Resiliency Design of a Loosely-Coupled Database System

Jae Hwa Choi
(Department of Management, Dankook University)
Sung-Eon Kim
(Department of MIS, Catholic University of Taegu-Hyosung)

Abstract

In a loosely-coupled distributed database server system, a server failure and/or a communication failure can be masked by a resiliency mechanism. Recognizing that a distributed transaction executes at several servers during its lifetime, we propose a resiliency mechanism which allows *continuous transaction processing* in distributed database server systems in the presence of a server failure. The resiliency mechanism for transaction processing is achieved by keeping redundant information using a primary/backup approach. The purpose of this paper is to analyze the performance improvement opportunities with the resiliency mechanism and to present the design of the proposed system.

1. Introduction

Database applications are evolving to the point where failure to provide service is unacceptable [Gray86]. This increased demand for continuous operation of database systems in recent years makes it necessary to maintain

backup systems, which allow the system to continue its operation even in the presence of primary failures [Borr84, Burk90, IBM87, Lyon90]. The use of backups allows both hardware and software failures to be masked at the application level. The goal is to make application services available to users despite hardware or software failures by backup systems which run on distinct hardware processors and maintain information about the global service state.

In a parallel/distributed database server system, most transactions are distributed transactions which are processed by multiple database servers. For nondistributed transactions, the effect of failures is limited to that database server and it is handled by the transaction recovery mechanism locally. In a conventional distributed system, when failures occur during transaction processing, but before the commit operation, then the entire transaction should be restarted. However, aborting the whole effort of processing distributed transactions results in a significant performance deterioration. This is particularly important for *long-lived transactions* as the loss of computing time is likely to be significant.

The performance improvement can be partly accomplished by minimizing the processing efforts lost due to failures [Shet87, LeeY87]. Recognizing that a distributed transaction executes at several servers during its lifetime, we propose a resiliency mechanism which allows *continuous transaction processing* in a distributed primary/backup database server system in the presence of a server failure. We investigate the performance improvement opportunities by reducing the loss of computing time due to

the failure. The approach considers a transaction component within a server as a unit of recovery and tracks the progress of the execution of a transaction on distributed servers.

In order to perform continuous transaction processing despite failures, every state change of a transaction processing may be saved in a backup server. Obviously, this pessimistic synchronization may give more burdens than benefits to the system. The tracking needs not be done continuously with the transaction progress. Instead, the state of all transaction processing in a system is saved periodically. This period is referred to as a *savepoint*. Then, the question is how we determine the savepoint.

The purpose of this paper is to analyze the performance improvement opportunities with the resiliency mechanism and to identify the optimization parameters for the resilient transaction processing system. In the next section we describe the system architecture and a model for the distributed transaction processing. In Section 3 the implementation of the continuous transaction processing system is briefly described. Section 4 shows the performance implication of the resiliency mechanism. The conclusion is given in the final section.

2. The X' System

We are developing a distributed primary/backup database server system, called X'. The system is basically a number of clustered database servers as shown in Figure 1.

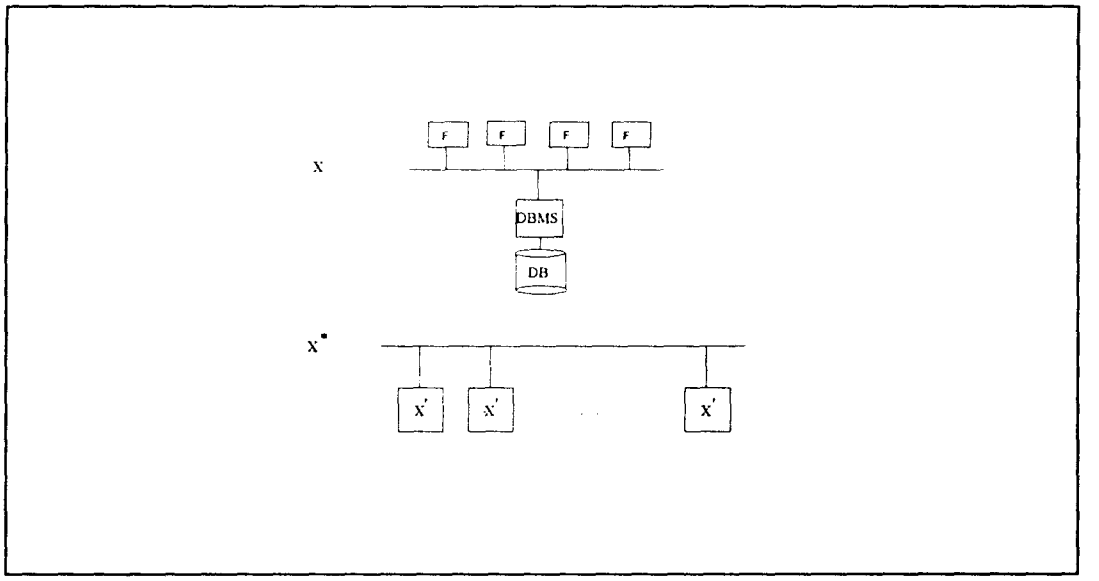

Figure 1. The X* System

In the figure a server is represented by X' which is an extension of a conventional database system, X, serving front end users, F, and the entire system is represented by X* which is a multiple of X'. The basic component X' is a database server based on the client/server architecture and is an extension of X, or XDB, a relational database system commercially available and is fully compatible with IBM's DB2. Our project is to integrate existing XDB database servers running under OS/2 and interconnected through a local area network [XDB91]. One of the goals of X* is not only to provide high level availability but also to achieve high performance. The system is intended to provide the capability of continuous transaction processing in the presence of failures. X* differs from other primary/dedicated backup systems in the sense that none of the database server is a dedicated backup for anther database server. A database server can become a backup database server for another server simply by keeping

partially replicated backup data. The primary/dedicated backup system can not survive the simultaneous failure of both primary and backup. In order to sustain multiple failures, X' may keep two or more identical copies of each relation in the system, one being the primary copy and the others being backup copies. Thus, one or more database servers become backup database servers for the server with the primary copy. In this way we can have a distributed *cross backup* database server system on a local area network. The survival of at least one backup copy allows the distributed transaction processing to continue.

Each database server has a copy of the global directory. Thus, every database server in the system knows which database server has the primary and backup copies of a table. All read and write requests go to the primary copy only. Propagating updates to the backup copy is done synchronously during the transaction commit phase by the primary server which contains the primary copy. We define a failure without distinction as the total loss of processing power and access to data at one server for a period of time, and possibly permanent loss of data at that server. When a server with the primary copy fails, all servers must be informed of the failure. The global directories are updated. One of the backup copies bocomes the primary copy. The new primary copy is used for all transaction processing. A certain number of backup copies may be always maintained. However, in order to simplify the discussion, we consider only one backup server for a primary server.

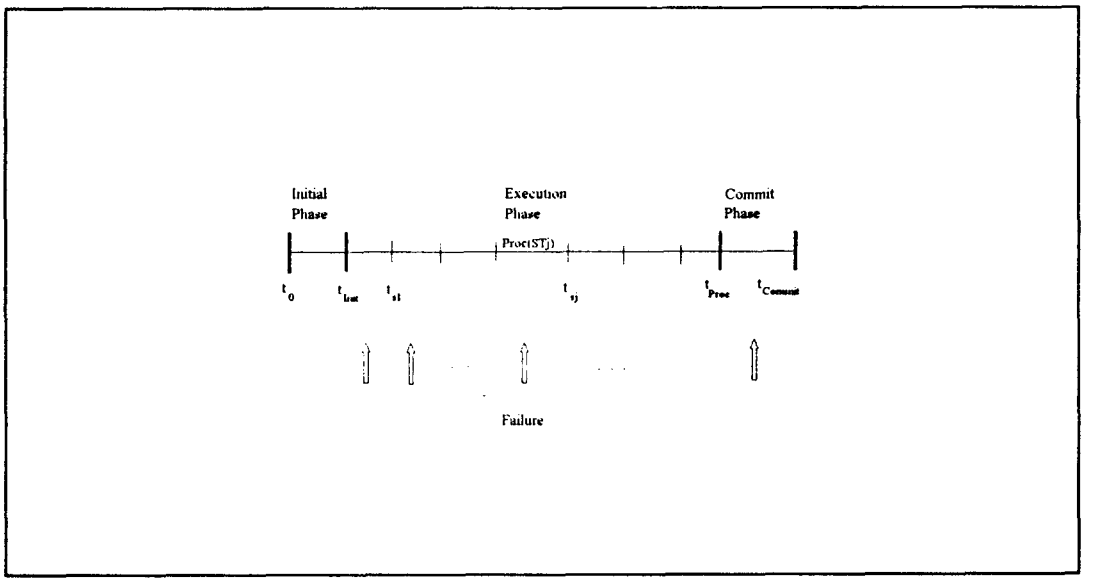

Figure 2. Transaction Processing Phases and Failure Window

As shown in Figure 2, distributed transaction processing can be divided into three phases: initial phase, execution phase, and commit phase. During the initial phase a transaction schedule is developed. In the transaction schedule, subtransaction dependency is represented by a data flow graph. The execution phase consists of one or more sequential steps that are divided according to the subtransaction dependency. In each step of the execution phase one or more subtransactions are processed, possibly in parallel. The commit phase involves the distributed two-phase commit.

In distributed transaction processing, subtransactions are executed in multiple servers. Figure 2 shows that failures, from various reasons and at different nodes, may occur at any time during the distributed transaction processing. The effect of a failure during the commit

phase is on the consistency of the distributed database. However, the effect of a failure before the commit phase is on the performance. Restart of the transaction entirely will lose valuable computing time. From the figure 2, if the effect of a node failure can be contained to a processing step, the performance loss will be reduced. This confinement of the effect of failure is the key idea for the resiliency mechanism with the distributed primary/backup database server system.

3. Resilient Transaction Processing System

The transaction processing in each node X' involves three components of a database server. The *global transaction manager (GTM)* controls the execution of transactions. GTM decomposes a transaction (T) into subtransactions (ST) and schedules them for execution. The *local transaction manager (LTM)* interacts with the *database manager (DM)* for executing subtransactions and distributed commits. DM performs retrieval of data from the database and update to the database, and controls the concurrent accesses to the database. The inter-server communications are done by the *communication manager (CM)* through the message passing scheme.

For a distributed transaction processing a server plays the role of either GTM, LTM, or both. We refer to a server with the role of GTM and a server with the role of LTM as S_{GTM} and S_{LTM} respectively. For each transaction there is only one S_{GTM} at which the transaction is initiated. A distributed transaction processing involves more than one S_{LTM}. Incidentally, for the nondistributed transaction, S_{GTM}

and S_{LTM} are the same server. Also, we refer to a backup server for S_{GTM} and a backup server for S_{LTM} as S_{BGTM} and S_{BLTM} respectively.

We classify the activities in a distributed transaction processing into two processing modes:

transaction processing mode and

restart processing mode.

During the transaction processing mode the system not only executes transactions as in the conventional distributed database systems but also maintains status information about transactions being processed in the system. This information is collected and maintained in the structure called the *transaction table.* In the event of failure, restart processing mode is invoked to perform continuous transaction processing.

In the following we first describe the structure and content of transaction tables. Then, we explain the general algorithm of two processing modes. Here, we assume the tracking is done synchronously with the transaction progress.

3.1 Transaction Table

The description of the resiliency mechanism starts with the concept of *transaction state.* As transaction processing progresses, the state of a transaction is changed. After the initial phase is completed by S_{GTM} without failure, the state of the transaction becomes "initiated". During the execution phase the state of a subtransaction is changed to "processed" as the completion of each subtransaction processing is known to S_{GTM}. Each S_{LTM}

also changes the state of a subtransaction(s) to "processed" after it completes execution. When all subtransactions are "processed", S_{GTM} changes the state of the transaction to "prepared", meaning the transaction is ready to be committed. This state information is crucial for continuous transaction processing.

To keep track of the state of transaction processing by the system, each server maintains *transaction tables*:

T_Table (T_i, T_{trans}, $User_{id}$, S_{GTM}, S_{BGTM}, State)
ST_Table (T_i, ST_j, ST_{trans}, S_{LTM}, S_{BLTM}, State)
LST_Table (ST_j, ST_{trans}, ST_{type}, S_{GTM}, S_{BGTM}, S_{LTM}, S_{BLTM}, State)

Although the global state information in T_Table and ST_Table can be integrated into one table, we decided to store them in separate tables for clarity. T_Table contains a unique identifier of a transaction, the primary server and backup server, and the processing state for each transaction. For the purpose of restart processing, the actual transaction itself is stored in the T_Table. ST_Table contains information about the global state of subtransactions. It contains a unique identifier for a subtransaction, the primary server and backup server, and the processing state for each subtransaction. For the purpose of restart processing, the actual subtransaction is stored in the ST_Table. Subtransaction types are either "read" or "update". LST_Table contains information about the local state of subtransactions. It contains the type of subtransaction, the servers and backup servers for the subtransaction and its transaction, and the processing state for each subtransaction. The actual subtransaction

is stored in the LST_Table for the purpose of restarting by a backup server.

3.2 Transaction Processing

During the initial phase, S_{GTM} saves the transaction, prepares a schedule for the transaction and starts to coordinate participating S_{LTM}s in the processing of the transaction. First, when S_{GTM} receives a transaction T_i, it selects a backup server, S_{BGTM}, to backup itself for T_i and enters a transaction record into the T_Table. After the transaction is decomposed into subtransactions, S_{GTM} also adds a record for each subtransaction of the transaction into the ST_Table. This phase corresponds to the period of between t_0 and t_{Init} (Figure 2 and 3). Then, subtransactions are sent to all participating S_{LTM}s. The execution phase begins when a server receives a subtransaction ST_j of a transaction T_i. Using the dependency relationship among the subtransactions, the execution phase is further divided into execution steps. Figure 3 illustrates *k* execution steps in the execution phase. The processing is executed independently by all S_{LTM}s at different point of time or in parallel. Each participating server, S_{LTM}, designates a server as its backup server, S_{BLTM} for the subtransaction, ST_j. Thus, if *m* S_{LTM}s participate in a transaction processing, *m* backup servers, S_{BLTM1}, ..., S_{BLTMm}, are elected as the backup server for subtransactions, as shown in Figure 3. Then, S_{LTM} adds a subtransaction record in the LST_Table. Once the backup recording is done, S_{LTM} asks its DM to process database requests and waits for the completion.

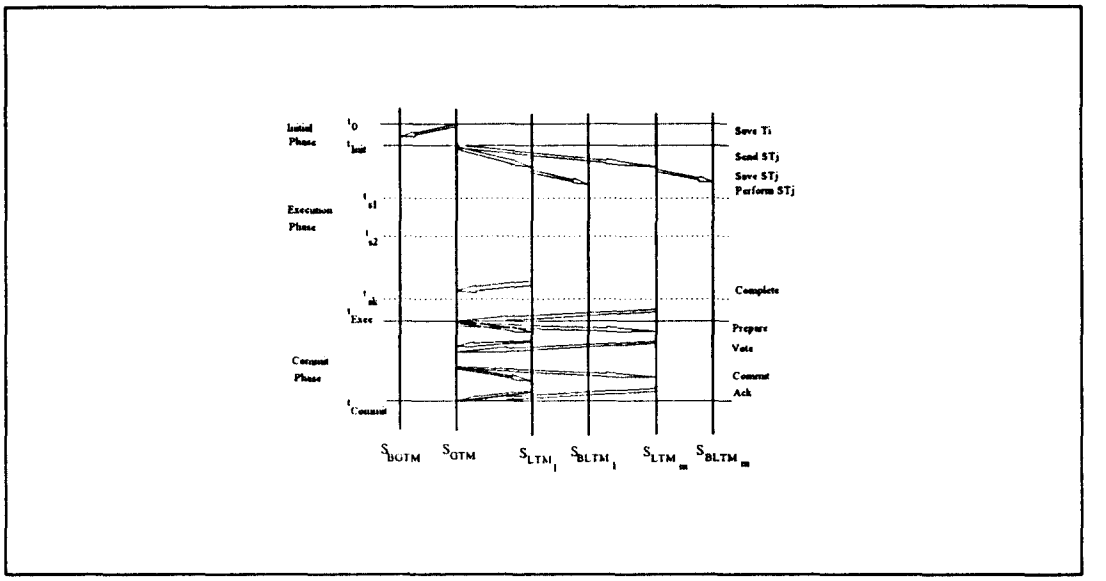

Figure 3 Resilient Transaction Processing with Cross Backup

When the execution is complete, S_{LTM} changes the state of subtransaction in LST_Table and returns results or appropriate messages to S_{GTM}. As S_{GTM} receives the reply from S_{LTM}, it also records a change in the state field of ST_Table. The execution phase completes at t_{Exec} at which *k* execution step finishes (Figure 3).

S_{GTM} begins the two-phase commit by sending Vote_Action message to all updating S_{LTM}s. S_{GTM} activates timeout and wait for reply from S_{LTM}s. If the decision is "commit", S_{GTM} writes a commit log record and sends Commit_Action to updating S_{LTM}s. If the decision is "abort", S_{GTM} writes an abort log record and sends Abort_Action to updating S_{LTM}s.

Each S_{LTM} acts according to S_{GTM}'s decision and removes the relevant LST_Table record. When S_{GTM} receives the Abort_Transaction from the user, it acts similarly. When S_{GTM} decides to commit or abort, the relevant records are removed from the T_Table and ST_Table.

Figure 4 shows the flow of data and control messages during the transaction processing. The global state of transaction processing is reflected into T_Table and ST_Table in S_{GTM} and the local state of a particular subtransaction is reflected into LST_Table in a S_{LTM}. The resiliency is prepared at backup servers, S_{BGTM} and S_{BLTM}s. Backup for the transaction information is sent from S_{GTM} to S_{BGTM}. That is, records for relevant transactions and subtransactions in T_Table and ST_Table are same in the two servers. Backup for the subtransaction information is

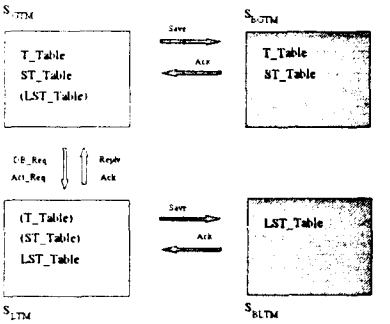

Figure 4. Information Flow between Primary and Backup System

sent from S_{LTM} to S_{BLTM}. Also, records for relevant subtransactions in LST_Table are the same in the two servers.

3.3 Restart Processing

Once the transaction processing phase is in the commit phase, the failure is handled according to the two phase commit protocol. Thus, the discussion about restart processing in resilient transaction processing deals mainly with the initial and execution phase. In this paper we briefly describe the algorithm for restart processing. We refer to the server which ceases to operate as the *failed server* and the rest of servers in the system as *survived servers* in the following.

A server failure is immediately informed to all database servers in the system. If a survived server finds any transaction and/or subtransaction affected by the failure, it has to perform restart processing. In order to determine what each survived server needs to do, it examines its T_Table, ST_Table, and LST_Table and identify transactions and subtransactions and its relationship with the failed server for each of them. Restart processing for the failed server depends on the state of transactions and subtransactions which are being processed at the time of failure. Each survived server, S_v, performs the following Procedure IB_LIST() to develop lists of incomplete and backup transactions and subtransactions. The examination of S_{GTM} and S_{BGTM} fields in the T_Table tells whether a transaction needs either a restart processing or a backup selection. Transactions in the incomplete transaction (IT) list represent that the transactions that were being processed by the failed primary server and have not been completed. Transactions in the backup transaction (BT) list represent that the transactions that were saved at the failed backup server and have not been completed yet.

Examination of S_{LTM} and S_{BLTM} fields in the LST_Table also tells whether a subtransaction needs either a restart processing or a backup selection. A subtransaction in the incomplete subtransaction (IST) list means that the subtransaction was being processed by the failed primary server and has not been completed. A subtransaction in the backup subtransaction (BST) list represents that the subtransaction was saved at the failed backup server and has not been completed yet. If a survived server finds any transaction and/or subtransaction affected by the failure from Procedure IB_LIST(), it has to perform appropriate action. The necessary processing is decided by the case which was caused by the failure. In the following we classify and briefly describe appropriate actions by the affected survived server.

Table 1 Restart Processing Cases

Case	Survived Server	Failed Server	Survived Server Action	Incomplete or Backup Transaction / Subtransaction
1	S_{BGTM}	S_{GTM}	Restart Transaction	IT
2	S_{BLTM}	S_{LTM}	Restart Subtransaction	IST
3	S_{GTM}	S_{BGTM}	Select Backup	BT
4	S_{LTM}	S_{BLTM}	Select Backup	BST

Case 1: The survived server is a backup server for transactions that are initiated by the failed

server. The survived server has to take over processing of the transactions. As the primary server has failed, the backup server becomes the primary server for those transactions, selects a new backup server for each transaction, and performs appropriate actions for the continuous transaction processing. The appropriate action is determined by examining the state information in the T_Table record. S_{BGTM}, performs Procedure RESTART_GTM for the incomplete transactions.

Case 2: The survived server is a backup server for subtransactions that are being executed by the failed server. The survived server has to take over processing of the subtransactions. The backup server becomes the primary server for those subtransactions, selects a new backup server, and performs appropriate actions for the subtransaction processing. The appropriate action is determined by examining the value of state and the type of subtransaction in the LST_Table record. S_{BLTM} performs Procedure RESTART_LTM for the incomplete subtransactions.

Case 3 and 4: The failed server is the backup server for the transactions and subtransactions processed by the survived server. Thus, new backup servers for the affected transaction and subtransactions have to be selected. A new backup server will be created by copying transaction records from the primary server. S_{GTM} and S_{LTM} perform Procedure MAKE_BACKUP.

These processes for restart processing should have higher priority than new transactions that entered the system after the failure of a server. This will be needed to provide fast response time. Since the effect of a server failure is distributed to several backup servers, the task by individual backup servers is not expected to be large.

4. Performance Implication

The primary server sends transaction table records to the backup server to keep the backup server informed of its current state. Ideally, these records in the backup server have to be identical to the ones in the primary. However, it may be costly to send each transaction table record to the backup server whenever a state change is made to any transaction table. Thus, to reduce the overhead, an alternative is to collect transaction table records and send them as a batch. Transaction table records are saved into the backup server at synchronization points. We refer to the time to save transaction records *savepoint cost*. If every state change of each transaction processing is saved in the backup system, the resiliency mechanism can perform continuous transaction processing without any restart processing in the event of failure. However, in order to minimize the overhead, we save redundant information periodically. Restart processing may be necessary for those portions of processing which are completed, but not reflected in the backup server. We call this loss due to periodic savepointing *reprocessing cost*. A proper interval between savepoints must be found. We

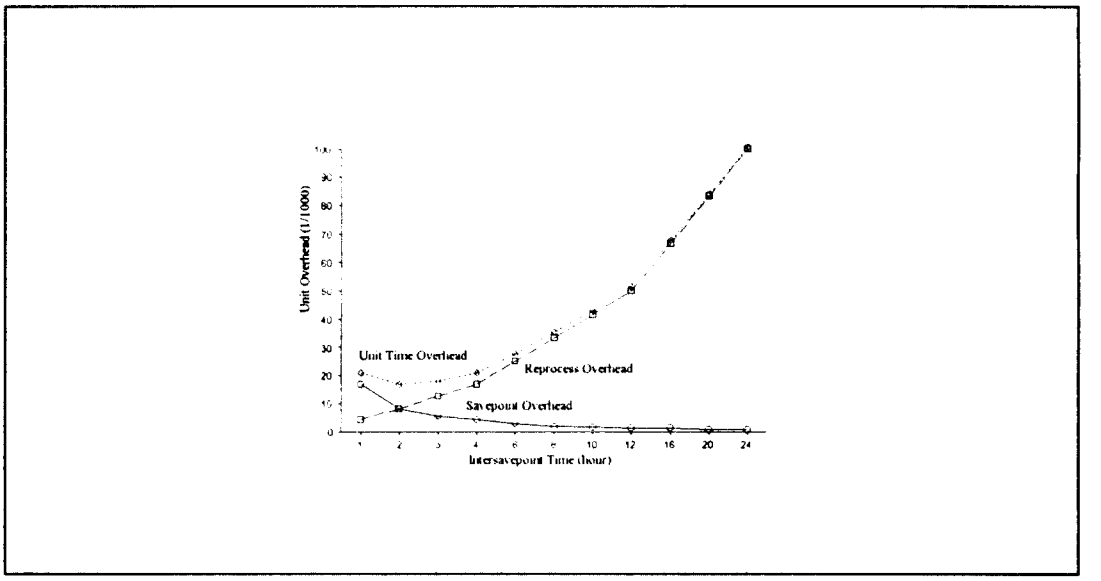

Figure 5. Unit Time Overhead

this point, we developed two figures using the following data: expected time to perform a savepoint = 1 min = 1/60 hour, failure rate of a server = 1/24 hour, reprocessing (processing) rate = 1/5.

Figure 5 shows a graph of unit time overhead as a function of intersavepoint time. While the reprocessing overhead increases monotonically, the savepoint overhead decreases rapidly and flattens. As the intersavepoint increases, the reprocessing overhead portion of the unit time overhead dominates that of the savepoint overhead. Thus, the unit time overhead decreases rapidly, stays relatively flat at optimality, and increase gradually. The optimal intersavepoint can be found at which the unit time refer to the time between two savepoints as the *intersavepoint*. If the intersavepoint is too small, savepointing overhead is high, and if the intersavepoint is too large, reprocessing overhead is high. To illustrate

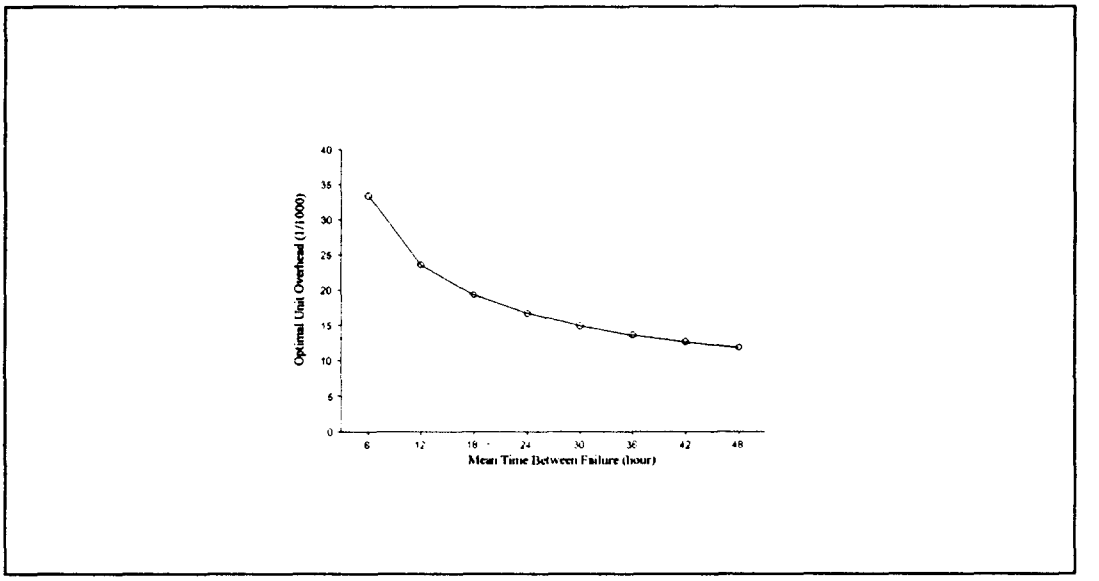

Figure 6. Optimal Unit Overhead vs. Mean Time Between Failure

overhead is at the minimum. Figure 6 show the optimal unit time overhead as a function of MTBF. As the MTBF decreases, the unit time overhead decreases.

5. Conclusions

In this paper we described an implementation of the resiliency mechanism which allows the continuous transaction processing on a loosely-coupled distributed database server system. Existing database servers are extended and integrated to build a resilient distributed database system. Unlike the primary/dedicated backup system, cross backup on a local area network is proposed. Multiple backups by the cross backup can provide higher reliability and use resources more effectively than the single backup by the primary/dedicated backup system. To keep track of a distributed transaction processing on

multiple servers, the state of a transaction is maintained.

In the case of failure survived servers restart affected transactions and subtransactions based on the state information maintained.

Our motivation is to improve the availability and performance of a loosely-coupled distributed database server system. The mechanism does provide performance improvement over conventional distributed systems by incurring overheads to reduce lost efforts in distributed transaction processing. We believe that, even with the overhead, the primary/backup approach will provide performance advantages for certain applications with long-lived transactions. The design of a beneficial resiliency mechanism requires an efficient implementation with proper system parameters. One additional parameter is the number of backup copies for individual tables in the distributed database. The determination of optimal number of backup copies is a future task.

References

[Borr84] Borr, A., "**Robustness to Crash in a Distributed Database: A Non Shared-Memory Multiprocessor Approach**," Tandem Technical Report 84.2, Sept. 1984

[Burk90] Burkes, D. L. and Treiber, R. K., "**Design Approaches for Real-Time Transaction Processing Remote Site Recovery**," Proc. of IEEE Compcon, 1990

[Gray86] Gray, J., "**Why do Computers stop and What can be done about it?**," Fifth ACM/IEEE Symposium on Reliability in Distributed Software and Database Systems, January 1986

[IBM87] IMS/VS Extended Recovery Facility (XRF), Technical Reference, 1987

[LeeY87] Lee, Y., Yu, P. S., and Iyer, B. R., "**Progressive Transaction Recovery in Distributed DB/DC System**," IEEE Transactions on Computers, Vol.C-36, No. 8, August 1987

[Lyon90] Lyon, J., "**Tandem's Remote Data Facility**," Proc. of IEEE Compcon, 1990

[Shet87] Sheth, A. P., Singhal, A., and Liu, M. T., "**Performance Analysis of Resiliency Mechanisms in Distributed Database Systems**," Proc. of IEEE 1987

[XDB91] XDB Reference Manual, XDB Systems, Inc., Laurel, MD, 1991