of Computer Science, University of Helsinki.

[13] Mohan, C., Lindsay, B., and Obermarck, R. 1986. Transaction management in the R* distributed database Management System. ACM transaction on Database Systems, 11, 4(Dec. 1986), 378-396.

[14] Pang, C. L., and Lam K. Y. 1998. On Using Similarity for Resolving Conflicts at Commit in Mixed Distributed Real-time Databases. In Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications.

[15] Qin, B., and Liu, Y. 2003. High Performance Distributed Real-time Commit Protocol. Journal of Systems and Software, Elsevier Science Inc., 68, 2(Nov. 2003), 145-152.

[16] Qin, B., Liu, Y., and Yang, J. C. 2003. A Commit Strategy for Distributed Real-Time Transaction. Journal of Computer Science and Technology, 18, 5, 626-631.

[17] Ramamritham, K., and Chrysanthis, P. K. 1996. A Taxonomy of Correctness Criteria in Database Applications. Journal of the VLDB, 5, 1(Jan. 1996), 85-97.

[18] Shanker, U, Misra, M., and Sarje, Anil K. 2005. Dependency Sensitive Distributed Commit Protocol. In Proceedings of the 8th International Conference on Information Technology, Bhubaneswar, India, 41-46.

[19] Shanker, U. 2006 Some Performance Issues in Distributed Real Time Database Systems. Doctoral Thesis, Department of Electronics & Computer Engineering, Indian Institute of Technology Roorkee, India.

[20] Shanker, U., Misra, M., and Sarje, Anil K. 2001. Hard Real-Time Distributed Database Systems: Future Directions. In Proceedings of the All India Seminar on Recent Trends in Computer Communication Networks, Department of Electronics & Computer Engineering, Indian Institute of Technology Roorkee, India, 172-177.

[21] Shanker, U., Misra, M., and Sarje, Anil K. 2008. Distributed Real Time Database Systems: Background and Literature Review. International Journal of Distributed and Parallel Databases, Springer Verlag, 23, 2(April 2008), 127-149.

[22] Shanker, U., Misra, M., and Sarje, Anil K. 2006. SWIFT-A New Real Time Commit Protocol. International Journal of Distributed and Parallel Databases, Springer Verlag, 20, 1(July 2006), 29-56.

[23] Shanker, U., Misra, M., Sarje, Anil K., and Shisondia, R. 2006. .Dependency Sensitive Shadow SWIFT. In Proceedings of the 10th International Database Applications and Engineering Symposium, Delhi, India, 373-376.

[24] Soparkar, N., Levy, E., Korth, H. F., and Silberschatz, A. 1994. Adaptive Commitment for Real - Time Distributed Transaction. In Proceedings of the 3rd International Conference on Information and Knowledge Management, Gaithersburg, Maryland, United States, 187-204.

[25] Taina, J., and Son, S. H. 1999. Towards a General Real-Time Database Simulator Software Library. In Proceedings of the Active and Real-Time Database Systems.

[26] Ulusoy, O. 1992 Concurrency Control in Real-time Database Systems. Doctoral Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, USA.

[27] Xin, T. 2006 A Framework for Processing Generalized Advanced Transactions. Doctoral Thesis, Department of Computer Science, Colorado State University, USA.