INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
CHOU, CHUEN-PU J.

SYSTEM DESIGN OF THE DISTRIBUTED LOOP DATABASE SYSTEM (DLDBS)

The Ohio State University

University Microfilms International

300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1981
SYSTEM DESIGN
OF
THE DISTRIBUTED LOOP DATABASE SYSTEM (DLDBS)

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
Chuen-Fu Chou, B.S., M.S.

****

The Ohio State University
1981

Reading Committee:

Dr. Ming T. Liu
Dr. Douglas S. Kerr
Dr. Jayashree Ramanathan

Approved by

Dr. Ming T. Liu
Advisor
Department of Computer
and Information Science
To my parents and my wife
I would like to express my sincere gratitude to my advisor, Professor Ming T. Liu, for his constructive suggestions and constant encouragement during the development of this research. His guidance and support have been invaluable in the completion of my doctoral program.

I would like to thank the members of the reading committee, Professors Douglas Kerr and Jayashree Ramanathan, for their criticism and valuable insight with regard to this dissertation.

I would also like to thank D. P. Tsay, J. J. Lin, Rick Tobin and many graduate students for creating an interesting working environment and for their assistance. Roberto Pardo, through many discussions during the early days of this research, also helped to clarify some of the ideas in this dissertation.

Finally and most importantly, I would like to thank my wife Jui-Yu for her support, patience, encouragement, and understanding.
VITA

January 1, 1951 ....... Born in Taichung, Taiwan, R. O. C.

1973 ............... B. S., Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

1976-1977 ............ Associate Instructor, Department of Computer Science, The Indiana University, Bloomington, Indiana

1977 ................. M. S., Department of Computer Science, The Indiana University, Bloomington, Indiana

1977-1980 ............ Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Computer Organization and Architecture.
Dr. Ming T Liu

Computer Programming, including Systems Programming.
Dr. Sandra Mamarak

Theory and Processing of Programming Languages.
Dr. Jayashree Ramanathan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiii</td>
</tr>
<tr>
<td>Chapter 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Distributed Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Distributed Double-Loop Computer Network</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Distributed Database Systems</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Objectives of the Dissertation</td>
<td>15</td>
</tr>
<tr>
<td>1.5 Related Work and Contributions</td>
<td>23</td>
</tr>
<tr>
<td>1.6 Organization of the Dissertation</td>
<td>29</td>
</tr>
<tr>
<td>Chapter 2 SYSTEM ORGANIZATION OF</td>
<td>33</td>
</tr>
<tr>
<td>DISTRIBUTED LOOP DATABASE SYSTEM</td>
<td></td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>33</td>
</tr>
<tr>
<td>2.2 Network architecture of DLDBS</td>
<td>35</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Hardware Configuration</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Communication Subsystem Architecture</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Distributed Database Architecture</td>
</tr>
<tr>
<td>2.3</td>
<td>Software Components in Distributed Loop Database Management System</td>
</tr>
<tr>
<td>2.3.1</td>
<td>System Configuration of DLDBS</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Software Components of Loop Request Nodes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Software Components of Loop Data Nodes</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>CONCURRENCY CONTROL IN DATABASE MANAGEMENT SYSTEMS</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>Database Consistency</td>
</tr>
<tr>
<td>3.3</td>
<td>Transaction</td>
</tr>
<tr>
<td>3.4</td>
<td>Concurrency</td>
</tr>
<tr>
<td>3.5</td>
<td>Concurrency Control</td>
</tr>
<tr>
<td>3.6</td>
<td>Serializability</td>
</tr>
<tr>
<td>3.7</td>
<td>Concurrency Control in Centralized DBMS</td>
</tr>
<tr>
<td>3.8</td>
<td>Concurrency Control in Distributed DBMS</td>
</tr>
<tr>
<td>3.9</td>
<td>Reliability of Concurrency Control Mechanisms</td>
</tr>
<tr>
<td>3.10</td>
<td>Summary</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>SURVEY AND COMPARISON OF CONCURRENCY CONTROL MECHANISMS IN DISTRIBUTED DATABASES</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>Locking-Based Solutions</td>
</tr>
<tr>
<td>4.3</td>
<td>Rejection-Based Solutions</td>
</tr>
<tr>
<td>4.4</td>
<td>Timestamp-Based Solutions</td>
</tr>
<tr>
<td>4.5</td>
<td>Motivations for Designing a New Solution</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
</tr>
<tr>
<td>5</td>
<td>THE CONCURRENCY CONTROL MECHANISM AND CRASH RECOVERY FOR THE FULLY DUPLICATED DLDBS</td>
</tr>
<tr>
<td>5.1</td>
<td>Background</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of the Algorithm</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Preliminary Definitions</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Overview</td>
</tr>
<tr>
<td>5.3</td>
<td>The Algorithm (Normal Case)</td>
</tr>
<tr>
<td>5.4</td>
<td>Robustness of the Algorithm</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Requirements of Robustness</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Communication Link Failures</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Node crashes</td>
</tr>
<tr>
<td>5.4.3.1</td>
<td>Detection of Crashing</td>
</tr>
<tr>
<td>5.4.3.2</td>
<td>Reliable Broadcast</td>
</tr>
<tr>
<td>5.4.3.3</td>
<td>Recovery Algorithm</td>
</tr>
<tr>
<td>5.5</td>
<td>Correctness</td>
</tr>
</tbody>
</table>
Chapter 5
5.5.1 Correctness of Robustness .......... 146
5.5.2 Correctness of the Algorithm ...... 150
5.6 Performance of the Algorithm ........ 152
5.7 Summary .................................. 156

Chapter 6
THE CONCURRENCY CONTROL MECHANISM FOR
THE PARTIALLY DUPLICATED DLDBS .......... 158
6.1 Introduction ......................... 158
6.2 Overview of the Algorithm ............ 161
   6.2.1 Preliminary Definitions .......... 161
   6.2.2 Overview ......................... 165
6.3 The Algorithm (Normal Case) .......... 175
   Algorithm A .......................... 175
   Algorithm B .......................... 180
   Algorithm C .......................... 187
6.4 Correctness of the Algorithm .......... 191
6.5 Comparison with SDD-1's Mechanism ..... 192
6.6 Summary ............................... 199

Chapter 7
RELIABILITY ISSUES IN THE PARTIALLY
DUPLICATED DLDBS .......................... 200
7.1 Introduction ......................... 200
7.2 Communication Link Failures .......... 202
7.3 Node Crashes .......................... 203
   7.3.1 Reliable Broadcast ............... 203
7.3.2 Reliable Update ......................... 205
7.3.3 Withdrawal of a Crashed Loop Node .. 212
7.3.4 Reinstatement of a Repaired Node ... 214
7.4 Reliability of Query Processing .......... 215
7.5 Reliability of the Concurrency Control Mechanism .......................... 216
7.6 Correctness of the Reliability Mechanism in DLDBS .......................... 217
7.7 Summary ......................................... 222

Chapter 8 DATABASE DISTRIBUTION AND DISTRIBUTED DATABASE ARCHITECTURE .......................... 224
8.1 Introduction .................................... 224
8.2 Design Consideration of Distributed Database Architecture .................. 225
  8.2.1 ANSI/X3/SPARC Database Architecture ........................................ 227
  8.2.2 Alternatives of Distributed Database Architecture ......................... 230
8.3 Selection of a Database Architecture for DLDBS ............................. 239
8.4 Database Distribution in DLDBS .................................................. 244
  8.4.1 Basic Components of Database Distribution .................................. 245
  8.4.2 Database Distribution Operators .......................................... 246
8.5 Summary ........................................... 249

Chapter 9 DATABASE DIRECTORY MANAGEMENT ............................. 250
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>The Conflict Relationship Between</td>
<td>164</td>
</tr>
<tr>
<td></td>
<td>Two Transaction Operations</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Directory Management</td>
<td>263</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>System Architecture of DDLCN</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Prototype of DDLCN</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Organization of the Dissertation</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>Network Architecture of DLDBS</td>
<td>36</td>
</tr>
<tr>
<td>2.2</td>
<td>Four Level Database Architecture for Centralized Database System</td>
<td>46</td>
</tr>
<tr>
<td>2.3</td>
<td>Distributed Database Architecture for DLDBS</td>
<td>50</td>
</tr>
<tr>
<td>2.4</td>
<td>System Configuration of DLDBS</td>
<td>52</td>
</tr>
<tr>
<td>2.5</td>
<td>System Configuration of a Loop Node</td>
<td>54</td>
</tr>
<tr>
<td>2.6</td>
<td>Local DBMS of a Loop Node</td>
<td>56</td>
</tr>
<tr>
<td>2.7</td>
<td>Software Components for LRN</td>
<td>58</td>
</tr>
<tr>
<td>2.8</td>
<td>IDCS for LRN</td>
<td>60</td>
</tr>
<tr>
<td>2.9</td>
<td>Software Components for LDN</td>
<td>63</td>
</tr>
<tr>
<td>2.10</td>
<td>IDCS for LDN</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Update Anomaly</td>
<td>104</td>
</tr>
<tr>
<td>5.1</td>
<td>Algorithm Protocols</td>
<td>131</td>
</tr>
<tr>
<td>5.2</td>
<td>7 Cases When a Crash Happens</td>
<td>148</td>
</tr>
<tr>
<td>6.1</td>
<td>Algorithm Protocols</td>
<td>167</td>
</tr>
</tbody>
</table>
6.2.a Execution According to Timestamp Order ............ 169
6.2.b Execution After Switching the Order Between Read and Write Operations ............ 173
6.3.a Transmitting Phase ..................................... 178
6.3.b Execution According to Timestamp Order ............ 179
6.3.c Execution After Optimizing Phase ....................... 186

7.1 Algorithm Protocols ........................................ 204
7.2 15 Cases When a Crash Happens ............................ 219

8.1 Database Architecture - a Reference Structure ............ 228
8.2 Distribution at External View Level ....................... 231
8.3 Distribution at Conceptual View Level .................... 234
8.4 Distribution at Internal View Level ....................... 236
8.5.a Distribution at File View Level ......................... 237
8.5.b Distributed Database System Based on Distributed File System ............... 238
8.6 Horizontal Partition Followed by Vertical Partition ........ 248

9.1 DLDBS Directory Distribution ............................. 258

10.1 Sample Database ............................................. 270
10.2 Database Partitions ........................................... 278
CHAPTER 1

INTRODUCTION

With recent advances in communication and mini/micro technologies and the decline in cost of processing units and memory hardware, the development of computer networking is growing at a rapid rate within industrial, governmental, and university communities. A major motivation for the development of computer networking in various application areas is to achieve resource sharing - sharing of computer resources such as hardware, software, data, etc. The basic form of resource sharing achieved in most network systems today can be called explicit resource sharing. This means that the computer network simply provides services for accessing remote resources. The user views the network as a collection of computing systems with various resources and capabilities. From many resources (hardware and software) available in the network, the user must explicitly choose
the resources which are used to run his job.

The well-known ARPANET is representative of a computer network which provides explicit resource sharing. Users view the network as a collection of independent hosts with different resources and capabilities. In this environment, users have to explicitly manage the resources. If a user's program needs a resource such as a file, it is typically necessary to have the program and the file located at the same host. In case they are remote from each other, the user may have to make the file explicitly local to the program using perhaps a file transfer protocol. Once the resources become local, the program can start execution. Thus, at the execution time, all necessary resources are local instead of distributed.

As computer networks influence more and more sectors of our society, users have an increasing demand for having an easy and efficient means of using the resources available on the networks. In order to achieve this, the network system should take the responsibility automatically to manage resources for the users. Hence, users need not master several diverse operating systems and protocols to access various resources available on the network. This form of resource sharing achieved in a computer network is called
implicit resource sharing. The word "implicit" implies "system transparency." In contrast to the explicit resource sharing the user of a computer network with implicit resource sharing views the entire network as one large computing system. A computer network system providing both explicit as well as implicit resource sharing can be called a distributed system.

1.1 Distributed Systems

The phrase "distributed systems" has been the theme for many conferences, workshops, and numerous commercial advertisements. In general, what people call "distributed systems" vary in character and scope from each other. It is common for some vendors to consider distributed systems as those systems which involve more than one processor or a single host processor with a collection of remote intelligent terminals (e.g. terminals with some local editing and formatting capability). Those multiple processors are usually linked in a fixed master/slave relationship. In order to make distinctions between these types of "distributed systems" and the distributed system we deal with in this dissertation, a more specific definition of this term has to be given. Enslow [ENS78] has defined
that a "distributed data processing system" must encompass the following five points:

(1) a **multiplicity** of general-purpose resource components, including both physical and logical resources, that can be assigned to specific tasks on a dynamic basis;

(2) a **physical distribution** of these physical and logical components of the system interacting through a communication network;

(3) a **high-level operating system** that unifies and integrates the control of the distributed components. Individual processors each have their own local operating system, and these may be unique;

(4) **system transparency** that permits services to be requested by name only; The server does not have to be identified;

(5) **cooperative autonomy** that characterizes the operation and interaction of both physical and logical resources.

From Enslow's viewpoint, three key dimensions can be used to describe most computing systems. One dimension is the decentralization in hardware organization, ranging from a single central processor system to a multiple processor
and multiple computer system (computer network). The second dimension is the decentralization in control point organization, ranging from central control through a fixed master/slave relationship, up to a multiple control fully cooperating system. The third dimension depicts the decentralization of database organization, ranging from a centralized database through a fully duplicated database and continuing into a partially duplicated database. Given these dimensions (hardware, control, database), we can categorize those systems which are qualified to be distributed computing systems [ENS78].

The pioneer work in the design of distributed systems can be found in some local networks such as the Distributed Computer System (DCS) [FAR72], the ETHERNET [MET76], and the Distributed Double-Loop Computer Network (DDLCN) [LIU79], as well as in long-haul network systems such as the ARPANET with its RSEXEC [TH073] and its NSW [CRO75] systems. In these systems, users view a network as a unified computing system with a great number of available resources. Users need not be aware of the actual system's organization, nor should they have to distinguish between local and remote resources.
1.2 Distributed Double-Loop Computer Network

Conceived as a means of investigating fundamental problems in distributed networking and computing, the Distributed Double-Loop Computer Network (DDLCN) is designed as a fault-tolerant distributed processing system that interconnects midi, mini, and micro computers using a double-loop structure [LIU79]. The network is designed in such a manner that its users will see only a single, integrated computing facility with great power and many available resources without being aware of the system's actual organization and method of operation. DDLCN is the successor to our previous single-loop network, called DLCN (the Distributed Loop Computer Network) [LIU78]. Like DLCN, DDLCN encompasses the five characteristics of a "distributed data processing system" as defined by Enslow.

The design and development of distributed system hardware as well as software for DDLCN can be viewed as a hierarchical structure shown in Figure 1.1. In this section, we will make a brief review of research concerning DDLCN at each layer. Several new features and innovative ideas have been integrated into the hardware, communications, systems and applications of DDLCN so that it can realize its potential of becoming a powerful unified
Figure 1.1 System Architecture of DDLCN
distributed processing system.

At the bottom, there is a hardware layer. This layer includes computing and communication hardware, interfaces, and message communication protocols.

Computing and Communication Hardware. A seven-node prototype of DDLCN, interconnecting six LSI-11 microcomputers and one DECsystem-20 computer system, is currently being implemented under an NSF grant (see Figure 1.2). The communication network of DDLCN is a distributed control, fault-tolerant, double-loop network. The message transmission mechanism in DDLCN uses a shift-register insertion scheme for the transmission of multiple variable-length messages [REA76]. It combines the best features of the other two widely known mechanisms for distributed-control loop networks: the Newhall transmission mechanism (for transmission of a single message of variable length), and the Pierce transmission mechanism (multiple messages of fixed length).

Interfaces. The interface design of DDLCN is unique in that it incorporates tri-state control logic, thereby enabling the network to become fault tolerant in instances of link failure by dynamically reconfiguring the logical direction
**LSI-11/23:**

- 128 K Bytes MOS RAM
- Dual Floppy Disk
- VT-100 Terminal

**LIU (LOOP INTERFACE UNIT):**

- 16-bit bit-sliced microprogrammable microprocessors (AM2900 based)

**DECsystem-2020:**

- 512 K Words MOS RAM
- 2 Disk Drives
- Magnetic Tape Drives
- I/O Devices & Terminals

*Figure 1.2 Prototype of DDL&CN*
of message flow [WOL78]. The addition of the tri-state logic to the interface is in the form of a hardware component which is controlled by the microprogrammable microprocessor (AM2900) within the interface.

Message Communication Protocols. The message communication protocol to be used for DDLCN is called the Distributed Loop Message Communication Protocol (DLMCP) [REA76]. This protocol is similar in external format to most of the modern bit-oriented protocols such as SDLC (Synchronous Data Link Control).

At the communication layer, there are some interprocess communication (IPC) protocols which allow communication among remote processes. The IPC protocols in DDLCN are unique in that they can support message exchanges among \( n \) processes. Hence, they are not constrained to handle only 2-process communication as the conventional IPC protocols do, but rather \( n \) remote processes that communicate and cooperate concurrently during execution. We call this type of IPC protocol the multi-destination protocol (or the \( n \)-process protocol). Three classes of \( n \)-process protocols have been proposed in [PAR79]: unreliable \( N \)-process protocols, reliable best-effort-to-deliver protocols, and reliable guarantee-to-deliver protocols, each of which has a
different degree of reliability.

At the system software layer, we have a Distributed System Programming Language which has special features useful in a distributed environment. This language is a tool for the implementation of all kinds of distributed processing algorithms (DPAs). Examples of DPAs are algorithms developed to solve synchronization problems in the distributed database system such as updating multiple copies of data. A distributed synchronization model at the same level has a similar function as in a centralized system. However, it is far more difficult to build a generalized synchronization model in a distributed environment than in a centralized system. Some distributed synchronization models are currently under consideration, but the generality of these models still remains to be seen, if we really want to use them in the design of a distributed operating system (DOS).

At the application layer, there are distributed processing algorithms which implement the functions of a distributed operating system (DOS), a distributed programming system (DPS), and a distributed database system (DDBS).
1.3 Distributed Database Systems

Database management concepts and technology have passed out of their infancy, but they have not yet reached full technological maturity. Data management technology is growing rapidly, acquiring strength and provoking many questions regarding its place in the future. Historically, information management technology and organization have tended toward the centralization of data processing and storage facilities. Motivations for such centralization of data processing functions have focused on the issues of security control, data integrity, and standard management. Also, before today's dramatic decline in hardware prices, the cost of owning the necessary local processing facilities exceeded the financial capability of small cooperations. Therefore, a large centralized database system provided affordable data management services to the users in that environment.

Today, business organizations and databases have grown in size and complexity; the users have become more sophisticated in their information needs, and the geographic locations of the origin and use of data have become increasingly dispersed. As a result, the concept of centralization of a huge amount of data not only overloads
the single processor and degrades the performance, but also is not cost-effective. Further, the low cost of communication processing and storage facilities has encouraged the distribution of data processing and the design of distributed systems. Many issues on the design of computing systems have been changed due to the development of national and local networks. In this, the design of database systems is no exception.

What is a distributed database system? Our definition is the following: "A distributed database system is one in which access to the database is satisfied independently of the physical location of the database and of the database management system (DBMS). The distributed database system provides users a logically integrated access to the data while the physical partitioning of a database over different possible computing facilities is transparent to users." In short, the distributed database system is a logically integrated and physically distributed database system.

Currently the pressure to move toward distributed database systems is not yet high, because many companies are just beginning to understand the opportunities and organizational implications of distributed processing. The potential advantages of distributed database systems over
physically centralized database systems have been frequently described in the literature [ROT77, PAR79]. They are summarized as follows:

(1) **Accessibility and reliability.** Accessibility of information can be increased by replicating it at several nodes. The overall reliability of the system can be enhanced by providing redundancy of critical data in instances of communication link failures or node crashes.

(2) **Fast response time.** By proper allocation of data segments so that they are stored close to where they are most frequently accessed fast response time can be attained [MEN78].

(3) **Modular upward scaling.** As databases increase in size and usage, it is possible to increase a system's capability by adding new database nodes without disrupting existing operations. The upgrading of centralized systems is difficult to achieve without major service disruption.

(4) **Lower cost.** It is cheaper to expand a small minicomputer/microprocessor-based database than a large centralized database. By storing data where it originates and is used, communication costs are lower—an important consideration because memory prices are dropping faster than communication
The distributed database systems encompass many promising concepts to meet the needs of data management in the future. Some research problems have been attacked but many still need to be solved. The current research in distributed databases by our DDLCN group will investigate these major new problems.

1.4 Objectives of the Dissertation

The Distributed Double-Loop Computer Network (DDLCN) is envisioned as a powerful, unified distributed processing system. One of the major distributed services supported by DDLCN is the distributed database service. The objective of the dissertation is to design a distributed database system for a distributed system in general and for DDLCN in particular. We call such a distributed database system the Distributed Loop Database System (DLDBS). DLDBS is intended to support a database distributed across DDLCN over seven nodes. Functionally, DLDBS provides the same capabilities that one expects of any centralized database system, and users interact with it precisely as if it were centralized.
Network services dealing with data are not necessarily distributed. This means that sole incorporation of databases into computer networks does not make them distributed databases. Thus, the design of a Distributed Loop Database System should have the following properties:

1. The DLDBS is a logically integrated and physically distributed database system; users are unaware of data location.
2. Actual data in DLDBS are partially duplicated among nodes.
3. The distributed services provided by DLDBS are reliable.

The first property is our definition for distributed databases as well as the major component of Enslow's definition for distributed systems. It's inherent to all distributed services in distributed processing systems. Users should not need to know where data are located. In fact, the system can redistribute data without users ever knowing about it. The second property imposes far reaching requirements. Actually, many of the promises (e.g., increasing reliability and accessibility improving responsiveness and throughput) in distributed databases can only be attained by duplicating data. The third property is one of the major advantages of a distributed database over a
centralized one. Some databases contain such a valuable resource that some users cannot afford to lose the service.

DLDBS supports a relational model [COD70]. Users interact with DLDBS in a high-level data language called the Distributed Data Language (DDL). DDL is a relational algebra oriented data language. It can be used as a tool for application users to do manipulations on a database. It can also allow a qualified user (or enterprise administrator) to define, partition, and destroy a database relation. DDL consists of several commands, each of which is considered a user transaction. A user transaction is an atomic unit of related requests from users to DLDBS.

A traditional single-node database environment consists of database data, a database directory/dictionary (schema), and a DBMS [BRA76, TSI78]. Placing these components at the nodes in a network environment produces a distributed database environment. Figure 1.3 shows several technical problems and design decisions considered in the distributed database environment. Each of these problems occurs due to the transition of a database system from a centralized database environment into a distributed database environment.
Figure 1.3 Organization of the Dissertation
The major problem in designing DLDBS is to design the Distributed Loop DBMS. Since we want to avoid major surgeries on existing database management systems and to explore only those problems dealing with a distributed environment, our approach will disassociate as much as possible the problems occurring only in distributed databases from the problems occurring in conventional centralized databases. As a result, the design of the Distributed Loop DBMS turns out to be designing a network-wise component called the Inter-Database Control Software (IDCS). IDCS consists of a group of distributed processing algorithms (DPAs) to solve the problems of distributed concurrency control, distributed crash recovery, and distributed query processing. These distributed processing algorithms often involve the cooperation of many remote processes at different nodes. The notion of DPAs is close to that of a "distributed computation" used by Kahn [KAH77]. In short, it captures the idea of the execution of an algorithm involving the interaction of two or more remote processes.

There are three other design decisions that must be made in designing a DLDBS. The first one involves how we shall distribute the database. The second is what kind of distributed database architecture we should have by
incorporating database distribution. The third is how we should manage the database directory in a distributed environment. On the following pages, we will briefly describe each of the problems attacked in this dissertation. Note that none of these problems can be treated independently; they are all closely interrelated and must be considered as a whole.

(1) **Distributed Concurrency Control.** In a distributed database with redundant copies of data, it is necessary to ensure that an update always "sees" a consistent version of data. On the one hand, copies of data should indeed have identical values after the update. (Of course, at a given point in time during the update, copies may have different values; however, if no other updates are received after that update, the copies should eventually converge to identical values.) This type of consistency is usually called "mutual consistency." On the other hand, every copy should perform the updates in the order equivalent to a serial scheduling. This corresponds to the so-called "internal consistency." The distributed concurrency control is a system mechanism that maintains both types of consistency.
(2) **Reliability.** Since a major motivation for distributing data is to increase its availability, it is important to design system mechanisms that can cope with different types of failures. If a host fails at a critical time, data can be left in inconsistent states control information can be lost, transactions may not finish completely, etc. These problems arise in the long-haul as well as in the local network architectures. On the other hand, communication system failures (e.g., message loss, network partition) can also be a source of serious problems.

(3) **Distributed Query Processing.** If a distributed database is a read-only database, it is advantageous to duplicate the entire data at many (perhaps all) hosts. However, total duplication may not be possible if storage costs are too high. As a result, the database may only be partially duplicated so that some queries may have to collect data from different nodes. Distributed query processing is a system mechanism that implements a strategy for efficiently retrieving the data of such distributed queries.

(4) **Database Distribution and Distributed Database Architecture.** According to the ANSI/X3/SPARC report [TSI78], the database representing the application view of
the information is called the external database; the database representing the enterprise's description of the information is called the conceptual database; the database representing the inside representation of the information is called the internal database. A significant problem in the distributed database environment would be determining at which level (external, conceptual, internal, or even at physical file level) a database should be distributed. Then a distributed database architecture can be determined by incorporating such database distribution. Another related problem of having a methodology to do systematic database decomposition is also important.

(5) Directory Management. The database directory is the repository of information about the database. At the minimum, it contains schemata and mapping definitions. In the distributed environment, it also contains a location directory to identify the location of database segments. The directory can be global or local, distributed or centralized, duplicated or non-duplicated. If the contents of the directory are not permanently static, careful synchronization of directory changes is required.
1.5 Related Work and Contributions

During the past several years, many solutions to the distributed concurrency control problem have been proposed in the literature [BER79]. In a locking-based approach [ALS76, ST078], a requested data item has to be locked before an access starts. Once a data item is locked, other transactions which want to access it are deferred until the lock is released. In a rejection-based approach [ELL77, TH079], an update is first propagated to other nodes to decide the acceptability of the update. If a consensus cannot be reached, the update transaction will be rejected; otherwise, the actual update can be carried out.

However, global locking incurs high internode communication overhead, creates a potential bottleneck, has low concurrency, and needs another algorithm to do network deadlock detection and resolution. Use of rejection to resolve transaction conflicts, on the other hand, increases communication traffic and delay if the rejected transaction is resubmitted later for processing. Since asking other nodes for agreement of updating a data item is equivalent to testing whether the data item is locked, the amount of concurrency that can be attained in the rejection-based approach is similar to that of the locking-based approach.
The concurrency control mechanism of SDD-1, which uses the timestamp-based approach [BER80], is one of the best-known solutions. SDD-1 assumes that most of the transactions a system will execute are known a priori at system design time; thus, a static set of transaction classes can be established. Then, it tries to detect the potential concurrency among conflicting transactions by statically analyzing transaction classes at database design time. As a result, it can reduce the amount of synchronization required for executing transactions. However, it cannot fully exploit the potential concurrency among conflicting transactions at run time; sometimes it may degenerate into a global locking case [M0H79] depending on how the timestamp in read condition is chosen (for details, see Section 6.5). SDD-1's mechanism is designed basically for a point-to-point long-haul network; it does not take advantage of the broadcast facility which exists in a local network with broadcast channels such as DDLCN or ETHERNET [MET76]. SDD-1's solution is rather complex; it may not be suitable for a low-cost local-network environment.

Our major contribution in this area is that we present a new way of handling the concurrency control problem. The notions of our solutions are quite different from those of
others. Our solutions are simple and efficient, especially for a local network such as the DDLCN, which uses multi-destination protocols. Two new concurrency control mechanisms, one for fully and one for partially duplicated DLDBS, are developed. The two mechanisms have characteristics as follows:

(1) It uses distributed control.
(2) It is robust with respect to both the communication system and database nodes failure.
(3) It does not create a communication traffic bottleneck nor a transaction execution bottleneck in the network.
(4) It does not reject transactions due to transaction conflicts.
(5) It does not do global locking and does not use timestamps to label database data.
(6) It prevents deadlocks so that no separate distributed deadlock detection mechanism is needed.
(7) It is simple and easy to implement.

The concurrency control mechanism designed for a partially duplicated DLDBS is especially interesting. It not only has the characteristics as described above, but also can exploit potential concurrency among conflicting
transactions. By using some local computation, our solution can reduce communication traffic and transaction execution delay due to concurrency control. In the presentation we show that our solution does attain more concurrency than other algorithms which use either a locking-based or a rejection-based approach. Also in many cases, our solution can provide more concurrency and less delay than the SDD-1's solution which is notable for the high concurrency it can attain.

The increment of concurrency in our solution is based on the cost of broadcasting transaction messages. However, broadcasting a message in a DDLCN environment would not pay more cost than sending a single destination message. In fact, our initial motivation for designing a new solution instead of choosing an available one is to find a way which takes advantage of special characteristics of a local network, thus resulting in a simple and efficient solution to handle the concurrency control problem.

As mentioned previously, improved reliability is one of the major advantages of a distributed database over a centralized one. If a single failure at a node or at a communication link could result in the shut-down of the whole network, a distributed system will actually be less
reliable than a single node system. Therefore, in case a failure occurs, important problems to consider are how to maintain the continuing operation of the network and how to do recovery without destroying the consistency of the database.

Little work has been done in this area, and it is insufficient. Some work in distributed databases simply ignores the problem [MIL80]. Some solutions may depend on suspending the whole network until the crashed node is recovered [PAR79]. One technique has been proposed by Alsberg and Day [ALS76] and Menasce, et al. [MEN78]. This technique is presented in the context of centralized locking which does not fit the distributed control environment in DLDBS. The recovery mechanism proposed in [ELL77] is for a distributed environment; however, it only works in the case where the receiving node of a transaction is crashed but the sending node is not.

Our major contribution in this area is that we design a crash recovery mechanism for the fully duplicated as well as partially duplicated distributed database systems. This mechanism uses distributed control, enables the continuing operation of the network in spite of node crashes and communication link failures, and preserves the consistency
of the database. It is robust not only for receiving node failure and for sending node failure but for nested node failures as well.

The literature about designing data languages and distributed query processing mechanisms is abundant [WON77, AST75, ST076]. Our work in this area is that we design a relational algebra oriented language for DLDBS users and employ a distributed query processing mechanism which is suitable for a mini/micro computer environment such as DLDBS. The problems of designing a distributed database architecture by incorporating database distribution and employing a way of maintaining and managing the database directory are design decisions to make. Our contributions in these areas are that we make sound decisions to facilitate the implementation of DLDBS in a mini/micro computer environment, while at the same time we make sure that DLDBS meets the three properties as described in Section 1.4 and serves as a truly distributed database system.

Finally, DLDBS is designed in a structured, modular way so that each component (e.g., distributed concurrency control and distributed crash recovery) can be implemented as an independent software module (certainly, they are
closely inter-related and coordinated). This software engineering approach facilitates the future expansion/shrinkage of network size of DDLCN without causing major surgery on the existing software. The design of a Distributed Loop Database System is intended to be general so that many new concepts can be applied to implement distributed database services in other distributed systems. This dissertation demonstrates that it is feasible to integrate database management, computer networking, and distributed processing technologies into a unified system. The design of DLDBS is by no means complete. We hope this dissertation can serve as a foundation of future research by our DDLCN group.

1.6 Organization of the Dissertation

Figure 1.3 illustrates the organization of the dissertation. In the first half of Chapter 2, we introduce the overall architecture of DLDBS. For the communication subsystem, we suggest a four-layer protocol hierarchy. For the database system, we suggest a four-level distributed database architecture. In the second half of Chapter 2, we present the design of the Distributed Loop DBMS. We first consider a system configuration for the Distributed Loop
DBMS. Then we specifically identify various software components contained in the Distributed Loop DBMS. The inter-relationships, among these components are also presented.

Chapter 3 through Chapter 6 deal with the distributed concurrency control problem, which is one of the difficult technical problems in designing a Distributed Loop DBMS. Chapter 3 clarifies various terminologies and issues which are important to understand the concurrency control problem.

Chapter 4 gives a survey of several proposed concurrency control mechanisms in distributed database systems. This chapter not only presents the fundamental concepts of each solution, but also outlines their advantages and disadvantages and makes comparisons among them. The basic reasons of why they preserve the database consistency are also given. Chapter 3 and Chapter 4 also serve as introductory material for those who are not familiar with concurrency control problems and as background to appreciate our proposed solution.

In Chapter 5 we present the concurrency control mechanism and crash recovery mechanism for the fully duplicated DLDBS. Arguments for the correctness of both
mechanisms are given, and performance analysis of the mechanism is also made.

In Chapter 6 we present the concurrency control mechanism for the partially duplicated DLDBS. The implications and the functioning of the mechanism are illustrated through examples. The arguments for the correctness of the mechanism is given, and a comparison with other solutions, especially with the SDD-1's mechanism, is also described.

In Chapter 7, we present a reliability mechanism to ensure the continuing and correct operation of the partially duplicated DLDBS in the cases of communication link failures and node crashes. We specify the procedures to enforce the reliable broadcast, to enforce atomic operations of a transaction, to withdraw a crashed node, and to reinstate a repaired node, etc. An argument is also given to show this mechanism is robust in the cases of either a sending or a receiving node crash.

Chapter 8 addresses the question of at which level the database is best suited to be distributed. We first describe and compare several alternatives of distributed database architecture by incorporating database
distribution. We decide on one and explain why. At the end we specify the basic component of database distribution and the operators for partitioning the database.

Chapter 9 explains our strategy to maintain and manage the database directory in DLDBS. The reasons for and advantages of doing that are detailed.

In Chapter 10 we present the data manipulation language and data definition language in DLDBS as an interface between DLDBS and its users. Also, a distributed query processing mechanism to process users' requests is considered. Finally, in Chapter 11 some conclusions of this work are drawn and suggestions for future research are given.
2.1 Introduction

The purpose of this chapter is to present an overall organization for Distributed Loop Database System (DLDBS). As has been pointed out in [ROT77], the development of a distributed database system needs the combination of computer networking and database technology. Hence, in the designing of DLDBS, we should first undertake a study of the communication subsystem and communication protocols which support the DLDBS. From Section 2.2.1 to Section 2.2.3 we introduce a network architecture for DLDBS. This topic discusses the hardware configuration, the communication subsystem architecture, and the distributed database architecture.
For the communication subsystem, we suggest a four-layer communication protocols hierarchy. The functions and responsibilities at each layer are identified. For the database system in DLDBS, we suggest a four-level distributed database architecture in which the database is distributed at the conceptual level. The meaning of each database level is also described.

Following the presentation of the DLDBS architecture, in Section 2.3 we design the Distributed Loop DBMS. Section 2.3.1 presents a system configuration for DLDBS. Two virtual nodes, Loop Request Nodes (LRNs) and Loop Data Nodes (LDNs), are identified. Section 2.3.2 specifies the software components and their functional relationship in LRN. Section 2.3.3 specifies the software components in LDN. A collection of these software components forms a network-wise component called Inter-Database Control Software (IDCS). Implementation of an operational DLDBS relies on the existence of solutions to the problems in implementing IDCS.
2.2 Network Architecture of DLDBS

Figure 2.1 illustrates an overall network architecture for DLDBS. Along the vertical axis, the communication subsystem is defined by a four-layer (transmission, transport, session, application) protocol hierarchy. Below the protocol hierarchy is the computing and communication hardware. Along the horizontal axis the database system is defined by a four-layer (application (external), conceptual, internal, file) database architecture. Assume that an end user issues a transaction request to DLDBS. If the transaction can be satisfied locally, it is forwarded directly to the local DBMS. The transaction is translated and executed; the database data is retrieved or updated through the file system. On the other hand, if the transaction involves remote operations, a transaction message is formatted and transmitted intact to the remote nodes with the help of various layers of protocols. In the following three subsections, the hardware configuration, the protocol hierarchy, and the distributed database architecture of DLDBS are discussed.
Figure 2.1 Network Architecture of DLDBS
2.2.1 Hardware Configuration

The DLDBS runs on the network called DDLCN, which interconnects six LSI-11 microcomputers and a DECsystem-20 computer and is currently being implemented at the Ohio State University. The communication link has a double loop structure. The Loop Interface Unit (LIU) for DDLCN incorporates tri-state control logic, thereby enabling the network to become fault-tolerant in instances of link failure [WOL78]. The implementation of LIU uses LSI bit-sliced microprogrammable AM2900 family. The loop communication channel adapter consists of a USRT, an RS-422 line driver, and an RS-422 line receiver. The data communication rate of the communication channel is up to 10 Mbps at 40 feet or up to 100 Kbps at 4000 feet [TSA79].

2.2.2 Communication Subsystem Architecture

The identification of protocol hierarchy and the implementation of protocols at each level is an on-going research project in the DDLCN group. The basic protocol layers for the communication subsystem which supports DLDBS consist of the transmission layer, the transport layer, the session layer, and the application layer. Each layer
derives services from the next lower layer. Each lower layer interprets the commands of a higher layer.

(1) **Transmission Layer.** The transmission layer includes functions pertaining to the transfer of data between geographically distinct locations. The transmission protocol for DDLCN at this layer is the Distributed Loop Message Communication Protocol (DLMCP). The major distinct function of DLMCP is to do addressing and multi-destination routing of data packets. Because the broadcast transmission is imbedded in the hardware configuration (i.e., the loop structure), the multi-destination routing can be implemented efficiently which typically is not true in the ARPANET type network. Several algorithms for doing different types of multi-destination routing have been detailed in [PAR79].

(2) **Transport Layer.** Since no transmission system is totally reliable and some are far less than perfect, the transport protocols are designed to check the integrity of the transmission layer. In a real transmission system, messages may be lost, damaged, duplicated, or delivered out of sequence; the amount of information transmitted by a source might exceed the capacity of the receiver. The transport protocols thus have the duties to do error control (e.g. use a time-out and retransmission mechanism), to do
flow control, to enforce message sequencing, etc.

The protocols for DDLGN at the transport layer are the multi-destination protocols (or n-process protocols). The rationale for having multi-destination protocols is given as follows. In a resource sharing network, such as ARPANET, the users of the protocols are services (i.e., function-oriented protocols) which are typically implemented as a pair of remote processes. For example, a user TELNET process and a remote-server TELNET process will cooperate as long as a (human) user needs access to a remote time-sharing system. This two-process communication protocol is natural in the ARPANET because such a network was not designed to be a distributed system.

In designing a Distributed Loop DBMS, many distributed processing algorithms (DPAs) require the exchange of multi-destination messages in order to exploit the inherent parallelism. Using the two-process communication protocol in DPAs with n processes (n>2) may require to handle n pair-wise independent communications. In addition, for the virtual-circuit type communication using the two-process protocol, a separate connection is created between a sender and a receiver; a separate sequence number and a message buffer are allocated for each pair-wise communication.
Since all destinations are expecting the same message, one sequence number and one copy of that message kept at the sender's node suffices. Also in the transaction-oriented application (such as DLDBS application), many short simple messages are exchanged. A virtual-circuit connection typically is not necessary. Therefore, for achieving parallelism, sending n single destination messages is not appropriate. Three classes of n-process protocols have been proposed in [PAR79]. They are the unreliable N-process protocol, the reliable best-effort-to-deliver protocol, and reliable guarantee-to-deliver protocol, each of which provides a different degree of reliability.

(3) **Session Layer.** A session is a cooperative relationship between application-entities characterizing the communication between them [ZIM80]. The purpose of the session layer is to support the interactions between cooperating application-entities. Services provided by a session layer include the following:

(a) binding two application entities into a relationship and unbinding them,

(b) control of data exchange and synchronization of operations between two application-entities,

(c) allowing a set of messages to be related together,

(d) establishing recovery.
Although some one-shot messages can be sent to a destination without establishing a session, most of the communications in distributed processing algorithms require sending the messages via an established session.

(4) **Application Layer.** Protocols at the application layer directly serve the end user by providing the distributed services appropriate to an application. An application within DDLCN is composed of cooperating application processes that intercommunicate according to application layer protocols. Examples of application processes may be a FORTRAN program accessing a remote database, a process control program executing in a dedicated computer, etc. A distributed processing algorithm is composed of a group of cooperating remote application processes at different nodes. There are a potentially large number of protocols in the application layer to carry out the application oriented communications between application processes. Some examples of these protocols are the file transfer protocol, remote job entry protocol, deadlock detection and recovery protocol, commitment protocol, etc.

Although the design of the distributed database service at the application layer should not relate to the details of the communication subnet, the performance of such a design
will be affected by the types of subnets underneath. For example, a point-to-point daisy-chain oriented concurrency control mechanism in a distributed database system [TH079] cannot take advantage of the broadcast facility of a network which has broadcast channels. On the other hand, a broadcast oriented concurrency control algorithm will not be suitable for a network where broadcasting is an expensive activity.

Generally speaking, there are two types of design for the communication subnet: point-to-point subnet and broadcast subnet. In a point-to-point subnet, the network contains many links, each of which connects a pair of nodes. If two nodes that do not share a link wish to exchange a message, the message must transmit indirectly through other nodes. During its journey, the message is received and stored at each intermediate node before it is forwarded on to the next node. Therefore, a point-to-point subnet is also called a store-and-forward subnet. The well-known ARPANET is a network of this type. In the broadcast subnet, the transmission link is not point-to-point; there is a communication channel shared by all nodes. A message sent by any node is received by all the others. The satellite network, the packet radio network, the bus network, and the loop network are examples of a broadcast network.
From the geographical point of view, networks can be also classified into two types: local networks and long-haul networks. Local networks are distinguished from long-haul networks by the following three characteristics [TAN81]:

1. A diameter of not more than a few kilometers.
2. A data rate exceeding 1 Mbps.
3. Ownership by a single organization.

The geographic characteristics of a local network yield economic and technological considerations that are quite different from those in a long-haul network. The key difference is that a local network can be of low cost and with high bandwidth. For example, a simple twisted-wire pair can support communication in the 1-10 Mbps range over the distance of a kilometer between repeaters. The hardware needed to drive and control the transmission media is cheap. However, for the long-haul network, the wideband common carrier circuit is expensive and bandwidth is not high (e.g., 50 Kbps in ARPANET).

These simple differences create several new opportunities for a local network. First, low communication cost and high bandwidth make the implementation of a truly distributed computing system feasible and cost-effective.
High communication cost has been a major reason why people suspect the feasibility of truly distributed services implemented on a computer network. Another major attraction is that many local networks have a common broadcast channel which makes the implementation of multi-destination protocols cost-effective. In the implementation of multi-destination protocols in a local network, the idea of creating a virtual-circuit connection before sending data packets to avoid putting full source and destination addresses in each data packet becomes unjustified. This is because the high bandwidth in the local network could afford us to include the full address in each data packet just like a datagram. Another reason is that the distributed applications in a network involve many transaction-oriented short messages employing no more than one packetful of information in each communication. For such an exchange it becomes doubtful whether the overhead in setting up the virtual connection is worthy. Examples of such distributed applications are home banking, cash dispenser, airline reservations, etc. Transaction messages in these applications will contribute a big part to future network's traffic.
DDL CN is designed as a local network with broadcast channels. The implementation of multi-destination protocols is one of the major on-going research projects [PAR79]. The decision of designing DLDBS as a truly unified distributed service is justified by the fact that multi-destination protocols can be efficiently developed in DDL CN. The design of new distributed concurrency control mechanisms is another example of our taking advantage of multi-destination protocols. The concurrency control mechanism in DLDBS is efficient and provides good concurrency among transactions execution. The increment of concurrency is based on the cost for broadcasting transaction messages. However, the cost of broadcasting a message in DDL CN would not be different from that of sending a single destination message.

2.2.3 Distributed Database Architecture

The study group on database management systems of the Standard Planning and Requirements Committee (SPARC) of the American National Standards Committee on Computers and Information Processing (ANSI/X3) has proposed a framework for the potential standardization in the area of database management systems [TSI78]. A simplified schematic view of the ANSI/X3/SPARC architecture is depicted in Figure 2.2,
Figure 2.2  Four Level Database Architecture for Centralized Database System
which identifies a three-level view of the database.

(1) **External View Level.** An individual user (an application programmer or an online user) will generally be interested in only some portion of the total database. In each general application area, an application administrator provides multiple external schemata which define the application view of the database. The application administrators determine the objects of interest for each specific class of applications, and they introduce various data models through which these objects are presented. Since the database structure at this level is modelled from the viewpoint of the user, this level is also called the "user view level" or the "application view level."

(2) **Conceptual View Level.** The conceptual view is a view of the entire information content of the database. It represents the enterprise's description of the information as modelled in the database. The enterprise administrator defines the conceptual schema which contains the definitions of entities in the conceptual database and their properties and relationships. The conceptual schema may include additional features, such as access control and validation procedures. No entities or properties may be referenced in the database unless they are defined in this schema. The
conceptual view is also called the "enterprise database view."

(3) **Internal View Level.** The internal view is an inside (but not physical) representation of the entire database. It describes the storage structure of the database and specifies efficient access paths to the internal records in the database. The database administrator is responsible for defining the internal schema which defines the internal view of the database. The internal schema relates to the performance strategies employed by the database management system (but not performance strategies employed by storage media and devices). It specifies whether the data is stored in a flat, an inverted, a network, or other form.

Below the ANSI/X3/SPARC three-level database architecture, a file level may exist actually to access the data. The file level may be supported by a conventional file system (or needing some modifications) in an operating system. Thus we add another level in the database architecture.

(4) **File View Level.** The internal view of the database is an abstraction from physical files managed by the file system. The physical file deals in terms of hardware
constructs such as tracks, cylinders, and the like. At this level, the database is viewed as a set of physical files stored in the storage media using different file organizations and accessed using access methods in the file system.

The four-level database architecture provides a model for centralized database systems. A natural question to a distributed database designer is how to distribute the four-level architecture in a distributed system environment and produce a distributed database architecture. The distributed database architecture selected for DLDBS is depicted in Figure 2.3. In such an architecture, a database in DLDBS is distributed at the conceptual view level. Some reasons for doing this are its naturality, good portability, good adaptability, and autonomous management. Discussions of other alternatives of distributed database architecture, the rationale of choosing the one above, the basic components of database distribution, and the operators for partitioning the database are detailed in Chapter 8.
Figure 2.3 Distributed Database Architecture for DLDBS

LEV: Local External View
GCV: Global Conceptual View
LCV: Local Conceptual View
LIV: Local Internal View
LFV: Local File View
2.3 **Software Components in**

**Distributed Loop Database Management System**

2.3.1 **System Configuration of DLDBS**

In the Distributed Loop Database System there are two types of virtual nodes (see Figure 2.4). First, there are Loop Request Nodes (LRNs), which accept the user's request, parse and translate user queries, create transaction messages, and supervise the distributed query processing of transactions. Second, there are Loop Data Nodes (LDNs) which contain the data, the schema, and the DBMS needed to satisfy the user's requests. In Figure 2.4, the **User Processes** (UP) may be either processes representing some remote online users or processes on behalf of application programs. The **data** is a collection of files to represent the contents of the database. The **Database Management System** (DBMS) is a collection of programs which control the operation and access to the data. The **Inter-Database Control Software** (IDCS) is a network-wise component which consists of a group of distributed processing algorithms (DPAs) to solve the problems of distributed concurrency control, distributed crash recovery, distributed query processing etc.
Figure 2.4  System Configuration of DLDBS
A physical node in the network is called a Loop Node (LN). A Loop Node can serve as an LRN, or an LDN, or both. In DDL CN, the DEC-20 computer and LSI-11/23 microcomputers equipped with enough disk storage will be used as both LRNs and LDNs. Figure 2.5 depicts a system configuration of a Loop Node which serves as both an LDN and an LRN. The job and process management component is a component in a conventional operating system to do job scheduling, process scheduling, resource management, etc. The file system is a component to handle local storage management. The communication processor is responsible for the transmission of messages. It handles message management, message routing, message buffering, and inter-process communications via different levels of protocols as described in Section 2.2.2.

As we mentioned earlier, a single-node database system consists of three major parts: the database data, the database directory/dictionary, and the database management system. The database data are the contents of the database. The database directory contains statistics information, transaction definition, cataloged queries, and schemata which describe the database. The DBMS is the most important part which consists of various programs and processes to control the access to and operation on the database.
Figure 2.5  System Configuration of a Loop Node
Before we identify the software components in DLDBS, it seems necessary to first identify the basic system functions of a local DBMS at a Loop Node. Figure 2.6 illustrates the functional relationship among several software processors to perform database definition, data manipulation, backup, and recovery functions in a DBMS.

The DBMS provides users with a data definition language based upon the relational model. The data definition processor interprets user specifications and stores them in a database directory. In DLDBS, the user specifications include relation name, attribute name, size, storage strategies, etc. Database definition, the required first step in any database activity, affects the performance of all subsequent functions. Database initialization processor prepares the empty database files for accommodating relation tuples. This step is necessary before an actual tuple is added to the database.

The DBMS also provides users with a data manipulation language based upon the relational model. It allows users to retrieve, insert, delete, and update database data. Data definition and manipulation languages are the subjects discussed in Chapter 10. The transaction processor parses and translates a user's requests into transaction internal
Figure 2.6  Local DBMS of a Loop Node
form, which is performed by a collection of library routines, referred to collectively as database manipulation processor, to actually retrieve and change the data.

The backup and recovery processor consists of a set of system utilities to ensure database integrity and to do crash recovery. It produces a dump file and journaling file (system log or audit trail) to do recovery. It produces a checkpoint to do restart. It also collects and prints statistics and performs other functions.

2.3.2 Software Components of Loop Request Nodes

As described in the previous section, the Distributed Loop Database System consists of two types of nodes: the Loop Request Nodes and the Loop Data Nodes. Figure 2.7 shows the software components of the Loop Request Nodes. User Processes (UP) may be either processes representing some remote online users or processes on behalf of application programs. In a typical scenario, a user requests service of DLDBS by means of a relational algebra source transaction. The user process receives the request, does some preprocessing (e.g., syntax checking), and passes it to the transaction processor. A transaction typically
Figure 2.7 Software Components for LRN
consists of three types of operations: Read operations (query operations), Compute operations, and Write operations (value assignment operations). Read operations query the database to obtain values of data items, called Read Values. By using Read Values obtained in Read operations, the transaction performs a computation to generate new values called Write Values. Then Write operations record Write Values on the permanent database. In case the request involves a query, the transaction processor parses the query and produces a tree representation. If the optimizing scheme in [SMI75] is implemented here, a tree transformer can do some algebraic transformation to increase the query execution efficiency. Then the transaction is passed to the Inter-Database Control Software (IDCS).

Figure 2.8 depicts the software components of Inter-Database Control Software for the Loop Request Node. The driver accepts the user transaction and invokes the network concurrency control processor. This software component implements a distributed processing algorithm (DPA) to maintain database consistency among the LDNs (this DPA is detailed in Chapter 5 and Chapter 6). Since a transaction may contain a query, the driver calls for assistance of the network query processor. The tree representation of query is interpreted by the network query
Transactions/Data from/to Remote Processes

Figure 2.8 IDCS for LRN
processor and is decomposed into several distributed simple queries, each of which corresponds to a Read operation and can be processed at a single LDN. The function performed by the network query processor is to employ a strategy efficiently to process these distributed simple queries whose answer set spans more than one LDN. Note that this software component also implements a DPA (which is described in Chapter 10). After the query is processed by LDN, the final Read Values are sent to the LRN. If the transaction is a pure query transaction, the Read Values are final responses to the users. If the transaction is an update transaction, local computations are performed using the Read Values to produce Write Values. Note that the update transactions are the major concern of the network concurrency control processor. Once the update transaction is found to be safe for processing (integrity checking), the Write Values are destined to the LDNs which contain the write data items. In order to ensure reliable updates, the network concurrency control processor calls for assistance from the network crash recovery processor working together to ensure that the transaction can be executed as an atomic unit, should crashes occur. The network crash recovery processor is also an example of a DPA implemented in DLDBS (this DPA is described in Chapter 5 and Chapter 7).
Since a user transaction may or may not be satisfied locally, any remote access always needs to refer to a component called the **location directory** which contains information indicating the nodes where the various units of data reside in the distributed environment. Finally, during the progress of the transaction activity, all these software components send performance information to the **statistical collector** for gathering transaction statistics (e.g., delay time, locality interference, etc.).

### 2.3.3 Software Components of Loop Data Nodes

**DLDBS** is intended to support the database distributed across DDLCN over Loop Data Nodes. In our distributed database architecture, the schema and the control programs (DBMS) are stored at the same node as the data they deal with. Figure 2.9 presents the software components of a Loop Data Node. The communication processor is responsible for transmitting messages to and receiving messages from the DLDBS software on the other nodes. The software processors in the DBMS shown in Figure 2.9 are the same as those of the DBMS described in Section 2.3.1. They are the "expects" on the local database. The transaction processor translates transaction operations (subtransactions). The database
Figure 2.9  Software Components for LDN
manipulation processor makes mapping between schemata, invokes system library functions, and constructs execution environments. It performs some special functions not provided by operating systems such as granularity locking and uses the file system to perform physical I/O. The Int-r-Database Control Software for LDN is depicted in Figure 2.10. The functions of its software components are similar to those described in Section 2.3.2. Hence, we do not repeat them again.

2.4 Summary

This chapter presents an overall organization of the Distributed Loop Database System. In the first half of the chapter, we introduce a network architecture for DLDBS. In the communication subsystem, we suggest a four-layer communication protocol hierarchy. They are the transmission layer, the transport layer, the session layer, and the application layer. The functions and responsibilities at each layer are identified. In the database system we suggest a four-level distributed database architecture for DLDBS in which the database is distributed at the conceptual level. The meaning of each database level is also described.
Transactions/Data from/to Remote Processes

Driver

Network Concurrency Control Processor

Network Crash Recovery Processor

Statistical Collector

Local Update to Local DBMS

Figure 2.10 IDCS for LDN
In the second half of the chapter we design the Distributed Loop DBMS. The system configuration of the DLDBS is first given. The DLDBS consists of a collection of virtual nodes: Loop Request Nodes and Loop Data Nodes. The basic system functions of a local DBMS at a node are discussed. Then we concentrate on designing a network-wise component called Inter-Database Control Software. The functional structure of an LRN and an LDN are presented. The software components (each of which implements a DPA) of an IDCS and the inter-relationship among these components are also described.
3.1 Introduction

In the previous chapter, three software components of the Inter-Database Control Software in the Distributed Loop DBMS were identified. Implementation of an operational DLDBS relies on the existence of solutions to implement these components. From Chapter 3 to Chapter 6 we will concentrate on designing the distributed concurrency control mechanism for DLDBS. This mechanism has many interesting characteristics as described in Section 5.1 and Section 6.1. Its simplicity makes it very suitable to be implemented in a low cost mini/micro computer based local network.

In this chapter we present the working definitions of some frequently used terminologies and give a brief description of several concepts that are important to the
understanding of the following chapters. The organization of the chapter starts by explaining the concept of database consistency. Then, in Section 3.3 we define the meaning of a transaction. In Section 3.4 and Section 3.5 we address the importance of concurrent execution of transactions and explain the necessity of concurrency control. In Section 3.6 we address the concept of serializability as a correctness criterion of a concurrency control mechanism. In Section 3.7 we describe the most popular concurrency control mechanism in a centralized DBMS. The deadlock problem and the recovery problem are also considered. In Section 3.8 we discuss the concurrency control problem in a distributed system environment. Several issues whose implications in the distributed environment are different from the centralized environment are discussed. Finally Section 3.9 considers the reliability of a concurrency control mechanism. The importance and implication of reliability are also given.

3.2 Database Consistency

A database is an information model which reflects a selected part of the real world. Data items in the database relate to each other in certain ways. Examples of such
relationships are as follows: in a company database, the salary of a department manager is greater than the salary of any other employee in the department, and in an air-line reservation database, the total number of reserved seats is less than the number of seats in an airplane. These relationships among data items are called integrity constraints (ICs), which are a property of the real world. Data which violate the property are not correct. Consequently, the ICs are concerned with the correctness of the data in the database so that the values of the data items returned to a user from the database are accurate. A database is said to be in a consistent state if it satisfies all its ICs.

In the process of proving the correctness of a program, formal specification of the concept of the program is a necessary step. As a counterpart, the specification of a valid database state in terms of a set of ICs is important so that one can deal with the correctness of the data in a database with respect to these ICs. Much research has been directed to this area; for details see [MCL76, ST075].

After the ICs are specified, the next problem is to determine how to enforce the ICs. Typically in a database management system (DBMS), there is a validation checking
subsystem to monitor IC violations whenever a request arrives. Therefore, data are protected against invalid alteration, and the database can remain correct with respect to the ICs. However, even in a well-controlled system, loss of integrity is still possible due to various errors and failures such as human errors, system errors, hardware failures, etc. Usually, several routines such as a dump routine, journaling routine, backout routine, etc. are required to detect, recover, and maintain the integrity of the database. The way to implement a validation checking mechanism and to preserve the integrity of the database also attracts many researchers, for details see [ESW76a, ST075].

3.3 Transaction

A sequence of related actions on the database which are considered by a user to be indivisible are grouped as a single transaction. For example, a single command in the query language which may be translated into several run-time calls on a database management system can be a transaction. A routine which consists of statements in the programming language intermixed with statements in the query language to access the database system can be a transaction. In system R [AST76], a user can define a transaction using
BEGIN-TRANSACTION and END-TRANSACTION operators.

If a transaction is considered to be valid by the validation checking mechanism in DBMS, it is a unit of consistency; i.e., if it executes alone, it preserves the consistency of the database. For the rest of this dissertation, a transaction is always referred to as a valid transaction. Although the validation checking problem of rejecting an invalid transaction is important, it will not be discussed further.

In order for a transaction to transform a database from one consistent state into another, the database may in many cases become temporarily inconsistent (this is explained in Section 3.5). Consider that if a system crash occurred so that the execution of a transaction cannot be completed, the database would be left in the inconsistent state. Therefore, even though a transaction is defined to preserve consistency of a database, the occurrence of crashes can affect the consistency of the database. Consequently, a transaction is also a unit of recovery. That is, if some actions of a transaction cannot be completed, the entire transaction is undone and the database is rolled back to the previous consistent state.
3.4 **Concurrency**

Concurrency of activities has long been recognized as an important feature in many computer systems. A transaction typically involves some input (Read) or output (Write) actions. In a single processor environment if transactions are restricted to execute only one at a time, the system performance may be degraded because most of the processor time is spent waiting for I/O. Furthermore, a long running transaction may delay a short running transaction for a long time period. Thus, the system response time, which is crucial in a timesharing or a real time application, may not be acceptable. As a result, even in a single processor system, concurrent execution of transactions implemented by using a multiprogramming technique is needed to improve system response time and performance. Two transactions are said to execute **concurrently** if one transaction starts to execute before the other transaction finishes.

Since each of several independent processors has the ability to execute a transaction program in a multiprocessor or a computer network environment, the potential for parallel execution of transactions is obviously increased. Concurrent execution turns out to be an inherent nature of
such a system.

3.5 **Concurrency Control**

Although a valid transaction can individually preserve the consistency of a database, concurrent execution of different transactions must be regulated; otherwise, a user may receive inaccurate information or the consistency of the database may be destroyed. One type of problem arises when a transaction reads inconsistent data produced temporarily by another transaction. To illustrate the issue, imagine a banking system in which a transaction T1 runs to produce a total balance report for a customer TOM, and, at the same time, a second transaction T2 runs to transfer $50 from TOM's savings account to his checking account. Assume that each account balance is $500 before execution; after execution, one would expect the total balance report to be $1000 because neither of the two transactions changed the total balance. Suppose T1 and T2 execute concurrently as follows:

- **T2 reads TOM's savings balance, subtracts $50, and writes a new savings balance back.**
- **T1 reads TOM's savings balance.**
- **T1 reads TOM's checking balance.**
T2 reads TOM's checking balance, adds $50, and writes a new checking balance back. The total balance report which is the sum of the savings and checking balances will display a total $450 + $500 = $950, which is incorrect. After the fifty dollars have been withdrawn from the savings account but before they have been credited into the checking account, the database is temporarily in an inconsistent state. The inaccuracy of the total balance report is produced because T1 observes inconsistent data from the temporarily inconsistent database.

Another type of problem arises when each of two concurrent transactions updates each other's Read data thus producing a cyclic race condition. As an example, consider a database that includes the data items X and Y and the integrity constraints between X and Y are X ≤ Y. Assume X and Y have the initial values 1 and 2 respectively. Suppose that two transactions T1 : X := Y and T2 : Y := X run concurrently. After they are complete, one would expect X and Y to be equal and the database to be consistent. Suppose the actions of T1 and T2 are interleaved as follows:

T1 reads Y
T2 reads X.
T2 writes Y.
T1 writes X.
The values of X and Y would end with 2 and 1 which violates the integrity constraint.

From the information given above, it can be seen that in order to preserve database consistency and to ensure that users receive accurate information, a mechanism to control the way in which the actions of several transactions are interleaved is necessary. Since a transaction is viewed as an atomic unit to a user, whereas an action in a transaction is an atomic unit to the system, the system's atomic actions are at a finer granularity than its user's atomic actions. How the system can support the user's viewpoint so that the behavior of a system conforms to the user's expectations that each transaction is processed as an indivisible computation, is the problem called concurrency control.

3.6 Serializability

In order to deal with the correctness of a concurrency control mechanism, certain correctness criteria are needed. Although preserving database consistency is the purpose of
a concurrency control mechanism, this notion is not enough
to be served as a proof criterion. This is because we have
first to define all permissible states of a database by
defining various integrity constraints among numerous data
items in the database. Using the banking system example as
described previously in order to say that the concurrent
execution of T1 and T2 cannot preserve database
consistency, an integrity constraint is needed: "The total
balance of a customer must be equal to the sum of his
savings account balance and checking account balance."
However, such an integrity constraint is application
dependent and thus is difficult to define for a general
database system. In addition, a complete set of integrity
constraints about a database would be as large as the
database itself. Explicitly enumerating all such
constraints for checking consistency would not be
practical.

The correctness criterion of concurrency control
universally accepted by almost all the literature is
"serializability." The method of avoiding inconsistency is
by guaranteeing that the execution schedule of concurrent
transactions is serializable. A schedule (or log) of a set
of transactions is a sequence of atomic actions from the
set of transactions. A schedule of a set of transactions
is **serial** if all the atomic actions of each transaction are put contiguously in the schedule. That is, a serial schedule of a set of transactions represents a serial execution of these transactions; each of which runs alone to completion before the next one begins. From Section 3.3, if a user transaction runs alone, it will leave a consistent database state if the database state is initially consistent. Therefore, a serial schedule of a set of transactions will, by induction, result in a consistent database state. This fact is obvious because a serial schedule represents an execution in which no transactions execute concurrently; thus making no occurrence of inconsistency due to concurrency. Two schedules are **equivalent** if in both schedules each transaction produces the same output thereby leading to the same final state of the database. An interleaved execution of a set of transactions is **serializable** (or serial reproducible) if its schedule is equivalent to a serial schedule. This means that for all initial database states, the serializable schedule produces the same output and the same final database states as some non-interleaved serial execution of the same set of transactions. Serializability of a schedule only requires that there exists **some** serial schedule equivalent to it. In fact, a serializable schedule may be equivalent to **several** serial schedules.
To fix our idea let us consider the second example in Section 3.5 in which two transactions T1 and T2 run concurrently. The serial execution of T1 and T2 would be either T1 T2 which results in $X \cdot Y = 2$, or T2 T1 which results in $X = Y = 1$. Neither of them is equivalent to the interleaved schedule which results in $X = 2$ and $Y = 1$. Thus, the interleaved execution of T1 and T2 in Section 3.5 is not serializable.

3.7 Concurrency Control in Centralized DBMS

In the centralized database management system (DBMS), locking of data items is the most popular concurrency control technique to synchronize concurrent transaction activities. Whenever the DBMS executes a Read or a Write action, it has to first set a lock on the requested data items. Any sequence obtained by collating the Read, Write, LOCK and UNLOCK actions of transactions is a schedule for these transactions. A schedule is legal if it only allows lock actions on free data items but never allows a lock action on an already locked data item [ESW76]. A legal schedule may not be a serializable schedule. Eswaran [ESW76] has proposed and proved that if the transactions in a schedule are each well-formed and two-phase, then any
legal schedule is serializable and thus consistent. A transaction is said to be \textit{well-formed} if the transaction always sets a lock on the requested data item before accessing the item. A transaction is said to be \textit{two-phase} if it never sets a new lock on any requested data item after it has released a lock.

One problem the locking based concurrency control mechanism must deal with is \textit{deadlock}. A deadlock is a situation in which two transactions are unknowingly waiting for data items that are locked by each other and thus unavailable. Techniques for handling deadlock are not new; they typically include deadlock prevention or deadlock detection and resolution. Deadlock prevention can be achieved by never executing a transaction until all the requested data items are available or requesting locks on data items in a prespecified order. A simple way to detect and resolve a deadlock may be that a transaction can only be allowed to run for a limited time interval; otherwise, a timeout will occur to deny a possibly indefinite waiting. This method is acceptable for a lightly loaded system, but is not appropriate when a system becomes more congested so that more and more timeouts are generated not for deadlock but for congestion delay. Deadlock can also be detected by maintaining a resource allocation wait-for graph in the
DBMS. Whenever there is a cycle in the wait-for graph, there is a deadlock. This deadlock detection scheme has two drawbacks. First, it involves substantial overhead to perform the detection, and second, it may be expensive to maintain a resource allocation graph. These drawbacks are especially true in a distributed system. A way of resolving the deadlock situation once it is detected would be to abort a victim (e.g., the "youngest" transaction), to back up the victim, to release the locks, and to restart the victim later.

Deadlock is only one of the reasons that a transaction may fail before completion. Other reasons for a transaction failure may be a user's decision to abort the transaction or a system crash such as hardware failures, system errors, etc. A failed transaction may leave a database in an inconsistent state; thus, a system recovery procedure is typically required to bring the database to a new consistent state.

During a transaction backup, it may be necessary to roll back or undo the effects of the failed transaction. However, backing up one transaction may require backing up another. If a transaction T2 reads the effect of an update from a transaction T1, and if T1 is then aborted and has
been rolled back, T2 will have read a value which no longer exists. Thus T2 must also back up. This cascaded back up situation can be avoided if a transaction holds all the locks until the end of the transaction. In fact, nearly all commercial systems require that a transaction hold all its locks until the transaction is completed.

3.8 Concurrency Control in Distributed DBMS

If the distributed database can be updated, which seems very often to be the case, it is necessary to ensure that the distributed database remains consistent. The notion of database consistency in a distributed environment is more complex than in a centralized environment and concurrency control mechanisms designed for a centralized environment cannot be simply extended to a distributed environment. Several issues whose implications are different in the two environments and which are important to our future discussion will be described in this section.

The notion of database consistency in a distributed environment concerns two aspects. The first one, the so-called internal consistency, says that the internal data relationships (integrity constraints) within a database
must not be violated if the transaction working alone on the database can preserve them. Clearly, this is the same concept as the database consistency in the centralized system. Another aspect the so-called mutual consistency which is a unique problem in a duplicated distributed database, says that all copies of data will eventually converge to the same value when updated activities cease. We use the term "eventually converge" because the inherent nature of a distributed system such as the variable communication delay existing in the communication subsystem prevents activities at each node from progressing at the same pace synchronously. Therefore, it is impossible to require two data copies having the same value at any instant in time.

Serializability is again almost universally adopted as a correctness criterion for internal consistency. That is, the concurrency control mechanism must ensure that the effect of a set of concurrent transactions on the global database and their outputs must be the same as the effect and outputs of some serial schedule of the same transactions. We have defined a schedule as a sequence of atomic actions of transactions. Instead of all the atomic actions of transactions being processed at one node, in a distributed database system different subsets of atomic
actions of transactions are processed at different nodes. The sequence of local actions at a node is called a local schedule. Local schedules at different nodes together represent the physical behavior of a distributed system. Local schedules are difficult to deal with since each of them only represents a partial view of the total behavior. In [ROT77a], a global schedule is defined by shuffling a set of local schedules and the global schedule preserves all the information given by the set of local schedules.

In a distributed database, a concurrency control mechanism is said to be correct if:

1) It preserves mutual consistency of the multiple copy data.
2) It preserves internal consistency of the database.
3) It is deadlock free.

However, in order to be a good mechanism, the concurrency control mechanism is typically required to be robust with respect to failures and to have good performance.
3.9 Reliability of Concurrency Control Mechanisms

In a centralized environment, in case a critical failure occurs at the system, we typically cannot do much about it other than to shut it down and fix it. In a distributed environment, if a single computing node has a 10% chance to go down per day, a ten-node computer network will have a failure almost every day. Therefore, failures are an inevitable event in a distributed system. If a failure at a node would result in the shut down of the whole network, a distributed system would actually be less reliable than a single node system. However, if the failure of a node does not affect the continuing operation of other active nodes, reliability turns out to be a major promise of a distributed system. This is because the duplicated copies of resources which can exist in a distributed system offer a possibility of continuing the services despite the failure of one copy of the resources.

Since it is not practically required or theoretically possible to achieve 100 percent robustness [BER79], a practical crash recovery mechanism is only required to have the ability to handle failures up to some degree. Actually, the degree of robustness that a mechanism can attain depends on how much we are willing to spend. A concurrency control
mechanism is said to be robust if the recovery mechanism can satisfy the degree of robustness it claims under the assumptions it specifies. It is generally believed that a well-defined crash recovery mechanism is more valuable than a mechanism that just vaguely claims to be "very robust."

In a distributed system, failures can come about from two sources. One source is from the communication subsystem of a network; the other is from the processing system of a node. A broken communication link is a kind of communication subsystem failure. Several link failures may result in a partitioned network in which some nodes are entirely isolated from other nodes—a serious and difficult problem. Note that the reliability issues about communication protocols such as handling loss of messages, duplicate of messages, etc. are considered as separate problems at a different layer. Typically, a concurrency control mechanism and its recovery procedures are assumed working on top of a communication subsystem which has some sort of communication protocols. Distributed crash recovery will be the major topic in Chapter 5 and Chapter 7.
3.10 Summary

In this chapter we have introduced many important concepts and terms related to concurrency control. They include database consistency, integrity constraints, transactions, concurrency, concurrency control, serializability, schedules, serial schedules, serializable schedules, schedule equivalence, local schedules, and global schedules. We describe the concurrency control mechanism in a centralized DBMS and present important issues about concurrency control in the distributed environment. Finally, the importance and implications of reliability of a concurrency control mechanism are given.
CHAPTER 4

SURVEY AND COMPARISON OF
CONCURRENCY CONTROL MECHANISMS IN DISTRIBUTED DATABASES

4.1 Introduction

In the previous chapter we have clarified some of the terminology and issues which are important in understanding the concurrency control problem. In this chapter we will review several proposed solutions to concurrency control in the distributed environment, each of which can best represent the approach it employs. Because of the complexity of these solutions, interested readers should consult the original papers for greater understanding.

During the past few years, several solutions to the concurrency control problem in the distributed environment have been proposed in the literature [BER79]. There are some ways to classify these solutions. From the
architectural point of view, several solutions are proposed for a fully duplicated distributed database system (DDBS) [ELL77, TH079] and several are for a partially duplicated DDBS [ST078, BER80]. From the control point of view, some solutions use centralized control [ASL76] and some use distributed control [ELL77, CH080a]. Here, we will classify these solutions into three categories according to the way they handle transaction conflicts: locking-based algorithms [ALS76, ST078], rejection-based algorithms [TH079, ELL77, MUL75], and timestamp-based algorithms [BAD78, BER80]. In the locking-based algorithms, a requested data item has to be locked before an access starts. Once a data item is locked, other transactions that want to access it are deferred until the lock is released. The major disadvantages of a locking-based algorithm are that it lacks reliability, it creates a bottleneck in the network, and it needs a separate algorithm to do network deadlock detection and resolution.

In the rejection-based algorithms where locking is totally avoided, an update transaction is first propagated to every node at which the requested data of the transaction are located to decide the acceptability of the update. If a positive consensus has been reached, an actual update can be carried out. If a consensus cannot be reached, the update
transaction will be rejected and can be submitted again. Since asking other nodes for agreement of updating a data item is equivalent to testing whether the data item is locked, the amount of concurrency that can be attained in this approach is similar to global locking. However, two major advantages can be seen. First, algorithms in this approach typically use distributed control, thus implying bottleneck-free and robust solutions. Second, the deadlock problem is prevented. The reason for the latter is because a concurrent conflicting transaction is rejected instead of waiting. Thus, there is no possibility of a deadlock.

In the timestamp-based algorithms, a transaction is given a globally unique timestamp as its identifier as soon as it is received. The concurrency control mechanism will enforce an execution schedule of conflicting transaction operations equivalent to a global serial schedule of the same transactions based on their timestamps. The relative order of non-conflicting transaction operations in a schedule is not important. Although we cannot determine that all solutions using this approach have significant improvement of concurrency over the locking approach, it shows several of the following important advantages:

(1) Algorithms using this approach typically have distributed control. Thus, they are
bottleneck-free and can withstand network crashes.

(2) The distributed deadlock problem is avoided because transactions are executed in a prespecified order.

(3) This approach lends itself well to the design of an algorithm which reduces the internode communication messages and delay.

In order to enforce conflicting transaction operations to execute according to the timestamp order, each transaction operation, before initiating its execution, must make sure there is no other conflicting transaction operation with a less recent timestamp outstanding in the system. Consequently, a transaction, instead of being deferred by a lock set by a conflicting transaction, will be deferred by waiting for the completion of execution of all preceding conflicting transactions. This is the reason we do not claim that a timestamp algorithm can have better concurrency than a locking algorithm.

Nevertheless, in many patterns of conflicting transactions, such waiting is not necessary [BER79]. That is, it is still possible to execute conflicting transactions concurrently. The concurrency control mechanisms of SDD-1 [BER80] and of DLDBS [CHO80] exploit this idea. Thus, they
can attain more concurrency than algorithms of the other two approaches, in which all concurrent and conflicting transactions except one will always be deferred or rejected.

The remainder of this chapter is organized so that in Section 4.2 two locking-based solutions, a simple centralized locking algorithm and a distributed locking algorithm, are presented. Then, in Section 4.3 two rejection-based algorithms, Thomas' algorithm and Ellis' algorithm, are described. In Section 4.4 two timestamp-based algorithms, Badal's algorithm and SDD-1's algorithm, are discussed. We not only present the fundamental concepts of each solution, but also outline their advantages and disadvantages and make comparisons among them. The basic reasons of why they preserve database consistency are also given. Finally, in Section 4.5 we explain the motivation for the need of a new solution in DLDBS.

4.2 Locking-Based Solutions

Most locking-based algorithms to synchronize concurrent updates in distributed DBMS are extensions of locking methods in centralized DBMS. A simple locking-based
solution using centralized control is described as follows:

[Simplified Centralized Locking Algorithm]

A master node works as a central locking controller to receive all transaction requests from any other node in a network. The master node keeps all necessary information to synchronize transaction requests and to detect and to resolve deadlocks. The procedures for the master node are stated as follows:

1. receive a transaction from any node in the network;
2. issue lock requests on behalf of the requesting node;
3. process the lock requests exactly as a locking method does in a centralized DBMS;
4. access the data at any node that has a copy;
5. release locks.

Two issues have to be noted: (1) if a logical data item has many stored copies, a lock on a logical data item cannot be released before all copies are updated; (2) both query and update transactions have to follow the locking algorithm; otherwise, a query could read inconsistent data.
Correctness

Internal Consistency: This can be preserved for the same reason as a locking algorithm in a centralized DBMS (e.g., to issue, process, and release locks following well-formed and two-phase locking rules).

Mutual Consistency: Setting or releasing a lock on a logical data item implies setting or releasing the lock on all physical copies. Therefore, all physical copies of a data item are accessed globally as a single one; mutual consistency is preserved.

Deadlock-free: A centralized deadlock detection and resolution mechanism as the one in a centralized DBMS can handle this problem.

Advantages
(1) It is a simple solution.
(2) Deadlock handling will not be more difficult than in a centralized environment.
(3) Internode communication overhead and delay are not high. A good degree of concurrency can be attained.
Disadvantages

(1) Storage overhead (e.g., locking tables) at the master node to keep track of current information about locking and deadlock detection may be high.

(2) The master node may be a communication traffic bottleneck as well as a transaction execution bottleneck.

(3) This is a rather unreliable method; if the master node fails, the whole network fails.

Disadvantages (2) and (3) can be serious drawbacks that preclude an implementation of such an algorithm in a distributed environment. In order to make the centralized locking algorithm robust to a master node failure, Alsberg and Day [ALS76] proposed an algorithm in which the master node employs a backup copy. Any actions on the data at the master node will also propagate to the backup copy so that a master node image is always recorded. Therefore, if the master node fails, the backup node can immediately take over as a new master node and another backup copy is selected. Alsberg and Day's algorithm only deals with single node failures; multiple node failures and nested node failures are not considered. A rather involved mechanism proposed by [MEN77] is designed for handling the case of multiple node failures.
In order to avoid the bottleneck problem, the INGRES group proposed an algorithm [ST078] using a "semi-distributed" control. This algorithm basically employs an idea similar to Alsberg and Day's. However, instead of choosing a master node as a primary copy for all data items, each group of duplicated data items has a primary copy, and the primary copies for different groups may be stored at different nodes. In this scheme, the bottleneck problem is apparently reduced; a primary copy node failure will not be as disastrous as a master node failure, but the deadlock detection will be more involved.

[Distributed Locking Algorithm]

There is another different locking approach [ROT77b] where the control is not centralized to a single master node, and the locks are distributed with the data items. In this scheme, the node issuing the transaction will serve as a temporary master node to lock, access, and release a lock on physical data. The control in this scheme will be more distributed than a centralized locking scheme as described above; thus, the bottleneck problem is reduced. The distributed locking scheme typically has a high internode communication overhead and long delay [ROT77]. Besides, a distributed deadlock detection mechanism is required.
4.3 Rejection-Based Solutions

Basically, rejection-based algorithms [TH079, ELL77, MUL75] are extensions of the distributed locking algorithm. In these algorithms the action of locking is totally avoided. Therefore, the internode messages to set a lock, acknowledge a lock, and release a lock are eliminated. For a group of concurrent and conflicting transactions, all except one are rejected; rejection instead of waiting precludes the possibility of deadlock.

In order to reach the agreement to accept a transaction and reject the other, Mullery and Ellis [MUL75, ELL77] employ similar concepts, but Thomas [TH079] employs a different idea. All of them use a fully distributed control, thus avoiding the bottleneck problem and having good potential to withstand network failures. However, according to the literature published so far, these algorithms are not generalized to the partially duplicated DDB. The degree of concurrency attained in these algorithms is not high.
In [ELL77], a decentralized control algorithm is proposed. The basic idea of this algorithm is that every node can independently issue and broadcast a transaction to the remaining \( n-1 \) nodes (\( n \) is the number of nodes in the network). Then the node waits for \( n-1 \) positive acknowledgements, which, in fact, inform that no concurrent conflicting transactions exist. The actual update can proceed after \( n-1 \) positive acknowledgements are received. The progress of a transaction at a node can be divided into three phases:

(1) **Receive-update Phase.** In this phase, a transaction has to acquire a timestamp and to wait at its initiating node until a possible current transaction initiated at the same node finishes processing. A timestamp can be considered as a transaction number with a local integer number in the high order bits and a node number in the low order bits. Thus, it is globally unique.

(2) **Prepare-update Phase.** In this phase, a transaction is broadcast to \( n-1 \) nodes and waits for \( n-1 \) positive acknowledgements. When each has acknowledged the prepare-update request, the originator will enter the
do-update phase (see (3) below). However, when at least one remote node responds with a negative acknowledgement, the originator will reject the transaction. A node will respond with a negative acknowledgement if the incoming transaction conflicts with and has a lower timestamp than the transaction which is initiated at the node and currently is in the prepare-update phase.

(3) **Do-update Phase.** In this phase, the originator broadcasts an update message to all nodes and the actual update is performed there. When every node responds with an ACKd, the processing of the transaction is completed.

**Correctness**

Internal Consistency: Within a group of concurrent and conflicting transactions, only one transaction with the highest timestamp (priority) is accepted and enters the Do-update Phase. Therefore, there is no chance of internal inconsistency.

Mutual Consistency: A transaction can enter the Do-update Phase only after every node with a copy of the database agrees; then the actual update will be performed at every copy. Therefore, all conflicting update transactions
proceed in the same sequence at every copy of the database. Mutual consistency is preserved.

Deadlock-free: Conflicting transactions are rejected instead of waiting. The deadlock problem is thus avoided.

Advantages
(1) It is a simple solution using distributed control.
(2) Physical locks and the deadlock problem are totally eliminated. The bottleneck problem is reduced.
(3) This algorithm is more robust than the centralized locking algorithm.

Disadvantages
(1) The originator of a transaction has control all the time over the progress of the transaction. That is, it is the temporary master node. This implies the algorithm is robust only with respect to failures of nodes other than the one controlling the update.
(2) During the lifetime of a transaction processing at a given node, no other transaction can be issued at the same time. This algorithm typically does not exhibit a high degree of concurrency.
(3) Conflicting transactions will be rejected and resubmitted later; re-submission of a transaction
increases communication traffic and upsets the user.

(4) The algorithm assumes entire duplication of the database at all nodes.

[Thomas' Algorithm]

In [TH076, TH079], a majority consensus algorithm is proposed. The idea is that every update transaction will be voted upon by the Database Managing Process (DBMP) at each node as to whether it should be rejected or accepted. A majority of OK votes is required for accepting a transaction and a DBMP will not vote OK for two conflicting transactions simultaneously. This prevents two conflicting transactions from being accepted at the same time since it is impossible for both of them to reach majority simultaneously. When a node recognizes that a majority of OK votes of an update transaction have been reached, it will tell all other nodes to do the actual updates. Again the progress of a transaction can be seen in three phases:

(1) **Receive-update phase.** In this algorithm each data item in the database has a timestamp associated with it. During this phase, upon receiving a transaction, the application process will request the values and the associated
timestamps of requested data variables of the transaction. After the application process computes the new values for the update variables, an update request is propagated in a sequential daisy-chain manner among nodes in the network. The update request is also tagged with a globally unique timestamp in the same fashion as in the Ellis' algorithm. This timestamp is used to guarantee mutual consistency when an actual update to the database copy is applied whereas the timestamp of the data item, which comes from the update request timestamp when an update is applied, is used to ensure the internal consistency.

(2) Prepare-update phase. In this phase an update request is propagated host by host to get a vote. A DBMP will vote REJ if any variable of the request is obsolete, i.e., any of the timestamps of the requested variables of the incoming request is smaller than that at the local database copy. This is because the timestamp of any database variable reflects the time it was last updated. A DBMP will vote OK and will mark the request as pending if each variable is current and if the request does not conflict with any pending requests. It also may vote PASS if each variable is current but the request conflicts with a pending request of higher priority. Otherwise, it defers a vote and remembers the request for later reconsideration. (It votes REJ if the
pending request is later accepted; it votes OK if the pending one is rejected."

The reason for taking such votes is to determine whether the data values on which the update request is based are still current during the propagation from node to node in order to exclude mutually other concurrent conflicting transactions. If the accumulated votes at a node are not yet enough to make a decision to accept or reject a transaction, all such votes and the update request are forwarded to the next node until a resolution is made. An update request is accepted if a majority of OK votes is reached or rejected if any node casts a REJ vote (for details, see [TH079]). To summarize, the voting rules and request resolution rules will accept only one of several concurrent conflicting transactions and reject the rest.

(3) **Do-update phase.** In this phase the decision on the acceptance or rejection of an update request and a value assignment message to modify all copies of update variables are propagated to all nodes. The value assignment message is tagged with the same timestamp as that of the update request. Since the variable delay exists in the communication subsystem, two value assignment messages of two consecutively accepted update requests may be received
and executed in a different order at different nodes (see Figure 4.1). This could result in mutual inconsistency of duplicated copies of the database at different nodes. This algorithm employs a mechanism to solve this problem by the method described next. A value assignment action of an accepted update request will change a local data copy at a node if and only if the timestamp of the local data copy is older (smaller) than the timestamp of the value assignment message. Whenever a data copy is modified, it is stamped with the timestamp of the value assignment message as its new timestamp value. This timestamping mechanism ensures that the effects of update requests on a data item are applied according to the increasing timestamp order. Therefore, when all update activities cease, different copies of a data item will converge to the same value. Thus mutual consistency is preserved.

Correctness

Internal consistency: The voting rules and request resolution rules will accept only one of several concurrent and conflicting transactions and reject the rest. As a result no chance of internal inconsistency exists.
R1 is the update request of transaction T1
U1 is the value assignment message of R1
Assume that X = 1, Y = 0 initially,
U1 is Y ← 2; U2 is Y ← 5.

Figure 4.1 Update Anomaly
Mutual consistency: The timestamping mechanism as described above ensures mutual consistency.

Deadlock-free: Conflicting transactions are rejected instead of waiting. The deadlock problem is thus avoided.

Advantages

(1) This algorithm is more robust than Ellis'. The control of the progress of an update in the voting process is attached to the message which carries the transaction and the outcome of the votes; there is no master node at any time. This means that any node can fail before it votes or after it votes without leading to the failure of the network as a whole.

(2) The deadlock problem and bottleneck problem are totally eliminated.

(3) It is one of the pioneer mechanisms proposed in this area. Some of his ideas (e.g., the timestamping mechanism) can be found in more recent designs.

Disadvantages

(1) This algorithm requires a daisy-chain communication. Therefore, it is not suitable for a network with a
(2) The synchronization delay of a transaction is a significant drawback - the delay is proportional to the number of nodes in the network.

(3) The timestamp embedded in each data item increases database storage.

(4) This algorithm needs to reject transactions and it requires entire duplication of the database at all nodes (so far, no version has come out for a partially duplicated case).

4.4 Timestamp-Based Solutions

Recently a few authors [BAD78, BER80, CH080] have employed a different approach in which many of the problems are solved by using timestamps. A timestamp is a uniquely generated identifier of a transaction. It is an ordered pair \((E, N)\), where \(E\) is a local eventcount that is incremented each time it is used. Thus, no two transactions emanating from the same node have the same number \(E\). \(N\) is a preassigned number for each node. All the eventcounts are integers generated independently at a local node, and they can be kept within tolerable synchronization by setting the local \(E\) equal to the \(E\) of an incoming transaction request.
from another node if the incoming E is higher than the local one.

In this approach timestamps are used to achieve serializability among transactions, to prevent deadlock, and to do automatic crash recovery [CH080]. Solutions in this approach typically use a fully distributed control and thus can withstand network crashes and are bottleneck-free.

[Badal's Algorithm]

In [BAD78], a read driven protocol was proposed. The concept of "read driven" is in the sense that Write operations of a transaction are triggered by subsequent Read operations of another transaction. The basic procedures of this algorithm have the following four steps:

1. SUM (set up messages). Consisting of Read events and Write events of a transaction are sent out and placed at respective Read nodes and Write nodes.
2. A preferred Read node broadcasts a REQ (request) message to n-1 nodes (assuming n nodes in the network).
3. Upon receipt of a REQ message, every other node responds with a REQ-ACK message containing all Write messages addressed to any Read node of the transaction.
4. If the preferred Read node has received n-1 REQ-ACK
messages, then it broadcasts one READ COMMAND containing all Write messages to the nonpreferred Read nodes of the transaction. A Read node will execute Write events of earlier timestamped transactions according to the timestamp order if the Write values have been received. It will execute a Read event if all preceding Write events interfering with the Read event have been executed.

Correctness

Internal consistency: Serializability of transactions can be attained in the algorithm because a Read operation is performed only after all conflicting Write operations with smaller timestamps have been carried out, because Write operations are performed in the order according to their timestamps, and because a Write operation is performed only after preceding conflicting Read operations are done. As a result, all interfering actions of transactions are executed in the order of their timestamps. Consequently, the effects are equivalent to a serial execution of transactions according to the timestamp order. Internal consistency is preserved.
Mutual consistency: Mutual consistency is also preserved because operations on all copies of a data item are performed in the same timestamp sequence.

Deadlock-free The deadlock problem is entirely eliminated because operations are performed in a pre-specified order.

This algorithm has the same advantages as those solutions using the timestamp approach: robustness, deadlock-free, and bottleneck-free. However, this algorithm does not necessarily have better concurrency than locking or rejection-based solutions, because the execution of a Read operation will be delayed until the execution of all preceding conflicting Write operations is completed. In many conflict patterns among transactions, this delay is not necessary. The solutions of SDD-1 to be described below and of our DLDBS can improve such a delay in a significant way.

[SDD-1 Algorithm]

The System for a Distributed Database (SDD-1) is a prototype distributed database system designed and implemented at Computer Corporation of America. The system consists of a collection of database nodes interconnected through a long-haul network. The database is physically
distributed with partial duplication over the network. In SDD-1 one type of virtual machine, Transaction Modules (TMs), will accept transactions. Another type of virtual machine, Data Modules (DMs), actually will house the database and perform operations on the database. SDD-1 has a communication subsystem which ensures that messages are received in the same order that they were sent.

A series of technical reports and papers have been released and published [ROT77a, BER78, BER80] concerning a technique for solving the concurrency control problem and have received wide publicity in this area. The focus of the concurrency control technique is to reduce run-time synchronization by preanalyzing the conflicting transaction relationships, thus avoiding excessive internode synchronization cost. The basic idea of this mechanism is the subject of the next discussions.

In order to maintain mutual consistency of different physical copies of a logical update data item, SDD-1's mechanism borrows the timestamping mechanism originally proposed by Thomas. The timestamping mechanism ensures that two Write messages of two transactions updating the same logical data item will perform actual updates in the same order at different DMs, each of which contains a physical
copy of the update data item. The rule of the mechanism is that each physical data item is attached a timestamp; a data item can be updated by a Write message if and only if the data item's timestamp is less than the Write message's timestamp.

SDD-1 assumes that most transactions a system will execute are known a priori at system design time, and when designing the database, a static set of transaction classes is established. A transaction class is defined by a read set which contains all read variables of the transaction class and a write set which contains all update variables of the transaction class. A real transaction can fit in a transaction class if its read (write) set is a subset of the read (write) set of the class.

An important idea of this mechanism called conflict graph analysis is applied on the statically defined transaction classes. The major purpose of this analysis is to identify the potentially dangerous conflicts among transaction classes which have to be synchronized. This also means that some unnecessary synchronization for controlling non-dangerous conflicts can be avoided. Some conflicts among transactions are not dangerous because they will not lead to non-serializability. The conflict graph
analysis which is done off-line determines the type of synchronization required for each transaction class which otherwise will be done at run time by exchanging synchronizing messages. This analysis technique is the reason for the mechanism being able to reduce synchronization cost and delay.

By analyzing the conflict graphs of transaction classes, the synchronization rules (protocols) required for each transaction class to avoid non-serializability can be categorized into four types. Each of them has different synchronization cost and are described as follows:

**Protocol P1**

(protocol selection rule) A Read message of a transaction in transaction class i will obey Protocol P1 with respect to a transaction in transaction class j if there is a diagonal edge <ri, wj> in the conflict graph and there are no cycles including <ri, wj>.

(effect) Protocol P1 is to prevent Read messages of one transaction and conflicting Write messages of another transaction from being processed in different relative orders at different DMs.
[implementation of Protocol PI] A TM sends a Read message of a transaction in transaction class $i$, attached with a read condition $(TS, (j_1, \ldots, j_m))$, to all destination DMs. $TS$ is a timestamp (not necessarily equal to the transaction's timestamp) and $j_i$ represents a transaction class $i$. A DM can process a Read message only when (1) all Write messages from classes $(j_1, \ldots, j_m)$ with timestamps earlier than $TS$ have been processed and (2) no Write messages from classes $(j_1, \ldots, j_m)$ with timestamps later than $TS$ have been processed.

In order to satisfy (1), the Read message can be processed only after a DM receives a Write message from every transaction class in the read condition and each of the Write messages has a timestamp later than $TS$. In order to satisfy (2), as soon as a DM receives a Write message timestamped later than $TS$, it must hold the Write message until it processes the Read message; for a Write message received with a timestamp earlier than $TS$, it can process the Write message immediately. Of course, when the Read message is received, some Write messages from transaction class $j$ may have been processed already. In this case, in order to satisfy (2), the Read message must be rejected. Note that if any Read message is rejected, all Read messages on behalf of the same transaction have to be resubmitted.
with a higher timestamp. Clearly the synchronization delay and the communication cost of this mechanism will depend on the choice of timestamps for read conditions.

**Protocol P2**

[protocol selection rule] A read message of a transaction in transaction class $i$ will obey P2 with respect to a transaction in transaction class $j$ and a transaction in transaction class $k$, if there is a cycle in the conflict graph including edges $<ri, wj>$ and $<ri, wk>$ but not $<ri, wi>$.

[effect] Protocol P2 is to prevent a Read message from seeing Write messages from two other transactions in reverse timestamp order, i.e., it prevents a case where Write$_k$ precedes and conflicts with Read$_i$ and Read$_i$ precedes and conflicts with Write$_j$ but $T_{Sj} < T_{Sk}$.

[implementation of protocol P2] This is the same as the implementation of P1.
Protocol P3

[protocol selection rule] A Read message of a transaction in transaction class i will obey P3 with respect to a transaction in transaction class j, if there is a cycle in the conflict graph including edges \(<r_i, w_i>\) and \(<r_i, w_j>\).

[effect] Protocol P3 is to prevent two transactions that read each other's output from both reading before either writes, i.e., preventing a classical race condition.

[implementation of Protocol P3] This is the same as the implementation of P1 except one difference: P1 allows any timestamp to be TS in the read condition while P3 requires TS = TS_i, where TS_i is the transaction timestamp along with the transaction i.

Protocol P4 is able to execute an unexpected transaction which does not fit into any of the transaction classes. The effect of P4 is similar to shutting off the system, executing the unexpected transaction, and returning to normal operation.
In the discussion of the implementation of PI, we assume a Read message has a read condition. Actually, a TM sends Read messages attached with a list of read conditions, each of which consists of a list of transaction classes obeying the same protocol. A DM processes a Read message only when all read conditions are satisfied.

4.5 Motivations for Designing a New Solution

The concept of centralized control in the centralized locking solution and some drawbacks (unreliable solution, network bottleneck of the aforementioned algorithms) preclude the possibility of implementing such algorithms in our DDLCN which is a local computer network. The distributed locking algorithm requires $5n$ messages in order to synchronize a transaction among $n$ nodes in a network: $n$ lock request messages, $n$ lock grant messages, $n$ message to transmit update, $n$ update acknowledgements, and $n$ messages to release the lock. The high internode communication overhead and long delays discourage us from adopting such a solution.

Both Thomas' algorithm and Ellis' algorithm seem to be attractive to us. In Thomas' algorithm, a given update is serially propagated host by host, which is inefficient for use in DDLCN, since DDLCN has a multi-destination routing
facility. The complexity of the algorithm and the storage overhead due to the timestamp labelled on database data are also reasons for not using this algorithm in DLDBS. Ellis' algorithm is an attractive candidate for a local network that has a fast broadcast facility. However, this algorithm uses a global locking mechanism, and thus has low concurrency and is robust only with respect to failures of nodes other than the one controlling the update. Both of these two algorithms use rejection to resolve transaction conflicts, which may increase communication traffic and processing delay.

SDD-1 assumes that most transactions a system will execute are known a priori at system design time; thus, a static set of transaction classes can be established. Then, it tries to detect the potential concurrency among conflicting transactions by statically analyzing transaction classes at database design time. As a result, it can reduce the amount of synchronization required for executing transactions. However, it cannot fully exploit the potential concurrency among conflicting transactions at run time; it sometimes may degenerate into a global locking case [MOH79], depending on how the TS in read condition is chosen (for details, see Section 6.5). SDD-1's mechanism is basically designed for a point-to-point long-haul network;
it does not take advantage of the broadcast facility which exists in a local network with broadcast channels such as DDLCN or ETHERNET [MET76]. SDD-1's solution is rather complex; it may not be suitable for a low cost local network.

In Chapter 5 and Chapter 6, two concurrency control mechanisms one for fully duplicated DLDBS and one for partially duplicated DLDBS, are given. They use distributed control and avoid major disadvantages of either the locking-based solution or rejection-based solution.

4.6 Summary

In this chapter a list of concurrency control mechanisms in the distributed database system have been discussed. They are categorized into three approaches: the locking-based solutions, the rejection-based solutions, and the timestamp-based solutions.

In the locking-based solutions, a requested data item has to be locked before an access starts. Some major disadvantages of this approach include a lack of reliability, creation of a network bottleneck, a need for deadlock detection and resolution, and high internode
communication cost and delay. Two algorithms, a simple centralized locking algorithm and a distributed locking algorithm, are discussed.

In the rejection-based solutions, an update transaction is first propagated to other nodes to decide the acceptability of the update. If a consensus cannot be reached, the update transaction will be rejected; otherwise, the actual update can be carried out. The major disadvantages of this approach are that its application is restricted to a fully duplicated DDB and the degree of concurrency of concurrent updates is not high. Two algorithms, El-Wis' algorithm and Thomas' algorithm, are discussed.

Finally, in the timestamp-based solutions, conflicting transaction operations are executed in a schedule equivalent to a serial schedule of the same transactions based on their timestamp order. Two algorithms Badal's algorithm and SDD-1's algorithm, are discussed. This chapter not only presents fundamental concepts of each solution but also outlines their advantages and disadvantages and makes comparisons among them. The basic reasons why they preserve database consistency are also given.
5.1 Background

In the design of the Distributed Loop Database System (DLDBS) for DDLCN [CH079], we have considered two types of virtual nodes. First, there are Loop Request Nodes (LRNs), which accept the user's requests, create transaction messages and supervise the distributed query processing of transactions. Second, there are Loop Data Nodes (LDNs) which contain the physical data. In a fully duplicated DLDBS, each physical Loop Node (LN) in the network can accept and create transaction messages and has a complete copy of the entire database data. Therefore, a Loop Node serves as both an LRN and an LDN.
Distributed transaction processing in DLDBS demands concurrent accesses to the database by independent transactions issued from different nodes. The problem of concurrency control arises whenever concurrent accesses are allowed. Concurrent accesses must be regulated by a concurrency control mechanism to ensure that two conflicting transactions do not perform inconsistent modifications to the database.

A correctly functioning concurrency control mechanism of a distributed database (DDB) must satisfy the following conditions:

(1) It preserves mutual consistency of the DDB. Because of variable communication delay, it is impossible to ensure that all duplicated database copies have identical value at all times. However, mutual consistency of database copies should be preserved so that all database copies will eventually converge to identical values if all update activities cease.

(2) It preserves internal consistency of the database. Each copy of the database must remain consistent within itself. The property of serializability [ESW76] can ensure the correctness of internal consistency.

(3) It is deadlock-free; for example, a transaction execution eventually terminates.
This chapter describes a simple and efficient concurrency control algorithm for DLDBS. The discussion to follow is restricted to a case called the "fully duplicated distributed database," meaning that a complete copy of the entire database is maintained at each node. The algorithm has characteristics as follows:

1. It uses distributed control.
2. It is robust with respect to both communication system and database node failures.
3. It neither creates a communication traffic bottleneck nor a transaction execution bottleneck in the network.
4. It does not reject transactions due to transaction conflicts.
5. It does not use global locking mechanisms and does not use timestamps to label database data.
6. It prevents deadlocks, so that no separate distributed deadlock detection mechanism is needed.
7. It can automatically execute more transactions issued from a heavily loaded node than from a lightly loaded node.
8. It requires only \( n \) messages per update for synchronization (where \( n \) is the number of database nodes in the network) and has good throughput and low delay.
9. It is simple and easy to implement.

This chapter is organized in the following way: Section 5.2 presents working definitions of some used terms and the overall procedures of the algorithm. Section 5.3 formally proposes the algorithm and explains its meaning. In Section 5.4 the reliability mechanism for maintaining the continuing operation of DLDBS in spite of node crashes and link failures is given. Section 5.4 also discusses several topics: detection of a crash, reliable broadcast, withdrawal of a crashed node, reinstatement of a repaired node, and recovery of one or multiple node crashes and nested node crashes. Section 5.5 gives the argument for the correctness of the algorithm and the reliability mechanism. Finally, in Section 5.6 a performance analysis of the algorithm in terms of three measures (message traffic, delay, and throughput) is made.

5.2 Overview of the Algorithm

5.2.1 Preliminary Definitions

**Priority**: A priority assigned to a transaction is an ordered pair \( P = \langle E, N \rangle \), where \( E \) is an increasing integer
generated by a local counter of an LDN that is incremented each time a transaction arrives and $N$ is a preassigned number for each LDN. Thus, no two transactions emanating from the same LDN have the same number $E$. $P_1 = <E_1, N_1>$ of a transaction is said to have a higher priority than $P_2 = <E_2, N_2>$ of another transaction if either (i) $E_1 < E_2$ or (ii) $E_1 = E_2$ and $N_1 < N_2$. Note that $E$ may be reset to zero during system maintenance.

Transaction Conflict: A transaction is characterized by the following two sets of data items:

1. The Read Set (RS) - the set of input data items.
2. The Write Set (WS) - the set of output data items.

Two transactions $T_1$ and $T_2$ are said to be in conflict if the following conditions occur:

(a) the intersection between the Read Set of $T_1$ and the Write Set of $T_2$ is not an empty set, or
(b) the intersection between the Write Set of $T_1$ and the Read Set of $T_2$ is not an empty set, or
(c) the Write Set of $T_1$ and the Write Set of $T_2$ is not an empty set.

The notions of internal consistency and mutual consistency are described in Section 3.8. The definition of transaction, schedule, serial schedule, and schedule
equivalence that appear in this paper can be found in Section 3.3 and Section 3.6.

5.2.2 Overview

This subsection presents an informal outline of our concurrency control algorithm. The algorithm is assumed working on top of a communication subsystem which has reliable end-to-end protocols [PAR79]. In normal cases (no node crashes or link failures), the protocols in the communication subsystem can guarantee that (1) a transaction message will eventually be delivered to all destinations and that (2) transaction messages from a node are delivered in the order they were sent.

A Loop Node in the network serves as both an LDN and an LRN. The distributed software residing at each Loop Node to enforce mutual consistency among database copies is called the network concurrency control processor, which is a component of the Inter-Database Control Software (IDCS). Each local DBMS at a Loop Node has its own processes to handle local concurrency control when local transactions are executed concurrently. It is assumed that distributed transaction processing is initiated by user processes, each of which is local to one of the Loop Nodes. User processes
may be either processes representing some remote on-line users or processes on behalf of application programs [CH079].

Without loss of generality, it is assumed that when a user tries to access DLDBS by sending a transaction, a user process is created in a Loop Node. After some integrity checking is done and the transaction is considered valid (consistency at this level is not our interest here), the user process sends this transaction (in case of an update transaction) to IDCS. Then, a network-level concurrency control will be enforced. After the network concurrency control processor finishes its job, the local DBMS starts to execute the update transactions and local concurrency control will be enforced there. Note that in the case of a retrieval transaction, the user process may pass the transaction directly to its local DBMS.

For convenience of discussion, the progress of an update transaction is divided into three phases. Phase I (the transmitting phase) and phase II (the selecting phase) are handled by the network concurrency control processor, whereas phase III (the processing phase) is handled by its local DBMS. Note that the different phases of different transactions or the same phase of different transactions may
overlapping in the same node.

**Phase I: Transmitting Phase.** After the network concurrency control processor of a Loop Node receives a transaction from a user process, a priority as described in Section 5.2.1 is assigned to the transaction. The Loop Node then broadcasts a transaction message to every Loop Node. As soon as a Loop Node receives a transaction message it puts the transaction message in a queue called the **Execution Waiting Queue** (EWQ). After the broadcast is completed, the network concurrency control processor is ready to accept the next transaction from other user processes. (A broadcast in the algorithm is said to be completed when the sender is informed by the communication subsystem that every destination has received the broadcast message.) Note that in normal cases, a broadcast message in this phase is guaranteed to be delivered to all destinations by the communication subsystem using reliable end-to-end protocols as assumed above.

**Phase II: Selecting Phase.** Transactions waiting in an EWQ will be selected by the network concurrency control processor and then dispatched to the local DBMS for processing. Since the "higher than" relation among priorities of different transactions forms a total ordering
[LAM78] for preserving mutual consistency, we will enforce every Loop Node to dispatch transactions in its EWQ according to this total ordering. For this purpose, we always dispatch the transaction which has the highest priority among all transactions which have not yet been dispatched. However, the highest priority transaction in an EWQ may not be the true highest one, since it is possible that the true highest priority transaction is still on its way in transmission. If we defer this selection until there is at least one transaction from every Loop Node in the EWQ, the highest priority transaction in the EWQ is the true one. The reason is that since transaction priorities from the same node are assigned, sent out, and received in a decreasing order of priority, an arriving transaction from a Loop Node will always have a lower priority if the EWQ already has a transaction issued from the same node.

Assertion 1: If any two transaction messages issued from the same node move into EWQ in a decreasing order of priority (increasing value of E) and if the selection rule in A3 (see Section 5.3) is followed, then all transaction messages can be dispatched in a total ordering according to their priorities.

Proof: This is discussed above.
**Phase III:** Processing Phase. After transactions are dispatched to the local DBMS for processing, the concurrency control problem is changed from a distributed environment to a centralized environment. For exploiting concurrency, we like the transaction processing to be executed in an interleaved manner. The problem in this phase is: "Given a set of transactions arriving in a priority order, can we concurrently execute these transactions in such a way that a schedule of transactions is equivalent to a serial schedule of transactions determined by the priority order?" A simple solution, for example, can be that the local DBMS always locks (using a granularity lock scheme described in [GRA78]) all necessary resources before a transaction is allowed to be executed and unlocks resources after transaction execution is finished. Therefore, non-conflict transactions can be executed in an interleaved manner, and conflicting transactions are executed in a serial manner according to a priority order.

**Assertion 2:** If transactions are dispatched to the local DBMS according to their priority order, they can be executed so that a schedule of transactions is equivalent to a serial schedule of transactions determined by the priority order.

**Proof:** This is discussed above.
5.3 The Algorithm (Normal Case)

The algorithm in the case that the network communication and processing system are in normal operating condition, is stated formally below (see Figure 5 1).

**Data structure:** Each node has an Execution Waiting Queue (EWQ), a digital counter CNT for generating a non-decreasing integer E, a time-out CLOCK, and a flip-flop STATE (STATE = 0 means user processes can send a transaction to IDCS; STATE = 1 is the forbidden state).

**Initialization:** E <- 0, CLOCK <- 0, STATE <- 0, CLOCK starts ticking.

(A1) When STATE = 0, the network concurrency control processor of IDCS accepts an update transaction from a user process. If the network concurrency control processor has not received any transactions from user processes and the value of CLOCK exceeds the time-out period Tw, a dummy transaction is created. Then, STATE < 1. A priority is assigned to the transaction by the Loop Node. The priority is an ordered pair P = <E, N>, where E has the value of CNT, after CNT is incremented by one. N is a preassigned number for each Loop Node. (Because all the values of E are integers generated independently by a local Loop Node, they
Figure 5.1 Algorithm Protocols
may be unsynchronized eventually if the generating speed of E on each node is different. However, all the values of E can be kept within tolerable synchronization by setting a local E equal to the E of an incoming transaction message from another Loop Node if the incoming E has a value higher than the local one [LAM78].

(A2) A transaction message (Priority, Transaction) for the transaction is then put into the local EWQ and also broadcast to all other Loop Nodes. After the broadcast is completed, the local Loop Node sets STATE <- 0, CLOCK <- 0, and lets CLOCK start ticking. On the receiver side, as soon as it receives a transaction message, it saves the transaction message in the EWQ.

(A3) A transaction is selected from an Execution Waiting Queue (EWQ) by the network concurrency control processor for dispatching to the local DBMS if both of the following two conditions are satisfied:

(a) There is at least one transaction from every Loop Node pending in the Execution Waiting Queue of the Loop Node. (A transaction is said to be pending if it has been accepted by a Loop Node, but has not been selected and dispatched for processing. The transaction could be a dummy transaction.)
(b) The transaction selected has the highest priority among all the (real and dummy) transactions pending in the Execution Waiting Queue.

(A4) After a transaction is selected and dispatched to the local DBMS, it will process the transaction in a way to preserve the consistency of the database; that is, transactions are executed in such a way that the schedule of transactions is equivalent to a serial schedule of transactions determined by their priority order.

(A5) As soon as a local DBMS finishes the execution of a transaction, an ACKd is sent to the sender of the transaction. After the sender receives an ACKd from each of the receivers, the sender notifies the user process of the completion of the update transaction by sending a DONE message to it.

If the network concurrency control processor does not receive a transaction request in a time period Tw, a dummy transaction is broadcast to all other Loop Nodes to prompt processing of transactions. Note that a dummy transaction only contains its priority P, and processing of a dummy transaction actually takes no computing time. Furthermore, if the network concurrency control processor detects that
its local EWQ has no real transaction pending, the dummy transaction generating process can be temporarily suspended until a real transaction enters the EWQ again. This fact is useful when the DLDBS may not receive any transaction request for a time period, and when time wasted in performing useless work can be saved by stopping the generation of a transaction in that time period.

**Assertion 3:** If the EWQ of a node has no real transaction pending, dummy transaction generation on this node can be stopped until a real transaction enters the EWQ again.

**Proof:** The generation of dummy transactions is to ensure that a real transaction in a node can be dispatched without being delayed indefinitely due to the slow sending of real transactions from some other nodes. Let Se be a node whose EWQ contains no pending real transaction. Assume that the node Se continuously sends a dummy transaction (Td) under this circumstance. We would like to prove that the dummy transaction Td generated from the node Se is useless for the purpose of dummy transaction generation.

When the node Se has no real transaction pending, there are two cases possible in the network:

1. There is no real transaction pending at any node. In this case, sending a dummy transaction Td from Se is
obviously useless.

(2) There is a real transaction (Tr) pending at some node (St). Since Tr is in St and since according to the requirement in the transmitting phase a transaction message is received by every node, Tr is also in Se eventually. However, because of the transmission delay, there are two cases possible. They are described as follows:

(a) Tr is on its way in transmission to Se. As soon as Se receives Tr, by the requirement of the algorithm, Se will resume generating dummy transactions which can cause Tr to be dispatched without the need of Td from Se. Therefore, letting Se continuously send a Td during the time interval between the transmission of Tr from St and the arrival of Tr at Se is useless for the purpose of a dummy transaction generation.

(b) Tr is already received by Se. Since the local EWQ of Se has no real transaction pending, Tr must have been dispatched already. In this node, as we recall the rule in the selecting phase, Tr can be dispatched only when there are a set of transactions which qualify Tr to be selected. From the requirement in the transmitting phase, this set of transactions also exists on St eventually. Therefore, Tr in the EWQ of St will be dispatched without the need of Td from Se. To continuously send a Td from Se in this case is also
Thus, we conclude that a node that keeps on sending a dummy transaction when its EWQ has no real transaction pending is useless for the purpose of dummy transaction generation. Thus assertion 3 is true.

From the information given above, we can see that dummy transactions are seldom generated in a heavily loaded network, and the generation of dummy transactions is also suspended during the time period when the DLDBS is not receiving any real transaction requests. These facts imply that the communication cost incurred by the generation of dummy transactions in our algorithm is very low.

5.4 Robustness of the Algorithm

5.4.1 Requirements of Robustness

The preceding section has outlined the operation of the algorithm under a normal condition. However, in practice there are many ways in which abnormal conditions may arise. By an abnormal case, we mean the case of node crashes and communication link failures.
By robustness of the algorithm, we mean that the following three requirements are maintained in an abnormal case:

(R1) The system will continue operating in spite of node crashes and communication link failures.

(R2) A transaction message will eventually be put into the EWQ of either every node or no node.

(R3) If a transaction message is put into an EWQ, it will be dispatched and executed to completion sooner or later; all transactions are eventually dispatched in a total ordering according to their priorities.

Note that R1 is the basic requirement of robustness, and R2 and R3 are conditions needed for the DLDBS to recover from crashes and failures and to lead correctly to a mutually consistent state.

5.4.2 Communication Link Failures

Our algorithm requires that each node has a (direct/indirect) path to every other node. Therefore, as long as no node is partitioned from the network, communication link failures do not create any difficulty to the algorithm.
When the network becomes partitioned, the partition which has a majority of nodes in the network still can continue operating and treats the nodes in the other partition the same as crashed nodes. Note that only one partition is allowed to operate; otherwise, inconsistency among databases in different partitions may occur. Using the recovery algorithm for node crashes which will be described below the network can return to a consistent state after the partitions are repaired. In DDLCN, network partition is rare due to the tri-state control mechanism built into the interface [WOL79].

5.4.3 Node Crashes

This subsection will first show how the algorithm can continue operating in the case of one or more node crashes. Then, it shows how the distributed database will recover from anomalies and lead to a consistent state when a crashing node has been repaired. Note that the node crash considered here may be a memory loss, which means that transaction information stored in the EWQ, the CNT value, and the ACKd information will be lost after the crash.
5.4.3.1 Detection of Crashing

If after one or more times of retransmission of transaction messages a sender cannot receive any response from the receiver node, it is reasonable (in the degree of reliability we desire) to conclude that a receiver node is unreachable due to either a node crash or a link failure. Any node which detects a crash has the responsibility to notify other nodes in the network of the crash.

5.4.3.2 Reliable Broadcast

To satisfy the requirement R2, we need a "reliable broadcast" facility [ROT77], which guarantees that a broadcast transaction will reach either every destination or none when the sender crashes during the broadcasting. To provide a reliable broadcast facility for the algorithm we require that every node keeps a Last Transaction Array (LTA). This array consists of the last transaction message (real or dummy) received from each node. Recalling that the broadcast protocol in the algorithm is a "one-message-at-a-time" protocol, we see that the existence of the last transaction message from a node implies that all previous transaction message broadcasts issued from the same
node are already complete. Only a broadcast for the last transaction message from a node can be an incomplete broadcast. Therefore, as soon as a node ascertains that another node is unreachable, the last transaction message received from that crashed node is broadcast to all other active nodes. Duplicated messages thus produced are rejected. (The priority of a transaction message is uniquely generated; thus, LTA can be used to reject transaction message duplicates.)

Let us consider a simple example to illustrate the idea described above. Suppose that a node S1 broadcasts a transaction message to nodes S2, S3 and S4. We further suppose that S1 crashes during the broadcasting in such a way that S2 receives the transaction but S3 and S4 do not. After S2, S3 and S4 learn of the crash, they broadcast their last received transaction messages from S1 to each other. Thus, the last broadcast message from S1 received by S2 will later be received by S3 and S4. The messages broadcast by S3 and S4 are duplicated messages and thus are rejected.
5.4.3.3 Recovery Algorithm

Withdrawal of a crashed node. Suppose that a node (Sd) detects a crash happening on a node (Sf) after Sd tried to send a transaction message to Sf and eventually concludes that Sf is unreachable. The sender node Sd then puts this transaction message as well as all transaction messages whose ACKs from Sf have not yet been received into a recovery array created for Sf. The node Sd also notifies the node (where the recovery array locates) of the timestamp of the last transaction message whose ACKd from Sf has been received by Sd. A recovery array, a buffer storage to buffer all transactions which will later be executed by the crashed node, may be duplicated to achieve robustness in the event that any active node may crash before Sf recovers. The node Sd also broadcasts the unreachable node's ID to all other nodes and deletes Sf from its local up-list. The up-list keeps the IDs of all active Loop Nodes in the network. Any active node receiving the broadcast from Sd will also buffer its transaction messages, notify the node (where the recovery array locates) of the timestamp of the last transaction message whose ACKd from Sf has been received, and update its up-list. Since the node where the recovery array locates receives from every active node the timestamp of the last transaction message whose ACKd from Sf
has been received, it can conclude which transaction may be the last one completed at node Sf before a crash occurs. All transaction messages issued from node Sf having a larger timestamp than the timestamp of that transaction will also be buffered in the recovery array.

It is possible that Sf is also a sender at the time of its crash. In this case the facility to achieve reliable broadcast as described in the previous subsection is initiated. After every active node receives the last broadcast message from Sf, this transaction message is stored in the recovery array of Sf because this transaction message may not survive in Sf after a memory loss crash.

From then on, broadcast messages are sent to database nodes which are in the up-list and the selection rule in A3 is also changed to consider only the transaction messages from active Loop Nodes. Any transaction messages (real transaction messages only) and ACKds destined for the unreachable node Sf are sent to the active nodes which have a copy of the recovery array for the crash node and are logged into the recovery array.
Reinstatement of a repaired node. Upon its recovery, node Sf first makes a request to a node having a recovery array copy (if that node is not active, it contacts another one) to obtain its recovery array (let us call it the first version recovery array). Then, it begins to process transaction messages buffered in the recovery array according to their timestamp order. In a normal case, at the initiating node of a transaction, the receiving of an ACKd of the transaction from other nodes is recorded in a bit map created for the transaction. This bit map now provides us with information to identify the transaction messages which are stored in the recovery array but were completed before the Sf crashed, and they will not be processed again.

During the processing of buffered transaction messages, the active nodes in the network keep on sending newly generated transaction messages into the recovery array of node Sf. Therefore, after the processing of the buffered transaction messages in the first version recovery array finishes, the recovery node Sf needs to ask again for the recently generated transaction messages buffered in the recovery array (let us call it the second version recovery array). In the meantime, the recovery node Sf broadcasts an <I-am-up> message to all other active nodes as well as makes
a request for the current up-list (other nodes may crash during the crash period of Sf). Any active node which receives the <I-am-up> message will update its up-list and consider that the recovery node Sf has rejoined the network. When recovery node Sf receives the first transaction message from another node, it synchronizes its CNT value and resumes its normal operation in the network.

Reconstruction of an EWQ. After node Sf rejoins the network and receives a copy of its second version recovery array, it puts all transaction messages in the recovery array into a newly constructed EWQ. After the EWQ is reconstructed, the dispatching of transactions is restarted in node Sf according to the selection rule of the algorithm. Apparently, while the recovery node reconstructs its EWQ and executes those buffered transaction messages, it may receive new transaction messages generated after it rejoins the network. Those newly generated transaction messages can be directly put into an EWQ, because the dispatching of transactions is according to the order of their priorities instead of according to the order of their entry into EWQ.

Under a typical operational environment in which the system is not overloaded, all transaction messages accumulated in the EWQ of a recovery node will be executed
and the operation in the network will return to its normal condition. Since the second version recovery array only contains recently generated transaction messages which were not included in the first version recovery array, transaction messages in it are few. Therefore, the time delay for a repaired node to resume its normal role in the network is short, and again the algorithm for reinstatement is simple.

**Nested Crashes.** The last problem not yet addressed is what happens when a recovering node fails while it is in the midst of a recovering operation. The solution to this problem can be achieved by following the procedure described next. When a buffered transaction message completes its execution, the recovering node sends a special acknowledgement to other nodes which have a copy of the recovery array. Then this transaction message is deleted from the recovery array. Should a crash happen before all buffered transaction messages are completed, those transaction messages which have not been completed (still kept in the recovery array) can go through recovery procedures in the same fashion as described above and be executed after the node recovers.
5.5 Correctness

This section presents arguments to show the correctness of our algorithm. The first subsection shows that the algorithm is robust, and the second shows the algorithm is correct in both normal and abnormal cases.

5.5.1 Correctness of Robustness

No mechanism yet developed can achieve 100 percent robustness [BER79]. The degree of robustness that can be attained depends on how much cost we care to pay. An algorithm is said to be robust if the recovery mechanism can satisfy the degree of robustness it claims under the assumptions it specifies.

The degree of robustness for our algorithm is that it can satisfy conditions R1, R2, and R3 of Section 4.1 under multiple node crashes and link failures. The assumption for robustness is that the ACKd information which can be stored in only one memory word as a bit map is stored in a stable storage (one that holds its data throughout system crashes or can easily recover its data after a crash).
To show the robustness of our algorithm, let us consider the 7 cases shown in Figure 5.2. A transaction will be in one of the 7 cases when a crash happens.

A. In case the crash node is a sender.

Case 1: A crash happens before a transaction is broadcast. After recovery, all information about the transaction is lost, and the user process must send the transaction again.

Case 2: A crash happens when a transaction broadcast is going on. The reliable broadcast facility discussed in Section 5.4.3.2 can ensure that the transaction message will be received by all destinations or none at all.

Case 3: A crash happens before all ACKs for a transaction are received from the destinations. Since ACK information is stored in stable storage, those ACKs received before the crash survive, but those ACKs generated during the crash are buffered in a recovery array and will be available to the recovering node.

Case 4: A transaction processing is completed. A crash will have no effect on it.

B. In case the crash node is a receiver.

Case 5: A crash happens before a transaction message is received by the crashed receiver. In this case, this unreceived transaction message will be buffered in a
Figure 5.2  7 Cases When a Crash Happens
recovery array by the sender according to the recovery algorithm and will be received by the crashed receiver after the receiver recovers.

Case 6: A crash happens during the interval when a transaction message is waiting for dispatch or is under execution. According to the recovery algorithm, all these transaction messages will be put into the recovery array and will be executed while recovering. To make the transaction execution idempotent (repeatable), all outputs that the transaction performs are placed in a temporary file, and they will be recorded on a permanent database only after the transaction execution is entirely completed.

Case 7: A transaction execution is completed on the receiver node. A crash will have no effect on it.

The recovery algorithm shows that the network can continue to operate in the face of node crashes and link failures; thus, R1 is true. Reliable broadcast in Case 2 and the discussion in Case 5 ensure that a transaction message will exist in the EWQ of every node or of none. The recovery array in Case 6 ensures that R2 is true even though some transaction messages which are already in the EWQ may temporarily be lost due to node crashes. Thus, R2 is true. According to Cases 3, 5, and 6, a transaction generated before a crash (either in transmission, in the EWQ, or under
execution) will eventually be dispatched according to a priority order and executed to completion, although it may be interrupted and delayed by the node crash. According to the recovery algorithm, transactions generated during crashes are buffered in a recovery array; by Assertion 1, they can be dispatched according to a priority order during node recovery. Therefore, all transactions, regardless of when they are generated (before, during or after crashes), can be dispatched according to their priority order. This is true for an active node as well as for a recovering node. Thus, R3 is true.

5.5.2 Correctness of the Algorithm

In this subsection, we will show that our algorithm can satisfy the following three conditions in both normal and abnormal cases:

(C1) It preserves mutual consistency of the multiple copy database.

(C2) It preserves internal consistency of the database.

(C3) It is deadlock-free.

Instead of directly proving C1, C2 and C3, we will first argue that the algorithm can satisfy the following three conditions:
(T1) A transaction message will eventually be put into the EWQ of every node or of none.

(T2) All transactions can be executed in such a way that a schedule of these transactions is equivalent to a serial schedule of transactions determined by their priority order.

(T3) The algorithm is deadlock-free.

In a normal case, T1 is true in the previous discussion in the transmitting phase. In an abnormal case, T1 is the same as R2; thus, we know T1 is true. In a normal case, steps A1 and A2 of the algorithm ensure that two transaction messages issued from the same node enter the EWQ in a decreasing order of priority. From Rule A3 and Assertion 1, all transactions can be dispatched according to their priority order; from Assertion 2, T2 is true in a normal case. R3 also tells us that all transactions can be eventually dispatched according to their priority order and can be executed to completion; from Assertion 2, T2 is true also in an abnormal case. Once a transaction is put into an EWQ, it will be forced to be dispatched for processing sooner or later by an incoming real or dummy transactions. Thus, there is no permanent waiting and T3 is true.
T1 implies that every Loop Node has the same set of transactions and T2 says that these transactions will be equivalently executed in the same order; thus, mutual consistency among database copies can be preserved. Therefore, C1 is true. The serializability implied in T2 ensures internal consistency of database [ESW76]; thus, C2 is true. T3 is the same as C3.

5.6 Performance of the Algorithm

We will now analyze the performance of this algorithm in terms of three measures: message traffic, delay, and throughput.

(1) Message traffic. This is the number of messages required to accomplish an update transaction. Let us assume that there are n multiple copies of the database. In counting the number of messages transmitted, we do not include the subnet acknowledgement. Consideration of traffic depends on the following types of protocols used:
(a) Multi-destination protocol: This is the case in DDLCN [PAR79].
1 to broadcast a transaction message to other nodes.

n-1 to transmit ACKd.

(b) Point-to-point protocol: this is the case in ARPANET.

n-1 to send a transaction message to other nodes. ACKd to the sender can be piggy-backed on the next transaction message (real or dummy) to the sender.

(2) Delay. There are four distinct time periods for completing a transaction: (a) time taken to broadcast the transaction message; (b) time waiting in EWQ for dispatching; (c) time taken to process the transaction; (d) time taken for transmitting an ACKd from a destination to the sender. The delay time discussed here is the sum of time taken in (a), (b), and (d) ((c) is not included, since it depends on the nature of the transaction and types of the database). Let us assume the time taken to broadcast a message is Tb, the time taken to transmit a single message is Ts (in DDLCN, Tb = Ts), and the average inter-arrival time between two update requests at a node i is Tai. The waiting time of a transaction in an EWQ closely relates to the time-out period Tw (for sending a dummy transaction) and Ta. Note that Tw is a design parameter of the system. In the best case, where Tw = 0 for each node i, the worst
delay for any transaction is $2T_b + T_s$ (one $T_b$ for (a), one $T_b$ for (b), and one $T_s$ for (d)); the shortest delay for a transaction is $T_b + T_s$. (As soon as the transaction enters the EWQ, it is qualified to be dispatched.) In the worst case, where $T_w = T_{ai} - T_b$, the worst delay for a transaction is $T_b + (T_w + T_b) + T_s = T_b + T_{ak} + T_s$ (one $T_b$ for (a), one $T_{ak}$ for (b), and one $T_s$ for (d)), where $T_{ak} \geq T_{ai}$, for each node $i$. If $T_{ak}$ is long, we may adjust the time-out period $T_w$ so that $T_w + T_b$ is a fraction of $T_{ak}$.

If, as in the case of $T_w = 0$ for each node $i$, in $2T_b$ time units no real transaction is issued from any node, the EWQ of any node will contain no real transaction, and dummy transaction generation is suspended. Because a dummy transaction is generated only when it is necessary and itself is a very short message, it will not increase communication traffic. Therefore, in a local network such as DDLCN which has high communication bandwidth, a choice of $T_w = 0$ for each node $i$ may be justified.

(3) **Throughput.** This is defined as the number of transactions executed per time unit. In the best case in which $T_{ai} \leq T_b$ for each node $i$, the throughput of the system is $n/T_b$ transactions per time unit since each node can contribute one real transaction in every $T_b$ time units,
and on the average, there are n transactions that can be dispatched in every Tb time units. When Tai >= Tb for some node i, assuming Tak is the longest inter-arrival time, the throughput will be in the range n/Tak <= throughput <= n/Tb. Now, from the nature of the algorithm, we can see an interesting feature. The system can automatically execute more transactions issued from a heavily loaded node than from a lightly loaded node. For example, if Tai = 3 Ta2, then every time the system executes one transaction issued from node 1, it also executes 3 transactions issued from node 2.

The performance of this algorithm compares favorably with other techniques such as the majority consensus algorithm [TH079] which requires in the best case n + n/2 + 3 messages per update and (n + n/2 + 3)Ts for delay; the decentralized parallel algorithm [ELL77], which requires 2n messages per update and 2 Tb + 2 Ts for delay, has the best throughput of n/(2 Tb + 2 Ts), assuming no transaction rejection would happen.
5.7 Summary

We are now in a position to summarize some characteristics of the algorithm.

1. The control of the algorithm is fully distributed. After the transmitting phase is complete, the sender does not have control over the progress of the transaction. Each node processes transactions independently according to the rules of the algorithm.

2. The distributed nature explains the robustness of the algorithm. The algorithm is robust with respect to node crashes and link failures.

3. No rejection of transactions occurs. Rejection of a transaction causes resubmission of the transaction at a later time and thus increases inter-node communication traffic and delay.

4. This algorithm uses neither a global locking mechanism nor timestamps to label database data.

5. The algorithm is deadlock-free, which is achieved as a side effect of the algorithm, and no separate distributed deadlock detection method is needed.

6. This algorithm can, by its nature, execute more transactions issued from a heavily loaded node; therefore, the system will not be clogged at busy nodes.

7. This algorithm is simple and easy to implement.
This algorithm requires only \( n \) messages per update transaction for synchronization, where \( n \) is the number of database nodes in the network. It has a good throughput of \( n/Tb \) transactions per time unit and the shortest delay of \( Tb + Ts \) time units in the best case. We have claimed that this algorithm can preserve mutual consistency and internal consistency in both normal and abnormal cases. The algorithm is easy to implement in a low-cost local computer network like the DDLCN.
CHAPTER 6

THE CONCURRENCY CONTROL MECHANISM FOR
THE PARTIALLY DUPLICATED DLDBS

6.1 Introduction

In the previous chapter we presented the concurrency control mechanism as well as the crash recovery mechanism for a fully duplicated Distributed Loop Database System (DLDBS). The fully duplicated distributed database is a special type of distributed database. Another general type of distributed database is called the partially duplicated distributed database where the database is decomposed into partitions and partitions may be replicated at one or many nodes. Although it is clearly impractical to fully duplicate the database data at every node, in some applications it may be desirable to maintain fully duplicated directories for a partially duplicated distributed database. Thus, the concurrency control mechanism described in the previous chapter can be applied.
to handle updates to the database directory for these applications.

In this chapter we shall present a new approach to handling the concurrency control problem in a partially duplicated DLDBS. By using some local computation, our approach can reduce communication traffic and transaction execution delay due to concurrency control, and avoid major disadvantages of other algorithms that use either a locking-based or a rejection-based method. Most interesting to us is the fact that this solution in many cases can provide more concurrency and less delay than SDD-1's solution, which is notable for the high concurrency it can attain. Our algorithm is simple and efficient, especially for a local network such as DDLCN, which uses multi-destination protocols [PAR79]. The algorithm has characteristics as follows:

1. It uses distributed control.
2. It is robust with respect to both communication system and database node failures.
3. It does not create a communication traffic bottleneck nor a transaction execution bottleneck in the network.
4. It does not use a global locking mechanism and does not use timestamps to label database data.
5. It does not reject transactions due to transaction conflicts and, sometimes, it allows interleaved execution among conflicting transactions.

6. It prevents deadlocks so that no separate distributed deadlock detection mechanism is needed.

7. It exploits potential concurrency among conflicting transactions, thus attaining high concurrency and low delay.

8. It is simple and easy to implement.

This chapter begins with Section 6.2 which presents working definitions of some frequently used terms and the overall procedures of the solution. Next, Section 6.3 formally presents the algorithms and explains their meaning through examples. In Section 6.4 we give the argument for the correctness of the solution. Finally, in Section 6.5 comparisons with other solutions, especially SDD-1's, are described.
6.2 Overview of the Algorithm

6.2.1 Preliminary Definitions

A transaction is characterized by two sets of data items: (1) the Read Set (RS), the set of input data items; and (2) the Write Set (WS), the set of output data items. In a relational database system, the RS and WS are specified by the domain of the relation to be accessed and by the predicate which is a qualification of the items to be accessed.

Every transaction is appended with a timestamp $TS = <E, N>$, where $E$ is an increasing integer generated by a local counter of a node, and $N$ is a preassigned node number. A TS can also be considered as a transaction number with a local integer number ($E$) in the high order bits and a node number ($N$) in the low order bits. Thus, it is globally unique. $P_1 = <E_1, N_1>$ of a transaction is said to be smaller than $P_2 = <E_2, N_2>$ of another transaction if either (i) $E_1 < E_2$ or (ii) $E_1 = E_2$ and $N_1 < N_2$.

A transaction processing conceptually consists of the following three steps of operations:
(1) **Read operations:** Read operations query the database to obtain values of data items, called Read Values. A transaction may have several Read operations which may span over different nodes, called Read Nodes. The size of data items that a Read operation can access is determined by the atomic action which a node can support. The input data items of a Read operation is called the input set of the Read operation (to distinguish from the Read Set of a transaction). We use Ren c to denote a Read operation of a transaction with timestamp \((e, n)\) to be processed at node \(c\).

(2) **Computation operation:** By using Read Values obtained in read operations, the transaction performs a computation to generate new values, called Write Values, for the data items in WS. In order to reduce the communication cost and to increase parallel processing, another system mechanism, called distributed query processing, implements a strategy for efficiently retrieving the Read Values. Without loss of generality, this paper assumes a simple distributed query processing scheme that all requested Read Values of a transaction are moved to the originator of the transaction for computation.

(3) **Write operations:** Write operations record Write Values on the permanent database. A transaction may have several Write operations which may span over different nodes, called Write Nodes. The size of the data items that
a Write operation can access is determined by the atomic action which a node can support. The output data items of a Write operation is called the output set of the Write operation. We use \( W_{e,n} \) to denote a Write operation of a transaction with timestamp \((e, n)\) to be processed at node \( c \). Note that for convenience, we also use one subscript to represent a timestamp affixed to a transaction and a transaction operation, written as, for example, \( T_i \) and \( R_j \), where \( i, j \) are timestamps.

If a logical data item is physically duplicated at several LDNs, then only one Read operation is issued to a particular stored copy when a transaction wants to read the data item. For writes, however, a Write operation is required for each stored copy. Two transactions are in conflict if the Read Set or Write Set of one transaction intersects the Write Set of the other. Two transaction operations from different transactions are said to be in conflict if their relative order to each other in a global schedule \([\text{ROT77a}]\) cannot be arbitrarily switched. They are a Write - Write, a Read - Write or a Write - Read pair executing on the same node; the intersection of the sets of their accessed data items is not an empty set. A Read and a Write operation from the same transaction are always defined to be in conflict. Table 6.1 illustrates the conflict
<table>
<thead>
<tr>
<th></th>
<th>Two Transaction Operations are at the Same Node</th>
<th>Two Transaction Operations are at Different Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R - R</td>
<td>no conflict</td>
<td>no conflict</td>
</tr>
<tr>
<td></td>
<td>have conflict if output set of one intersects the output set of the other</td>
<td>no conflict</td>
</tr>
<tr>
<td>W - W</td>
<td>have conflict if (a) the input set of one intersects the output set of the other, or (b) R and W belong to the same transaction</td>
<td>have conflict if R and W belong to the same transaction</td>
</tr>
</tbody>
</table>

Table 6.1  The Conflict Relationship between Two Transaction Operations
relationship between any two transaction operations.

6.2.2 Overview

This subsection presents an informal outline of our concurrency control mechanism. The mechanism is assumed to be working on top of a communication subsystem which has reliable end-to-end protocols [PAR79]. In normal cases (no node crashes or link failures), the protocols in the communication subsystem can guarantee that (1) a transaction message will eventually be delivered to all destinations and that (2) transaction messages from a node are delivered in the order they were sent.

In the distributed loop database environment, the database is partitioned and partitions are replicated in nodes called Loop Data Nodes (LDNs). The distributed software residing at each LRN and LDN to enforce consistency for databases is called the network concurrency control processor which is a component of the Inter-Database Control Software (IDCS). Each local DBMS has an Execution Monitor (EM) actually to execute the Read or the Write transaction operations.
Without loss of generality, it is assumed that a remote on-line user or an application program tries to access DLDBS by issuing a transaction. For convenience of discussion, the progress of transactions is divided into three phases. Phase I (the transmitting phase) and Phase II (the optimizing phase) are handled by the network concurrency control processor, whereas Phase III (the processing phase) is handled by the Execution Monitor (see Figure 6.1).

Phase I Transferring Phase. After the network concurrency control processor of an LRN receives a transaction from a user process, a timestamp as described in Section 6.2.1 is assigned to the transaction. Assuming that each LRN has a global enterprise directory (see Section 9.3.2), the Read Set, the Write Set, the Read operations, and the Write operations of the transaction can be identified. A transaction message <RS, WS, a list of Renc, a list of Wenc> is then broadcast to every node in the network. Note that every Read or Write operation of the same transaction bears the same unique timestamp of the transaction.

Every LDN copies the transaction message and puts the transaction message in a queue called the Execution Waiting Queue (EWQ).
Figure 6.1 Algorithm Protocol
Whenever there are \( k \) transaction messages having the same \( E \) value in the transaction timestamps and pending in \( EWQ \), they will be selected together from \( EWQ \) (where \( k \) is the number of LRNs in the network). To guarantee that transactions are selected at a minimum rate, dummy transactions are generated. The information about WS and RS of each selected transaction is saved for use in the next phase, and Read operations and Write operations which can be executed locally are arranged as a local schedule according to their timestamp order. The algorithm A in Section 6.3 is a formal description of this phase.

Let us consider a simple example. In Figure 6.2.a, it is assumed that two transactions, \( Y := X, Y' := X \) and \( Z := Y + Y' \), are respectively issued from Loop Request Nodes 1 and 2 close together in time with timestamps \( <E_1, 1> \), \( <E_2, 2> \). (We use integers for LRN node numbers and alphabets for LDN node numbers.) \( E_1 \) and \( E_2 \) may be any integer numbers, and from Rule A1 of Algorithm A, \( E_1 \) and \( E_2 \) can be tightly synchronized at all time. Without loss of generality, we assume \( E_1 = E_2 = 3 \). Assume that Loop Data Node a has data items \( X, Y', Z \) and LDN b has data item \( Y \). After transaction messages TM31 and TM32 are created, broadcasted from node 1 and node 2, and qualified to be selected as a group at node a and node b, the transaction operations are arranged...
Transaction issues from node 1
\[ Y := X \quad Y' := X \]

Transaction issued from node 2
\[ Z := Y + Y' \]

\[ R_{31a}(X) \]
\[ T_{M31}: W_{31a}(Y') \]
\[ W_{31b}(Y) \]

\[ R_{32a}(Y') \]
\[ T_{M32}: R_{32b}(Y) \]
\[ W_{32a}(Z) \]

LDA LRN LRN LDN
node a node 1 node 2 node b

\[ X, Y', Z \]

\[ R_{31a}(X) \]
\[ T_{0}+Ts \]
\[ W_{31a}(Y') \]
\[ T_{0}+4Ts \]
\[ R_{32a}(Y') \]
\[ W_{32a}(Z) \]

\[ X \]
\[ T_{0}+2Ts \]
\[ W_{31b}(Y) \]
\[ T_{0}+3Ts \]
\[ R_{32b}(Y) \]

TO : initial time
Ts : average time taken to transmit a single message
Equivalent result : T31 \(\rightarrow\) T32
Total delay during execution : 4 Ts

Figure 6.2.a Execution According to Timestamp Order
according to their timestamp order as depicted in Figure 6.2.a.

Phase II: Optimizing Phase. After transaction operations are arranged as a serial schedule according to the timestamp order, a simple method but with less concurrency is to execute those Read and Write operations of transactions in such a way that conflicting operations of transactions at the same node will be executed in the order of their timestamps and that nonconflicting operations will be executed in any order since the serializability [ESW76] will not depend on the order of nonconflicting operations. The degree of concurrency it can achieve is similar to that of the protocol P3 of SDD-1 or the algorithm proposed in [BAD78].

However, a better approach is possible. The k transactions selected as a group are considered as concurrent transactions. Since each of the k transactions is issued from different nodes and the timestamp order of these transactions does not reflect the order of the transaction's arrival at the DLDBS, the system need not necessarily execute these transactions in the order indicated by their timestamps. In many conflict patterns of concurrent transactions, trying to execute these
transactions so that the result is equivalent to a serial execution of transactions according to timestamp order may lose some attainable concurrency and create unnecessary delay.

Let's consider the same example as in Phase I. In Figure 6.2.a, since a Write operation cannot start before its Write Value is available, W31b (Y) cannot start until R31a (X) has been executed by node a and the Write Value has been transmitted to node b. The same situation is true for W32a (Z). Since W31b has a smaller timestamp than R32b and since they conflict with each other, W31b must be executed before R32b if we use the method described above which requires executions according to the timestamp order. The sequence in which DLDBS executes the transaction operations is shown as dotted and solid lines in the Figure 6.2.a. Here we assume execution starts at time T0 and local execution delay is negligible as compared to internode communication delay Ts, where Ts is the average time taken to transmit a single message. The execution result would be equivalent to T31 <- T32 (we use notation "<-" to represent "is executed before"). Thus, the total execution delay in this case is 4 Ts.
Apparently, the major run-time delay in the example above is a Write operation waiting for its Write Value to be available. If we can execute the Read operation of each transaction as early as possible, we can reduce such delay and create more parallelism. That is, we can try to rearrange the sequence of the local schedule at each node by moving Read operations as close to the top of the schedule as possible. Obviously, this kind of movement is not always possible, because it may create a nonserializable global schedule.

To summarize, in Phase II, we will use Algorithm B (as described in Section 6.3) to move some Read operations as close to the top of each local schedule as possible and at the same time keep the global schedule still serializable. Figure 6.2.b shows that R32b (Y) can be moved before W31b (Y) by applying Algorithm B.

Phase III: Processing Phase. In this phase, the Execution Monitor will actually execute the transaction operations at each node by applying Algorithm C (see Section 6.3). The basic idea of Algorithm C is that only conflicting operations at the same node must be executed in the "position order," but nonconflicting operations can be executed at any order. The execution of transaction
Equivalent result: T32 ← T31
Total delay during execution: 2 Ts

Figure 6.2.b Execution after Switching the Order between Read and Write Operations
operations according to Algorithm C will produce a schedule which is equivalent to the serializable schedule produced by Algorithm B. Note that in phase II, we do not rearrange the position order of Writes; thus, the conflicting Writes in phase III will be executed in their timestamp order. This will facilitate the handling of crash recovery to be discussed in Chapter 7.

From Figure 6.2.b, we can see the order in which the system executes the Read and Write operations. R31a (X) and R32b (Y) can be executed at node a and node b in parallel without any waiting. W31b (Y) and W32a (Z) can be executed as soon as their Write Values are available. The execution delay is reduced from 4 Ts in Figure 6.2a to 2 Ts, and the result is equivalent to a serial execution of T2 <- T1.

In short, our approach is to improve, if not to optimize, the execution schedule of each group of concurrent transactions so that we can increase concurrency and reduce delay. Since there is only local computation involved when we do the improvement, this approach will not incur the overhead of internode communication traffic and delay.
6.3 The Algorithm (Normal Case)

The algorithm, in the case that the network communication and processing systems are operating in normal conditions, is stated formally below.

Algorithm A (for Phase I):

Data structure: Each LDN has an Execution Waiting Queue (EWQ); each LRN has a variable V, a flipflop STATE, and a digital counter CNT for generating an increasing integer E.

Initialization: CNT <- 0, V <- 0, STATE <- 0.

(A1) When STATE = 0, the network concurrency control processor of the IDCS accepts a transaction from an user process. If there is no real transaction arriving and if the value in V (see below) is greater than the value in CNT, a dummy transaction is created. Then, STATE <- 1. A timestamp is assigned to the transaction by the LRN. The timestamp is an ordered pair P = <E, N>, where E has the value of CNT, after CNT is incremented by one, and N is a preassigned number for each LRN. Whenever an incoming transaction message from another LRN is received and the local V is smaller than the incoming E, the value in V will
be incremented to the value of the incoming $E$.

(A2) A transaction message, including the Read Set and Write Set of the transaction, the list of Read and Write operations along with the transaction timestamp and the processing node number, is then broadcast to all other nodes in the network. After the broadcast is completed, the local LRN sets $STATE \leftarrow 0$. On the receiver side, an LDN copies a broadcast transaction message and saves the transaction message in an EWQ; an LRN adjusts its $V$ value using the incoming transaction message timestamp according to Rule Al.

(A3) Whenever there are $k$ transaction messages having the same $E$ value in their transaction timestamps and pending in the Execution Waiting Queue, the $k$ transaction messages are qualified to be selected as a group by the network concurrency control processor, where $k$ is the number of LRNs in the network.

(A4) After $k$ transaction messages are selected, the network concurrency control processor ignores dummy transactions if any, and saves information about the Read Set and Write Set of each real transaction. For transaction operations which can be executed locally the following rules will generate a local schedule at the node:
(1) If two operations have the same timestamp, put Read before Write.
(2) If two operations have different timestamps, put them in the timestamp order.

Figure 6.3.a is an example to show how Algorithm A works. Assume that in the network there are four LRN nodes: node 1, 2, 3, and 4 and three LDN nodes: node a which has data items T, Y; node b which has X, Z, U; and node c which has X, Y. We further assume that the values in V and CNT for every node in the network are initially 0. Transactions T11, T12 and T13 arrive at node 1, 2, and 3, respectively. After the Read Set, Write Set, and a list of operations for a transaction message are created at time to, the transaction message is broadcast. At time t0 + Tb (Tb is the average time taken to broadcast a message), node 4 receives transaction messages from the other nodes and sets V = 1. Then from Rule (A1), a dummy transaction TD14 is broadcast to every other node. At time t0 + 2Tb, TM11, TM12, TM13, and TD14 are qualified to be selected from EWQ and a schedule of local operations is generated at each LDN (see Figure 6.3.b). Note that a dummy transaction only contains its timestamp and processing of a dummy transaction actually takes no computing time.
Transactions: $X := T \quad U := X \quad Z := X + Y + U$

Read Set: $\{T\} \quad \{X\} \quad \{X, Y, U\}$

Write Set: $\{X\} \quad \{Y, U\} \quad \{Z\}$

<table>
<thead>
<tr>
<th>Time</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>TM11</td>
<td>TM12</td>
<td>TM13</td>
<td>TD14</td>
</tr>
<tr>
<td>$t_0 + Tb$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_0 + 2Tb$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3.a Transferring Phase
Figure 6.3.b  Execution According to Timestamp Order

Equivalent result: T11 ← T12 ← T13
Total delay during execution: 6 Ts
In the busy time period (e.g., in the afternoon) during which every node in the network is busy sending real transaction messages, dummy transactions are seldom generated; in the idle time period (e.g., at midnight) during which transactions arrive only once in a while (or not all), a dummy transaction is also generated once in a while (or not at all). The dummy transaction generation mechanism ensures that dummy transactions are generated only when they are necessary. (If no new transactions enter the DLDBS, after the value of CNT is equal to that of V at a node, this node can automatically stop the dummy transaction generation until a new transaction enters the DLDBS and causes the value of V to be greater than that of CNT again.) It also ensures that a real transaction only waits in EWQ for a short finite time. In the best case, a real transaction can be selected as soon as it enters EWQ; otherwise, it will wait in EWQ for an interval of one Tb (see Figure 6.3.a).

**ALGORITHM B** (for Phase II):

(B1) From the Read Sets (RS) and Write Sets (WS) of the k concurrent transactions, the system can generate the following collection of conflicting pairs S:
\[ S = \{<r_i, w_j>|RS_i \cap WS_j \neq \emptyset \text{ and } i < j\} \cup \{<w_i, r_j>|WS_i \cap RS_j \neq \emptyset \text{ and } i < j\}, \]

where \( i, j \) are timestamps.

(B2) Let \( TK \) represent the set of \( k \) concurrent transactions. A relation "\(<-\)" over \( TK \) can be induced and consists of \( T_i <- T_j \) if there is \(<r_i, w_j> \) or \(<w_i, w_j> \) in \( S \), where \( i < j \). After the relation set "\(<-\)" is generated, we further create the transitive closure of "\(<-\)" , denoted by \( t(\,<-\, )\).

(B3) Execute the following procedure in such a way that the relation "\(<-\)" will include any \( T_i <- T_j \) if and only if transaction \( i \) conflicts with transaction \( j \). We use \( t \) to represent the smallest timestamp in \( k \) transactions, \( h \) to represent the highest timestamp and \( i+1 \) \((i-1)\) to represent the timestamp next to and higher (lower) than the timestamp \( i \).

\[
\begin{align*}
\text{For } j &:= t + 1 \text{ to } j = h \\
\text{DO } &\text{ For } i := t \text{ to } i = j - 1 \\
&\text{DO IF a } <w_i, r_j> \text{ is in } S \text{ and there is no } T_i <- T_j \text{ in } t(\,<-\,) \\
&\text{ THEN generates a relation pair } T_j <- T_i \text{ and include } T_j <- T_i \text{ in relation } <- \\ 
&\text{ recalculate } t(\,<-\,) \\
&\text{ ENDIF}
\end{align*}
\]

\[
\begin{align*}
&i := i + 1 \\
&\text{ ENDDO}
\end{align*}
\]

\[
\begin{align*}
&j := j + 1 \\
&\text{ ENDDO}
\end{align*}
\]

END
(B4) A Read operation, \( R_i \), and a Write operation, \( W_j \), which is positioned immediately preceding \( R_i \), can switch their relative order if the following situations occur:

(a) \( R_i \) and \( W_j \) are not in conflict, or
(b) \( R_i \) and \( W_j \) are in conflict but \( T_j \prec T_i \) is not in the relation \( \prec \).

Note that the relative order of any two Read operations are always switchable.

(B5) Scan downward from the top of the local schedule until a Read operation is found. At this point, the system attempts to move the Read up as high as possible by repeating Rule (B4). When the Read cannot be moved up further, the system resumes scanning the schedule from the point it was interrupted.

In step B1, an element in \( S \) represents a conflicting pair of two different transactions. (There may be several conflicting pairs for two conflicting transactions, and there may be several conflicting transaction operation pairs corresponding to the same conflicting pair.) Thus, \( S \) consists of all such conflicting pairs of \( k \) concurrent transactions and represents how these \( k \) transactions interfere with each other. In a schedule of transaction operations, the relative order of two conflicting
transaction operations corresponding to a conflicting pair determines whether the schedule is serializable or not. Two equivalent schedules will have the same set of conflicting pairs, and the corresponding conflicting transaction operations will have the same relative order in both schedules.

In step B2, since our principle idea is to try executing a Read as early as possible, the relative order of Ri and Wj corresponding to a <ri, wj> pair, where i < j, should be "Ri <- Wj", which implies Ti <- Tj. Because Write operations always execute in the timestamp order, the relative order of two Write operations, Wk and Wl corresponding to a <wk, w1> pair where k < l, must be "Wk <- Wl," which implies Tk <- Tl. So far, the relation <- over TK will not have a cycle because for any Ti <- Tj in the relation, Ti and Tj are arranged in the timestamp order. For <wi, rj> where i < j, we hope we can execute Rj before Wi. However, in many cases, this might create a nonserializable schedule and result in an inconsistent database, because including a Tj <- Ti in the relation <- may create a cycle in the relation.
In step B3, we decide in which relative order the Rj and W1 will be executed if a <wi, rj> pair exists. When we add a new element into the relation <- due to considering a <wi, rj> pair, we make sure the t(<-) can remain antisymmetric. After step B3, we have a relation <- whose t(<-) is a partial order (transitive and antisymmetric). From graph theory, we know that if the transitive closure t(<-) is antisymmetric, <- can be topologically sorted. Thus, we can generate a topological order for all transactions sequenced by the relation <- and the relation set <- represents the execution order of any two conflicting transactions in a serial schedule.

In step B4 and B5, we actually switch the Read and the Write operations if such switches do not violate the ordering relationship between two conflicting transactions shown in <-.

**Assertion 1**: The Algorithm B generates a serializable schedule for k concurrent transactions.

**Proof**: From the previous discussion, Algorithm B will produce a relation <- which expresses the execution order between any two conflicting transactions and the transitive closure of <- is a partial order relation. Thus, we can generate a serial schedule (L') for the k transactions in
the following way: using a topological sort, we can arrange all conflicting transactions according to a topological order; for a transaction which does not conflict with any other transactions, it can be put in any position in the schedule. Since the schedule \((L)\) generated by Algorithm B has the same set of conflicting transaction operations as the \(L'\) has (because \(L\) and \(L'\) are composed of the same set of transactions) and any two conflicting transaction operations occur in the same relative order in both schedules \(L\) and \(L'\) (from Rules \((B4)\) and \((B5)\)), we know \(L\) and \(L'\) are equivalent [ESW76]. Therefore, \(L\) is serializable and is equivalent to \(L'\).

Using the same example as in Figure 6.3.a, we obtain Figure 6.3.c after the execution of algorithm B at each node. A relation \(<-\) will include \(T_{12} <- T_{11}, T_{13} <- T_{11}, T_{13} <- T_{12},\) and a modified local schedule at each node can be produced. Using the method in [ROT77a], we can have a global schedule \(L\) which is the combination of three local schedules. This schedule \(L\) is equivalent to a serial schedule \(L'\) according to the order \(T_{13} <- T_{12} <- T_{11}\).
After rule B1 \[ S = \{ \langle w_{11}, r_{12} \rangle, \langle w_{11}, r_{13} \rangle, \langle w_{12}, r_{13} \rangle \} \]

rule B3 Relation \[ \leftarrow = \{ T_{12} \leftarrow T_{11}, T_{13} \leftarrow T_{11}, T_{13} \leftarrow T_{12} \} \]

rule B5

Equivalent result: \[ T_{13} \leftarrow T_{12} \leftarrow T_{11} \]
Total delay during execution: 2 Ts

Figure 6.3c  Execution after Optimizing Phase
ALGORITHM C (for Phase III):

(C1) Each LDN processes transaction operations from the beginning (top) of a local schedule.

(C2) A Read operation can be processed if all conflicting Write operations positioned (not timestamped) above it have been processed. A Write operation can be processed if the Write Values for the operation are available and all conflicting Read and Write operations positioned above it have been processed.

(C3) After the execution of a transaction operation is completed, the operation is deleted from the local schedule. If an operation cannot be processed instantly, the system tries to execute the one positioned immediately below that operation.

(C4) When the processing hits the bottom of the schedule, it loops back to the top of the remaining schedule.

(C5) As soon as the execution of Write operations of a transaction is completed at a node, an ACKd is sent to the sender of the transaction operations. After the sender receives an ACKd from each of the receivers, the sender
notifies the user process of the completion of the transaction.

It is easy to implement the checking of an execution qualification for a Read or a Write operation. When a Write operation cannot be processed, the Write data items are put into a table, called the Write Wait Table. In the same way the read data items can be put into a Read Wait Table. That no Read (Write) data items of a Read (Write) operation which is trying to be executed appear in the Write Wait Table implies that all conflicting Write operations positioned above the Read (Write) operation have been processed. Similarly, that no Write data items of a Write operation which is trying to be executed appear in the Read Wait Table implies that all conflicting Read operations positioned above the Write operation have been processed. Thus, step C2 can be implemented by simply checking the two tables.

Now we can see that the basic delay for executing Algorithm C is for a Write operation to wait for the arrival of its Write Value. Such delay of a Write operation will further delay the execution of a later conflicting Read operation and subsequently will delay the Write Value of another Write operation. One way we can improve is to execute Read operations as early as possible so that the
delay time of the arrival of a Write Value for a Write operation can be minimized. The necessity of executing Algorithm B before executing Algorithm C now becomes obvious. Note that the execution of Algorithm C for a group of transactions (this phase takes more delay) and the execution of Algorithm B for the next group of transactions can be overlapped. Furthermore, as soon as the execution of the local transaction operations of a group of k transactions is completed, the execution of transaction operations for the next group of transactions can start without waiting for the completion of other transaction operations which were executed at other nodes and which were generated from the previous groups of transactions.

In Figure 6.3.c, we can see that R11a (T), R12b (X), and R13c (Y) can be executed in parallel respectively at nodes a, b, and c. Therefore, execution delay is reduced from 6 Ts in Figure 6.3.b to 2 Ts in Figure 6.3.c. Certainly, such a significant improvement in the execution delay does not always exist, if those concurrent transactions, by their nature, do not have such potential parallelism. (e.g., two transactions X := Y and Y := X cannot be executed concurrently.) Our approach will exploit such potential concurrency among transactions as long as it exists.
Assertion 2: The execution of Algorithm C on the schedule L produced by Algorithm B will generate a schedule L" which is equivalent to L and the execution of Algorithm C is deadlock-free.

Proof: Since L and L" have the same set of transaction operations and since in step C2 the execution of Algorithm C does not change the ordering relationship between any two conflicting transaction operations whether they are a Read-Write pair, a Write-Read pair or a Write-Write pair, L" is therefore equivalent to L.

In Algorithm C a transaction operation only waits for the completion of a conflicting transaction operation which is positioned above it at the same node. Even though a Write operation may wait for its Write Value from a remote node, the generation of such a Write Value will never depend on any transaction operation which is positioned below the Write operation. Therefore, there is no circular waiting and Algorithm C is deadlock-free.
6.4 Correctness of the Algorithm

In this section, we will show that our algorithm can satisfy the following three conditions:

(D1) It preserves internal consistency of the database.

(D2) It preserves mutual consistency of the multiple copy data items.

(D3) It is deadlock-free.

From the information detailed in Assertion 1, we see that schedule L generated from Algorithm B is a serializable schedule. From the information provided in Assertion 2, we see that schedule L'* generated from the execution of Algorithm C is equivalent to L. Thus, L'* is serializable and ensures internal consistency of database; D1 is true. Since every duplicated copy of a data item has the same set of Write operations working on it and since these Write operations will execute in the same order (timestamp order), mutual consistency among data item copies can thus be preserved; D2 is true. In Algorithm A, once a transaction message is put into an EWQ, it will be forced to be selected for processing sooner or later by incoming real or dummy transaction messages. Thus, there is no permanent waiting and Algorithm A is deadlock-free. Algorithm B is obviously
deadlock-free and, as can be seen in Assertion 2, Algorithm C is also deadlock-free. Therefore, D3 is true.

6.5 Comparison with SDD-1's Mechanism

In this section, we will compare our mechanism with SDD-1's solution and with the locking-based or rejection-based approach. It is certainly not fair to compare two solutions which are not designed for running in the same environment. Our comparison is made under the following term: "In case we need a concurrency control mechanism to be implemented in a local network with broadcast channels which mechanism is more desirable?" Since broadcast is an expensive activity in a point-to-point network, the comparisons described below may not be appropriate under point-to-point network environment. In a point-to-point network SDD-1's solution (or the other solution) may be a better one to use.

It is quite obvious that our mechanism can attain more concurrency than other mechanisms that use either a locking-based or rejection-based approach. In locking or rejection-based solutions conflicting and concurrent transactions are actually executed in a serial manner.
Using the same examples as in Figure 6.2.a and Figure 6.3.a, the amount of concurrency attained by using locking or rejection-based algorithms is similar to the concurrency attained by the schedules in Figure 6.2.a and in Figure 6.3.b. As we described earlier, the schedules in Figure 6.2.b and Figure 6.3.c generated by the optimizing phase can achieve much more concurrency and lower delay (e.g., from 6 Ts to 2 Ts) than the schedules in Figure 6.2.a and in Figure 6.3.b. Therefore, it is obvious that our mechanism has advantages in concurrency and delay. Moreover, our mechanism prevents deadlock, thus avoiding the overhead due to distributed deadlock detection and resolution, which is typically required in locking-based solutions. Our mechanism does not require rejection of any transaction; thus it also avoids the overhead due to the rejection and later resubmission of the transaction, which typically occur in rejection-based solutions. Therefore, our mechanism is a desirable one compared to locking or rejection-based solutions.

The following comparison with SDD-1's solution which uses a timestamped approach is an informal one by using some examples to illustrate the amount of concurrency allowed and the delay incurred in both mechanisms. In the comparison, we assume that all transactions in each example enter the
DDBS close together in time. In SDD-1, transactions are classified by the database administrator into classes, and a conflict graph analysis is conducted at database design time to reduce run-time synchronization.

By analyzing the conflict graphs of transaction classes, the synchronization required for each transaction class with respect to other transactions can be categorized into three types of protocols P1, P2, and P3. There is also a protocol P4 to handle non-predefined transaction classes.

In the implementation of protocols, any two conflicting Read and Write transaction operations are executed in their timestamp order at Data Modules [BER80]. However, the timestamp (TS) of a Read operation of a transaction which obeys P1 and P2 can be smaller than the transaction timestamp; thus, the execution of a Read operation does not need to wait as long as it once had to do if the TS had to be equal to the transaction timestamp which is the case of protocol P3. There is a danger that a Write operation timestamped later than TS has been processed before the Read operation is received. In such a case, the Read message which carries the Read operation must be rejected and all Read messages issued for the transaction must be resubmitted with a higher TS.
CONCURRENCY: If Read operations of a transaction conflict with Write operations of another at only one Data Module, the concurrency attained in both algorithms is very similar. However, if they conflict at two or more nodes, the SDD-1's solution cannot always achieve the same degree of concurrency as ours. For example, in Figure 6.2.a, transaction operations of T31 and T32 conflict with each other at both nodes a and b. From the protocol selection rule of SDD-1, transaction T32 should obey protocol P1 with respect to transaction T31. If the TS of R32b (Y) (R32a (Y')) is selected to be smaller than W31b (Y) (W31a (Y')), R32b (Y) (R32a (Y')) may be executed before W31b (Y) (W31a (Y')) which is the same as ours as shown in Figure 6.2.b. Unfortunately, there is a chance that W31b (Y) has been processed when R32b (Y) arrives; then R32b (Y) and R32a (Y') have to be rejected. In order to reduce the possibility of being rejected, the TS of R32b (Y) would prefer to be larger which consequently increases the delay of executing the Read. When TS of R32b (Y) is chosen to be not smaller than that of W31b (Y) (e.g., the same as the timestamp of T32), the execution sequence will come out to be the same as in Figure 6.2.a. Apparently, only in the best case in which the timestamp is chosen to be as small as possible (thus achieving best concurrency) without forcing the Read to be rejected, the amount of concurrency and
synchronization cost are close to ours. In fact, it is difficult to achieve this without extra overhead. In other cases, either the amount of concurrency or the overhead incurred from rejection will be worse than in Figure 6.2.b.

For protocol P2 a similar fact can be observed. In Figure 6.3.a, transaction T13 should obey P2 with respect to T11 and T12. At node c, the execution sequence of the three transaction operations can be any of the following cases depending on how TS of R13c(X, Y) is chosen (in SDD-1, R13c(X, Y) is the combination of R13c(X) and R13c(Y)): W11c(X), W12c(Y), R13c(X, Y) which has the worst delay; W11c(X), R13c(X, Y), W12c(Y); and R13c(X, Y), W11c(X), W12c(Y) which has the shortest delay. A discussion similar to one above can be applied here: the shortest delay case has the largest possibility so that the R13c(X, Y) is rejected. However, our mechanism can ensure the shortest delay case without any possible rejection overhead.

For transaction obeying protocol P3, the same concurrency can be obtained in both solutions. However, the Read message in SDD-1 still has a chance of being rejected. There is another important aspect: in some cases, the transaction which must obey protocol P3 in SDD-1 actually
can obtain Pl's concurrency in our algorithm. For example, in Figure 6.3.a, transaction T12 must obey P3 with respect to transaction T1; therefore, R12b(X) must be executed after W11b(X) as the case in Figure 6.3.b. However, in Figure 6.3.c, our mechanism can allow R12b(X) to be executed before W11b(X), which looks as if T12 were executed according to protocol Pl. If in SDD-1's solution, R12b(X) is executed before W11b(X) which implies T12 < T11 and if the sequence W11c(X), R13c(X, Y), W12c(Y) occurs at node c which implies a path T11 < T13 < T12, then T11 < T12 would contradict T12 < T11. However, since our algorithm ensures the sequence R13c(X, Y), W11c(X), W12c(Y), such a problem would not occur. Therefore, some transaction classes which satisfy protocol P3's protocol selection rule in SDD-1 may just require Pl's synchronization cost in our mechanism without violating serializability.

DELAY: Since SDD-1's algorithm cannot obtain the same degree of concurrency without rejection overhead, the average delay for completing a transaction is worse than ours. In order to have the same amount of concurrency as in Figure 6.2.b and Figure 6.3.c, SDD-1's solution has a good chance of having a Read message rejected. Rejection of a Read message of a transaction causes resubmission of all Read messages of the transaction and thus increases
Internode communication traffic and delay. Note that the delay in the transmitting phase of our solution also occurs in SDD-1's solution in which a Read message is held until a Write message or a NULLWRITE message timestamped later than TS arrives from every transaction class in the read condition [BER80].

**OTHER ISSUES:** SDD-1's approach requires static preclassification of transaction classes that may be good only for some kind of applications. Once a new transaction class needs to be included, the conflict graph has to be reanalyzed. For transactions not requiring any synchronization among them (e.g., queries) and falling into the same transaction class, the class pipelining rule requires them to execute in a serial in SDD-1's approach; however, our mechanism can execute them in parallel.

For Read only transactions (queries) our mechanism can always move those Read operations to the top of local schedules and allow them to be executed before other concurrent update transactions; thus, queries can have a good response time. Transactions of this type are believed to constitute a major part of input transactions and their response time is crucial.
6.6 Summary

In this chapter, a concurrency control mechanism for a partially duplicated distributed database is presented. This mechanism uses distributed control, does not set locks, does not reject transactions, and prevents deadlocks. This mechanism can not only reduce communication traffic and delay, but also exploit potential concurrency among concurrent transactions; thus, it compares favorably to that of others. This mechanism can also be robust with respect to node crashes and link failures. Major ideas of handling a crash recovery for the mechanism are the topics of the next chapter. Our solution is simple both in concept and in implementation, which we believe is an important advantage especially for a low-cost local computer network like DDLCN.
7.1 Introduction

In the previous chapter, we have dealt with the operations of the concurrency control mechanism under normal conditions. In order to assure the continuous and correct operation of the DLDBS under abnormal conditions, reliability mechanisms have to be considered. By abnormal cases, we mean cases of node crashes and communication link failures.

In considering the reliability in the DLDBS, we actually deal with not only the reliability mechanism for distributed concurrency control, but also the reliability mechanism for distributed query processing. Since the operations of a distributed concurrency control and a distributed query process intermix as a whole, the
uninterrupted services of the DLDBS in spite of crashes and failures must depend on the reliable operations of both. Note that this is different from the discussion in Section 5.4 in which the problem of distributed query processing does not exist at all.

This chapter's organization begins with Section 7.2 which addresses the case of communication link failure. In Section 7.3.1 and Section 7.3.2 we describe some facilities to enforce atomic operations in a distributed environment. In Section 7.3.3 we present the procedures to withdraw a crashed node. In Section 7.3.4 we present the procedures to reinstate a repaired node. Then, Section 7.4 concerns the reliable operations in distributed query processing. In Section 7.5 we examine the reliable operations in distributed concurrency control. In the last section we present some arguments to show that the combination of those techniques proposed in the earlier sections can make the transaction processing in DLDBS resilient to both communication link failures and node crashes.
7.2 Communication Link Failures

As long as the communication link failures do not cause a node to be isolated (partitioned) from the network, such failures have no effect on the normal operations of the algorithm. (Such failures certainly affect the normal operation in the subnetwork.)

If the communication link failures do cause the network to become partitioned, the partition which has a majority of nodes in the network still can continue operating and treats the nodes in the other partition the same as crashed nodes. Using the recovery algorithm for node crashes to be described below, the network can return to a consistent state after the partitions are reconnected. In [BAD78], some efforts have been tried to allow all partitions to continue operating. The solution rests on keeping the local logs while each partition is executed on potentially inconsistent data. Network partitioning may be a significant problem depending on the topology of the network. In some topologies, even a single communication link failure could cause network partitioning. However, in the DDLCN, network partitioning is a rare event due to the double loop communication links and the tri-state control mechanism built into the interface.
7.3 Node Crashes

This subsection will show how the algorithm can continue operating in the case of one or more node crashes and how the distributed database recovers from anomalies and is led to a consistent state when a crashed node has been repaired.

7.3.1 Reliable Broadcast

As we can see in Figure 7.1, after a transaction is received at an LRN, a transaction message is created and is broadcast to other Read Nodes and Write Nodes. However, if the sender LRN crashes during the broadcasting, the broadcast message may have been sent to some LDNs but not to all. In order to guarantee that a broadcast message will reach either every destination or none, a reliable broadcast facility is required. Our idea to implement a reliable broadcast takes advantage of the fact that the broadcast protocol in our algorithm is a "one-message-at-a-time" protocol. We require that every node keep a Last Transaction Array to record the most recent transaction received from every other node. Thus, a partial broadcast can be converted into a complete broadcast if the sender...
Figure 7.1  Algorithm Protocols
crashes during the broadcasting. The details can be seen in Section 5.4.3.2.

7.3.2 Reliable Update

Assume that the distributed query processing finishes successfully. If the transaction is an update transaction, a series of messages carrying Write Values are created and are sent to appropriate Write Nodes. If the sender LRN crashes before all messages carrying Write Values are sent out, it is possible that only some Write Nodes, but not all, will receive the Write Values and perform the updates. Thus, only partial results of the update transaction record in the database. Consistency of the database is destroyed.

In order to avoid having partial results of an update transaction left in the database, all Write operations of a transaction must act like an atomic operation (either all Writes are done or none of them are done). Our approach to resolving this problem is based on a scheme known as two-phase-commit protocol [GRA78]. The central idea of the protocol is that a transaction is made atomic by being performed in two phases. A brief description of Gray's two-phase-commit protocol is given as follows:
**Phase I:**

A commit coordinator sends a REQUEST COMMIT message to each cohort. (Cohorts are a collection of processes each of which executes on different nodes. A commit coordinator is a process which communicates with all cohorts participating in a transaction. These are the terms used by Gray.) The commit coordinator then waits for a reply from each cohort. If the coordinator receives an ABORT message from any cohort, it broadcasts ABORT to all cohorts, writes an ABORT entry in the log (a log is also called an audit trail or a journal) and terminates. If the coordinator receives an AGREE message from every cohort, it starts Phase II.

A cohort receives a REQUEST COMMIT message from the commit coordinator. After the cohort completes requested work, successfully writes an UNDO-REDO log, and also writes an AGREE COMMIT entry in the log, it replies with an AGREE message to the commit coordinator. Otherwise, it replies with an ABORT message to the commit coordinator.
Phase II:

After the commit coordinator receives an AGREE from every cohort, it writes a COMMIT entry in the log and broadcasts a COMMIT message to each cohort. If the coordinator receives a positive acknowledgement from each cohort, it writes a COORDINATOR COMPLETE entry in the log and terminates.

After a cohort replies with an AGREE or ABORT message, it waits for a verdict from the coordinator. If the verdict is ABORT, the cohort undoes the transaction; if the verdict is COMMIT, it releases locks and resources and replies with a positive acknowledgement.

In order to know the details of the protocol and to see how system recovery can take advantage of the protocol in the event of restart, the reader can refer to [GRA78].

If a commit coordinator in Gray's two-phase protocol crashes before every cohort participating in a transaction receives from the coordinator a COMMIT or an ABORT message, the decision whether to commit or abort the transaction on those uninformed cohorts is suspended until the commit coordinator recovers (when it recovers, if COMMIT is
recorded in the log, the commit coordinator rebroadcasts a COMMIT message. Otherwise, it broadcasts an ABORT message. Since the database services still continue in spite of the crash, many transactions received later which conflict with the suspended transaction will be aborted or be deferred.

The implementation of a reliable update protocol to perform reliable updates in DLDBS will be different from Gray's in such a way that the decision to commit or abort a transaction will be made as soon as a node crash occurs. The rules are listed as follows:

For a Sender:

Phase I

1. Send a message consisting of Write Values and a commit request to each active Write Node of a transaction.
2. Wait for a reply from each active Write Node.
3. If an ACK message is received from every active Write Node (it means every active Write Node is ready to commit), start Phase II.
Phase II:

1. Broadcast a COMMIT message to all active Write Nodes.
2. Wait for ACKd from each active Write Node.
3. If an ACKd message is received from every active Write Node, send a DONE message to the user process.

For an Active Write Node:

Phase I

1. Wait for a message consisting of Write Values and a commit request from the sender.
2. After performing the Write operation, the Write Values of the Write operation are recorded in a temporary file.
3. If successful then send an ACK message to the sender.
Phase II:

1. Wait for a verdict from the sender.
2. If the verdict is ABORT, dispose of the temporary file; thus, no update results are seen in the database.
3. If the verdict is COMMIT, commit the temporary file to the permanent database. Send an ACKd to the sender.

The reliable update protocol may not be completed in the event of a node crash happening either at the sender or at the receiver ends. Recovery procedures for this protocol must be specified:

Recovery Procedures:

1. If a Write Node crash occurs at either Phase I or Phase II, the recovery procedures described in Section 5.4 3.3 and Section 7.3.3 are invoked to buffer Write operations and Write Values in a recovery array. The active Write Nodes are the Write Nodes whose node ID is shown in the up-list.
Since a sender only waits for a reply from an active Write Node, the decision to commit or abort a transaction will not be deferred by a Write Node crash.

2. If the sender crashes before completing step 1 of Phase II (i.e., before the broadcasting of a COMMIT message is completed), each Write Node of a transaction does as follows:

(a) After the write node detects the failure of the sender node it will either:
   (i) broadcast a COMMIT/ABORT message to other active nodes if it has received a COMMIT/ABORT message from the sender.
   (ii) wait for a long enough finite time interval if it has not received a COMMIT/ABORT message. If no COMMIT/ABORT message arrives within the time interval, it broadcasts an ABORT message to other active nodes.

(b) Any node which receives an ABORT will abort the temporary update results.

3. If the sender crashes after completing step 1 of Phase II, the unreceived ACKd messages from Write Nodes will be buffered in a recovery array as
described in Section 5.4.3.3.

In the discussion above, it can be seen that the decision to commit or abort a transaction can always be made as soon as a crash occurs (no matter whether it is a sender node crash or a receiver node crash). The combination of the reliable update protocol, its recovery procedures, and the recovery procedures of the concurrency control mechanism described in Section 5.4.3.3 and Section 7.3.3 guarantees atomic reliable transaction updates in the event of either a sender or a receiver node memory loss crash. Arguments showing the correctness of this are given in Section 7.6.

7.3.3 Withdrawal of a Crashed Loop Node

This subsection presents the procedures to withdraw a crashed Loop Node from the network. Since Write operations are executed according to timestamp order and their timestamps are globally unique, the same procedures as described in Section 5.4.3.3 can be applied here. A Loop Node is a physical node which serves as either an LRN (a sender) or an LDN (a Read Node or a Write Node). If a node (Sd) detects that a crash happened on a node (Sf) after Sd tried to send a message to Sf and eventually concludes that
Sf is unreachable, the withdrawal procedures are invoked as follows:

**If the Crashed Node Is a Sender of a Transaction:**

1. Reliable broadcast is initiated.
2. Node Sd deletes node Sf from its local up-list and notifies other nodes of the crash.
3. Any other active node which recognizes the crash deletes node Sf from its local up-list and aborts or commits the transaction as described in Section 7.3.2 and Section 7.4.

**If the Crashed Node Is a Receiver of a Transaction:**

1. If the receiver is a Read Node, the sender (Sd) of the transaction aborts the transaction following the rules in Section 7.4. If the receiver is a Write Node, the sender (Sd) of the transaction buffers unacknowledged Write operations in a recovery array as described in Section 5.4.3.3.
2. Node Sd deletes node Sf from its local up-list.
3. Other active nodes delete node Sf from their local up-list and repeat step 1.
7.3.4 Reinstatement of a Repaired Node

The procedures to reinstate a repaired node are the same as the procedures in Section 5.4.3.3. We outline them as follows:

For a Recovery Node:

1. Request and process its recovery array.
2. Request the second version recovery array and the up-list.
3. Broadcast the "I-am-up" message to all nodes.
4. Wait for transaction messages from other nodes and synchronize CNT value.
5. Resume the normal operation in the network.

For Any Active Node:

1. When receiving an "I-am-up" message, update its local up-list.
2. Consider Sf as having rejoined the network.
7.4 Reliability of Query Processing

In Section 7.3.3, we have mentioned that the transactions which have not been completed at the crash moment will go forward either to be committed or to be aborted. This subsection presents the conditions to commit or abort a transaction if the query processing of the transaction is not completed when the crash occurs.

Failure of a Read Node

If any Read Node of a transaction fails before the query processing of the transaction is completed, the transaction is aborted by the sender of the transaction. The aborted transaction may be restarted if the data being read by the Read operation at the crashed node is available at other nodes.

Failure of a Sender

If the sender of a transaction fails before the transaction query processing is completed, the transaction is aborted with an ABORT message being broadcast by a Read Node or a Write Node. The Read Node has a Read operation of the transaction unprocessed. The Write Node has a Write
operation of the transaction and it does not receive a commit request from the sender.

7.5 Reliability of the Concurrency Control Mechanism

A transaction processing conceptually consists of the following three steps of operations: read, compute, and write. If a crash happens at a Write Node of a transaction before the Write step starts (before the reliable update protocol starts), the sender of the transaction can buffer the Write operation at the crashed node according to the rules described in Section 5.4.3. The case in which the crash happens at a Read Node or at the sender before the Write step starts is described in Section 7.4.

If a crash occurs after the Write step of the transaction starts, the reliable update protocol and its recovery procedures can abort, commit, or buffer Write operations according to the rules in Section 7.3.2.

The DLDBS maintains its services in spite of a node crash; however, not all transaction requests can be honored. After a crash occurs, an LDN will reject or process a newly arrived transaction request according to the
following rules:

1. If all Write Nodes and Read Nodes are available, the transaction is processed as usual.
2. If all Read Nodes are available and some Write Nodes are not available, the transaction is processed, and those Write operations whose Write Nodes are not available are buffered in a recovery array.
3. If some Read Nodes of the transaction fail, the transaction is rejected.

The selection Rule A3 of Algorithm A in Section 6.3 is also changed to consider only transaction messages from active nodes.

7.6 Correctness of the Reliability Mechanism in DLDBS

The design of a reliable distributed database is a complex task. It involves many difficult design issues that must be faced by the designer. Although the problems that the designer encounters may be varied due to the different concurrency control mechanisms which the different systems adopt, the problems that we have discussed so far (e.g. how to enforce reliable broadcast, how and when to commit or to abort a transaction, how to buffer transactions, how to
withdraw and to reinstate a node) are quite essential. In this section, we present some arguments to show that the combination of those methods proposed in previous sections can make the transaction processing in the DLDBS resilient to both communication link failures and node crashes. Also, we hope the arguments may give the reader a specific and clear picture about how a transaction is handled if a crash occurs before the transaction processing is totally completed.

In Figure 7.2, we consider 15 cases. A processing transaction may be in some of the 15 cases when a crash happens.

A. In Case The Crash Node Is a Sender

Case 1: A crash happens before a transaction is broadcast. After recovery, all information about the transaction is lost, and the user process must send the transaction again.

Case 2: A crash occurs when a transaction broadcast is going on. The reliable broadcast discussed in Section 5.5.2.1 can ensure that the transaction message will be received by all destinations or none at all. The condition is necessary for a
Figure 7.2  15 Cases When a Crash Happens
value assignment type of transaction.

Case 3: A crash happens before all Read Values are collected at the sender node. In Section 5.5.2.5, the Read Nodes which have a Read operation will broadcast ABORT to abort the transaction.

Case 4: A crash happens before the computation of the transaction is completed. In Section 5.5.2.5, the Write Nodes which expect a commit request will broadcast ABORT to abort the transaction.

Case 5: A crash happens before ACK is received from every Write Node. In Section 5.5.2.2, the Write Node which expects COMMIT will broadcast ABORT to abort the transaction.

Case 6: A crash happens before all ACKds for the transaction are received from Write Nodes. In Section 5.5.2.2, recovery procedures of the reliable update protocol guarantee that the COMMIT/ABORT message is delivered to either all Write Nodes or none at all. In Section 4.4.4.2, since the ACKd information is stored in stable storage, those ACKds received before the crash survive and those ACKds generated during the crash are buffered in a recovery array and will be available to the recovering node.

Case 7: A transaction processing is completed. In this
case a crash has no effect on it.

B. In Case The Crash Node Is a Receiver

Case 8: A crash happens before the transaction message is received by the crashed Read Node. The sender of the transaction will abort the transaction.

Case 9: A crash happens before the Read Value at the crashed node is sent to the sender. The sender of the transaction will abort the transaction.

Case 10: A crash happens before the transaction message is received by the crashed Write Node. The unreceived transaction will be buffered in a recovery array by the sender and will be received by the crashed node after the node recovers.

Case 11: A crash happens after the transaction was received but before the Write Value and the commit request are received. The Write operation at the crashed node will be buffered in a recovery array by the sender or be aborted if the transaction is aborted.

Case 12: A crash happens before the sender receives an ACK for a Write operation from the crashed node. The sender will buffer the Write operation in a recovery array.
Case 13: A crash happens before a COMMIT is received by the crashed node. Since the sender node cannot receive an ACKd for this COMMIT, it buffers the Write operation at the crashed node in a recovery array.

Case 14: A crash happens after a COMMIT is received but before the commit action is done and the ACKd is replied. Since the sender cannot receive an ACKd from the crashed node, it buffers the Write operation at the crashed node in a recovery array.

Case 15: The result of a Write operation has been committed to the physical database. In this case a crash has no effect on it.

7.7 Summary

In this chapter, we presented a reliability mechanism to assure the continuing and correct operation of a partially duplicated DLDBS in cases of communication link failures and node crashes. We started this chapter by specifying the procedures to enforce the reliable broadcast, to enforce atomic operations of a transaction, to withdraw a crash node, and to reinstate a repaired node. Then, we concern ourselves with the reliable operations of
distributed query processing and of distributed concurrency control. Detailed arguments to show that those techniques proposed in this chapter can provide reliable transaction processing in DLDBS are also given.
CHAPTER 8

DATABASE DISTRIBUTION AND DISTRIBUTED DATABASE ARCHITECTURE

8.1 Introduction

In the previous chapters we have proposed solutions to two technical problems: distributed concurrency control and crash recovery for implementing DLDBS. In this and the next chapters we will concern ourselves with two design decisions: determining a distributed database architecture by incorporating database distribution (Chapter 8) and database directory management (see Chapter 9).

In this chapter we first design a distributed database architecture for DLDBS by incorporating database distribution, and then a methodology to do database decomposition is described. This chapter begins with Section 8.2.1 which briefly describes a four-level database architecture in a centralized environment. Based on the
centralized database architecture, four types of distributed database architecture are considered in Section 8.2.2. These kinds of architecture are created by incorporating database distribution at the different levels of the centralized database hierarchy. In Section 8.3 the trade-offs among the four database architectures are compared. The distributed database architecture where databases are distributed at the conceptual-view level is selected for DLDBS. Finally, in Section 8.4 the basic components of database distribution are described. Two partition operators to allow a user to do database decomposition are also introduced.

8.2 Design Consideration of Distributed Database Architecture

In a distributed database environment, a user at a host computer is not restricted to control local data only; the user is able to gain access to global data in the network as well. In order to support the idea of providing global data to a user, the following two alternatives are possible:
(1) Although a user at a node can view a logically integrated database, the entire database is actually located at every node. This is the fully duplicated distributed database case. Apparently this approach is rather expensive and impractical unless the database is of small size.

(2) Although a user at a node views a complete database, the database is actually distributed at different nodes. If the distributed database allows parts of the data to be duplicated at some nodes, it is a partially duplicated distributed database. In this case the locations of data distribution and duplication are invisible to users. Users enter queries and updates into the distributed database system in the same way as they would in a non-distributed system.

DLDBS is designed to support partially duplicated distributed databases because much of the promises of a distributed system such as reliability, accessibility, and flexibility can be achieved only through duplicating data. In designing such a system, a significant problem is determining at which level the database should be partitioned (to partition databases at the physical database level may not always be a good idea). Then, a distributed
database architecture can be determined by incorporating such a database distribution.

In order to determine a feasible architecture for DLDBS, alternative architectures have to be identified, compared, and evaluated. The remaining discussion and the comparison are based on the DBMS framework proposed in the well-known ANSI/X3/SPARC report [TSI78]. In Section 8.2.1 we briefly review the centralized database architecture proposed in the report. In Section 8.2.2 we extend the centralized database architecture into a distributed environment. We identify several distributed database architecture alternatives created by incorporating database distribution at different levels of the centralized database hierarchy. We explain their meaning and compare their trade-offs.

8.2.1 ANSI/X3/SPARC Database Architecture

The ANSI/X3/SPARC study group has proposed a framework for the potential standardization in the area of database management systems. A simplified schematic view of ANSI/X3/SPARC architecture is depicted in Figure 8.1 which identifies three levels of view for the database. The
Figure 8.1  Database Architecture - a Reference Structure
meaning of the three levels are described in Section 2.2.3. The external view defines multiple application views of the database. The conceptual view defines the entire information content of the database. The internal view is an inside representation of the entire database. Below the three-level database architecture, a file level may exist to actually access the data. The file level may be supported by a conventional file system in an operating system.

The ANSI/X3/SPARC database architecture provides an important model for centralized database systems. It can also play a key role in the distributed database systems. A natural question for a distributed database designer is how to distribute the database at various levels of the ANSI/X3/SPARC architecture for a computer network. Using the architecture in Figure 8.1 as a reference structure, the distribution of the database at different levels leads to several distributed database architectures. These architectures are created by incorporating database distribution at the external-view level, at the conceptual view level, at the internal view level, and at the file view level. In the next subsection we will have a short evaluation of each.
8.2.2 Alternatives of Distributed Database Architecture

Database distribution at the external view level

In Figure 8.2 a distributed database architecture where the database is distributed at the external-view level is presented. In this type of distributed database, the location distribution of database fragments are explicitly known to a user. The user has a global external view of the distributed database; however, the distributed database management system does not support the global external view. Therefore, in order to retrieve (query) remote database data, the user has to do the following steps himself:

1. Decompose the query into several subqueries, each of which refers to data located at a single node.
2. Determine the execution order of these subqueries.
3. Specify the location of a remote host and access remote data located on the host.
4. Receive and buffer the partial results of the subqueries.
5. Do operations on partial results of subqueries to produce the final results of the query.
User Global View

LEV

LCV

LIV

LFV

Data

LEV

LCV

LIV

LFV

Data

LEV : Local External View
LCV : Local Conceptual View
LIV : Local Internal View
LFV : Local File View

Figure 8.2 Distribution at External View Level
In this type of architecture a complete conventional database management system without modification is included at each node. Several computers each having its existing database system are connected to provide distributed database services. This approach is an easy way to create a distributed database where portable database management systems are available.

However, according to the definition of a distributed system provided by Enslow [ENS78], this type of database system is not a truly distributed database system. This is because it violates one of the properties essential to a distributed data processing system: system transparency. Another disadvantage of this approach is that it is difficult to achieve cooperative autonomy, another essential property of a distributed system. The architecture shown in Figure 8.2 lacks a platform to achieve cooperation among independently developed local database systems. Thus, a collection of independent systems cannot cooperate to produce a coherent system.
Database distribution at the conceptual view level

In Figure 8.3, a distributed database architecture where the database is distributed at the conceptual view level is illustrated. The key concept of this architecture is that the distribution of data is related to the view of an enterprise. In most applications, the geographical localization of data has a conceptual meaning. For example in a library database the books whose subjects relate to engineering science are typically located in the engineering library. Consequently, the physical database can be distributed in a meaningful way by using the localization properties of objects as the criteria.

The Global Conceptual View (GCV) in Figure 8.3 is the total integration of the local conceptual view of the enterprise database at each node. Since the Global Conceptual View provides a global view of the distributed database to the external-view level, the external view turns out to be a global external view; thus, all users' views are global. The Global Conceptual View (GCV) allows a user to access the distributed database as he would access a non-distributed database. The user does not need to make a distinction between remote data and local data. This is an important difference from the previous architecture.
GCV: Global Conceptual View

Figure 8.3  Distribution at Conceptual View Level
**Database distribution at the internal view level**

In Figure 8.4, a distributed database architecture where the database is distributed at the internal-view level is illustrated. In this architecture not only is the storage structure of the database an object to be distributed, but also the access paths may be considered as objects of distribution. Since all access paths existing in each local database are available at the network level, the access optimization may be achieved.

**Database distribution at the file view level**

In Figure 8.5.a a distributed database architecture where the database is distributed at the file-view level is illustrated. In this architecture, the whole distributed database system is actually based on a distributed file system which provides a global file-view level (see Figure 8.5.b). Objects of distribution are files, pages, or records. The locations of distributed files are transparent to users of the distributed file system. There are some advantages of putting the distributed database on top of the distributed file system. A major attraction would be that, in many cases, a distributed file system is a necessary part of a distributed operating system (such as RSEXEC [TH073])
GIV : Global Internal View

Figure 8.4    Distribution at Internal View Level
GFV : Global File View

Figure 8.5.a Distribution at File View Level
Figure 8.5.b  Distributed Database System Based on Distributed File System
and NSW [CR075] network operating systems). In case we try to implement a distributed database in a distributed system where the distributed operating system already exists, this architecture can simplify the design efforts because all problems of data distribution are already solved by the underlying distributed file system. However, this approach is not an efficient one. The results of this approach would be the increase in communication data traffic and the decrease of parallelism in transaction processing. The reasons are described in the next section.

8.3 Selection of a Database Architecture for DLDBS

Following the discussions in previous sections, we are now in a position to select one architecture for DLDBS. Since the first architecture where a database is distributed at the external-view level fails to represent a truly distributed database system it will not be given our consideration.

In making a choice between the second architecture where a database is distributed at the conceptual-view level and the third architecture where a database is distributed at the internal-view level, we feel that the second
architecture is a promising one. Some reasons are given below:

(1) **Naturality** To associate data distribution with conceptual meaning in the real world is a natural way to distribute a database. In real applications this can be verified from a management point of view (e.g. engineering school librarians are experts in managing books and documents relating to engineering science) and from efficient utilization of resources.

(2) **Portability:** The second architecture is easier for expansion/shrinkage of a distributed database or incorporating of an existing portable local database. This is an important aspect in a mini/micro computer network.

(3) **Adaptability:** Since most existing database management systems enable a qualified user to design a database at the conceptual level, the second architecture can thus take advantage of existing facilities. Since each local subset of a distributed database is designed based on data models and access methods supported by a local DBMS, this architecture results in good adaptability when incorporating a new local part into a global system.

(4) **Autonomous Management:** In the second architecture, each local enterprise has the right to design a local
enterprise database. A global enterprise administrator has the duty to coordinate distributed local enterprise administrators. Thus, management burden can be distributed into different locations. Local autonomy is an important promise in distributed systems, because many administrative as well as technical decisions tend to be local.

In order to make comparisons between the second architecture and the fourth architecture where the database is distributed at the file view level, it is necessary to understand the difference between a file system and a database system. A database system implements the data abstraction concept to the physical data. It gives users with a logical view of the database (logical database). The record fields and the relationship between record types are known by the database system (defined in schema and mappings). Users query the logical database according to record field contents. The database system can efficiently access a set of records whose contents satisfy the query by following the access paths specified by secondary indices.

A file system also accesses records. However, the record fields and the relationship between different record types are unknown to the file system. They are written in
the logic of the application programs which access the records using the file system. Records in a file are typically identified by their physical contiguity (e.g., sequential file) or by an unique key (e.g. index sequential file). Only single-record accesses are possible via the unique record key. For example, in order to answer the query, "Find all students whose age is 20," applied on a student relation file, an application program has to examine sequentially every record in the student file and to collect all students whose age is 20. If the student file is a keyed file, many file accesses are issued. This is because the file system can do single-record accesses only. In a file system, the lock granule is typically the entire file. No concurrent updates to a file are allowed. In the database system, a lock granule can be an individual record.

In the fourth architecture, the distributed database system is built on top of a distributed file system. The problems of data distribution are solved by the underlying distributed file system (the database system is not aware of the database distribution). In such a system, transaction processing can be performed in one of the following two ways: (1) Transfer all necessary relation files to the transaction processing node by means of file transfer facilities in the distributed file system. By this, a large
amount of unnecessary communication cost and delay are generated. A huge file may be transferred just because of accessing a single record. Also, parallel processing of transaction operations is reduced. (2) Translate the transaction to the global file level and perform any possible remote file access. The remote file access may be processed very inefficiently because the distributed file system only provides single-record access via the record key. A single-record remote access involves the transmission of an access command and the return of a single record. In a bad case, if the secondary indexing is not sufficient for processing a query in the database system at the upper level, a large number of records must be checked sequentially. Each of these checks results in the sending of a command message and receiving of a single record. Thus, a large amount of message transmissions and delay are incurred. The communication cost and time delay may seriously affect the performance and the response time of the system.

Distributing a database at the conceptual-view level can overcome the problems mentioned above. In this architecture, a user's request is translated into several conceptual requests (subqueries). All requested data located at the same node can be accessed by only one
conceptual request. A set of requested data records are returned in one message. In addition, if the distributed database is built on top of a distributed file system, the whole database management system has to be fully duplicated at every node. In contrast, if the database distribution is at the conceptual level, only parts of the database management system which are related to local physical data are stored locally.

8.4 Database Distribution in DLDBS

From the discussion in the previous section, we can see that the database architecture where the database is distributed at the conceptual-view level is adopted in DLDBS. The conceptual data model supported by DLDBS is relational because of its flexibility and its structural simplicity in defining external and conceptual schemata. The relational model also makes the partition and distribution of the database naturally defined. In order to distribute a conceptual database, a methodology to do systematic database decomposition is necessary. In DLDBS, a database relation can be decomposed into fragments (subrelations) by means of two types of partition rules described in [CHA77]: vertical partition and horizontal
partition (in the reverse, fragments can be composed into a relation). In Section 10.2 a data definition language is also specified to allow users (or enterprise administrators) to create fragments distributed in the DDLCN.

8.4.1 Basic Components of Database Distribution

The basic components of database distribution and allocation are elementary fragments, the finest subrelation which will not be further decomposed. An elementary fragment which is viewed by the DBMS to be local is a local fragment. Otherwise, it is a virtual (non-local) fragment. Local fragments and virtual fragments are logical fragments, the basic units of the distribution of conceptual databases to Loop Data Nodes (LDNs). A logical fragment is either entirely present or entirely absent at each LDN.

A logical fragment may be stored redundantly at more than one LDN in a partially duplicated DLDBS. The physical storage of a logical fragment is a physical fragment, a subrelation which is actually stored in a storage medium. Note that the distribution of a database into logical fragments and the actual assignment of physical fragments are totally transparent to application users who access the database.
8.4.2 Database Distribution Operators

The distribution of conceptual databases and physical fragment assignments begin with the partition of relations in the database into logical fragments. Each logical fragment is a subset of a relation created by applying iteratively the following two partition operators [CHA77]:

1. **Horizontal Partition Operator** HP

   HP : R1 -> (R11, ..., R1n)

   where R1 is a global relation at the conceptual-view level. Rli is a horizontal partition defined by the distribution-predicate i and the value in an attribute (partition-key attribute). The distribution-predicate is a simple boolean expression of the form: attribute-name rel-op constant, where rel-op is >, =, <, etc. The Rli's must satisfy the following two conditions:

   (a) The Rli's have the same attribute set.

   (b) The values in the partition-key attribute of Rli's are mutally exclusive.

For example, HP : R1(a1, a2, a3, a4, a5) -> (R11, R12)

   R11=R1(a1, a2, a3, a4, a5) where a4 = 'EE'

   R12=R1(a1, a2, a3, a4, a5) where a4 = 'CS'
(2) **Vertical Partition Operator VP.**

VP : R1 -> (R11, ... , R1n)

where R1 is a global relation at the conceptual-view level, and R1i's are vertical partitions and satisfy the following conditions:

(a) The set of non-key attributes in any partition is mutually exclusive with the set of non-key attributes in other partitions.

(b) Every partition maintains the same set of primary key values. The order (ascending, descending, or unsorted) of the primary key values in every partition is also the same.

For example, VP : R1(a1, a2, a3, a4, a5) -> (R11, R12)

R11 = (a1 a2, a3, a4)

R12 = (a1, a5)

The iterative application of partition rules results in a tree-like structure (see Figure 8.6).
Figure 8.6  Horizontal Partition Followed by Vertical Partition
8.5 Summary

In this chapter we extend the ANSI/X3/SPARC database architecture into a distributed environment. Four types of distributed database architecture are considered and compared. The distributed database architecture, where databases are distributed at the conceptual level, is selected for DLDBS. Then the basic components of database distribution are described. Two partition operators to enable a user to do database decomposition are also introduced.
9.1 Introduction

In the previous chapter we described several alternatives for incorporating database distribution into centralized database architecture and thus created several distributed database architectures. The architecture where databases are distributed at the conceptual-view level is suggested for the Distributed Loop Database System (DLDBS). In this architecture, the DLDBS is considered to be an aggregation of several autonomous local databases. However, a user has the illusion that a complete and non-distributed database is resident at the local node. The distribution and replication of data are invisible to users. The objective of relieving users of the need to be aware of distributedness is similar to goals pursued in other distributed systems developed elsewhere (e.g., DCS [FAR72],
In modern database management systems, an important component called the database directory/dictionary is employed [TSI78]. A database directory contains a repository of information about the database. This repository of information is used to assist both the management of the installation's database and also the production of application programs which operate on the database. The database directory is considered a meta-database from which a DBMS processor fetches information required by its execution. A database directory typically contains the following information:

1. At a minimum, it contains database schemata and mapping definitions between different levels of schemata (external, conceptual, internal). The database schemata actually describe the meanings, relationships, and formats of the database data. The mapping specifies the correspondence between different levels of schemata. The correspondence includes name association, data type conversion, reordering, encryption, etc.

2. It contains usage statistics of various data elements.

3. It contains access control procedures, protection declarations etc.
(4) It contains restart and recovery procedures, accounting and auditing data, etc.

In a distributed database system we must have an additional directory of information which specifies the location of each physical fragment of the database. This location directory provides the network-level access paths which direct the transaction to the set of nodes containing the requested data of the transaction. The location directory implements the data independent concept in such a way that the location change of a data fragment has no impact on the global view of the database.

9.2 Directory Management Alternatives

A DBMS processor must fetch information from the database directory in order to translate a user request, to determine the set of logical fragments the transaction needs to access, and to locate the physical fragments. The efficiency of transaction processing thus depends on how the database directory is distributed and replicated. Several possible alternatives to distribute and to duplicate database directories are the following [ROT77]:

(1) Centralization without duplication: The complete
directory is stored at one node. Every transaction must access the central directory. This centralized approach is not favored in a truly distributed system.

(2) Centralization with duplication: Each node has its directory for its local data. A complete directory copy is duplicated at one node. Thus, a local transaction can be entirely processed locally; only non-local transactions need to do remote access to the central directory.

(3) Distribution without duplication: Each node has a local directory for its local data. A local transaction can be entirely processed locally. A non-local transaction needs first to issue a broadcast to other nodes to determine whether they have the physical fragments the transaction requests.

(4) Distribution with full duplication: Each node has a complete directory for global data in the network. All directory accesses can be performed locally.

(5) Distribution with partial duplication: The data directory is partitioned and distributed in the network nodes. Some partitions may be duplicated at some nodes. In order to know how the data directory is distributed and duplicated, a directory of database directories is needed.
9.3 Database Directory Management in DLDBS

9.3.1 Design Consideration for Data Distribution and Duplication

There are some trade-offs associated with various degrees of data distribution and duplication: communication cost, storage cost, reliability, and efficiency of transaction processing. Centralization of data tends to save storage cost; however, efficiency of transaction processing and reliability of the distributed database system are poor. Duplication of data will increase efficiency of transaction processing; however, the storage cost is high and the communication overhead required to apply updates to each copy of the data is increased. Thus, a design decision may depend on what bandwidth the communication media can support, how much storage is available, how reliable the system needs to be, etc. There are two other important criteria that need to be considered. They are related to the observable properties of data and are described below.

(1) Mutability of data: Mutability is a property of data which relates to the frequency of change of the data over time. High frequency of change of the data discourages the
duplication of the data because an update to the data must be posted to every copy of the data. This complicates the process to do data updates. On the contrary, high frequency of data retrieval encourages data duplication because most retrievals will not require access to remote nodes in the network.

(2) **Locality of reference to data:** Locality of reference says that a database can be geographically or functionally split into regional components so that most updates are local to their region. For example, a banking database may be geographically distributed, with each branch serving a subset of customers who live around the branch. Therefore, most queries and updates at the branch can be processed locally. Another way to distribute a database is by function locality. For example if a school database contains an order file and an inventory file for office equipment, a student file and a course scheduling file, a logical distribution would be to store the order file and the inventory file at one node, the student file and the course scheduling file at another node. Thus, most requests can be processed locally.
These design considerations can be applied not only for considering data distribution and duplication but also for considering data directory distribution and duplication. This is because the data directory is actually a set of data.

9.3.2 Directory Distribution and Duplication in DLDBS

In Section 8.3 the DLDBS is determined to be a logically integrated and physically distributed database system. A global conceptual view is presented to the users and the global conceptual view is partitioned into logical fragments. A logical fragment has one (or more, in case of duplication) physical fragment which is the smallest unit to be allocated at a node.

In Figure 2.4, it is shown that wherever database data are stored, there also exist DBMS capabilities (DBMS control program) to access and process the data. If some part of the data is duplicated, the appropriate DBMS functions are also duplicated. Except the DBMS control program and data, the management of the database directory is the key point to the realization of the DLDBS architecture.
The alternatives of managing the database directory are described in Section 9.2. Since different types of data in the database directory have different natures, they do not need to be managed in the same way. The major types of data in our consideration are as follows:

1. cataloged transaction definition and the user program interpreter;
2. the external schema definition;
3. the mapping definition between the external schema and the conceptual schema;
4. the conceptual schema definition;
5. the mapping definition between the conceptual schema and the internal schema;
6. the internal schema definition;
7. the location directory;
8. usage statistics of various data objects;
9. accounting and auditing data.

**Database directory distribution.** The general architecture of the DLDBS directory distribution is given in Figure 9.1. The external schema, the mapping definition between the external schema and conceptual schema, and cataloged transaction definition are the information closest to local users. Thus, they are located on the LRN which receives local user requests. Localization of this information at
Figure 9.1  DLDBS Directory Distribution
the node which defines it increases autonomy between nodes. The user program interpreter which typically contains a lexical analyzer and a parser also is located on the local LRN. Each LDN contains the internal schema to describe the storage structure of its local data and the file access methods to access the local data. Thus, the remaining issue is how the conceptual schema and location directory are maintained.

In order to present a global conceptual view to a user, each LRN will keep a global enterprise directory. The global enterprise directory is not a union of all local conceptual schemata. It just contains some necessary information to facilitate a user's request being decomposed locally into several conceptual-level subrequests without referring to remote nodes. This information includes the names of relations in the enterprise database, the attributes of a relation, integrity constraints and security considerations, etc. The complete description of local conceptual schemata and the mapping definition relating conceptual schemata to internal schemata for local database data are stored at the local LDN. Thus, this scheme implements the concept of logical integration of the distributed databases but avoids the full duplication of the whole conceptual schemata and the mapping definition at
every node.

In Figure 9.1, a conceptual view of an enterprise is decomposed into three logical fragments R1, R2, and R3. The physical fragment of R1 as well as its schema definition are redundantly stored at the LDN node a and node c. In the same fashion, fragment R2 is stored at node a and node b and fragment R3 is stored at node c. A user's request can be decomposed into several conceptual-level subrequests by making reference to a global enterprise directory. Each subrequest refers to a single database node. Then it is necessary to know which fragment among several fragment duplicates the subrequest should choose and where the fragment is physically located. In order to answer this question, a location directory is maintained at each LRN. The location directory will specify and select the individual occurrence of a fragment and implement the concept called data materialization [DAT77]. Materialization refers to the process of constructing an occurrence of the logical fragment from the corresponding physical fragment occurrences and presenting it to the application. A user of the DLDBS is given access to the global database through a specific supported materialization which is specified in the location directory. The location directory not only provides the materialization of the
conceptual database, but also contains the location where the physical fragments are stored. Thus, the location directory provides necessary information to determine the network access path of a user request.

Database directory duplication. Now we are in a position to describe the duplication of a database directory/dictionary in the DLDBS. As described in Chapter 2, there are two types of virtual nodes, LDN and LRN, in DLDBS. A Loop Node (a host computer) in the DDLCN can serve as an LDN, an LRN, or both. Referring to Figure 9.1, locally defined external schemata, external to conceptual mapping, and cataloged transactions are stored at each LRN. Since this information is local application dependent, there is no need to have duplication of it. The global enterprise directory and location directory contain very static data with high frequency of retrieval, low chance of updating, and small size. From the consideration described in Section 9.3.1, they can be fully duplicated in every LRN without much overhead. This decision enhances the efficiency of transaction processing and reduces communication overhead for doing access to remote directories. Finally, the conceptual schema, the internal schema, the conceptual to internal mapping, and the file access method are located with the physical fragment at each LDN. If a physical
fragment is duplicated, the corresponding schema is also duplicated. The decision of duplicating physical fragments of a logical fragment is implementation dependent. Table 9.1 presents a summary of database directory management in the DLDBS. The concurrency control mechanism described in Chapter 5 and Chapter 6 can be applied here for doing updates if the contents of the database directory need to be modified.

The distributed database architecture and the database directory/dictionary management scheme in DLDBS support several of the following important design decisions described previously:

(1) The DLDBS is a logically integrated and physically distributed database system. The distribution and allocation of physical fragments are transparent to users.

(2) The DLDBS is designed and can be implemented in an uniformly structured way. Future expansion or shrinkage of the network size is always possible without major surgery.

(3) Inter-Database Control Software is considered as a set of distributed processing algorithms (DPAS). The major efforts of designing a distributed database management system is the design of those DPAs. IDCS can be
<table>
<thead>
<tr>
<th>Directory Component</th>
<th>Distribution and Duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>external schema external to conceptual mapping</td>
<td>distributed at each LRN no duplication</td>
</tr>
<tr>
<td>global enterprise directory</td>
<td>fully duplicated at each LRN</td>
</tr>
<tr>
<td>location directory</td>
<td>fully duplicated at each LRN</td>
</tr>
<tr>
<td>conceptual schema internal schema conceptual to internal mapping</td>
<td>allocated with physical data fragments</td>
</tr>
<tr>
<td>statistics and accounting data</td>
<td>located at each loop node, no duplication</td>
</tr>
</tbody>
</table>

Table 9.1 Directory Management
devised and implemented as an independent software module for incorporating several local DBMSs. Any change at the local level should not affect the function of IDCS.

(4) The DLDBS is composed of a set of autonomous database nodes. It provides a truly distributed database service.

9.4 Summary

A database directory contains a repository of information about the database. This repository of information is used to assist both the management of the installation's database and also the production of application programs which operate on the database. This chapter presents a strategy to maintain and manage the database directory in a network environment. It supports the DLDBS design goals as described in Chapter 1. Several alternatives for distributing and duplicating directories are first described. The design criteria of managing directories are considered. Then the methods of managing the database directories in DLDBS are presented. The purposes of maintaining directories in the proposed way are also explained.
10.1 Introduction

In modern database systems, the user's interface with the database is typically a data language. The user can define, manipulate (query or update), and control the database via the data language. The data languages that were developed in database management systems are dependent on the details of a data model. In [COD74, MEL78], five types of data languages that have been developed for the relational databases are identified as follows:

1. **Element by Element**. In element by element languages, only one tuple of a relation may be referenced at a time. Users write procedural programs with tuple-at-a-time operators to access the base relation using the access path specified in an application system. The RSI (Relational
Storage Interface) in System R [AST76] is an example of the language of this type.

(2) **Relational Algebra.** Relational Algebra [COD70, DAT77] is a procedural data language. It consists of two groups of operations: traditional set operations which include union, intersection, difference, and extended cartesian product; and special relational operations which include selection, projection, join, and division.

(3) **Relational Calculus.** Relational Calculus [COD70, DAT77] is a non-procedural data language. It is a mathematically oriented notation which allows users to indicate desired results without many details about how to obtain them. Codd has shown that any relational calculus expression can be translated into relational algebra. The data language ALPHA [DAT77] is an example of a language of this type.

(4) **Mapping-Oriented Language.** In this type of language, users learn a powerful query specification format. SEQUEL [DAT77] and Query By Example [DAT77] are such languages. They are intended for use by nonspecialists in data processing as well as by professional programmers.
Natural Language. This type of language is intended to deal with casual users who have little knowledge of the database system. It allows users to express queries in a natural language, and the system is designed to ask the user for help if it cannot understand a query. The language "RENDEZVOUS" [COD78] is an example. Data languages expressed in the form of natural languages are extremely desirable but very difficult to implement.

In this chapter, we intend to propose a primitive data language for the DLDBS users and suggest a way to process distributed queries. Since this language is designed for the DLDBS, we call it Distributed Data Language. In Section 10.2.1 we present a data manipulation language to allow users to do retrieval operations and storage operations, as well as to define transactions. In Section 10.2.2 a data definition language is described in such a way that a user can create and define a database as well as specify the way to partition a database. Finally, Section 10.3 suggests a way to process distributed queries in the DLDBS.
The Distributed Loop Database System supports a relational interface to the users. The alternatives were to select one of the five languages just described and modify it to fit our environment. The decision made is to implement the relational algebra oriented language. Several reasons for the decision are described as follows:

(1) Relational algebra is fairly simple compared to other languages. Natural language is too difficult to implement in a prototype system. Element by Element language is in very low level form and is not appropriate to be a user interface. Relational calculus and mapping oriented languages which are higher-level forms of the relational algebra can be translated into relational algebra.

(2) Relational algebra is powerful and relationally complete. A language is relationally complete if any relation derivable from the data model by means of an expression of the relational calculus can be retrieved by using that language. Relational algebra can either be used as a high level interface for users or as a lower level language to support the possibility of
designing a more natural interface in the future. Thus, our approach would be to implement a basic set of commands to facilitate users doing database management and future development.

(3) Relational algebra has been well-studied by database researchers. A number of articles have been published on its background and design. The future implementation could profit from the knowledge and experience of others.

10.2.1 Data Manipulation Commands

We shall now consider the data manipulation language in the DLDBS. The syntax of the language is the combination of relational algebra and QUEL [ST076]. Our discussion starts with the description of three relational algebra operations: SELECT, PROJECT, and JOIN. Then, the basic language commands are presented. In the following discussion, all examples will refer to a sample database [DAT77] with relations as shown in Figure 10.1.

SUPPLIER (SN SNAME, STATUS, CITY)

PART (PN, PNAME, COLOR, WEIGHT, CITY)
### Supplier

<table>
<thead>
<tr>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
</tbody>
</table>

### Supply

<table>
<thead>
<tr>
<th>SN</th>
<th>PN</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>P1</td>
<td>300</td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
<td>200</td>
</tr>
<tr>
<td>S1</td>
<td>P3</td>
<td>400</td>
</tr>
<tr>
<td>S2</td>
<td>P1</td>
<td>300</td>
</tr>
<tr>
<td>S2</td>
<td>P2</td>
<td>400</td>
</tr>
<tr>
<td>S3</td>
<td>P2</td>
<td>200</td>
</tr>
</tbody>
</table>

### Part

<table>
<thead>
<tr>
<th>PN</th>
<th>PNAME</th>
<th>COLOR</th>
<th>WEIGHT</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Nut</td>
<td>Red</td>
<td>12</td>
<td>London</td>
</tr>
<tr>
<td>P2</td>
<td>Bolt</td>
<td>Green</td>
<td>17</td>
<td>Paris</td>
</tr>
<tr>
<td>P3</td>
<td>Screw</td>
<td>Blue</td>
<td>17</td>
<td>Rome</td>
</tr>
<tr>
<td>P4</td>
<td>Screw</td>
<td>Red</td>
<td>14</td>
<td>London</td>
</tr>
</tbody>
</table>

**Figure 10.1** Sample Database
SUPPLY (SN, PN, QTY)

The Supplier relation contains, for each supplier, a supplier number, name, status, and location. The Part relation contains, for each part, a part number, name, color, weight, and location. The Supply relation contains, for each shipment, a supplier number, a part number, and the quantity shipped. The underlined attributes are the key of each relation.

The SELECT operation forms horizontal subsets of relations. It extracts from one relation all those tuples which meet prescribed criteria and forms a new relation. The general syntax is SELECT relation-name WHERE qualification GIVING new-relation-name. The PROJECT operator forms vertical subsets of relations. It removes the unwanted attributes of the source relation and retains in the new only those specified. The general syntax is PROJECT relation-name OVER attributes GIVING new-relation-name. The JOIN operation forms a new relation by the concatenation of relations over a common attribute, where the values of that attribute match. The general syntax is JOIN relation-name1, relation-name2 OVER attributes GIVING new-relation-name.
**QUERY COMMAND**

Based on the relational algebra operations described above, all kinds of queries can be defined. The query command in our language is QUERY. The syntax is as follows:

```
RANGE OF variable-list IS relation-name
QUERY result-relation-name
```

relational algebra operations

Example 10.1 Get supplier numbers for suppliers who supply at least one red part.

```
RANGE OF P SP IS PART SUPPLY
QUERY RESULT
SELECT P WHERE COLOR=RED GIVING TEMP1
JOIN TEMP1, SP OVER PN GIVING TEMP2
PROJECT TEMP2 OVER SN GIVING RESULT
```

**STORAGE COMMAND**

Three basic storage commands are defined to allow users to add tuples to a relation, to remove tuples from a relation, and to modify a set of values within a relation. The three commands are INSERT, DELETE, and UPDATE.
The syntax of INSERT is as follows:

`RANGE OF variable-list IS relation-name
INSERT tuples INTO result-relation-name`

Example 10.2 Add a new supplier whose number is S4, whose
name is Jerry, whose status is 15, and who lives in Dayton
into the relation SUPPLIER

`RANGE OF S IS SUPPLIER
INSERT (SN=S4, SNAME=Jerry STATUS=15, CITY=Dayton) INTO S`

The syntax of DELETE is as follows

`RANGE OF variable-list IS relation-name
DELETE old-relation-name WHERE qualification
[GIVING new-relation-name]`

If GIVING is not in the command, the new relation name
will be the old-relation-name.

Example 10.3 Delete all parts stored in LONDON from the
relation PART as follows:

`RANGE OF P IS PART
DELETE P WHERE CITY=LONDON`
The syntax of UPDATE is as follows:

```
RANGE OF variable-list IS relation-name
UPDATE result-relation-name
relational algebra operations
BEGIN
  computation
END
```

Example 10.4 Increase supplier Jones' status by 15.

```
RANGE OF S IS SUPPLIER
UPDATE S
SELECT S WHERE SNAME=JONES GIVING TEMP1
BEGIN
  TEMP1.STATUS=TEMP1 STATUS+15
END
```

In order to ensure that a sequence of related operations on the database which are considered by a user to be indivisible can be executed as an atomic unit, the data language must have the facilities to implicitly or explicitly declare such an atomic act which is called a transaction. In the DLDBS, any data language command is considered to be a user transaction. Thus, the QUERY command in Example 10.1, the UPDATE command in Example 10.4, etc., are executed as atomic units. The data language in
the DLDBS also allows several statements to be grouped into a transaction by placing them between the statements BEGIN TRAN and END TRAN.

10.2.2 Data Definition Commands

A database is created on a node using a data definition language. The data definition language for the DLDBS contains the following five basic commands.

- **CREDB** - to create a database
- **DELEDB** - to destroy a database
- **DEFRL** - to define a new relation in a database
- **ASSIGNRL** - to partition a relation and to allocate a relation fragment
- **DELRL** - to destroy a relation in a database

The syntax for the five commands are as follows:

- **CREDB** user-code.database-name
- **DELEDB** user-code.database-name
- **DEFRL** relation-name (attribute-name,format; attribute-name,format;...) KEY is attribution-name....
- **ASSIGNRL** relation-name CONTAINS HF WHERE distribution-predicate: loc1,loc2,...;HF WHERE
distribution-predicate:loc1,loc2,...
VF(attribute-name,...):loc1,loc2,...
VF(attribute-name,...):loc1,loc2,...
distribution-predicate := attribute-name
rel-op constant
rel-op := <, =, >, etc.
DELRL relation-name

In CREDDB, the user-code is an identification code to authorize a person to create databases. In DEFRL, the data formats accepted by the DLDBS are integers, floating point numbers, and variable-length alphanumeric character strings. In ASSIGNRL, two types of partitioning techniques can be iteratively applied by using the key word CONTAINS. A relation can be decomposed into HF's (horizontal fragments) defined by distribution-predicates or VF's (vertical fragments) defined by the attribute-names. An HF may again contain several VF's and a VF may contain several HF's. Locales indicates the node number i where the physical fragment locates. A fragment can be replicated at more than one node. The deletion of a relation using the command DELRL implies the deletion of all its replicates.
Example 10.5 Create and partition a SUPPLIER relation and distribute the fragments on node 1, 2, and 4 (see Figure 10.2).

DEFRL SUPPLIER (SN I2; SNAME, A10; STATUS, I3; CITY, A15) KEY IS SN
ASSIGNRL SUPPLIER CONTAINS HF WHERE CITY=PARIS:
NODE1, NODE4; HF WHERE CITY <> PARIS: NODE2, NODE4;

10.3 Distributed Query Processing

The development of query processing techniques on centralized database systems has received considerable attention [SMI75, WON76]. In [WON76], a general procedure to decompose a multi-variable query into a sequence of one-variable queries is given. The environment of processing a query which accesses data at multiple nodes in a distributed database has the following two significant differences from a centralized database:

(1) Data transmission delay introduces substantial processing time delay.

(2) There is an opportunity for parallel processing since there are several computers involved in handling the query.
Figure 10.2  Database Partitions

<table>
<thead>
<tr>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>12</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>13</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>14</td>
<td>Jerry</td>
<td>40</td>
<td>Dayton</td>
</tr>
</tbody>
</table>

Allocated at node 1 and node 4

<table>
<thead>
<tr>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>13</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
</tbody>
</table>

Allocated at node 2 and node 4

<table>
<thead>
<tr>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>14</td>
<td>Jerry</td>
<td>40</td>
<td>Dayton</td>
</tr>
</tbody>
</table>
In fact, these two design considerations are the same as those in designing a distributed concurrency control mechanism. An effective distributed query processing technique should take these two issues into consideration.

In Wong's query decomposition scheme [WON76], the following two basic functions are employed:

1. **Reduction**: This function breaks up the multi-variable query into irreducible components.
2. **Tuple Substitution**: This function substitutes for one of the query variables a tuple at a time.

This scheme can produce efficient query processing when good choices of reduction and substitution variables are made. However, since bulk transfer of data is more efficient than small amounts of data in each of many separate messages; a direct application of the above scheme may lead to very inefficient processing in a distributed database environment [ROT77].

In [WON77] an extension of Wong's centralized decomposition scheme to a distributed environment has been made. In the extended algorithm, the tuple substitution is replaced by a relation move tactic in such a way that a bulk of relation fragments is moved instead of many
tuple-at-a-time messages being transferred. This algorithm begins with an initial solution of transmitting all data to a chosen node. The initial solution is in the following:

1. Perform all the local processing that can be done with no data transfer at all.
2. Make the minimum moves of the results to a single node where the query processing can be completed.

Starting from the unoptimized initial solution, an optimization scheme is applied to determine if lower communication costs can be obtained by employing a set of moves and local processing prior to the movement of subrelations to the node selected in the initial solution. If any gain can be obtained, the optimization scheme is recursively applied until no further improvement can be made.

The distributed query processing scheme in the DLDBS will adopt the approach discussed above. A user of the DLDBS issues a query by means of relational algebra oriented data language. After some preprocessing (e.g., syntax checking), the transaction processor (see Section 2.3.2) parses the query request and produces a tree-like representation. An optimizing scheme in [SMI75] can be implemented here to do some algebraic transformation so that
SELECT, PROJECT operators are kept as far down the tree as possible. This transformation can increase query processing efficiency. Then the query is passed to the Inter-Database Control Software (IDCS).

In the network query processor of an IDCS, the subset of the database required to satisfy the query is determined. The location of the requested data in the network is found by referring to the location directory. Since data may be replicated at network nodes, a materialization [DAT77] of the requested data is determined for the query processing. The decision of assigning a materialization to a query is based on some cost consideration.

If the query involves remote operations at other nodes, the distributed query is processed in the following three phases:

(1) **Local Processing.** This phase is used to send out and execute all subqueries, which can be processed at single node. In the DLDBS all unary operators (PROJECTION, SELECTION) and the binary operator (JOIN), in which both JOIN relations are located at the same node, are first processed at a single node.

(2) **Optimization Scheme.** A set of data moves and local
processing is performed. Sometimes these activities can reduce the size of data to be moved in the next phase, thus resulting in a lower cost for completing the query.

(3) Final Moves and Processing. In this phase all reduced data relations are moved to a final node, where the final processing of the query is done to get the result of the query.

In the first stage of the implementation of the DLDBS on a mini/micro computer based network, for simplicity the optimization scheme is not necessary. Thus, a simple strategy for distributed query processing only consists of Phase 1 and Phase 3. In order to reduce the cost of executing a distributed full-join operation (a join operation in which the two join relations are located at different nodes), a distributed full-join operation can be done by performing two separate semi-join operations [BER79a]. Since semi-joins always reduce the size of a relation, they give the power of a full-join operation without requiring that the data move of a full relation and thus reduce the data move cost.
Several techniques for designing optimization schemes for phase 2 can be found in [WON77, BER79a, ADI80, HEV79]. One of the schemes can be applied if the DLDBS intends to incorporate an optimization scheme in distributed query processing.

10.4 Summary

This chapter presents a primitive data language for a DLDBS and suggests a scheme to handle distributed query processing. The data manipulation language is in relational algebra form. Several commands are defined to allow users to query the database, to add tuples in a relation, to remove tuples from a relation, and to modify a set of values within a relation. Five basic data definition commands are identified to allow qualified users to create a database, to destroy a database, to define a relation, to partition a relation, to allocate a partition fragment, and to delete a relation. Finally, a distributed query processing scheme is described.
CHAPTER 11

SUMMARY AND FUTURE DIRECTIONS

11.1 Summary

The Distributed Double-Loop Computer Network [LIU79] is envisioned as a powerful, unified distributed processing system. One of the major distributed services supported by DDLBN is the distributed database service. The basic result of our research is that we have designed a distributed database service for a distributed system in general and for DDLBN in particular. We call such a distributed database system the Distributed Loop Database System (DLDBS). DLDBS is intended to support a database distributed across DDLBN over seven nodes. Functionally, DLDBS provides the same capabilities that one expects of any centralized database system and users interact with it precisely as if it were centralized.
Network services dealing with data are not necessarily distributed. This means that sole incorporation of database distribution into computer networks does not make them distributed databases. The design of the Distributed Loop Database System intends to meet the following design goals:

1. **DLDBS is a logically integrated and physically distributed** database system; users are unaware of data location.
2. Actual data in DLDBS are **partially duplicated** among nodes.
3. **The distributed services provided by DLDBS are reliable.**

A traditional single-node database environment consists of database data, a database directory/dictionary, and a DBMS. Placing these components at the nodes in a network environment produces a distributed database environment. In this dissertation we have proposed a system organization for DLDBS and have considered five problems in designing a distributed database system. Each of these problems occurs due to the transition of a database system from a centralized database environment to a distributed database environment.
A system organization of the Distributed Loop DBMS is first described. Its software components and the inter-relationship among these components are identified. The major problem in designing DLDBS is to design the Distributed Loop DBMS. The design of the Distributed Loop DBMS involves designing three distributed processing algorithms (DPAs): distributed concurrency control, distributed query processing, and distributed crash recovery.

Two new concurrency control mechanisms, one for a fully and one for a partially duplicated DLDBS, are presented. The mechanisms use distributed control and are deadlock free, simple to implement, and robust with respect to failures of communication links and hosts. They do not use global locking, do not reject transactions, and exploit potential concurrency among transactions.

Improved reliability is one of the major advantages of a distributed database over a centralized one. If a single failure at a node or at a communication link could result in the shut-down of the whole network, a distributed system will actually be less reliable than a single node system. Therefore, in case a failure occurs, how to maintain the continuing operation of the network and how to do recovery
without destroying the consistency of the database are extremely important problems.

In the dissertation, a crash recovery mechanism for the fully duplicated as well as the partially duplicated distributed database system is presented. This mechanism uses distributed control, enables the continuing operation of the network in spite of node crashes and communication link failures, and preserves the consistency of the database. It is robust not only for receiving node failures but also for sending node failures and for nested node failures.

Then we examine the problem of determining the distributed database architecture for DLDBS by incorporating database distribution. Our decision is to distribute the database at the conceptual-view level. The basic component of data distribution and two operators to do database partition are also described.

In order to support DLDBS architecture, a strategy to maintain and manage the database directory is proposed. The data that we are concerned with in the database directory include schemata, mapping, global enterprise directory, location directory, and statistics data. Since different
types of data have different natures, they are maintained in a different way.

Finally, we have designed a relational algebra oriented data language for DLDBS users. Several commands are defined to allow a user to query the database, to add tuples in a relation, to remove tuples from a relation, and to modify a set of values within a relation. Five basic data definition commands are identified to allow qualified users to create a database, to destroy a database, to define a relation, to partition and allocate a partition fragment, and to delete a relation. A distributed query processing scheme for DLDBS is also described.

DLDBS is designed in a structured, modular way so that each component (e.g. distributed concurrency control, distributed crash recovery) can be implemented as independent software modules (certainly, they are closely inter-related and coordinated). This software engineering approach facilitates the future expansion/shrinkage of the network size of DDLCN without causing major surgery on the existing software. The design of the Distributed Loop Database System is intended to be general so that many new concepts developed for it can be applied to implement distributed database services in other distributed systems.
This research demonstrates that it is feasible to integrate database management, computer networking, and distributed processing technologies into a unified system.

11.2 Future Directions

A topic that could be investigated in the future is to do some analytical or simulation studies on the proposed distributed concurrency control mechanism and the proposed distributed crash recovery mechanism. Another topic could be to conduct a complete specification of the data language for DLDBS users. The optimization phase of the distributed query processing scheme also needs to be detailed. The most interesting future direction would be to implement a prototype DLDBS system on DDLCN. Some of the ideas and experience can only be gained through a real running system.

Other topics which are described below are also important in designing a distributed database system.

Data Allocation and Migration. This problem is to find the optimum solution to place the physical fragments in the network and to select the time for migrating the fragments. The data fragments may need to migrate if the performance of
the distributed database system is unacceptable. The performance deterioration may be due to the change of the usage pattern or increase in data dispersion [LEV74, MAH76].

Data and Program Translation. In a heterogeneous distributed database system where the local nodes may be under different physical and logical data models, it is necessary to have a system transparent translation mechanism between different databases. The translation mechanism will provide a single global view of databases on the local nodes which are expressed in different data models. The major problem would be to design a global schema and an intermediate language to integrate data descriptions from various local DBMSs. An associated language translation system is also needed to interpret query programs into the intermediate code [MER74].

Management. The successful operation of a distributed database system will depend heavily on the successful management of the system. Although the distributed database system gathers all dispersed database systems into a single system, shall the management organization be distributed or centralized? Management is a broader and less addressed issue in distributed systems.
BIBLIOGRAPHY


COD78 E. F. Codd, R. S. Arnold, J. M. Cadiou, C. L.
Chang, and N. Roussopoulos, "Rendezvous Version I
An Experimental English Language Query Formulation
System for Casual Users of Relational Databases," IBM

CR075 S. D. Crocker, "The National Software Works: A New
Method for Providing Software Development Tools Using
the ARPANET," presented at Consiglio Nazionale delle
Richerche Instituto Di Elaborazione Della
Informazione, meeting on 20 Years of Computer
Science, Pisa, Italy, June 1975.

DAT77 C. J. Date, An Introduction to Data Base Systems,

ELL77 C. A. Ellis, "A Robust Algorithm for Updating
Duplicate Databases," Proc. Second Berkeley Workshop
on Distributed Data Management and Computer Networks,

ELL78 C. A. Ellis, "Consistency and Correctness of
Duplicated Database Systems," Operating Systems

ENS78 P. H. Enslow, "What is a 'Distributed Data
Processing System'?", Computer, Vol. 11, No. 1,

EPS78 R. Epstein, M. Stonebraker, and E. Wong,
"Distributed Query Processing in a Relational Data
Base System" Proc. 1978 ACM-SIGMOD Conf. on

ESW76 K. P. Eswaran, J. N. Gray, B. A. Lorie, and I.
L. Traigen, "The Notions of Consistency and
Predicate Locks in a Database System," Communication

ESW76a K. P. Eswaran, "Specifications, Implementations and
Interactions of a Trigger Subsystem in an Integrated
Database System," IBM Research Report RJ1820,
November 1976.

FAR72 D. J. Farber and K. C. Larson, "The System
Architecture of the Distributed Computer System--The
Networks and Teletraffic, Polytechnic Institute of
Brooklyn, pp.21-27, April 1972.

FOR78 H. C. Fordrick, et al., "Operating Systems for
Computer Networks," Computer, Vol. 11, No. 1,


