INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Implementing Real-Time Transactions Using Distributed Main Memory Databases

by

Smita Oza
B.E.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Science in Information & Systems Science
Ottawa-Carleton Institute for Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
October 1997

© 1997, Smita Oza

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27056-4

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
**Implementing Real-Time Transactions Using Distributed Main Memory
Databases**

submitted by
Smita Oza. B.E.
in partial fulfilment of the requirements
for the degree of Master of Science in Information & Systems Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

October 7, 1997

Abstract

Main memory database systems store their data in main physical memory and provide very high-speed access to data resources. They are suitable for certain time-critical applications such as in telecommunications, security trading, radar tracking, network routing etc. The performance and reliability of these applications are enhanced by replicating and distributing the data over a number of computers connected by the high speed local area network. This thesis studies the implementation of such a system whereby three main memory databases are integrated using the message passing interface (MPI). In any given configuration of the system, transactions can be processed in several ways. We examine two alternatives of transaction processing. In the first case, information on the distribution of data is known. In the second case, the information on the distribution of data is not known. Two transaction processing models are designed to mimic these alternatives. They are compared experimentally for the response time of read-only queries. The models are evaluated for exact-match queries and range-search queries using different cases of distributed database. Results in all experiments show that the response time of the queries improves significantly when the system keeps the information on the distribution of the data.

Acknowledgment

I would like to express my deep appreciation to **Dr. Ekow J. Otoo**, my thesis supervisor, for his valuable advice and guidance. His ideas and expertise have been essential for this thesis work.

I am also greatly indebted to all my friends at School of Computer Science, Carleton University, who extended their help in many different ways.

Finally and most importantly, I wish to thank **Sanjay Oza**, for his unselfish and devoted support, both morally and financially. Without his encouragement and patience, this study would have never been completed.

I am thankful to School of Computer Science, for providing me the opportunity and support for this study.

Contents

Abstract	iii
Acknowledgment	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Transaction Processing System	1
1.2 Disk Resident DataBases v/s Main Memory DataBases	2
1.3 Study Objective	3
1.4 Outline Of The Thesis	8
2 Background Study	9
2.1 Main Memory Databases	10
2.1.1 Design of the MM-Ode system	11
2.2 Data Replication and Performance	12
2.3 Transaction Management	14
2.3.1 Transaction Models	15
2.4 Design of the Database Management System	16
2.4.1 Concurrency Control	16

2.4.2	Commit Processing	18
2.4.3	Recovery	18
3	Implementation of the Distributed Database	21
3.1	Network Configuration	22
3.2	The MM-Ode Database System	24
3.2.1	System Set-up and Operations	25
3.3	The Message Passing Interface	31
3.4	The Integration of MM-Ode and MPI	36
3.4.1	Database Design	37
3.4.2	Installation of the Component Databases	37
3.4.3	The Interconnection of the Component Databases	38
3.5	Design of a Distributed Database Application	41
4	Design of Transaction Processing Models	44
4.1	The Execution Environment	45
4.1.1	System Configuration	45
4.1.2	Database Design	45
4.1.3	Workload	45
4.2	Assumptions for the Design	46
4.3	The Structure of the Modules	48
4.3.1	The Master-Slave Model	49
4.3.2	The Broadcast Model	53
4.3.3	The Model Implementation	56
4.4	Update Transactions	57

5	Experiment Results	59
5.1	Input Parameters	60
5.2	Comparison of exact-match queries	61
5.2.1	Partially Replicated Database	63
5.2.2	Fully Replicated Database	63
5.3	Comparison of Range-Search Queries	65
6	Conclusions	66
6.1	Summary	66
6.2	Main Contributions	66
6.3	Directions for future research	67
	References	69

List of Tables

5.1	System parameters for experiments of exact-match queries	62
5.2	System parameters for experiments of range-search queries	64

List of Figures

1.1	Two approaches to processing queries in a transaction processing system with distributed databases	6
3.1	Configuration of the distributed database in the given network	23
3.2	Distributed database using the MM-Ode system	38
3.3	Integration of MM-Ode and MPI	40
4.1	Flow of a transaction in master-slave model	50
4.2	Flow of a transaction in broadcast model	54
5.1	Comparison of models for exact-match queries : partially replicated database	62
5.2	Comparison of models for exact-match queries : fully replicated database	64
5.3	Comparison of models for range-search queries	65

Chapter 1

Introduction

Real-time transactions are best serviced by main memory databases. Examples of such transactions occur in air traffic control system, telecommunications, security trading, radar tracking, network routing and ATM banking service system. These transactions are characterized by deadlines. To obviate latency delay involved with disk-based transaction processing system, the entire database can be loaded into memory. However, the assumption of having enough memory capacity introduces some complexity in the design of such applications. These include: persistency of data, growth of the system, data availability and reliability. In general, these problems can be resolved by creating redundancy of data in the system and constructing a distributed database system over a high-speed local area network. In this work, we focus on the design of transaction processing (TP) system using distributed main memory databases for such real-time applications.

1.1 Transaction Processing System

A TP system is an *on-line multi-user* system that accepts user requests and returns responses to those requests. The act of generating a response usually implies accessing a database maintained by the TP system. The term *On-line* indicates that

the user has a direct connection to the system. Many users can access the system concurrently to retrieve the required information. Generally, quick responses are desired from a TP system, so that they affect the functioning of environment. In other words, the system appears *responsive* to the user. Therefore, quite often the TP system is designed as a *real-time* system [Hig89]. The term *real-time* means that responses have strict deadlines, and the system must return them within predefined time limit. This system works interactively with users. There can be successive operations where a user inputs next request based on the response from the previous one.

Like any complex system, TP systems have many service components working together to form a complete system. Some typical components include a set of user terminals, one or more sources of data, i.e. databases and a communication network connecting them. The basic function of a TP system is to serve a large number of users with the requested information promptly, based on the data stored in its database(s).

1.2 Disk Resident DataBases v/s Main Memory DataBases

Transaction processing systems are built upon the main source of information, known as databases. The standard term **database** means *a collection of integrated and shared data* [Dat95]. Databases are used to represent the real life systems conceptually in computers. Two operations are possible on the data: read and write. The change in the system's state is noted by writing contents of the database. An operation on the database is performed by running a transaction. *A set of programs that manages operations on database and controls the manipulation of data is known as a database management system(DBMS)* [Dat95]. It is designed to handle large volume of stored data efficiently. A DBMS and a database together form a **database system**.

Databases involve enormous amount of data. Therefore, traditionally the databases are stored on secondary storage devices such as magnetic disks. These are often referenced as *disk resident databases* or DRDBs. Conventional DRDB systems are optimized for the particular characteristics of disk storage mechanisms. Disk operations are slow and an access to data stored on disk takes longer time. Based on the storage characteristics of disk devices, the data organization and index structures are designed to perform I/O operations efficiently. However, there are certain applications with stringent timing requirements that can not tolerate the response times offered by DRDB systems.

To improve the performance of the system, faster access to data is needed. An obvious and feasible solution is to keep the data in main memory and thus, eliminate slow disk accesses as much as possible. This idea leads to the design of main memory databases. **Main memory database** (MMDB) *systems store their data in main physical memory* throughout the lifetime of the databases. Because data can be accessed directly in main memory, MMDBs provide very high-speed access. As a result they exhibit better response time and transaction throughputs [GMS92]. However, the performance gain does not come for free. Main memory is normally volatile and reliability becomes a critical design issue. Design of memory resident database systems is discussed in detail later.

1.3 Study Objective

In the past years, a substantial amount of efforts have been put into examining potential benefits of memory resident databases [AHK85, BHT87, DKO⁺84, LC86a]. Various aspects influencing the performance of the database, i.e. index structure for efficient memory access [LC86b], and query processing [LC86a, AP92] have been studied. Several experimental prototypes have been built to explore the suitability of concurrency control and recovery algorithms for transaction processing applications using main memory databases [GMS90, LN88]. This research suggests that MMDBs can be an attractive and viable option for applications with real-time constraints.

Examples of such systems include telecommunications (e.g. 800 telephone numbers need to be translated to real phone numbers), radar tracking (e.g. signature of objects need to be matched against a database of known aircrafts), security trading (e.g. trading opportunities must be discovered and executed before they vanish). and interactive services (e.g. ATM banking services where cash withdrawal or account summary should be returned fairly quickly).

These real-time applications require responses in reasonably short time. If an MMDB is used to store data, the access to the data can be made faster. With declining costs of semiconductor memory and multiplying chip densities, it is realistic to fit entire database into main memory. Nevertheless, it is possible to have applications that involve very large databases. The finite capacity of a single computer's physical memory may not be able to hold the complete operational data for such an application. A distributed database¹ can solve the problem. In today's technology, a typical workstation for database application contains 128MB to 512MB of RAM. By connecting several such systems together, sufficiently large pool of main memories can be achieved. The database can be distributed over a network of computers where a portion of data will be kept as main memory database on each different machine.

In our work, we make use of a network of main memory databases as described above. The objective is to design a high performance transaction processing system for the real-time applications. We are considering applications similar to the popular 800 number service system. The main TP application in this system requires searching the database. It mostly involves read-only transactions. An example of call routing/processing in the telecommunications service is explained below. A customer dials an 800 phone number, internally it gets translated to some real physical phone number and the connection is made to this real phone number. From system's point of view, all 800 telephone numbers subscribed are logical entities and

¹A distributed database is a collection of multiple(independent), logically interrelated databases distributed over a computer network [OV91b]. Databases are physically separated and connected by a computer network. They communicate using *only* a message passing system.

they map to some real phone numbers. The system supporting this service keeps the conversion table of phone numbers as a memory resident database for fast access. When a request for service gets registered into the system, it performs the read-only query against the database. In return, it gets the real phone number that eventually gets connected. This call set-up application is an on-line transaction processing application running read-only transactions.

It has been observed that the transaction rate is usually very high in this system and the transactions require fast responses. The distributed database is considered to improve the performance of the system. To provide high availability of data and to make the system reliable in the presence of failures, data may be replicated at several nodes. In designing the database, the data is partitioned and distributed across a number of computers, we call them database hosts. The distributed DBMS keeps the information about the distribution and replication of the data and provides transparent access to data. In this architecture, the transaction processing can be carried out in two different ways.

1. If the distribution of data is precisely known to the system, using that information the system can direct the request to a particular node where the requested data resides. This kind of transaction processing is described in the *master-slave model*. (Figure1.1(a)).
2. If the system does not keep the information on the distribution of the data, then searching for the requested data requires access to all database nodes. By taking advantage of the inherent parallelism of distributed system, the request can be issued to all nodes in parallel. This transaction processing model is described as the *broadcast model*. (Figure1.1(b)).

The model naming convention follows the communications pattern used in performing the transactions. We carry out a comparative study of design alternatives of transaction processing system using the distributed main memory database. In our work, the distributed main memory database is constructed using a main memory database system called MM-Ode. We concentrate on the implementation of two

(a) Master-Slave Model :- Request R1 needs data d1 which resides at DB2. Similarly, R2 needs data d2 from DB3. The system performs direct operations.

(b) Broadcast Model :- When system is not aware of data location, the requests are sent to each database node. The operations are carried out in parallel.

Figure 1.1: Two approaches to processing queries in a transaction processing system with distributed databases

transaction processing models for the TP application. The models are compared experimentally. Input data to the models include a synthetically generated workload based on a number of observed characteristics such as arrival rates, service rate etc. It is our view that results of the experiments will provide important insight for making decision regarding data distribution and transaction processing technique in the design of such real-time systems.

In order to have up-to-date information delivered by the TP system, the database must be updated regularly. The update operations modify the contents of the database. In multi-user environment of distributed system, all updates must be performed in an integrated manner. It may affect the response time of read-only applications adversely. Many existing commercial systems employ two alternative execution strategies in realizing improved performance of the system [OV91a]. The first alternative uses time multiplexing between read activity and update activity. The database is kept open for queries (i.e. read-only) during the regular operating hours while the updates are batched. The database is then closed to query activity during off-hours when batched updates are run sequentially. In certain situation, such segregation of these two activities is not acceptable. The second alternative is based on multiplexing the database. In this case, two separate identical copies of the database are maintained. One is called the query database, which is used for the querying applications. The other copy known as production database, deals with updates. The applications run in parallel and periodically, the production database is copied to the query database.

Various techniques have been developed to manage different aspects of distributed data management including concurrency control, commit processing and recovery and reliability. A few suitable techniques are discussed for the design of functional components of the transaction processing system for the application under consideration.

1.4 Outline Of The Thesis

We consider transaction processing system for real-time applications using distributed memory resident databases. In this regard, Chapters 2 sheds some light on the research related to the design of main memory database systems. It provides the background information needed to follow the framework of the TP models.

Chapter 3 describes the implementation of the distributed main memory database. The component databases are installed using the MM-Ode database system. The communications for the component databases is established with the message passing interface MPI. The features and application development environment of the MM-Ode system and MPI are discussed. The integration procedure is also presented in detail.

The design of the two transaction processing models is described in Chapter 4. Various assumptions made in the design of the models are described. The complete algorithms showing the flow of transactions in the TP models are presented.

Chapter 5 summarizes the experimental results obtained. A discussion of the results is presented. It shows the relative comparison of the performance of the two models for two types of read-only queries.

In Chapter 6, we conclude with our assertions based on the observation of performance of the models. A short summary of design of functionality components for the extension of the transaction processing system is presented.

Chapter 2

Background Study

The availability of relatively inexpensive random access memory makes MMDB a database of choice for many time-critical applications. Certain real-time applications in telecommunications, network operations, financial and defense systems require fast look-ups into databases. Substantial performance gain can be achieved by keeping the entire database into main memory in these applications. We build a transaction processing system for such applications using main memory databases. The transaction rate is normally high in these systems. Majority of the transactions are read-only queries while updates occur quite infrequently. In this situation, replication and distribution of the data can improve the performance and enhance the reliability of the system. However, the complexity of the management of data also increases in such a distributed system. A considerable amount of research has been done in the design of distributed database systems. This chapter examines various alternatives that are proposed or used for the design of transaction processing system using distributed databases. Section 2.1 presents features and requirements of memory resident database systems. The use of replicated data to improve the performance of the systems is discussed in Section 2.2. The concepts of transaction management and transaction models for the applications are discussed in Section 2.3. The design of various data management components and their implementation in some existing systems are described in Section 2.4.

2.1 Main Memory Databases

The 800 number system of telecommunications needs quick responses to the queries for the translation of phone numbers in the call set-up services. Similarly, the data stored in the routing tables must be accessed in very short time in the message routing systems of network operations. Another system of radar tracking demand fast retrieval of the data for the objects of matching signatures. These are only a few examples of time-sensitive applications where high speed access to the data is highly required. The list grows rapidly with the emerging numerous other such real-time applications.

These real-time applications were previously implemented as stand-alone programs running in memory which provided limited sharability and persistency of their data. With the increasing growth of applications and related data, the need for a structured system was soon realized. Conventional databases were found incapable of meeting the response time requirements due to slow disk I/O operations. Over the time, the memory capacity of computers have grown tenfold. In computers with large main memory, the entire database can be cached into the memory. The database applications can benefit from such an arrangement, although they do not take full advantage of the memory. Researchers have sought the solution in designing main memory database systems tailored to the needs of memory resident data.

Main memory database systems store their data into memory and provide direct access to the data. The access time of the data improves significantly because of the following reasons. The I/O operations are eliminated as the data reside in memory. The layout of the data and the access methods are not significant for memory resident data. Fast access reduces the processing time and hence, lock contention decreases. Consequentially, transaction context switches and associated cache flushes are eliminated. The cumulative effect is that the main memory databases exhibit better response time and transaction throughput.

A number of projects like OBE(IBM) [AHK85], MM-DBMS(University of Wisconsin) [LC86a, LC86b, LC87, DKO⁺84], tpk(Princeton U) [LN88] and System

M(Princeton U) [GMS90] have studied the design of main memory databases. Main memory databases can take advantage of efficient pointer following for data representation. Therefore the database objects such as tuples and relations are represented as a set of pointers. This scheme offers optimal memory usage and aids in fast data manipulation [AHK85, LC86a, DKO⁻84]. A special index structure called T-tree is developed for main memory databases [LC86b]. Other searching techniques like hash directory structures [AP92] and inverted indexes [BHT87, AHK85] are proposed. Various hashing algorithms for query optimization are evaluated in [DKO⁻84, LC86a]. Optimistic protocols [AHK85] and two-phase locking [GMS90] are used for concurrency control. The algorithms for group commit and precommit with other logging and checkpointing techniques are tested in [LN88]. A thorough analysis of the recovery algorithms is carried out in [GMS90].

2.1.1 Design of the MM-Ode system

Based on the assertions derived from this extensive research, a complete database management system for main memory databases can be developed. The MM-Ode system is one such database system built on top of the Dali main memory storage manager [JLRS94, BLR⁻97]. It is a centralized database system designed in a client-server fashion. The system is built with memory-mapped architecture. It stores the user database is stored into operating system files that can be mapped into the virtual address space of the process. Similarly, the system control data for concurrency control and logging are accessed through shared memory. The direct access to the user and control data eliminates expensive interprocess communications with the servers.

The MM-Ode system supports fault-tolerant applications using transaction paradigm. It provides an advanced, explicitly multi-level transaction model. The fine-grained item-level locking is used with highly efficient index structures like T-tree [LC86b] and extendible hashing [Dat95]. It increases the degree of multiprogramming. The concurrency manager uses two-phase locking and provides a mechanism to introduce new user defined lock structures. To allow highly concurrent transaction execution.

the logical logging is used. The persistent log contains only redo records of committed transactions and repeat history is fully supported with write-ahead-logging (WAL) principle. Fuzzy and action-consistent checkpoints are taken with ping-pong techniques. The system uses codewords and memory protection to prevent data corruption due to stray application pointers and ensures the integrity of the data stored in shared memory.

Thus, the MM-Ode system is designed for high performance applications with higher throughput and fast responses. It is claimed that point index lookup in a large table are typically ten times faster than those on a leading commercial relational database on the same platform and updates are three times fast ¹. The system is suitable for small, highly concurrent transactions. It can efficiently serve the real-time applications we have considered in our work.

2.2 Data Replication and Performance

The response time requirements of the real-time system can be met using the main memory database system like the MM-Ode system. However, at high transaction rates, the performance of a centralized system gets affected when the system remains constantly loaded. It incurs large delays in responses. In this situation, the further performance gains can be obtained by replicating the database at several machines. As the read-only transactions are predominant in the systems and the update rates are low, the distributed database is a viable option. The data is replicated and distributed over different computers connected by fast network. In such a system, multiple transactions are processed by different computers in parallel. As a result, the response time of queries is reduced. The distributed database system are configured in several ways. A few widely used architectures are:

- A *distributed database system* in its true sense, where the data is distributed over a network of workstations [OV91b, CP84]. Every node in the system

¹This information is collected from the product home page of DataBlitz on the Internet URL <http://inside.bell-labs.com/project/dali/>

functions as a complete database system. Underlying distributed database management system provides transparent access to the distributed data and supports distributed transaction management.

- A *client-server system* connecting powerful workstation clients with a centralized server. The server has a disk to store the database and the system log. The clients cache data into the memory and provide database operations. The client-server database management system supports persistency and reliability of the system monitored by the server. Many commercial OODBMS e.g. ObjectStore [LLOW91], O2 [Deu91] and prototypes ORION [KGBW90], EXODUS [FZT⁺92], ARIES/CSA [MN94] are designed in this manner.
- A *multi backend system* uses one computer as the controller and several computers with their disks as backends [Hsi83]. The backends are configured in parallel manner with replicating the same software and data. The controller and the backends are connected with high-speed network. The controller distributes the load to the backends and monitors the transaction processing with the help of a distributed database management system.

The distributed database system is generally designed to serve different groups of users dispersed geographically apart. The design of client-server system and multi backend system are almost the same. The only exception is that the clients may store only a part of the server database and user have direct access to all clients. In case of the multi backend system, access to the database is centralized.

In our system, the distributed database is configured as a multi backend database system. Multiple queries are processed in parallel by backend nodes. The query-coordinator is the same as the controller who distributes the queries to the backend database nodes. The coordination of the parallel transaction execution is carried out by the query-coordinator and the underlying distributed database management system. The design of this system is relatively simple as there is only a single access/control point for the management of data.

2.3 Transaction Management

In database applications, the operations on the database are grouped together into a transaction. The transaction is a program that issues write and read actions on the data. Read actions do not change the data and hence, read-only transactions are easy to design. Write actions change the contents of the database. Therefore, the order of execution of two transactions is important when one of them contains a write action and they both operate on a common data [BG81]. An execution of transaction transforms the initially consistent database into a consistent state. In this way, a transaction represents the logical unit of consistency. It can contain an arbitrary number of operations. A transaction must guarantee the ACID transformation [GR93]. This means:

1. A transaction is an atomic action. Either all the operations are executed and the effects are properly installed in the database, when the transaction is called committed, or all effects of the transaction are undone and the transaction is aborted.
2. The concurrent execution of several transaction affects the database as if they executed serially in some order. The interleaved order of the actions of a set of concurrent transactions is called a schedule. A schedule is serializable if the effects of running the schedule is the same as if the transactions had executed serially in some order.

(1) implies the failure atomicity that is established by the commit and recovery algorithms and (2) implies the execution atomicity that is supported by the concurrency control algorithms. The ACID concept is integrated into the design of transaction processing system. An application defines the semantics of the transactions. The system implements the logic of the ACID transactions with the components designed for concurrency control, commit processing and recovery mechanism.

2.3.1 Transaction Models

A transaction is the unit of work in an application. The structure of a transaction is defined to meet the needs of the application. The structure is called the *transaction model*. The basic structure of the transaction is a *flat* transaction. Its design is simple. Special functions are used to define a transaction. Everything inside these boundaries is treated as a unit of work. The invocation of `Begin_Work()` starts the transaction. All operations performed after that are part of the transaction until the function `Commit_Work()` is executed. If operations are completed according to the rules defined, the transaction terminates successfully and its effects are made durable. Otherwise, the transaction is rolled back and its effects are undone. The effects of the operations performed by the transaction are seen only when the transaction commits.

The flat transaction provides only one level of control to the applications. It is useful when the size of the application is small and the entire application works as an immutable piece of work. Depending on the processing logic of the applications, more flexible and expressive transaction models are developed. For applications with long-lived transactions, savepoints are introduced to reduce the amount of wasted efforts in case of roll back. A stronger version is called chained transactions allow commit of a partial work. Advanced models such as nested transactions [Mos85], sagas [GMS88], multi-level and distributed transactions [GR93], are designed to support more complex applications. A few specialized models, e.g. ConTract, Acta, Flex are described in [ELMB92]. In general, any new application can come up with a new transaction model.

The database management systems and transaction processing systems are designed with keeping the requirements of applications in mind. They support one or more transaction models. The MM-Ode system is a centralized system that supports multi-level transactions. The EXODUS system and the ARIES system, both support variants of nested transactions [FZT⁺92, MN94]. The distributed ² design

²The system is proposed for client-server and shared disk environment, no implementation details is known yet. The proposed design is found on the Internet URL <http://www.bell->

of MM-Ode system supports multi-level transaction model. The Encina transaction monitor supports nested transactions and the Tuxedo system supports multi-level transactions ³. The flat transaction is the basic building block for all transaction models, hence it is supported by all systems.

In our system, the transactions are mostly small in size, consisting of a single operation. The flat transaction can directly be used to define the unit of work. However, the query is sent for processing to remote database nodes. Therefore, the transaction needs to include the information of communications operations also. In this sense, our transaction model represents a simple distributed transaction.

2.4 Design of the Database Management System

The transaction models defines the processing logic of the application. The execution of the logical sequence of the application is supported by the database and transaction management system. The main components of this system are concurrency control algorithms, commit processing algorithms and recovery algorithms.

2.4.1 Concurrency Control

Concurrency control is the activity of coordinating concurrent transactions to preserve the correctness of the database operations. Concurrency control techniques provide ordering of the transactions with optimizing the system throughput. An in-depth survey of different concurrency control algorithms is presented in [BG81]. These algorithms are broadly classified into three categories: (1) locking (2) timestamp ordering (3) optimistic.

Locking has remained a very popular technique in the design of various concurrency control algorithms for centralized, distributed and client-server systems. Basic

labs.com/project/dali/papers.html

³Encina from Transarc Co. and Tuxedo from BEA systems are commercial products of transaction processing monitors. They provide transactional support in the distributed environment. More information can be found under URL <http://www.transarc.com> and <http://www.beasys.com>

elements in these algorithms are the lock types and the locking granules. Normally, locks are granted in shared mode for reading a data object and in exclusive mode for writing a data object. The size of the locking granules depends on the type of transactions and performance requirements of the applications. Basic two-phase (2PL) locking works on the ROWA (Read One Write All) principle. A variant called primary copy 2PL is developed for the design of distributed Ingres system [Sto79]. By releasing the requirement of lock on all copies, concurrency level can be increased. 2PL with a centralized lock manager is proposed in [GM81] and used in System R* [LLM⁺84]. A locking mechanism based on the majority consensus approach is described in [Tho79]. It requires a separate procedure for replica control [AA92]. 2PL is used in the design of the client-server MM-Ode system with item-level and coarser granularity. Similar technique is used in the design of ARIES/CSA database system [MN94]. Strict 2PL with page-level or coarser granularity is used in EXODUS system [FZT⁺92].

In any locking mechanism, there is a possibility of deadlock [GR93]. All systems include one or more node with deadlock detection algorithm. The detector node checks the status of the network by algorithm based on the waits-for graphs. Any deadlock situation is resolved by aborting a transaction to break the cycle according to preemptive or nonpreemptive policy used in the algorithm.

Timestamp ordering technique uses system defined timestamp to serialize the transaction. The algorithm needs to be integrated with the two-phase commit protocol to ensure the atomicity of the transaction. This algorithm is used in the design of SDD-I system [JBF⁺80]. In optimistic approach, transactions are allowed to execute till completion with the optimism that its operations will not conflict with other transactions [KR81]. At the commit of the transaction a validity test is performed to verify the serialization of the operations. The IMS-FastPath system uses optimistic algorithm [CP84].

We use basic 2PL algorithm for the design of concurrency control. The update transactions occur at low rate, therefore, the probability of conflict is low. With the basic 2PL algorithm the problem of replica control is also handled simultaneously.

2.4.2 Commit Processing

The completion of a transaction in distributed environment has to be an unilateral decision of all participants of the transaction. The commit operation is carried out in two stages by a protocol called two-phase commit protocol (2PC). When the transaction issues `Commit_Work()` operation, this protocol is invoked. In the first "prepare" phase, the outcome of the transaction is checked at each participant and vote is taken. In case of a no reply by any participant, roll back action is initiated. Otherwise, the second "commit" phase is activated and the effects of the transaction is made durable. The transaction is committed and the resources allocated to the transaction are released. The commit processing is carried out with standard 2PC in all systems. The commit process works in integration with concurrency control and recovery algorithms.

2.4.3 Recovery

The reliability of any system is determined by the recovery mechanism employed into the system. Logging of the normal processing operations is the central aspect in making recoverable system. Every operation generates corresponding undo and redo log records. The redo log of the transaction is written to the durable storage before the transaction commits. The write-ahead-logging (WAL) principle is adopted for recovery algorithm. According to this principle, all the changes made by a transaction (kept as log) must be forced to the durable storage before copying the dirty pages. It ensures that the actions taken by the transactions can be back tracked in case of a system failure. Similarly, the undo log is used to roll back a failed transaction. Along with system log, two other data structures, the active transactions table (ATT) and dirty page table (DPT) are maintained by the system. As the log grows without bound, the checkpoints are taken to limit the amount of system log stored on stable storage.

The distributed MM-Ode client-server system is equipped with a multi-level recovery mechanism. The system uses logical logging technique with less weaker op-

eration locks. The undo log of a committed operation is replaced by a compensating action written as undo log. Only the redo log of the transactions are written to disk. The system uses lazy commit to allow high concurrency. The system enforces WAL property whereby all redo log for a page are flushed to disk before the page is copied to disk. (The undo log are only written in case of checkpointing) The system uses fuzzy checkpointing taken by ping-pong techniques. Action-consistent checkpoints are also supported. The logging and checkpointing are carried out by the server. The clients ship log of the transaction to the server.

Similarly, in the EXODUS page-server system, the server maintains system log and takes the checkpoints [FZT+92]. The clients ship log of the transaction and updated page to the server before committing the transaction, called as force-to-server-at-commit policy. It employs steal/no force buffer management policy. The system supports savepoints, nested recovery and logical undo logging.

In contrast to EXODUS, in the ARIES/CSA system the log is buffered at clients [MN94]. The clients have local log managers that behave very much like a regular log managers, except that, instead of writing log to disk they store it in virtual storage. The client send the log to the server along with dirty pages after executing multiple transactions. However, the system uses update privilege owning policy to ensure that a page is modified by a single client at a time. The system uses steal/no force policy that allows the clients to send pages containing uncommitted updates to the server. The system also allows even dirty pages to be shipped from one client to another before committing a transaction at the client. The server and the clients, both take checkpoints. The clients checkpoints information is stored at the server.

In the ObjectStore system, all modified pages are sent to the server at committing the transaction. While pages accessed by the transaction continue to be cached at the client after the transaction terminates [LLOW91].

The recovery is performed by the server in the event of system failure. The recovery process has three stages. In the first stage called analysis pass, the latest completed checkpoint is restored. The dirty page table and active transaction table

are constructed from the system log. The second stage starts where the redo log is applied to the checkpoint image of the database based on the entries of the dirty pages. In the third stage, the effects of the uncommitted transactions is removed by applying undo log. This will generate new redo log of the operations performed in the third stage. In this way, the entire system is reconstructed and brought into the status as it was before the crash. The recovery is performed by this method in all the systems that use logging for recovery.

In our system, the MM-Ode system provides locking and logging facilities for operations carried out at the local node. However, the global monitoring of the distributed transactions are to be designed. The transaction manager in the query-coordinator maintains the log of the operations performed for processing the queries. It keeps record of the query being sent to the backend processor with the node identifier. The log may contain multiple such records for a transaction corresponding to each participating node. The log is also generated upon the arrival of the response from the backend database node. The log records are written to disk when each transaction is committed.

Chapter 3

Implementation of the Distributed Database

The focus of this thesis is on designing high performance transaction processing system for real-time applications using multicomputer clusters. These applications are characterized by high transaction rates with very low latency for transactions. In these systems, transactions are predominantly small, read-only queries. Quite often the response time of queries is in the order of milliseconds. It has been observed that update rates are relatively low. Therefore, the performance needs of the applications are met using distributed databases. The database is replicated and distributed over a network of workstations. There are a number of network configuration possible. They are presented in section 3.1. The main memory database system MM-Ode is used as the underlying DBMS in our implementation. The advantages and limitations of the MM-Ode system are discussed in section 3.2. The MM-Ode is a centralized database system. The interconnection of the distributed database components is established using a message passing communication system. The suitability of the message passing interface MPI is described in section 3.3. Section 3.4 presents the integration experience of these system components - MM-Ode and MPI, for the implementation of the distributed database. Section 3.5 illustrates the design of a transaction processing system using the distributed database.

3.1 Network Configuration

Based on the characteristics of the real-time applications discussed before, the performance of the system can be improved by designing a distributed database. The database can be replicated and spread over different machines connected by a local area network. The potential benefits of this architecture are:

- The availability of the data increases as the data are stored at more than one node.
- The design of the system can be made fault-tolerant. The system can remain operational in spite of a node failure as the data will be accessible from other operating nodes.
- The use of the network offers scalability to the system. The network can be extended to accommodate further growth of the system.

It is also obvious that such systems are preferred in situations of very large databases that exceed the memory capacity of a single machine.

The distributed database is constructed using a set of off-the-shelf workstations connected to a high-speed network. The network can be set up in a variety of ways. Common network topologies are bus, star, ring, hierarchical, mesh, complete network etc. Although the network routing algorithm relieves us from knowing the physical connections in the network, the knowledge of the network topology is useful in developing the distributed algorithms.

In our existing network, the distributed database is stored on a network of workstations. The network consists of 12 SunSPARCstation nodes connected with a 10Mbps 10Base2 high-speed network. It is an Ethernet, a *bus* topology network. The entire system runs under Sun Solaris 2.5.

Bus topology is quite popular in local area networks. The control of the network is simple as the traffic flows through a common communication channel, called a bus. A node broadcasts to multiple nodes and the bus allows all nodes to receive every

Figure 3.1: Configuration of the distributed database in the given network

transmission. The network failure may be guarded by providing a secondary bus. It is difficult to isolate any failed nodes tied to the bus. The networking software provides an uniform base for the communication system.

Under the existing network configuration, the transaction processing system for the distributed database can be designed in two different ways as depicted in Figure 3.1. In the first case, each database node communicates with all other database nodes. The query may originate at any database node. The host (where the query originates) is responsible to process the query. If the query data does not reside in the host database, it will send the query to respective remote database node(s)

for processing and collects the result back. A distributed DBMS is developed to support this type of transaction processing. The design of such DBMS is complex as there are multiple query access points.

In the second case, a node on the network (it may contain database as well, depending on the design) is designated as a query-coordinator. It is responsible to provide the global query interface and to monitor the query processing work. The query-coordinator is the only access point for all queries. The query-coordinator schedules query for processing by a node. It sends the query to the appropriate database node(s) for processing and collects its result. For all queries, the query-coordinator performs this task. Thus, the query-coordinator is the single control point in the transaction processing system and hence, its design is less complicated. However, the centralized aspect of this architecture requires special measures to be taken for designing a fault-tolerant system.

In our implementation, we design the transaction processing system based on the second option. We make use of 4 SunSPARCstation nodes: three nodes are database hosts and one node works as the query-coordinator. These nodes are interconnected using message passing interface LAM 6.0 [GLS96].

The database is distributed over three nodes. At each node, the distributed database component is installed with a centralized database system. In order to provide the fast access to the data, the database must be stored in the main memory. The selection of the right database system faces limitations of choices. There are very few main memory database systems built. In fact, we are aware of only one full-fledged main memory database system, i.e. MM-Ode. In our implementation, the database is locked in main memory using MM-Ode. The suitability of the system can be justified from the following discussion.

3.2 The MM-Ode Database System

MM-Ode is an object-oriented database management system built on top of Dali main memory storage manager developed at Bell labs. The Dali storage man-

ager is designed to provide the persistence, availability and safety guarantees for its memory-resident database. The database system is designed as a client-server system. It stores user data and system control data into files and allows direct mapping of these files into the user's address space. The system supports a variety of locks and multi-level transaction model.

The system adopts a toolkit approach in the design. The logging facilities can be turned off for data which need not be persistent. For example, the read-only applications do not update the data and hence, logging may not be required for them. On the other hand, if sequential operations are performed on the data by the applications, the locking can be selectively turned off to reduce the system overhead. The low level components are exposed to the user, so that critical system components can be optimized with special implementation.

The user interface of the MM-Ode system is based on the C++ data model. The database programming interface is designed in O++, which enhances C++ by providing facilities for creating persistent objects, versioning and queries.

It can be observed that the MM-Ode system is equipped with the basic components required for efficient transaction management. It can fulfill the response time requirement of the application by providing the direct access to the data. The structure of the system is best suited for applications where the transactions are small and multiple processes may access shared data.

3.2.1 System Set-up and Operations

The following points describe the basic set-up procedure to run the MM-Ode applications.

1. MM-Ode must be installed on Sun SPARCstations (or a Sun with a Sparc processor) running SunOS 4.1x or Solaris 2.3 with Sun CC 4.x compiler.
2. The path is established by adding the following to the path variable in `.cshrc` file, to allow invoking the server startup commands and the O++ compiler

without explicitly specifying the full path.

```
set path = ($path mm-ode-directory-path/bin)
```

3. MM-Ode databases use memory-mapped UNIX files for shared memory databases.

Within application programs, databases may be specified by providing a full path name to the memory-mapped database files. Alternatively, a database may be created or assumed to exist in an optional default directory. To use a default directory, the environment variable `ODE_DB_DIR` must be set to the full path name of the default database directory.

Suppose, the database files are to reside in the database directory `mydb` which is a subdirectory created under the home directory. We add the following to the `.cshrc` file.

```
setenv ODE_DB_DIR ~/mydb
```

4. Dali requires a system database to store control information about normal databases. By default, this system database will be located in the `system` directory of the release. To change this, the path variable `DALI_DBPATH` is set to the desired path. In addition, the file `system/dalirc` must be copied into the selected non-default system database directory. If the system databases are to be stored under `mydb`, the file `.cshrc` is modified to include

```
setenv DALI_DBPATH ~/mydb
```

5. The MM-Ode system works in the client-server manner. The DBMS is the server process and user defined applications are client processes. Before client programs can be run, the MM-Ode server must be started by executing the command

```
server1
```

Only *one* server can be running on a machine. The server is typically run in a separate window because it prints messages. If the server crashes, command `servern`

is used to restart the server without destroying the data. Servers can be shut down with an interrupt, e.g. Control-C.

The application development and execution environment of MM-Ode is supported by the O++ interface. MM-Ode consists of the O++ compiler and the object manager library. The O++ object manager library uses the Dali storage manager library. Database applications are written in O++. The O++ compiler produces C++ code, which is then compiled with the C++ compiler.

The database creation application is written in O++. Each O++ source file that uses O++ database facilities must include the header file *ode.h*. Class database, which is automatically made available by including the header file *ode.h*, provides functions for manipulating the database and naming persistent objects. The following database operations:

`open,`

`close and`

`remove`

can be invoked from outside a transaction body. All other operations must be invoked from within the transaction body. Objects of any class type and of the primitive type can be made persistent. Persistent objects are created and deleted using operators `pnew` and `pdelete`. The following example code explains various database operations.

```
// phone.h    Definition of class phone-connection
#ifndef phone_h
#define phone_h

persistent
class phone {
    public:
        indexable long pnumber;

        phone() { pnumber = 0; }
        ~phone() { pnumber = 0; }
```

```
        phone(long number)
            { pnumber = number; }
};

#endif

// createdb.c    Creation of database using a data file (C++ file operations)
#include <ode.h>
#include "phone.h"
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>

main()
{
    long num;
    database *db;
    ifstream datafile("set1.dat");

    if(!datafile) {
        cout << "data file can not be opened" << endl;
        exit(1);
    }

    if ((db = database::open("dataone")) == NULL) {
        cout << "dataone can not be opened" << endl;
        exit(1);
    }
}
```



```
trans {  
    cout << endl << "Creating the database....."<< endl;  
    while(datafile >> num)  
        pnew phone(num);  
    cout << "DONE!!"<< endl;  
}  
  
datafile.close();  
db->close();  
return 0;  
}
```

Suppose we have a telephony system. All phone number entities are represented by the object *phone*. The definition of *phone* is given in the header file *phone.h*. To make things simple, for the moment we assume that the object contains only one component called phone number *pnumber* as described in the file. The *phone* object can be created using the function *phone(long)*.

The database *dataone* is created to store the *phone* objects. The member function *open* of the class *database* is used to make connection to the database. The function returns the pointer to the database named *dataone* if it exists. Otherwise, it creates new database and associated pointer is returned to the calling function. The objects created by *phone(long)* are made persistent using function *pnew*. The C++ language provides the class *persistent* and functions for its operation. Another member function of class *database*, i.e. *close* closes the database no longer in use, upon its invocation in the program.

The query interface to the database is provided in C++ language as shown below. The *for* loop along with the predicate given in *suchthat* clause returns the objects in the database which satisfy the predicate. The objects can be manipulated inside the transaction body, if a transaction is performed in the update mode (i.e.

without the word readonly in trans block).

```
#include <ode.h>
#include "phone.h"
....
main()
{
    long num;
    persistent phone *ptrph;
    database *db;
    .....

    if ((db = database::open("dataone")) == NULL ) {
        .....
        .....
        readonly trans {
            for (ptrph in phone) suchthat (ptrph->pnumber == num) {
                < do something ..... >
            }
        }
        .....

        db->close();
        return 0;
    }
```

Limitations The system works in a pair of the client and the server. *Only* one server can be running on a machine at a time. There can be multiple clients connected to a single server. Before client programs can be run, the MM-Ode server must be started. The client-server design of the system causes some rigidity in using the

system for the distributed database in the present set-up of the network we use. This issue is presented in detail in the following sections.

3.3 The Message Passing Interface

MM-Ode is a centralized database system. For creating the distributed database using MM-Ode, we need to provide the means for communications among different MM-Ode systems. The MM-Ode systems are installed on different host machines on a network of workstations. A communication subsystem is developed on top of the networking software and the host operating system so that the MM-Ode hosts can exchange the information via the network. There are several message passing libraries available to accomplish this important task: such as MPI (The Message Passing Interface), PVM (Parallel Virtual Machine). The p4 system, Express. The Linda system, Chameleon.

We have access to two popular message passing systems. MPI and PVM. They both provide an unified framework within which the parallel and distributed applications can be developed in an efficient and straightforward manner. MPI is primarily designed to produce high performance parallel applications on massively parallel processing (MPPs) computers. Therefore the system is highly optimized for faster communications. In contrast, the main goal of the design of PVM is to provide portability and interoperability among machines in heterogeneous environment at the sacrifice of optimal performance. MPI is formally specified and is a standard for message passing system. We use MPI libraries to construct the distributed database. It is preferred considering the following features:

- MPI has more than one freely available implementation, e.g., LAM, MPICH and CHIMP. LAM is really good for application development and learning. LAM based MPI implementation is stable and thorough.
- MPI has the ability to specify a logical communication topology. New group formation is collective and group membership information is distributed.

- MPI has full asynchronous communication. The computation can overlap the send and receive operations in non-blocking mode.
- MPI efficiently manages the message buffers. Messages are sent and received from user data structures and not from staging buffers within the communication library. The memory-to-memory copying is avoided.
- MPI is totally portable. The application can be recompiled and run on any implementation of MPP or clusters (i.e. network of workstations). It can be used for implementation in a heterogeneous environment.
- MPI supports an application programming interface with convenient C bindings.
- MPI provides a reliable communication interface. The user need not cope with communication failure. It is designed to allow for thread-safety.
- MPI is formally specified and is accepted as a standard for message passing system. It proves the credibility and stability of the system.

The LAM implementation of the MPI is used for developing the applications using the distributed database. The LAM library offers a wide range of communication options for distributed application. We use asynchronous send and receive with few other basic functions. Following tools are used to create and run the distributed applications.

1. To run any LAM application, the following new variable and search path must be established for the shell by adding the commands to shell start-up file `.cshrc`

```
setenv LAMHOME < LAM installation directory >  
set path = ($path $LAMHOME/bin)
```
2. Any native C compiler is used to translate LAM programs for execution. LAM provides wrapping command called `hcc`, which invokes `cc` with the proper

header and library directories. Major internal libraries are automatically linked and MPI is explicitly linked.

```
hcc -o <executable file> <source file.c> -lmpi
```

3. Before starting LAM, we specify the machines that will form the multicomputer. Create a host file listing the machine names, one on each line.

```
# sample host file with 2 hosts
```

```
machine-1
```

```
machine-2
```

The first machine in the host file will be assigned nodeid 0, the second nodeid 1 etc. To verify whether the multicomputer is ready to run LAM, use `recon` tool. It checks if the user has access privileges on each machine in the multicomputer and if LAM is installed and accessible.

```
recon -v <host file>
```

If `recon` does not report a problem, we can proceed to start the LAM session with the `lamboot` tool.

```
lamboot -v <host file>
```

The `-v` option causes `lamboot` to report on the startup process as it progresses. It returns user's own shell prompt.

4. To execute a program, `mpirun` command is used. The `-c <#>` option runs copies of the given program on nodes selected in a round-robin manner.

```
mpirun -w -c 2 <executable file>
```

The invocation of executable program above assumes that the program is locatable on the machine on which it will run. `mpirun` can also transfer the program to the target node before running it.

5. The progress of the LAM processes can be monitored by commands `mpitask` and `mpimsg`.
6. To terminate the LAM session, `wipe` tool is provided. The host file argument must be the same as the one given to `lamboot`.

```
wipe -v <host file>
```

The distributed applications are built on the framework provided by MPI/LAM. The LAM network creates a multicomputer system. Each node in the network is identified by a nodeid. The host node from where the LAM network is established carries the nodeid 0. A distributed application is invoked from the host node0. The same copy of the executable program will run at each node in the LAM network.

The distributed LAM application is structured in a modular manner using C language. A piece of work performed by each node is put together in the form of a module or a function. These modules are assigned to nodes based on their nodeid in structuring the application. When the distributed application is executed, each node has a copy of the executable program. A node executes the piece of code assigned to it. During the course of execution, the nodes can exchange data and information through message passing. The messages are handled by LAM communications system. The example below shows the structure of the distributed application using MPI/LAM.

```
/* mpi.c   An example of distributed application using MPI */
#include <stdio.h>
#include <mpi.h>

main(int argc, char *argv[])
{
    int nodeid, netsize;
    ....

    MPI_Init(&argc, &argv);    /* initializing the LAM network */
    MPI_Comm_rank(MPI_COMM_WORLD, &nodeid);    /* identifying the
nodes */
    MPI_Comm_size(MPI_COMM_WORLD, &netsize);    /* getting the clus-
ter size */
```

```

    if (nodeid == 0) {      /* work assigned to local host */
        < operations performed by local host >
        .....
    }
    elseif (nodeid == 1) {  /* work assigned to node1 */
        < operations performed by node1 >
        .....
    }
    else {                 /* work assigned to rest of the nodes */
        .....
    }

    MPI_Finalize();        /* finalize the LAM application */
}

```

As shown in the mpi code listed above, all statements of the application must be included within the statements `MPI_Initialize()` and `MPI_Finalize()` in function `main()`. The nodes perform the code given inside the block according to their `nodeid`. There are few options available for designing the communications among processes running at different nodes. In synchronous mode, sender and receiver nodes participate in the message passing with coordination. While in asynchronous mode, the nodes perform their operations separately. The sender does not have to wait for the receiver to complete the operation. Hence, computation and communications can be overlapped in the execution of the application. This method resembles the practical scenario in distributed computing. We use `MPI_Isend()` and `MPI_Irecv()` functions to carry out message sending and receiving operations asynchronously.

The structure of send and receive statements are as follows:

```

MPI_Isend (void *buffer, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request request)

```

`MPI_Irecv (void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request request)`

<code>*buffer</code>	address of data to be sent [input]
<code>count</code>	number of elements to be sent [input]
<code>datatype</code>	MPI data types (for e.g. <code>MPLINT</code> for integer) [input]
<code>dest</code>	nodeid of destination process [input]
<code>source</code>	nodeid of source process [input]
<code>tag</code>	message tag (<code>MPI_ANY_TAG</code> as default) [input]
<code>comm</code>	communicator, group of processors (<code>MPLCOMM_WORLD</code> for current group) [input]
<code>request</code>	communication request handle [output]

Along with these two basic message passing functions, we use `MPI_Iprobe()` to check the system buffer for available messages. The message can be sent directly from the buffer of user defined variable or it can be constructed from a number of variables before sending it. `MPLPack()` and `MPLUnpack()` are used to create and extract messages of heterogeneous types. More details on MPI can be found in [GLS96].

Limitations MPI provides a static process control model. The nodes in such a model may be fixed at compile or spawn time. An application can not change the size during its execution. Failure of a single program module causes the entire application to fail by invalidating its message communicators.

3.4 The Integration of MM-Ode and MPI

We have seen the utilities and operations of the MM-Ode system and MPI. The distributed database is constructed using these two system components. It forms the basic testbed for the implementation of the transaction processing model. We present the integration of these components in detail.

3.4.1 Database Design

The database is distributed among three nodes. Based on the distribution of data, the distributed database falls under one of the three categories. In a fully partitioned database with no replication, each data exists at one node only. We will discard this case for data availability and reliability reasons in the model experiments. We use a replicated database to maintain high degree of data availability and to ensure reliability in case of a database node failure. With reference to replication, two cases are taken into account. In a fully replicated database, the complete database is stored at each node. While in partially replicated database, the database is split into three fragments. Each node stores two out of three fragments. In this way, each fragment exists at two nodes and the system can remain operational in case of a single node failure.

3.4.2 Installation of the Component Databases

Before the client application program can be run, the MM-Ode server must be started by executing the command *server1*. Only one server can be running on a machine. When the server is invoked from a user account, it searches for the default user directories for creating the user database files and the system database files. This information is provided in the user account start-up file. (i.e. *.cshrc* file that is used to initialize the user environment at the time of logging in) .

Once the server becomes active, the database creation application is executed. The database is loaded from a file at each node. The database is stored under the default database directories. More than one database can be created, but all databases will be stored under the same default database directory defined at the time of invocation of the server. The databases of a single user account are connected to one particular server at a time. The databases remain operative as long as the server is running on the machine. For the local database operations, the application can be run on the local server. When the application involves the use of distributed databases, the integration of the remote database servers becomes necessary. In the

Figure 3.2: Distributed database using the MM-Ode system

following section, we explain how this interconnection is made possible.

3.4.3 The Interconnection of the Component Databases

All the component databases are created at every database node prior to running the transaction processing application. The interconnection of the databases is established using MPI. The configuration of this network is shown in Figure 3.2. The MM-Ode system works in the client-server fashion. The MM-Ode servers run at database nodes in the LAM network. The transaction processing application uses the services of these servers. It acts as the client process. The application is designed in distributed manner using LAM libraries. It spawns processes on each node of the LAM network. The local LAM process works as a client for the database server at that node and carries out the required database operations.

In the current environment of our network, the NFS(Network File System) is cross mounted on all hosts. Therefore, the user files are shared among all the hosts. When we logon to any host on the network, the system always maps the same set of physical files. We have already mentioned before that the MM-Ode system allows only one server to be operating on a machine. Based on these two facts, We can see that only one server can be started from an individual user account. When we start a server on one machine, the user databases are attached to this server. If we try to start a server on another machine from the same user account, the MM-Ode

system does not permit such an action. It happens because the user databases are already controlled by the active server on some other machine.

The way the installation of MM-Ode works, we need n separate user accounts for n different servers to run on n different databases at n nodes. We have three database nodes. Hence, the servers have to be connected to three different databases residing under three different user accounts. On the other hand, the MPI/LAM network can be set from only a single user account. The LAM application spawns processes that will run under the same user account. Therefore, all the LAM processes inherit the same user environment. A process (working as a client) can connect to the server for the database services, only if the environment of the server process and the client process match. Considering these facts about the operations of MM-Ode and LAM, the integration of two system components becomes very crucial task. It has been achieved in the following two stages. (figure 3.3)

Stage 1: The component databases at different nodes are installed from the same user account, but with the different default database directories in other user accounts. We logon to three machines from the same user account. Next we change the path of database directories through resetting the environment variables ODE_DB_DIR and DALI_DBPATH to point to different user accounts. We then start the MM-Ode server under the modified environment on each machine. The server processes will have the same environment except the default database directories. Three different databases are created, one on each machine.

Stage 2: In order to make the LAM processes on different machine work as clients for the MM-Ode server running at those machines respectively; the environment of the client processes need to be changed to the same as the servers' environment. Therefore, when the LAM processes are initialized at each node, the default database directories in the LAM processes are reset to match with the servers' environment. This operation must be performed before any transaction processing operation can take place. It makes the client LAM processes compatible to work with the server processes.

In this way, the application client processes and the server processes are inte-

Figure 3.3: Integration of MM-Ode and MPI

grated for the database operations. The interaction between the LAM processes and the MM-Ode server is the central aspect in the design of a transaction processing system using the distributed main memory database.

3.5 Design of a Distributed Database Application

In previous sections, we have described the design of a database application and a distributed application in LAM network. We now present a distributed database application by combining the logic of both the applications. We assume that the database is divided into two partitions. The partitions are stored at two nodes separately. The LAM interface is provided between these two nodes. As we discussed before, the procedure to create compatible environment for database operation follows the sequence of steps given below:

1. Two nodes in the network are identified as *node0* and *node1*. At *node0*, we logon to *acct1* and start the MM-Ode server. The database *dataone* is installed under the *acct1*. It is attached to the server running at *node0*.
2. At *node1*, we remote logon to *acct1* through telnet. The MM-Ode server running at *node0* is already controlling the database of *acct1*. We want to invoke another MM-Ode server on *node1* to manage the second database. We change the environment variables for default database directories of *acct1* to point to some other user account *acct2* using the command *setenv*. The MM-Ode server is started on *node1* through *acct1*. The database *datatwo* is installed under *acct2* which is attached to this server.
3. We create the LAM network from *node0*. *node0* works as the local host and *node1* is the remote host.
4. For performing the distributed transaction processing application, the application is structured as shown in the following algorithm. We assume that *node0* acts as a query-coordinator.

```
function main()
{
    Initialize the LAM process
    Verify the size of the network
    Identify the nodeid

    if (nodeid == 0)           [i.e. local host as query-coordinator]
        Take a query
        Find out the database portion of the query data
        if Query belongs to "dataone"
            Perform local query processing and invoke
            database operations module
        if Query falls under "datatwo"
            Send the query to the remote database node1
            Wait for the result and collect it
            Get the next query and continue the sequence

    if (nodeid != 0)           [i.e. remote database host]
        Wait for a query and collect it
        Invoke the database operations module to
        process the query
        Send the result to the host [i.e. node0]
        Wait for the next query and continue

    When all queries are processed, finalize the LAM process
}
```

The database operations module performs the actual query processing against the database attached to the server. It first makes connection to the local database. The transaction block corresponding to the query gets evaluated next. The module

returns the result of the query to the calling function. In this way, the transaction processing system is designed for the distributed database applications.

Chapter 4

Design of Transaction Processing Models

Several real-time applications demand fast responses for the queries. The transaction processing system for the service is built upon the distributed main memory databases for obtaining the high performance. In the previous chapter, we developed such a distributed database by integrating the distributed database components with the message passing system. There are quite a few options for designing the transaction processing system using the distributed database. In this work, we concentrate on the following two alternatives:

- (1) The system keeps the complete data distribution information
- (2) The system does not keep the information about the distribution of data

Two transaction processing models are designed based on these alternatives. The objective is to compare the performance of read-only queries in both the models, in terms of the response time. Two sets of experiments are conducted. In the first set, the models are evaluated for the exact-match queries. The second set considers the range-search queries.

The transaction processing model describes the sequence of operations performed for processing a query in the transaction processing system. The structure of the transaction model for a given application is determined by various factors. These include the duration of the application, the amount and type of the computation, the response time requirement etc. In the application under consideration, the transactions are simple search queries. However, the applications run for long time.

The query response time is an important factor in the design of the models. We describe the structure of the transaction and its execution in the models in detail in the following sections.

4.1 The Execution Environment

The transaction processing model is made up of several modules. The modules are designed with the following execution environment in view.

4.1.1 System Configuration

The distributed database is mounted on a network of workstations as described in section 3.1. There are three database nodes with MM-Ode systems installed on each of them. In addition, one of the nodes in the network is designated as the query-coordinator. The database nodes and the query-coordinator node are interconnected by the LAM network in the design of the models. The layout of this network is the same as shown in Figure 1.1. The main function of the query-coordinator is to supply a synthetic workload into the system and to carry out the transaction processing.

4.1.2 Database Design

The database is distributed and replicated on three nodes. Two cases of the distributed database are considered. In the partially replicated database, the database is split into three fragments. Each node stores two out of the three fragments. In fully replicated database, the complete database is stored at all three nodes.

4.1.3 Workload

The workload for the transaction processing model is generated by an emulator function provided in the query-coordinator. It is responsible for feeding the transaction request queue for driving the model experiments. The queries generated imitate the request from user terminals. The C random number generator function `random()` is

used to create query requests. A query is in the form of a long integer value that is used to identify the objects.

4.2 Assumptions for the Design

In our work, the transaction processing models are designed in a client-server manner. The client is the transaction_manager residing at the query-coordinator and the server is the query_processor running at each database node. The query_processor in turn, works as a client for the local MM-Ode server. The query-coordinator and the database nodes are connected by the LAM interface. The following assumptions are made in designing the models.

1. The MM-Ode servers are made active before commencing the execution of the transaction processing application.
2. The query-coordinator carries out two major tasks in the system. It performs the transaction operations management as well as it supplies a synthetic workload into the system. A query generator module is designed to create the queries of the required form.
3. The transaction manager of the query-coordinator ensures the ACID execution of the transactions. The *flat* transaction model is used. The transaction is defined as the set of operations performed to process a single query.
4. The query-coordinator schedules these two modules as Poisson processes. Poisson process is assumed since most natural occurring phenomena have such inter-arrival time with random distribution.
5. The transaction manager in the query-coordinator is scheduled based on the observed query service rate at the current system state.
6. The query generator is scheduled based on the assumed query arrival rate.

7. The models are designed to perform read-only queries in single-user mode. They process only the read-only queries in the sequential order and hence, no special concurrency control and logging measures are taken into account in the design.

The transaction processing logic of two models differs at the following points.

(1) Master-slave model

- Whenever a server becomes free, it sends a request for a query to the query-coordinator. In this way, the query-coordinator always knows which node is idle and available for the service.
- The transaction manager knows the distribution of data across database nodes. When a query arrives, the transaction manager directs it to a particular node where the requested data resides. The transaction manager is also aware of the status of the remote nodes, i.e. busy or idle, when it gets the query is to be processed.
- Every query is processed by exactly one node.

(2) Broadcast model

- Each database node has a local query queue. The server checks for a query in its queue as soon as it becomes free. If a query is available it processes the query and sends the result to the query-coordinator. If no query is available for processing, it keeps busy waiting until next query arrives.
- The transaction manager does not keep the information of the data distribution. It does not know the status of the remote nodes as well. Whenever a query arrives, the transaction manager broadcasts the query to all the nodes.
- The transaction manager considers the first "yes" reply for response time calculation, in case of the exact-match search in the partially replicated database. Since transactions are real-time, there are deadlines after which the query is discarded.

- A query is processed by all the nodes.

4.3 The Structure of the Modules

The transaction processing model consists of a number of modules. The query-coordinator contains the query generator module and the transaction manager module. The query-generator creates the synthetic workload into the model. The transaction_manager forms the logic of transaction processing activity.

The function of the query_generator remains the same for both the models. For exact-match case, the query is a single key value. For range-search the query is composed of two key values representing the upper and lower limits for the search.

The transaction_manager provides the core function of the transaction execution. It monitors the atomic operations for all queries. When a query is sent for processing, the transaction_manager puts a tag on it representing the invocation of Begin_Work function. The successful end of the transaction is marked by receiving the result of the query. It is similar to Commit_work. The Rollback_Work operation exists in a primitive form. Any incomplete transaction is started after a particular set of queries are processed.

The scheduling of the query_generator and transaction_manager in the query-coordinator is simulated. The transaction_manager remains the same for all the database nodes.

query-coordinator => query_generator

begin

Generate next query data

Take start-time of the query

Enqueue the query into input_queue

if input_queue was empty

Schedule next processing event

Generate next query generating event

it into event_queue

end

4.3.1 The Master-Slave Model

The flow of a transaction through the model is shown in Figure 5.1. The algorithms of the query-coordinator and transaction_manager are defined below.

query-coordinator

global parameters: *event_queue, input_queue, output_queue,*
requester_list, statistical parameters

begin

Create the first event of query generation

For given number of queries do

Take the next event from event_queue

if (*event-type = GENERATE_QUERY*)

Call query-generator

else (*event-type = PROCESS_QUERY*)

Call transaction_manager

enddo

Calculate the results' statistics

end

query-coordinator => transaction_manager

begin

Check the system message buffer for request messages

Begin_Work

*Take next query from input_queue and take the
appropriate requester from the requestors' list*

Send the next query to the requester

Wait for the response

Figure 4.1: Flow of a transaction in master-slave model

```
        if result is available then invoke
            Commit_Work
        else prepare to restart the transaction and call
            Rollback_Work
        if there is a query to be processed
            Schedule next processing event
    end

query_processor (database node)

    begin
        Send a request for query and wait
        Accept the query and invoke process_query
        for database operations
        Get the result and send it to transaction_monitor
    end
```

The query-coordinator starts its execution with the creation of the first event of GENERATE_QUERY. The event is enqueued into the event_queue and the query-coordinator starts processing work. The query-coordinator takes the event from event_queue. If it is a GENERATE_QUERY event, it calls query_generator. The query generator creates a query and puts it into input_queue. The query structure includes a query identifier, the query data and a field for start-time of the query. While inserting the query request, if input_queue was found empty, then query_generator creates a PROCESS_QUERY event and enqueues it into event_queue. Every time query_generator is invoked, it creates a GENERATE_QUERY event and puts it into event_queue. The organization of input_queue is different for the two database cases. In a fully replicated database, there is a common input_queue. In a partially replicated database, separate queues are maintained for each fragment. The query_generator enqueues queries into the appropriate queue.

When a `PROCESS_QUERY` event is scheduled, query-coordinator invokes `transaction_manager`. `Transaction_manager` checks the system buffer for messages. All request messages are put into `requester_list`. The reply messages are collected and query response time are calculated. The completed queries are deleted from the query `output_queue` and respected transaction is marked as completed. After collecting all the messages, the `transaction_manager` sends one query to the requester. It transfers the query being sent for processing from `input_queue` to `output_queue` and marks the start of the transaction. Each time `transaction_manager` is invoked, it sends one query for processing and collects atleast one result. In this way, every time `transaction_manager` is scheduled, one service is performed. This tries to coordinate the real transaction processing service rate with the simulated one. If there is a query in the `input_queue`, `transaction_manager` creates a `PROCESS_QUERY` event and enqueues it. The same operation sequence continues until all queries are processed. If the result is not received for a query until the end, the query is restarted.

The `query_processor` runs continuously at the database nodes from the time it is initialized. At start, it sends a request for query to the query-coordinator indicating that it is available for the service. In return, the `query_processor` gets a query from the `transaction_manager` of the query-coordinator. It extracts the query data and invokes the database operations module. The `query_processor` obtains the result of the query. The result is sent back to the query-coordinator followed by a request message for next query. The server finishes its work when it gets a completion signal message from the query-coordinator.

For range-search queries, the processing sequence remains almost the same. The structure of query message and result message change. The database operations module of the server will be modified according to the new query processing requirements. A query may involve search in more than one database node in case of a replicated database. Therefore, a separate query status list is maintained in addition to `input_queue` and `output_queue`. It keeps the information about the results obtained from the servers in order to measure the response time of the query.

4.3.2 The Broadcast Model

The transaction processing algorithm in the absence of information about data distribution remains the same for all cases of the database. The algorithm for transaction processing is given below.

query-coordinator

```

global parameters: event_queue, input_queue,
begin
    Create the first event of query generation
    For given number of queries do
        Take the next event from event_queue
        if (event-type = GENERATE_QUERY)
            Call generate_query
        else (event-type = PROCESS_QUERY)
            Call transaction_manager
        enddo
        Calculate the results' statistics
    end

```

query-coordinator => transaction_manager

```

begin
    Begin_Work
        Broadcast the queries from input_queue
        Wait for the response
        if result is available then invoke
    Commit_Work
        else prepare to restart the transaction and call
    Rollback_Work
        if there is a query to be processed

```

Figure 4.2: Flow of a transaction in broadcast model

```
                                Schedule next processing event  
end  
  
query_processor (database node)  
  
begin  
    Check the system buffer for a query  
    Take the query and invoke process_query for  
    database operations  
    Get the result and send it to transaction_monitor  
end
```

The query-coordinator starts its operations by creating the first event of GENERATE_QUERY. It enqueues the event into event_queue. The processing work begins. The query-coordinator takes the first event from the event_queue. If it is a GENERATE_QUERY event, it invokes query_generator. The query generator creates a query and puts it into input_queue. If input_queue was empty, it schedules the next PROCESS_QUERY event. It also creates GENERATE_QUERY event and enqueues it.

Query-coordinator invokes transaction_manager, when the type of the event is PROCESS_QUERY. Transaction_manager checks the system buffer and collects all the result messages received from servers. The completed queries are deleted from output_queue and corresponding transaction is said to be committed. After that, transaction_manager sends queries from input_queue for processing. It broadcasts all waiting queries to the servers (see section 5.3.3 for further discussion). When a query is sent for processing, it is transferred from input_queue to output_queue and the transaction is marked as started. Transaction_manager schedules the next PROCESS_QUERY event. All the queries are processed in this manner. All incomplete transactions (for which the results are not received) will be restarted at the end.

The query_processor runs at the database node. The query_processor takes a request from the system buffer and extracts the query data. It invokes the database operations module to process the query. The query_processor gets the result that it sends to the query-coordinator. It then takes the next query and processes it. If no query is available, it keeps busy waiting for arrival of a query. This sequence continues till the server is notified of the completion.

For exact-match queries, the first "yes" reply for the query is considered for the response time calculation. In case of the range-search queries, all the responses are taken into consideration.

Statistical Parameters In both models, a set of statistical parameters is used. The query structure keeps the information of the starting time of the query, when it is generated. Another time parameter is used to take the waiting time of the query when it is sent for the processing. The query_processor of the database node keeps track of the query execution time. When the result is collected by the transaction manager of query-coordinator, total turnaround time of the query is calculated. Along with time, the buffer occupancy is also noted. When the query generator inserts new query into input_queue the buffer count is increased. The transaction manager decreases the counter when a query is sent for the processing. The total program run time and the simulation clock time are recorded. Various statistical parameters such as average query response time, average query execution time, average waiting time, average buffer occupancy, server utilization and throughput of the system are observed for the experiments.

4.3.3 The Model Implementation

The models are implemented in C language using MPI libraries and MM-Ode database operations. The DBMS of MM-Ode provides a database operations interface in O++, which is an extension to C++ for persistent database operations. The O++ compiler produces C++ code. The MPI libraries and C routines are in-lined with the C++ code and compiled under the Sun C++ compiler.

Limitations The MPI system buffer does not allow more than certain number of messages in the system at a point in time. The implementation of broadcast model has been modified to adapt to the limitation. Transaction_manager does not broadcast all the waiting queries from input_queue. Instead, it broadcasts a fixed number of queries (we observed maximum 6 messages) at a time to all the servers. Normally, this will not affect the query response time, as the query is either waiting in the input_queue or in the local buffer of the server. However, this limitation has significant impact in certain critical situation when the database processors work under different loading conditions. The experiments are carried out on a general-purpose network. Consequentially, the imbalance of the external loads on the database processors can impair the performance of the algorithm.

4.4 Update Transactions

We have considered the transaction processing system for the read-only applications so far. The database serving the real-time applications do change over the time. In order to keep the system up-to-date for effective operations, update transactions are to be incorporated into the system. The update operations change the contents of databases. Hence, they must be handled in an ordered manner. The persistency and consistency of the data are important data management issues concerning the update transactions. The type of the real-time applications we are considering here are time-sensitive. They demand fast responses for the queries. On the other hand, an update transaction takes relatively long time for data processing. It may interfere with the performance of the system. Therefore, update transactions are given special considerations.

In case of the application where the query services operate non-stop without any down time, a separate dedicated update transaction processing system is designed. It has its own copy of the database. The system works independently in parallel to the query processing system. The query system keeps the query database open for only read activities. The update transactions are run against the update database.

The system stores the information of the updated records. It propagates the update records to the query database at low traffic time. The update system also maintains the information on the status of the query database. It is responsible for applying proper consistency measures for the replicated data. The system employs recovery mechanisms to ensure the persistency of the data.

For the applications operating in regular service hours and allowing fixed down time, online updates can be handled efficiently. The database is kept open for the read-only applications during the service hours and updates are queued in this time. During the off hours, the batched updates are applied to the database in a sequential manner. The system maintains the integrity and persistency of the data. The updates of replicated data are applied to all the copies of the data. The log of the update operations is stored to provide the durability of the data.

The transaction processing system dealing with the update operations has components for data management. The replica manager ensures that the update is applied to all copies of the data. The log manager keeps the copy of old and new data before completing an update operation. It also takes the log of the operations performed by the replica manager. Normally, the updates are handled sequentially in the systems. Therefore, no global locking on the data is necessary as the transactions are run in isolation. However, when concurrent updates or delayed updates are allowed, a lock manager is designed to maintain the integrity of the data.

Chapter 5

Experiment Results

The results of the experiments conducted on transaction processing models are presented in this chapter. Two sets of experiments are carried out to see the performance of read-only search in the two models. The influence of any external load on the network is ignored. We assume that there is only one application of transaction processing model running in the network. In addition, we assume that only one traffic stream is present in the network that does not lead to any congestion. Also the involved processors are assumed to be running the transaction processes exclusively. The models are tested under this ideal condition for fair comparison.

The models are evaluated for response time of the queries with respect to the throughput of the system. The response time of a query is defined as the time interval between the arrival of the query into the system to the time the result of the query is obtained. The throughput of the system is the ratio of number of processed queries leaving the system within predetermined time interval. The response time is the main comparison criterion for the application under consideration. The throughput is measured at various system loads. By load, we mean the number of queries processed per unit time. We make relative performance comparisons with respect to the above two metrics.

5.1 Input Parameters

In this section, we provide comprehensive details of the appropriate system parameters taken for our models in the experiments.

Database Design

(A) Partially replicated database :- The database is created using `srandom()` function. The function returns a long integer value from range 0 to $2^{31}-1$. The size of the database is 6,000 objects. The database is split into three fragments, each of size 2,000 objects. The fragments are designed with the ranges

- 0 to 1.5×10^9
- $1.5 \times 10^9 + 1$ to 3.0×10^9
- $3.0 \times 10^9 + 1$ to $2^{31} - 1$

Each node stores two fragments making component database of size 4,000 objects.

(B) Fully replicated database :- The database is created using the same `srandom()` function. There are 6,000 objects in the database. The complete database is stored at every node.

Exact-match queries are performed on both the types of the database. The range-search queries are performed only on the partially replicated database. The database is loaded from a file at each node.

Workload

The queries are generated using `random()` function. Exact-match query has one key value as data, while range-search query contains two numbers of query data.

Arrival rate

The arrival of queries is assumed to be a Poisson process. The mean arrival rate for the queries is chosen according to service rate offered by the system at the

current state to match the real transaction processing operations.

Service rate

The results of the model operations are taken on a general-purpose network. The varying loading conditions on the network drastically affects the operations of the models. To derive a mean service rate, the system state is observed through a separate program. This program processes 100 queries of the same type on each node. The average query execution time is considered as a parameter in the design of the service rate. The comparison of the response time for different system throughput is made for results obtained at the same service rate.

When experiments for exact-match queries are conducted, the observed average query execution time is 10.728sec for the three nodes. The service rate is taken as 0.28queries/sec. For range-search queries, the execution time is 13.637sec, the corresponding service rate taken is 0.22queries/sec.

5.2 Comparison of exact-match queries

The transaction models are tested for exact-match queries into two different database designs, namely fully replicated database and partially replicated database. The features of the testing environment is presented in Table 5.1. The performance of the models is observed for various query arrival rate. The response of the query takes longer as the number of queries entering the system, for a fixed time interval, increases. The server processes query at a certain rate. At lower query arrival rate, as soon as a query comes in, it gets the server for processing. Therefore, the query waiting time is less. As the difference in time between two consecutive query arrival decreases, the servers remain busy processing queries one after another. The query waiting time gets longer and the response time of the query increases. It can be seen in Figure 5.1 and Figure 5.2.

Table 5.1: System parameters for experiments of exact-match queries

number of databases	3
database size	6.000 objects
database size of each node	6,000 objects (fully replicated) 4,000 objects (partially replicated)
mean service rate	0.28queries/sec
mean arrival rate	0.05 - 0.28 queries/sec
number of queries tested	10,000
number of batches	10
number of queries per batch	1.000

Figure 5.1: Comparison of models for exact-match queries : partially replicated database

5.2.1 Partially Replicated Database

As shown in figure 5.1, the response time of the query increases with the load on the system. The same trend is observed in both the models. However, the response time obtained in broadcast model is higher than in master-slave model for the same amount of throughput. The obvious reason behind this difference in response time is due to the parallel query execution in master-slave model. If more processors are available, then queries are distributed to them. There can be more than one query being processed in the system at an instance in time. On the other hand, every processor gets the same query in broadcast model. Therefore, waiting time of a query at the processor node is more in broadcast model. In partially replicated database, data resides at a few nodes and not at all the nodes. If one node gets busy because of the arrival pattern of the queries, the average response time of the queries goes up. Its effect can be observed in both the models. In master-slave model, the request are blocked in waiting queue. In broadcast model, the "yes" response gets delayed and the response time for the queries increase.

5.2.2 Fully Replicated Database

The performance of exact-match search for fully replicated database is shown in figure 5.2. In this case also, master-slave model performs better than broadcast model. In fact, in master-slave model, the query can be sent to any processor as soon as it becomes free. There is more degree of parallelism in query execution. This results in lower response time compared to the case of partially replicated database. And consequently, the difference between the response time of the two models in this fully replicated database is more than in the partially replicated database. It can be observed from the graphs in Figure 5.1 and Figure 5.2.

Figure 5.2: Comparison of models for exact-match queries : fully replicated database

Table 5.2: System parameters for experiments of range-search queries

number of databases	3
database size	6,000 objects
database size of each node	6,000 objects (partially replicated)
mean service rate	0.22queries/sec
mean arrival rate	0.05 - 0.22 queries/sec
number of queries tested	1,000
number of batches	5

Figure 5.3: Comparison of models for range-search queries

5.3 Comparison of Range-Search Queries

The results of two models for range search queries show the same trend as in the exact-match queries. The testing parameters of the models are given in Table 5.2. The average service rate is 0.22 queries/sec. In the experiments, 1000 queries were tested in 5 independent batches for range-search in both the models. Master-slave model performs better than broadcast model. However, the possibility of the parallel execution of range-search queries in the master-slave model decreases in certain cases. Therefore, the difference between the performance of the two models is not much. The results of the query processing are shown in Figure 5.3.

The results of the experiments of exact-match queries and range-search queries on both models show that the master-slave model returns faster response. One reason is the possible simultaneous execution of queries. It has been observed that the parallel algorithm gets affected with the external loads more severely. The comparison of exact-match search in two different cases of the databases also shows that the availability of the data improves the query response time.

Chapter 6

Conclusions

6.1 Summary

The goal of this thesis is to design the high performance transaction processing system for real-time applications. These applications are characterized by high transaction rates and low response time of the queries. The read-only queries are predominant in these systems. Main memory databases are used to meet the stringent timing requirements of the queries. The transaction processing systems are built using the distributed main memory databases stored on multicomputer clusters. The data are replicated and distributed over a network of workstations. It increases the availability of the data and the reliability of the system. In this configuration of distributed databases, the transaction processing system can be designed in a number of ways. Two transaction processing models are designed for read-only applications in the systems. The models are evaluated experimentally for the transactions that execute exact-match and range-search queries. The models are tested for the response time of the queries under various loading conditions.

6.2 Main Contributions

In this research work, we have modeled the transaction processing system for the read-only applications in real-time systems. The main results of this work are:

- We have implemented a distributed main memory database on the network of workstations. The distributed database is constructed using the main memory database system MM-Ode and message passing system MPI. The integration of these two system components has prepared the testbed for the development and execution of the transaction processing applications.
- Two transaction processing models have been studied for the design of the high performance transaction processing systems using the distributed main memory databases. The models are designed to process the transactions involving exact-match and range search queries. They describe the sequence of operations performed for processing the transactions based on the information on distribution of the data.
- The transaction processing models are evaluated experimentally with the synthetic workload to compare relative performance of the response time of read-only queries. It has been observed that the information on distribution of the data improves the response time. The replication of the data also contributes in the reduction of the response time due to increased availability of the data.

6.3 Directions for future research

Due to the limitations of time and the system resources, we could not complete the entire modeling of the transaction processing system for the real-time applications. However, we have developed and tested the transaction processing models for the main read-only applications in these systems. The results of these models provide the basis for making decision regarding the data distribution and the structure of the transaction processing systems. As read data do not change the contents of the database, various transaction management aspects such as concurrency control, commit processing and recovery are not considered in the design of the models. The existing models are to be extended further to incorporate these aspects for the update transactions.

A lock manager is to be designed to serialize the data access. Basic 2PL algorithm can be used to coordinate the insertion, deletion and modification of the data. The exclusive locks are granted for the update transactions. Record-level locking is a suitable choice. The locks must be acquired on all the copies of the data before accessing the data. The locks must be held until the completion of the commit process. Two-phase commit protocol must be used to ensure the integrity of the data. The modification of the data involves deletion of the old data and insertion of the new data. These two operations must be carried out as atomic actions. The ACID transaction structure can be used for the operation. To guarantee the ACID transformation and reliability, a log manager should be used. The transaction manager should use write ahead logging technique to guard the system against any failure. Checkpointing can be used along with the logging for fast recovery. Force-at-commit must be used to avoid repeat execution. Group-commit can be used to reduce the I/O overhead, if the frequency of the update operations is high. The recovery and restart procedure can be designed as discussed earlier.

Various algorithms have been developed for the design of the functionality components of the transaction processing system [Gra81, Elm92, GR93, JBF⁻80, CP84, OV91b]. Few test models have to be designed using different combination of the algorithms for the components' design. These models should be tested to find out the optimal structure of the transaction processing system.

References

- [AA92] P.C. Aristides and A. El Abbadi. Fast real-only transactions in replicated databases. In *Proceedings of International Conference on Data Engineering*, pages 246–253, 1992.
- [AHK85] A.C. Ammann, M.B. Hanrahan, and R. Krishnamurthy. Design of a memory resident dbms. In *Proceedings of IEEE COMPCOM Conference*, pages 54–57, 1985.
- [AP92] A. Analyti and S. Pramanik. Fast search in main memory databases. In *Proceedings of ACM SIGMOD Conference*, pages 215–224, 1992.
- [BG81] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.
- [BHT87] D. Bitton, M.B. Hanrahan, and C. Turbyfill. Performance of complex queries in main memory database systems. In *Proceedings of International Conference on Data Engineering*, pages 72–81, 1987.
- [BLR⁺97] P. Bohannon, D. Lieuwen, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. The architecture of the dali main-memory storage manager. *Multimedia Tools and Applications*, 4(2):23–151, March 1997.
- [CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill Book Company, 1984.
- [Dat95] C. J. Date. *Introduction to Database Systems*. Addison-Wesley Pub. Co., Reading, Massachusetts, 1995.

- [Deu91] O. Deux. The o2 system. *Communications of ACM*, 34(10), October 1991.
- [DKO⁺84] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of ACM SIGMOD Conference*, pages 1–8, 1984.
- [Elm92] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, Calif., 1992.
- [ELMB92] A.K. Elmagarmid, Y. Leu, J.G. Mullen, and O. Bukhres. *Introduction to Advanced Transaction Models*, pages 33–52. In Elmagarmid [Elm92], 1992.
- [FZT⁺92] M.J. Franklin, M.J. Zwillig, C.K. Tan, M.J. Carey, and D. DeWitt. Crash recovery in client-server exodus. In *Proceedings of ACM SIGMOD Conference*, pages 165–174, 1992.
- [GLS96] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1996.
- [GM81] H. Garcia-Molina. *Performance of Update Algorithms for Replicated Data*. UMI Research Press, Ann Arbor, Michigan, 1981. Revision of P.hD. thesis.
- [GMS88] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD Conference*, pages 249–259, 1988.
- [GMS90] H. Garcia-Molina and K. Salem. System m: A transaction processing testbed for memory resident data. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

- [GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowledge and Data Engineering*, 4(6):509–516, December 1992.
- [GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, 1993.
- [Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of Conference on Very Large Data Bases*, pages 144–154, 1981.
- [Hig89] W. Highleyman. *Performance Analysis of Transaction Processing Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
- [Hsi83] D. Hsiao, editor. *Advanced Database Machine Architecture*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
- [JBF⁺80] J. Rothnie Jr., P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A. Landers, C. Reeve, P.W. Shipman, and E. Wong. Introduction to a system for distributed databases. *ACM Trans. on Database Systems*, 5(1):1–17, March 1980.
- [JLRS94] H.V. Jagdish, D. Lieuwem, R. Rastogi, and A. Silberschatz. Dali: A high performance main memory storage manager. In *Proceedings of Conference on Very Large Data Bases*, pages 48–59, 1994.
- [KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the orion: Next generation database system. *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.
- [KR81] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2):213–226, June 1981.
- [LC86a] T.J. Lehman and M.J. Carey. Query processing in main memory database systems. In *Proceedings of ACM SIGMOD Conference*, pages 239–250, 1986.

- [LC86b] T.J. Lehman and M.J. Carey. A study of index structures for main memory database systems. In *Proceedings of Conference on Very Large Data Bases*, pages 294–303, 1986.
- [LC87] T. Lehman and M. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of ACM SIGMOD Conference*, pages 104–117, 1987.
- [LLM⁺84] B. Lindsay, L.M.Hass, C. Mohan, P.F. Wilms, and R.A. Yost. Computation and communication in r* : a distributed database manager. *ACM Trans. on Computer Systems*, 2(1):24–38, February 1984.
- [LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore system. *Communications of ACM*, 34(10), October 1991.
- [LN88] K. Li and J. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, pages 177–187, 1988.
- [MN94] C. Mohan and I. Narang. Aries/csa: A method for database recovery in client-server architecture. In *Proceedings of ACM SIGMOD Conference*, pages 55–66, 1994.
- [Mos85] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, Massachusetts, 1985.
- [OV91a] M. T. Ozsú and P. Valduriez. Distributed database systems: Where are we now? *IEEE Trans. on Computers*, 24(8):–, August 1991.
- [OV91b] M. T. Ozsú and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. on software engineering*, SE-5(3):188–194, May 1979.

- [Tho79] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved