NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MAINTAINING THE INTEGRITY OF DISTRIBUTED DATABASE

by

Fahad A. Al-Lahaidan

June 1982

Thesis Advisor: Norman F. Schneidewind

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Maintaining the Integrity of Distributed Database		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Fahad A. Al-Lahaidan		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 107
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Transaction handling methods
Classes of data
Distributed database systems
Distributed database integrity
Data independence
Recovery techniques
Distributed semaphore method
Database computer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The overall objective for distributed databases is that of sharing of data among several nodes. Increasing the number of users and the size of communication are two factors associated with distributed database systems. These factors, with others such as hardware, software and operations, are major factors which could originate threats to the distributed database integrity. Some discussion about these factors is presented.

Maintaining the data integrity has become a critical problem in distributed database fields. The problem requires a clear and precise view; it needs an early determination for meeting user requirements for integrity, since each organization has its own priorities.

This thesis examines integrity in general and presents some considerations and strategies to be spaced through different system levels, such as design, management, and operation and communication. The main idea of such approaches is to avoid the threats, or to reduce the risks.

Maintaining the Integrity of Distributed Database

by

Fahad A. Al-Lahaidan
Lieutenant, Royal Saudi Naval Forces
B.S.S.E., U.P.M. Dhahran, Saudi Arabia, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1982

ABSTRACT

The overall objective for distributed databases is that of sharing of data among several nodes. Increasing the number of users and the size of communication are two factors associated with distributed database systems. These factors, with others such as hardware, software and operations, are major factors which could originate threats to the distributed database integrity. Some discussion about these factors is presented.

Maintaining the data integrity has become a critical problem in distributed database fields. The problem requires a clear and precise view; it needs an early determination for meeting user requirements for integrity, since each organization has its own priorities.

This thesis examines integrity in general and presents some considerations and strategies to be spaced through different system levels, such as design, management, and operation and communication. The main idea of such approaches is to avoid the threats, or to reduce the risks.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Distributed database technology is a comparatively recent development within the overall database field. The greatest advantages of distributed database systems are:

- Efficiency of local processing for most operations.
- Data sharing between different computers (nodes) in the distributed system.

However, inherent in the distributed database system are the basic problems of a centralized database (e.g., security, concurrency control and integrity). These problems are more critical in the distributed database environment due to several factors such as the large domain of users, the multitude of interactions possible between programs of heterogenous computers, and the multiple copies of a database in the different sites.

Preserving the integrity of a distributed database is not an easy task, particularly when all or part of the database is replicated at different nodes. An update of such a database is subject to a number of problems concerned with coordinating a series of updates entered at different sites and insuring correct entry of updates into all copies of the database. Reliable communication between such nodes is vital so that no entry or communication message (broadcast, acknowledge) can ever be lost.

This thesis discusses the problem of database integrity from a broad perspective. Such a problem needs a clear view for understanding the means and causes which threaten distributed database integrity. Approaches for resolving this problem are examined with respect to data classes, database configurations and systems applications.

This thesis is divided into two main parts. Part one consists of the introduction and the nature of the problem. It sets the scene by explaining the nature of the problem and gives a background for aspects related to distributed database integrity. These aspects define the scope of such integrity and serve as a terminology reference for the following chapters. Chapter III, Distributed Database Integrity Threats, is a brief study of some of the major factors which threaten data integrity. These include hardware and software malfunction, operational and user errors, and communications failures.

Part two consists of Chapter IV which contains a detailed discussion of the different methods and approaches which have been proposed for maintaining the integrity of distributed database systems. These include: design considerations, management considerations, operations strategies and communication strategies. This part ends with Chapter V which contains conclusions reached by the author.

II. NATURE OF THE PROBLEM

A prerequisite to solving a problem is a clear understanding of the problem itself; the integrity of distributed database is no exception. This section will present a background and some of the characteristics of this problem.

A. BACKGROUND

1. Preliminary Definitions

Database can be defined as a collection of interrelated data items that are processed by one or more applications programs. These programs which control the data contained in the database are called a Database Management System (DBMS) [Ref. 1]. This system is composed of several internal functional areas, e.g., a record management, a lock scheduling and recovery control, allocation of data to transaction, and insurance of data sharing and recovery over the database. A Distributed Database Management System (DDBMS) is a collection of sets of data in a network. Each site in the network is a computer running a local DBMS. The network consists of two or more nodes, interconnected with a computer-to-computer communication system. Another important term is the Database Administrator (DBA). This refers to the person (or group of people) responsible for overall control of the database system, using a number of utility programs to help with database control. Examples of utility programs include

loading routines, data dictionary and recovery routines. Users interact with DDBMS by entering transactions (from different sites); this means a program or on-line query which accesses the database.

2. Transactions

a. Types of Transactions

A request to a DBMS or DDBMS system can take any one of the following forms:

(1) An inquiry. This type of request does not update the directory; or, the database processing is required in order to access the directory and the database. An example is read only.

(2) An update. This type of request changes the status of the database but does not necessarily require that the directory contents be modified. Processing is required for accessing the directory, possibly changing the contents of the directory, and for changing the contents of the database. Examples of this type include read, write, change, delete, and file manipulation.

The first type of request is the simplest form because it does not imply any change in database status; the other types include changes in the status of the database.

b. Transaction Handling Methods

The way a DBMS or DDBMS handles any type of transaction depends on the characteristics of the distributed system. Davenport [Ref. 2] defined some ways of handling transactions by a distributed database system:

(1) Application job chaining. The transaction is split into a number of components with each component executing application programs and accessing data within a database section within the confines of a single computing facility (node). When one component finishes, it passes intermediate results to and activates the next component in a remote computing facility. When all components have completed, the final results are transmitted back to the computing facility where execution of the first component took place. Those results are passed back to the terminal that originated the transaction. Application job chaining can be summarised as moving the process to the data.

(2) Transparent access. The transaction executes application programs within one computing facility but accesses data within database sections which are held on remote computing facilities. Transparent access can be summarised as moving the data to the process. The rate of change in database depends on the classes of data.

3. Classes of Data

A DDBMS is concerned with the types of data (residing in each node). One criteria for differentiating between data is the update mode. Different types of data have different types of updating. There are five classes of data [Ref. 3].

a. Class 1: Unchanging Data

This is data which is never or only infrequently changed, e.g., town names and streets; historical information.

b. Class 2: Simple Update Data

This is data which is updated by simple replacement, such data which is performed twice with no harm done, or data which is upgraded by adding new and separate records, e.g., airline timetables; price lists.

c. Class 3: Nonrepeatable Independent Update Data

This is data with an update which cannot be applied twice, but which is independent of any other update. The update can take place at any time (within limits), e.g., bank account balances.

d. Class 4: Time-Critical Update Data

If this type of update is reapplied at different times (e.g., after a restart), its effect may not be the same. Its effect is tied to other events or to other updates which occur independently, e.g., airline reservations.

e. Class 5: An Action Triggered Update

When this data is updated it may trigger the updating of different data or other actions in a different machine, e.g., an inventory balance with automatic recording done on a different machine if the balance falls below a certain level.

A DDBMS is concerned about transaction types, transaction methods, and classes of data. It is also concerned about the configuration of data distribution between sites.

4. Distributed Database Systems (DDBS)

A distributed database (DDB) can be implemented by storing some subset of the entities that make up the database at each site. Reference 4 gave the following formula to describe data distribution:

$$DB = \sum_{S=1}^{S=n} E_S$$

The formula is a distributed implementation of database where E_S denotes the set of entities[1] stored at Site S; DB is the set of all entities that make up the database.

There are many ways in which the entities that comprise the database may be divided among the various sites. They can be characterized as follows:

a. Fully Redundant DDB

   Every entitiy is stored at every site. See Figure 1a on page 18.

b. Partially Redundant DDB

   Some entities are stored at more than one site. See Figure 1b on page 19.

c. Partitioned DDB.

   No entity is stored at more than one site. See Figure 1c on page 20.

Each type of organization has its advantages, depending on the nature of the database and its use.

---

[1]An entity is the unit of data which is controlled by DBMS.

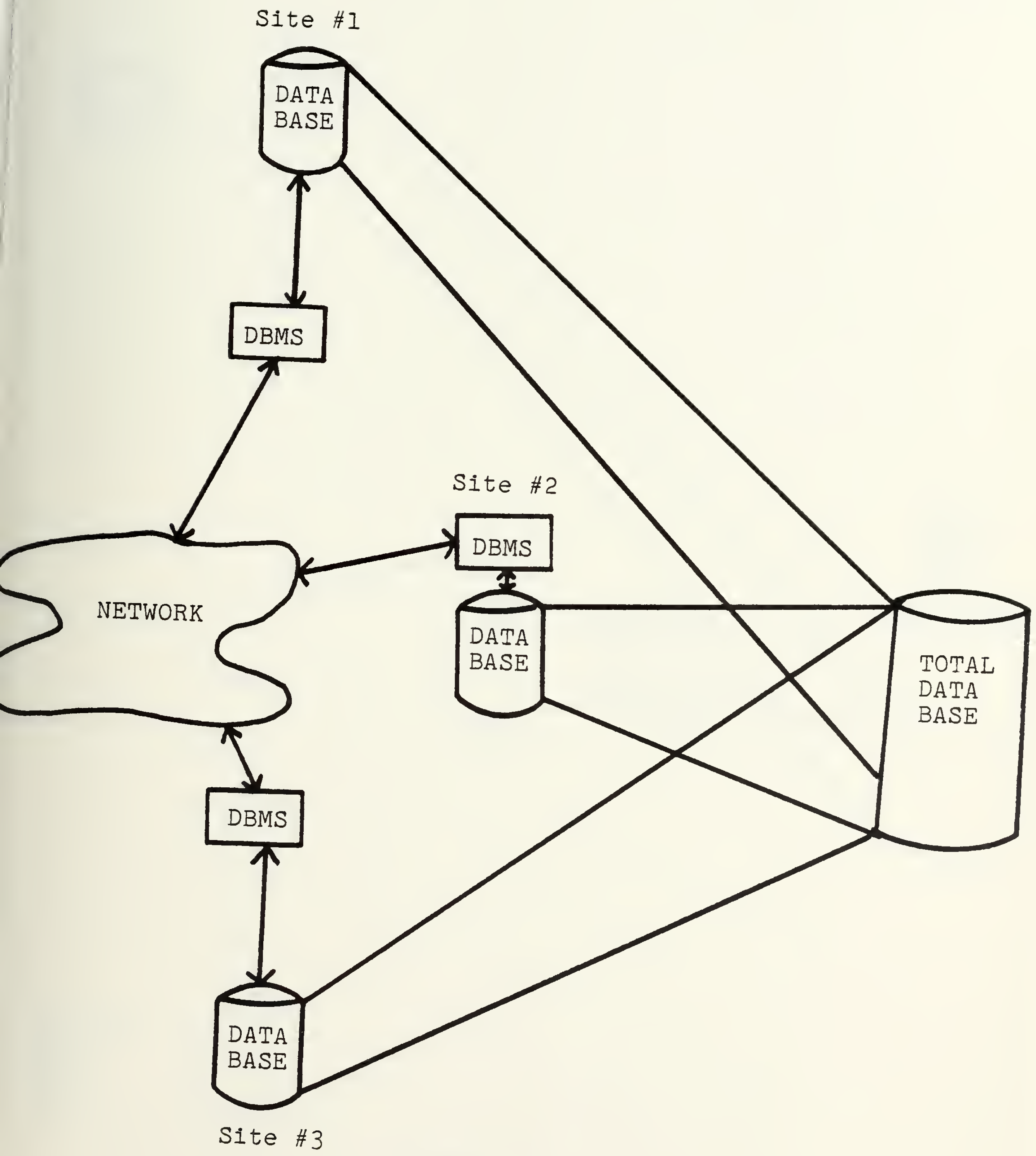


Figure 1a

FULLY REDUNDANT DDB

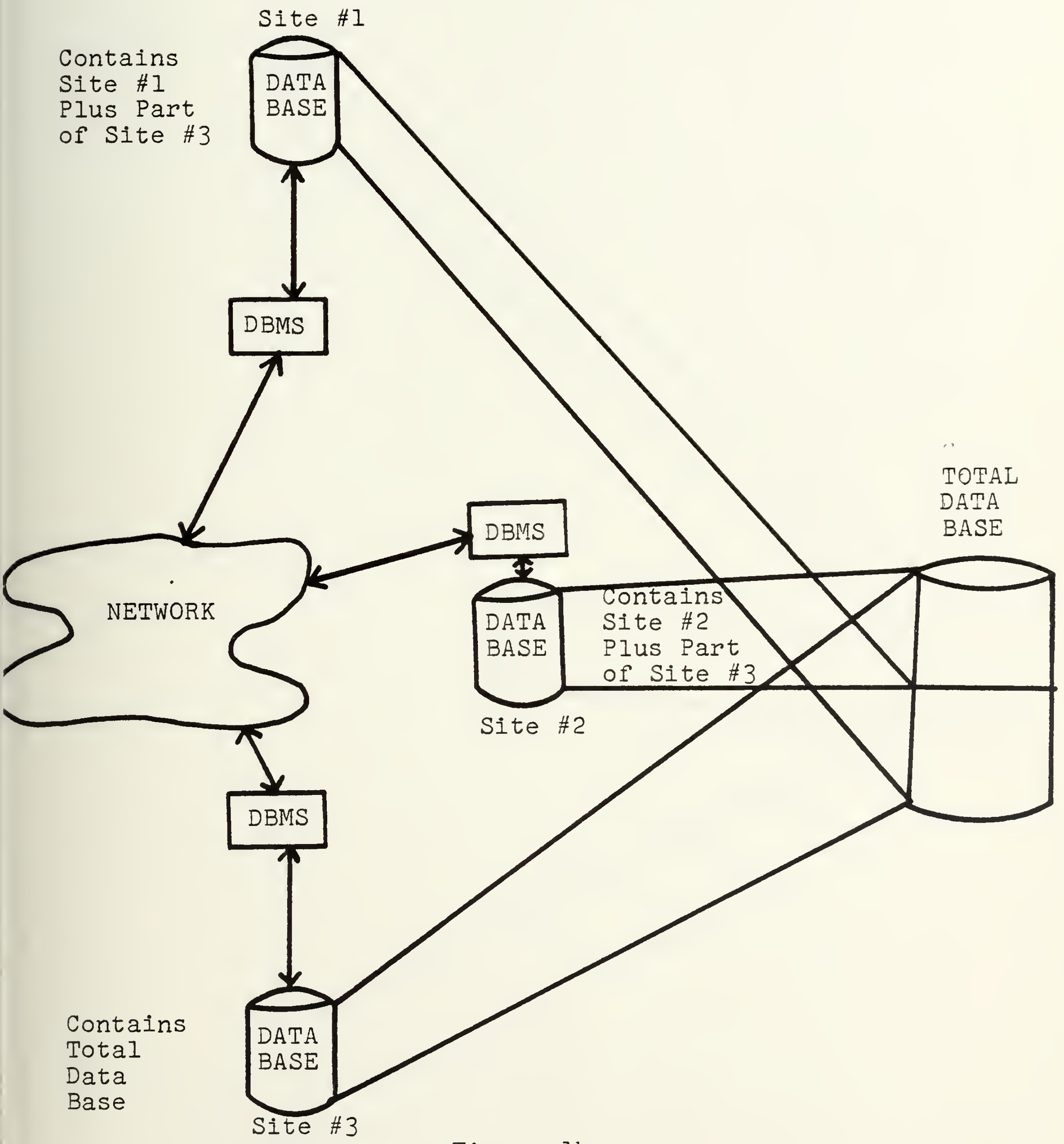


Figure 1b

PARTIALLY REDUNDANT DDB

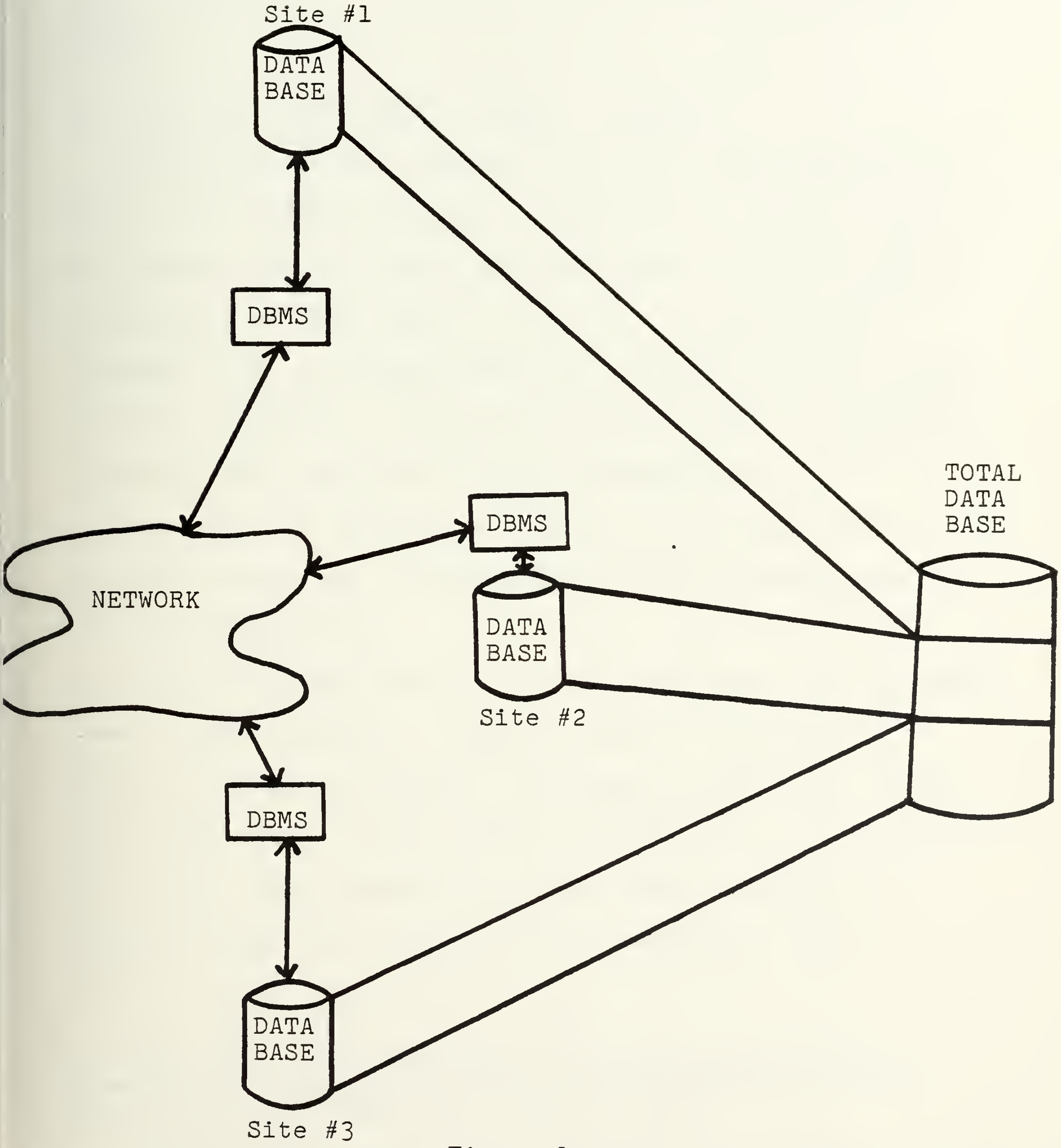


Figure 1c

PARTITIONED DDB

Certain areas of the background have little or no concern with integrity problems. These would include Class 1 in Classes of Data, and an inquiry (read only) transaction in partitioned DDB. Other areas are within the integrity scope.

B. SCOPE OF DISTRIBUTED DATABASE INTEGRITY

The scope of data integrity can be as wide and as complex as the database system is designed in order to protect its status. It can range from enforcing a simple semantic constraint[1] on data entry to the use of sophisticated hardware (e.g., database machines), software, and automatic recovery techniques. Techniques also include communication strategies and concurrent control mechanisms.

The range of the scope is based on the class of the data, the type and method of transactions, and the configuration of data distribution.

There are many points of view in defining data integrity. From a DBMS viewpoint, integrity could be defined as the ability of DBMS to preserve the status of the data elements in the database from any threats[2] leading it to an inconsistent state. The generality of this view includes other related terms, e.g., consistency of database. Maintaining

[1]There are different types of semantic constraints. Generally, it can be defined as an arrangement of values beyond which the input data should not gc.

[2]Discussion of the types of threats is in Chapter III.

the integrity of the database can be viewed as protecting the data against invalid alteration or destruction. Integrity is thus distinct from security, although the two issues are closely allied. Indeed the same mechanism may be used to achieve the preservation of both, at least to some extent. Reference 5 presents examples for such mechanisms. In discussing the integrity of distributed databases it is helpful to divide the previous view of data integrity into the global view and the local view. The first view is concerned with the integrity of the whole system (global); the second view is concerned with the integrity of the system in the site (node) level (local). In this thesis the assumption of the approach is the global DDBMS with heterogeneous or homogeneous local DDBMS.

III. DISTRIBUTED DATABASE INTEGRITY THREATS

The purpose of this chapter is to identify the threats which are likely to affect consistency of database. These threats could change information, destroy the whole database or a subset, or give an inaccurate state of the database.

Studying these threats and their origins assists the designer in planning for countermeasures to decrease the probability of threats occuring, or to decrease the impact of the threat should it occur. Such study needs to be taken in a general view so that the developed solutions will be easy to adopt for various systems in different situations. Some solutions have been proposed [Ref. 6 and Ref. 7]. However, they focus on a very limited area of the whole problem or are limited to a special type of database.

This chapter will examine the distributed database threats in order to see which is most and least critical for data integrity. Chapter IV will discuss different practical techniques which can be used to maintain the distributed database integrity. The importance of these techniques varies from one DDBMS to another according to its application.

DDBS integrity is threatened by several factors including:

- Hardware malfunction. This can result in failure of protection activities, disabling the memory read/write protective devices, and unknowing interruption of priviledged mode processing.

- Software errors. DDBMS application programs may contain undetermined errors which will arise over a period of time. These errors could also come from OS or utility programs.

- Operational problems. These happen during the transaction handling and data manipulation. For example, concurrency conflicts can induce improper sequences of operations and lead to inconsistencies.

- Communication failures. These result from abnormal conditions in the distributed environment and may lead to site crashes. A site crash in any node may prevent the completion of database updating in other nodes.

- User errors. These result from human interaction with the system, e.g., user update errors or a bad entry which introduces inconsistent data elements.

Other indirect errors may contribute to the DDBS threats. These include physical security of the computer system and the quality of DP management. Other than management and human errors, the DDBMS is responsible for realizing these problems and for ensuring the suitable strategy for resolving them. The remainder of this chapter will describe and analyze the above problems.

A. HARDWARE MALFUNCTION

Hardware failures can cause unintentional relevation, destruction, or scrambling of data elements in the database.

These failures can result from device deficiencies or from worn out parts. The limitation of using conventional computer architecture for database application increases the possibility of failures. Current computers are well-suited to scientific and traditional business applications. However, they are not well-suited to information storage and retrieval. Information storage and retrieval applications require addressing by content; while conventional computers are designed for referencing by physical address [Ref. 8]. This mismatch between conventional computer architecture and application requirements for information retrieval introduces inefficiencies in both the processor and storage areas. Data access tends to become computer-bound and tables required to locate data can consume more storage than the data itself.

B. SOFTWARE MALFUNCTION

The difference between intended and actual behavior is caused by "bugs" (program errors). Most large software systems are error-prone; these errors are supposed to be corrected during debugging. However, debugging is often considered a problem for three reasons: (1) the process is costly (takes too much effort); (2) after debugging the software still suffers from bugs; and (3) when the software is later modified, bugs turn up in completely unexpected places. Software faults account for approximately 20 percent of all failures. An analysis of software errors and their

causes is discussed thoroughly by Endres [Ref. 9] and by Schneidewind and Hoffman [Ref. 10].

Failures may be introduced into software at any stage of its development. These may include the following:

1. <u>During Specification</u>

   The analyst may omit to specify what a program should do under certain circumstances. The program may either do the wrong thing, or not do anything at all.

2. <u>In Design</u>

   The processing algorithms chosen to do a particular job may be wrong in that they fail to reflect real life.

3. <u>In Implementation</u>

   Through carelessness, misunderstanding, or lack of testing the program may not code what is required.

4. <u>In Maintenance</u>

   This is the most critical stage because while enhancing the program or correcting new faults, new faults may be introduced as unexpected side effects.

Errors in any level of DDBMS software or in the lower layers of software systems[1] could change the status of database to an inconsistent state. The difficulty here is that all of this can take place without notifying DDBMS. Types of real-time software errors are given in Appendix A.

[1]According to Lorin [Ref. 11], the software layers which usually come underneath the DDBMS and DBMS are the extended OS and the kernel.

C. OPERATIONAL ERRORS

Inconsistency in a database may occur temporarily as an inevitable consequence of an operation on the database. For example, if a data element is moved from membership of one set to another, there will be a brief period when it is attached to both or to neither. Conflict may occur in concurrent access to distributed database such as two users both attempting to modify the same data element.

Each modification of an entity (or data element) creates a new version of that entity. There exist two types of concurrency conflicts which can appear when actions simultaneously create new versions:

1. Lost Operation

This occurs when the new version of an entity is created by a transaction which utilizes obsolete versions of entities to produce the new one.

2. Inconsistency

Inconsistency appears when an integrity constraint is violated.

Simultaneous executions of transactions must be scheduled in order to prevent lost operations and inconsistency.

D. COMMUNICATION FAILURES

If each node in the distributed system network has a (direct/indirect) path to every other node (partitioned), communication link failures do not create any difficulty

since the partition which has a majority of nodes in the network can still continue operating and treats the nodes in the other partitions the same as if in crashed sites. This is a special case if only one partition is allowed to operate; but generally inconsistency among databases in different partitions may occur.

It is necessary to guarantee transaction atomicity in order to be sure that either all the transactions updates are committed in all the sites, or none of the updates are committed. For this purpose, some approaches have been proposed, e.g., two-step commitment protocol [Ref. 12]. Such approaches depend heavily on reliable communication between nodes (sites).

Communication link failures and site crashes are fundamental problems in distributed processing and local networking.

E. USER ERRORS

There are different types of errors in user-computer interface. This difference comes from several factors such as error revising, error origin, or unethical access. The degree of destruction in data is dependent on the type of data class. The actual causes of the error may come from unspecialized user or unintentional entry (bad entry). The upgrading in the degree of access authorization in distributed database environment tends to be less strict in the

ordinary database if there is conflict between the global and local access authorized administrations.

To prevent such errors it is desirable that tools be supplied to DDBMS in order to detect, investigate, and correct or avoid user errors; and to improve the mechanism for access authorization.

IV. MAINTAINING THE INTEGRITY OF DISTRIBUTED DATABASE

Distributed database systems pose problems of integrity much greater than those of centralized database systems, due to the multitude of interactions betwen different application programs, from heterogeneous nodes and concurrent updating the distributed database.

These programs must be prevented from interfering with one another. In addition, when updates occur in one site of the redundant DDB, this update should be read directly to the other copies in order to prevent inconsistency of database. Also, in the absence of effective communication, a crash site in one of the local databases may prevent the continuity of distributed database operations. Crash site or communication link failures need to be handled in such a fashion that gracefully degraded service is permitted. Moreover, the problems of long transmission delay and narrow bandwidth of most communications networks[1] exists in distributed systems.

There is considerable research containing reasonable solutions to some general problems of database systems; for example: database integrity, concurrency control and recovery techniques. Such approaches frequently work poorly in a

---

[1]Due to the internal system delays that result from secondary storage, main memory and CPU characteristics [Ref.13].

distributed environment because of the significant differences in hardware and software configuration.

Maintaining the integrity of DDB is not an easy task. In order to reach such an objective, careful, revised planning for this should start from the early stage of implementation of the distributed system through the system maintenance stage. Of course there is a limit to the extent to which this objective can be reached; in particular, human mistakes.[1] Apart from limitations of this or a similar nature, however, it should be possible to maintain a high degree of integrity in distributed database by implementing integrated planning.

This chapter contains some considerations and strategies which need to be taken in account in planning for the integrity of distributed database.

A. DESIGN CONSIDERATION

In designing distributed database, integrity issues should be the prime objective. The following factors need to be considered in order to achieve the first stage of the objective:

- An efficient hardware: special machines to suit database applications (database computer).

[1]The mistakes that can be made by the human operator include errors such as using the wrong versions of programs or damaging data volumes by careless handling.

- An effective network communication: to handle the data distribution gracefully.
- Reliable software: to cope with abnormal situations, over which the software designer has little or no control.

1. Efficient Hardware

In Chapter II it was seen that the limitation of using conventional computer architecture for database application is one of the hardware malfunction causes which threaten the integrity of database. There are different approaches to computer architectures which are more efficient for information storage and retrieval applications, specifically in database computers.

a. Database Computer

The database computer can be incorporated into a system in one of four ways [Ref. 14]:

- Back-end processor for a host.
- Intelligent peripheral control unit.
- Storage hierarchy.
- Network node.

Each of these approaches is independent and a system may include more than one of the architectures in its list. Each will be considered separately.

The back-end processor approach is usually though of as a master-slave configuration where the host passes high level access requests to the back-end. The

back-end is a general purpose computer which performs all of the database activities including access validation, storage management, update lockout, response formatting, and I/O operations. When the back-end processor has completed the access, it passes the response back to the host. The communication link between the host and back-end is usually an I/O channel, but it may be a telecommunication link.

The back-end processor can provide several benefits to the local database. Hardware specialization is possible, for example, leading to more efficient data and interrupt handling on a dedicated basis. Long register lengths, high speed floating, point, double-precision, multiplication and division hardware can be omitted. Furthermore, software specialization can reduce the overhead in handling interrupts and task switching.

The <u>intelligent peripheral control unit</u> approach moves out the highly repetitive aspects of data access to a mass storage controller in order to avoid the high overhead of the general purpose host hardware and software. The basic functions of device scheduling, head positioning, data recovery, searching, sorting, and error correction are implemented at this level. In addition to the usual I/O function, sequential associative access can also be implemented because of the close coupling between the intelligent control unit and the mass storage device. If the mass storage device is a disk, parallel read may be implemented to obtain storage

search speeds. The mass storage can also be a charge-coupled device (CCD) storage or bubble storage depending on the size and speed required. The controller is connected to the general purpose host through the normal I/O channel.

The <u>storage hierarchy approach</u> is a specialized architecture which can make database operations more efficient. The essence of this approach is that the same characteristic which makes a cache attractive for main storage access can also be used to improve access to mass storage. A wide variety of applications exhibit considerable locality of data reference. This is true of data reference by a processor to main storage for many applications, and has been exploited in the form of a cache, or high speed buffer. When the processor needs a word from main storage, the request is first made to the cache. If the desired word is in the cache the access is completed typically in 50 to 150 nsec. If the request is made to main storage it is typically completed in 800 nsec. A database cache is inserted in the system between main storage and disk.

The <u>network node</u> approach is a general purpose computer which communicates with several other nodes in the system; most frequently using data communication protocol and serial channels, but possibly using I/O channels. The benefit of this configuration is that several nodes (hosts) can access a single shared database. The network node can be implemented using a general purpose system only (which is

current practice), a general purpose host with a back-end processor, or a general purpose host with an intelligent control unit.

b. Integrity of DDB in Database Computers

From the viewpoint of integrity, the back-end processor approach is more beneficial than the other approaches. Using the back-end approach will improve the database integrity at local level. The back-end provides a single path to the database. This eliminates "back door" paths to the data through use of the same mass storage subsystem for both the database and normal system files. Application programmers can be prevented from programming the back-end computer and thus possibly introduce "sneak" access paths. Integrity at the local level is also improved by a single access path because locks on updates can be strictly enforced.

Site recovery can presumably be improved because a failure in the host computer will not compromise the database. Also, presumably the back-end computer has much less hardware and much simpler software than the host, thus extending the time between system failures. The host and back-end can check on each other's sanity, including keeping separate audit trails.

However, there are trade-offs in this approach. The second processor and the software will add cost and complexity in initial development and in maintenance. Two hardware systems and two software systems must be maintained,

thus increasing training and support costs. The reliability of the system will be degraded because having a second system will increase the failure rate which will threaten the integrity of data in case of partitioned DDB. In the other types of distributed database systems this failure has less threat (in cases of partially redundant DDB), or no threat (in cases of fully redundant DDB), since the entities are stored at more than one site.

Another advantage for the back-end processor, which is relevant to DDB, is the ability of this processor to decouple the database from the host to ease conversion or interface multiple heterogeneous hosts.

2. Effective Communication Systems

The communication system describes the way in which the links and nodes of a computer network are connected. Because no specific definition of the precise composition of computer networks exists, several methods of characterization can be used. One characterization involves the reasons for which a network is used. This includes computer resource sharing, database sharing, program sharing and program segmentation. The geometrical arrangement of system resources could be viewed from two points of view: in terms of topology, and in terms of communications structure [Ref. 15].

Network concepts can be classified according to how they contribute to the design of a distributed system or

distributed database. The manner in which work is partitioned in a computer network essentially determines how effectively the resources of the network are utilized.

a. Computer Network Message Techniques

In a computer network the techniques for routing messages from source to destination are generally classified as circuit switching, message switching, and packet switching. Through one or more of these techniques many computer networks provide packet switching capability and virtual circuits. A computer network performs a set of well-defined functions, uses a set of network components, and adheres to a collection of rules and protocols. A protocol is a set of conventions between communication nodes that governs the procedures and format of message transmission.

b. Network Management

Network management is the process which determines through what facilities a message will travel from its source to its destination. It is also concerned with the management of network resources--communication links, switching nodes, and communications processors. There are two basic types of network management systems:

- Master-Slave or "hierarchical,"
- Distributed or "horizontal."

The two types will be discussed separately.

The master-slave network management refers to the use of one or more master stations or processors that

control a plurality of slave processors or nodes. The routing of a particular message is directly controlled by the slave processors, but the general management is controlled by the master station or processor.

Distributed network management refers to the use of decision making facilities at each node (processor) with no one node given control over another node. Depending upon the type of network and the number of nodes, data communication networks are typically designed using one of these two types of network management systems.

Some of the issues that must be considered in determining the type of network management system for a given application are:

- Hardware and software availability.
- Reconfigurability and flexibility,
- Susceptibility to communication failures.

From the above issues, it can be pointed out that master-slave systems are much more structured and accountable, more available, in more widespread use, and often more flexible than distributed configurations. Distributed networks are more reconfigurable and may offer less susceptibility to communication failures.

Network and communication components which are part of the distributed system include the basic components of the database system, the schema, the data, and the programs. A distributed database system therefore should

be designed around two sets of objectives: database objectives and communications objectives. The database objectives are availability and integrity. Communication objectives involve the reduction of number and size of messages and the path length between network nodes (effectiveness). The objectives can be satisfied through the following alternatives:

- Splitting the database.
- Splitting the directories.
- Locating the database programs.

Distributed database system characteristics are achieved through various combinations of the above alternatives with any strategy of data distribution (fully redundant, partially redundant, partitioned).

c. Integrity of DDB in Computer Networks

There are many possible mechanisms which can be used to make the network computer function efficiently. These machanisms may reside in any of the distributed system locations. There are three basic types of control mechanisms that may be implemented in computer networks to support the integrity in DDB and to protect against error in access control, memory control, and integrity control.

(1) Access control. This refers to techniques for preventing unauthorized access to the computer network, application programs, memory, or operating systems. Control procedures can be added to this control to recover from

errors or failures and to ensure that no messages are lost or double processed.

(2) Memory control. This refers to techniques for setting predetermined criteria such as who can read or write what from database or memory. In effect, memory control is access control, not for just the system, but for specific areas of memory. Specialized techniques such as internal usage codes or memory encipherment may be implemented to deter an unauthorized penetration or produce inconsistent data should a detected penetration occur.

(3) Integrity control. This refers to techniques for determining the integrity of the computer network. That is, that it is operating as it was intended to operate. At the most basic level, all system operations--jobs, application programs, supporting systems, communications, and so forth--are given security codes and checks to ascertain whether such operations are occurring when they should be. More sophisticated mechanisms include internal auditing mechanisms and fail-secure and graceful degradation systems.

3. Software Reliability

Although many efforts to improve software quality and reliability have been made, it is hard to say if they will completely eliminate software failures. In the Bell Laboratory Electronic Switching Systems (which employ hardware redundancy and thoroughly tested software) software accounted for approximately 20% of all failures [Ref. 16].

Continuous software modification for large systems leads to additional failures. In many database applications such as computerized air-line reservations systems, isolated small breakdowns can be tolerated as long as the overall system remains operational. However in another application, traffic control systems for example, only moments of cessation of service can be tolerated; incorrect results are unacceptable. There are two basic concepts that make up reliability of software.

a. Correctness

A program is correct if it performs properly the functions that were intended and has no unwanted side effects.

b. Robustness

A program is robust if it will continue to do something reasonable in the presence of environmental changes (such as hardware failure) and demands (such as bad data) that were not foreseen. In addition to robustness, the terms fault-tolerant and error-resistant are often used to describe this property.

The need for reliability of operations in large automated real-time systems is becoming increasingly important, particularly in transportation applications and nuclear industry [Ref. 17]. For such systems it is important to have high confidence that the system will behave as expected for all possible environments. Software structures must be

investigated which provide fault tolerance in addition to fault avoidance. Correctness is a more narrow concept since it refers only to the operation of a system with respect to conditions that can be laid down in advance.

Robustness is concerned with making programs well-behaved in the face of unexpected events, so that it can cope with such situations. Coping means finding alternative ways of carrying out required functions, even though something is wrong. It may mean notifying a higher authority that something is wrong. It almost always means not propagating the error so that problems are contained and catastrophies do not occur. It may mean finding some way to recover from the malfunction.

Figure 2 illustrates the various steps of fault tolerance. A detailed discussion of these steps and the different techniques for fault tolerance is available in Reference 18.

(1) Error detection. The first step is to recognize or prevent system failures by designing proper checks for every critical step. A detected error is only a symptom of the fault that caused it and does not necessarily identify that fault. Usually there is many-to-many mapping between errors and possible reasons.

(2) Hardware reconfiguration. At this step a different strategy will be to ignore the fault and try to continue to provide service despite its continued presence.

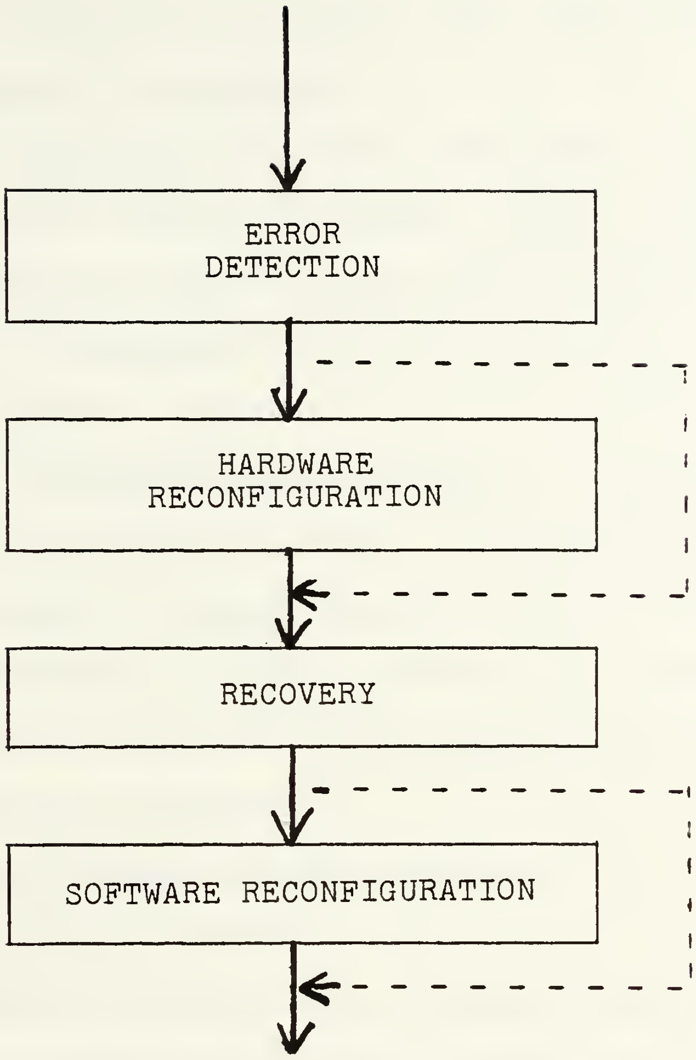


Figure 2

STEPS OF FAULT-TOLERANT PROCESSING

Reconfiguration necessarily involves some degree of performance and/or function degradation.

(3) Recovery. Once the system goes into an erroneous state, its resources (program states, databases) should be brought to a correct state before further processing can be continued. Forward or backward error recovery techniques are used.

(4) Software Reconfiguration. A different strategy in this step could be used. In the distributed software environment, a higher authority or the central module can be notified to take an action, i.e., isolating the software portion which contains the error (locking the site where the error originated).

4. Reliable Software and DDB Integrity

a. Distributed Software

Reliable software should support the data distribution in distributed systems. There are a number of functions required to be handled in a distributed database system. Ideally, there would be only one integrated piece of software. However, the most likely method of development, because of the amount of effort required, is that additional software will be written to interface to the standard components supplied by the manufacturer or software vendor in order to handle the distributed database aspect. The standard components would be the same for either a single computing facility or distributed system. These include:

- Standard operating system of each computing facility.
- Network communication software.
- DDBMS, DBMS.
- Control structure or network component (additional component).

Therefore, reliable measures should be applied for all the pieces of distributed software.

b. DDB Integrity

One way to ensure that incorrect data is not stored in the database is by defining integrity assertions on the distributed structure and semantics of database, and surrounding the local databases with an integrity monitor. Any access to the database would pass through the integrity monitor for verification. Transactions violating the assertions would be disallowed. There are three issues reported in Reference 18 regarding this approach for ensuring integrity of the distributed database:

- Design of integrity assertions.
- Language of the integrity assertions.
- Monitoring of integrity assertions.

(1) Integrity assertions. There are two types of integrity assertions that can be defined at the local database:

(a) Structural constraints. For example, we can declare that duplicate keys or records are not allowed.

Every table must contain only those items which are fully dependent on the attributes. No transitive dependencies among attributes are allowed.

(b) The actual values. These values of the constant are stored in the database. For example, we can limit the value of an item to be within reasonable bounds.

(2) Language of integrity. The language used to express integrity assertion could be the same as one used for accessing the data. DDBMS should enable the local DBMS's to define their assertion. (See [B. 1]) DBMS can use tables (such as header to data file) to describe integrity assertion. These tables are brought at the time of access of these files. It is important that DDBMS (in the case of partitioned DDB) maintain a global table which would contain the whole local table.

(3) Monitoring of integrity. The monitoring or validation of integrity assertions can be done before executing the transaction at run time, or after executing the transaction. Each DBMS should monitor the assertions for the data residing in its local node. The three methods are briefly described below:

(a) Pre-execution. This method requires:

(i) Simulating the transaction to find results that would be written if assertions are not violated (what is to be written?).

(ii) Checking the assertions.

(iii) Executing the transaction if all the assertions were found true.

(b) Run-time validation. This method requires:

(i) Executing a transaction, ignoring its "write" operations.

(ii) Checking the assertions.

(iii) Performing the "write" operations if all assertions are found true.

(c) Post-execution validation. This method requires:

(i) Executing transactions completely.

(ii) Checking the assertions.

(iii) Performing corrective actions.

Which of the above methods is best? This will depend on the types of transactions that will be entered in the database system. If we have the list of items for read and the list of items for write, the pre-execution validation cost is less than or equal to the run time validation cost, which is less than or equal to the post-execution validation cost.

c. Summary

Understanding the importance of the reliability issues for the DDB in the early stage of planning can help the system designers to build software that will be fault-tolerant and which will lead to robust processing. It will also support concurrent processing with the assurance of consistency of DDBS.

B. MANAGEMENT CONSIDERATIONS

1. Decentralized Authorization

The increase in size and complexity of database systems (i.e., Distributed Database over hererogenous hosts) requires the decentralization of some database functions to avoid performance bottlenecks and to improve accessibility without losing integrity of the data. One form of decentralization is delegation; just as the general administration of an enterprise is delegated in an hierarchical way which can be easily decomposed into autonomous functional units.

a. Authorization Functions

Decentralization of authorization functions in distributed database means that authorization functions, instead of being in the hands of DDBMS, are distributed to local DBMS's of the system. The DDBMS may wish to retain a separate administrative function in order to better control the database or delegate some of the administrative rights (i.e., the right to grant access to a particular class of database) to a local DBMS.

b. Decentralized Authorization Model

A model for decentralized authorization for partitioned database has been proposed by Wood S. Fernandez (Appendix B). This model is independent of database configuration (centralized or distributed). It can be adapted, after slight modification, for non-redundant DDB. Each node needs to have a replicated class-node directory (which typically will be small compared with the number of

authorization rules). Validation of an administrative or access request requires the reading of the directory to locate the node where the relevant authorization rules are stored and passing the request to the DMBS if the rule corresponding to the request is denied. Rules at other DBMS's need not be searched. The delegation of a class[1] requires access to authorization rules at possibly two DBMS's. Recall of a delegated class requires access to the rules stored at the nodes associated with the classes in the class structure subgraph. However, recall is not likely to be a frequent occurrence. Authorization related functions can therefore be performed with the minimum of inter-node messages.

c. DDB Integrity and Decentralized Authorization

In the proposed model there are no multiple delegations of specific administrative rights, e.g., it is not possible to give to one DBMS the right to define authorization rules for the objects in a class, and to another DBMS the right to define integrity constraints for these objects. However, delegation of specific administrative rights (access rights, types of access, and integrity constraints) to the local DBMS will improve the accessibility control to the database and support the overall integrity of database. In the case of partitioned database, it is helpful to distribute the integrity responsibilities among the DBMS's

[1]A class may be a set of relations.

so that every DBMS will be responsible for maintaining the integrity of the portion of the database which resides in its local node.

2. Data Independence

Data independence is a capability of a DBMS that insulates a program from interference with its use of data. Technically, this means that the way in which the data is organized in secondary storage and the way in which it is accessed are both dictated by the requirements of the application. For example, it may be decided that a particular file is to be stored in indexed sequential form. The application, then, must know that the index exists and must know the file sequence (as defined by the index). The internal structure of the application will be built around this knowledge.

Data independence is the additional function that preserves alternative views of the same stored data during evolution of the data environment. The importance of data independence is to reduce the effect of the application change on the statues of database. There are two types of data independence: static and dynamic [Ref. 19].

a. Static Data Independence

Static data independence is the ability to cope with change in the "everyone out of the pool" mode. All processing of that body of stored data is stopped. All the descriptions are rewritten. All the stored data is converted

(possibly automatically) to correspond to the new descriptions. All the application programs that access the stored date are converted (possibly automatically) to correspond to the new descriptions. When this conversion is complete for the entire database, then processing can resume. For the heterogenous nodes configuration, it may be possible that each local database will conduct the above separately.

b. Dynamic Data Independence

Dynamic data independence is the ability to cope with change when there are not two states (the pre-existing and the target), nor the ability to suspend processing during the conversion. Dynamic change can be characterized as the concurrent existence of different forms of representation: organization, indexing, access paths, materialization algorithms for the same kind of data. An example of dynamic change is the distributed database in real-time system often cannot be taken down while the stored data is reconfigured and reorganized.

DDBMS or DBMS dynamically provides the data to the user or the user's program as it expects to see the data. Therefore, the stored data need not be completely converted. The programs need not be recompiled for the processing of the stored data to proceed.

Dynamic variation therefore has two aspects: (1) the existence of different extents of the same kind of stored data with different sets of descriptors, and (2) the

possibility of "rolling conversion" concurrent with processing. In the first case, the mixture of different formats may coexist for an extended period of time. All of the old stored records remain in the earlier format, and all the new stored records conform to the new format. In the second case, the conversion mechanism shares the database with concurrent applications. Applications can utilize stored data subject to time-variable descriptors, with the system insulating them from the time variability.

c. Impact of Data Independence on DDB Integrity

Different applications (originating from different nodes) will need different views of the same data. For example, suppose that before the enterprise introduces its integrated database, we have two applications (from two nodes), AN1 and AN2, each owning a file containing the label "Part#." Suppose, however, that application AN1 records this value in decimals while application AN2 records it in binary numerics. It will still be possible to integrate the two files and to eliminate the redundancy (saving the updating process for one copy), provided that DDBMS performs all necessary conversions between the stored representation which is chosen (which may be decimal, binary, or something else again).

DBMS will have the freedom (at the local level) to change the storage structure or access strategy or both in response to changing requirements, without having to

modify existing applications. If applications are data-dependent such changes involve corresponding changes to programs. This leads to unpredictable errors, especially for large systems such as distributed database systems.

It follows that the provision of data independence should be a major consideration in managing the database system. Such consideration may take an important role in reducing the software errors which threaten DDBS integrity.

C. OPERATIONAL STRATEGIES

1. User Error Detection and Avoidance

At least two types of user errors exist which can threaten the database consistency:

- User programs can be incorrectly programmed.
- User programs input data can be incorrect.

Errors from the first type are generally detected at debugging time. However, it is obvious that some of them remain. Therefore, DBMS should prevent incorrect database updating due to incorrect programming. Errors from the second type usually happen when end users who are performing data entry are not specialized persons. To prevent such errors, the user program should verify the input data as completely as possible. However, it is not possible to avoid some typing errors, such as 3,000 in place of 3,200 for a salary. Even if all verifications are not possible, it is desirable that

tools be supplied in order to detect and correct or avoid user errors. Such tools could be intelligent terminals.

a. User Error Detection

The semantic integrity is a method utilized to detect user error by enforcing integrity constraints. These constraints are defined by the DBA and are verified by DBMS whenever the database is modified. At the end of a transaction, all integrity constraints should remain satisfied. However, some of them can be verified after performing a database update. That is the case for integrity constraints where only one data item or an individual record is involved. All other integrity constraints are checked at transaction end, before committing updates. For this purpose, the database which would be obtained with the transaction updates is considered and integrity constraints are evaluated. If one of them is false, the transaction updates are cancelled and the transaction is rolled back. It is not necessary to examine all integrity constraints, but only those whose value (true or false) could be modified by the transaction updates. Generally such integrity constraint verifications are very expensive and user error detection by integrity constraint monitoring appears as an inefficient mechanism.

b. User Error Avoidance

There are some efficient semantic integrity verification methodologies which have been proposed by Hammer [Ref.20] and Gardarin [Ref. 21]. In these approaches, semantic

integrity is maintained at compilation time rather than at execution time. Therefore, one can say that user error is avoided.

(1) The first approach. This is based on an analysis of operations performed by a transaction at compilation time. The integrity constraints studied are restricted to those constraining an individual object. Consider a pair of operation-integrity constraints: an assertion processor performs an analysis that produces an efficient test for the assertion under the operation. This process begins with perturbation analysis. This determines the effect that execution of the operation can have on the truth of the assertion. The information thus derived permits determination of a set of conditions under which the assertion can remain true after executing the operation. If the conditions are suspicious, then the assertion processor generates an efficiency test that will be performed at the time the operation will be invoked, and which will determine the assertion value. Moreover, whenever possible, the generated test can be evaluated before executing the operation, thus allowing the avoidance and execution and rollback of the operation. In addition, several equivalent tests can be generated. The test that should actually be used by the database system at run-time is the one that is expected to incur the lowest cost in its execution. Finally, this approach allows user error avoidance by perturbation analysis

at compilation time and prompt, efficient test evaluation at run-time, but only for restricted classes of integrity constraints.

(2) The second approach. This has been proposed by Gard and is based on program correctness. Transactions are written in PASCAL-like programming language. A data manipulation language based on predicate calculus is embedded in the programming language. An axiomatic definition of both PASCAL and the embedded data manipulation language is utilized in order to show that integrity constraints are constraints through the statements of the transaction with the Hoare axiomatic and predicate calculus theory. The formal proof of success requires inclusion by hand of correct tests in the transaction program. Finally, an automatic transaction consistency verifier is proposed which will definitely permit avoidance of inconsistencies induced by incorrect programming and/or incorrect data entry. However, to build such a transaction consistency verifier remains a difficult program proving task.

2. Recovery Techniques

Recovery is the process of repairing the faulty system or component, or putting right any damage it may have caused, and of restoring it to normal operation.

a. Recovery elements

Recovery of the database may involve a number of elements. These include:

(1) Database Dump. A periodic copy of the database is made.

(2) Logs (Journals). These are serial files which provide a continuous historical record of all the transactions of a certain type.

(3) Database Log. This contains two types of entry. First, <u>before image</u>, is a copy of the old, unchanged version of any block of database. Second, <u>after image</u>, is a copy of the new version of any block of database.

(4) Log Control Data. This allows a check to be made as to whether or not the log block was correctly written.

(5) Checkpoint. This is a stable point written on the log. In the event of recovery action being required, a search for incomplete transactions need only take place between the most recent check point and the end of the log file.

b. Distributed Database Recovery

There are different methods of recovery for the distributed database. Each method depends on the particular recovery points. The recovery situation of the recovery points may be of two types: <u>transaction recovery points</u> which lie either on transaction or integrity unit boundaries, and <u>system recovery points</u> which are check points [Ref. 22]. The methods of recovery are briefly described in Appendix C.

Recovery can be employed in updating[1] in two types of DDBS. These are as follows:

(1) Update of Partitioned Data. Transactions may cause amendments to any part of the database when they are processed. The restart and recovery actions performed depend much more on the transaction handling methods.

(a) Application Job Chaining. When application job chaining is the method of transaction handling, an important consideration is the scope of the integrity unit.[2] This is used in order to maintain content consistency as to how much of the database a transaction must have sole access to. If any integrity unit is required to span several nodes, then resources are blocked for a significant period of time. The aim should be to confine integrity units to within nodes. Therefore each node maintains its own logs of transactions and database changes. If a particular node has suffered a failure, then recovery is initiated. The method of recovery will depend on the type of failure. If the recording media has been damaged, then roll-forward,

---

[1]There are two types of updating: delayed update and immediate update. Our concern here is the latter, since the immediate update is critical for the integrity issue.

[2]Integrity unit: A component of database architecture which is responsible for monitoring the efficiency of integrity constraints.

roll-forward with roll-back or re-run is employed. If a transaction or database request has aborted, then roll-back is employed since the system is a distributed one. The status of the transaction is of interest to more than one node. Therefore, when recovery is initiated, messages indicating that fact should be passed back to the local nodes.

(b) Transparent access. With transparent access an integrity unit may span several nodes. A transaction in one node may require a section of database that is not in the local storage. The central or hierarchical control will grant the requests using mechanisms which insure the consistency of database. Each node should have a log that contains transactions and changes in database. If there is failure in recording media at a particular node, then the local database is recovered using roll-forward, roll-forward with roll-back, or re-run. If the failure is in the transaction then roll-back method is used. The messages between nodes depend on the type of the control. If the node issues a request for data which is not in the local database, the request is sent to the control and the node waits for the reply. If a failure occurs during the request processing (at the other node) then the reply may take an unacceptable length of time. Thus it is necessary to monitor control messages which receive a response that indicates that recovery is taking place (in the node holding the required

data). The receiving node should roll-back the transaction and the monitoring control should send a message to the original node to reinput the transaction at a later time when the original node has received a control message indicating that recovery has been completed.

(2) Update of redundant data. In this type of DDBS, update should take place on all copies which are redundant within short interval times using the transparent access as the handling method. This method is a viable solution for the problem of maintaining the integrity of DDBS if most of the transactions are from read only types with a small number of transactions from file manipulation types. The recovery at this type of update is the same as for update of a partitioned database. There is an extra degree of complexity due to requests for data being made simultaneously to several nodes. In case of failure in one node, two actions should be taken. First, the transaction being executed has to be rolled-back. Second, the database may have to be rolled-back in a number of nodes. Therefore, control messages will have to be passed to each node and be rolled-back.

3. Concurrency Control Mechanisms

In database environments, there are multiple users and programs which access a database concurrently. These require a concurrency control. The problem is to synchronize concurrent interactions so that each reads consistent data

from the database, writes consistent data, and is ultimately processed to completion [Ref. 23]. In a distributed database this problem is exacerbated because a concurrency control mechanism at one site cannot instantaneously know about interactions at other sites. Before discussion of the concurrency control mechanisms, it is necessary to present the following:

a. Definitions

- Serializability: If the reads and writes for each transaction among sequences of transactions are contiguous, such a log is called serial. This serial sequence of transactions preserves consistency since each transaction is executed alone. Serializability has been adopted almost universally as the correct criterion for DBMS concurrency control.

- Transaction failures: The concurrency controller must also guarangee termination, and must operate robustly and efficiently to maintain the integrity of DDBS. The failure in transaction is due to three problems:

-- Deadlock, i.e., two or more processes might be forced to wait for each other.

-- Some process may be indefinitely postponed by an unexpected conspiracy of events.

-- Cyclic restart, i.e., the transaction repeated reaches a blocked state and is aborted and restarted.

- Robustness: this means that the concurrency controller must operate correctly despite the component failures. There are three types of these component failures:

-- A failed site may hold information needed to synchronize progress transactions.

-- A failed site may hold stored copies of data items being updated by a transaction.

-- A transaction that is updating data at several sites may fail after performing some updates but not all of them.

- Efficiency: the efficiency of a distributed concurrency controller is determined principally by how much intersite communication it requires.

b. Types of Mechanisms

In this section the discussion will be on three concurrency control mechanisms which satisfy the following criteria: serializability, robustness, and efficiency.

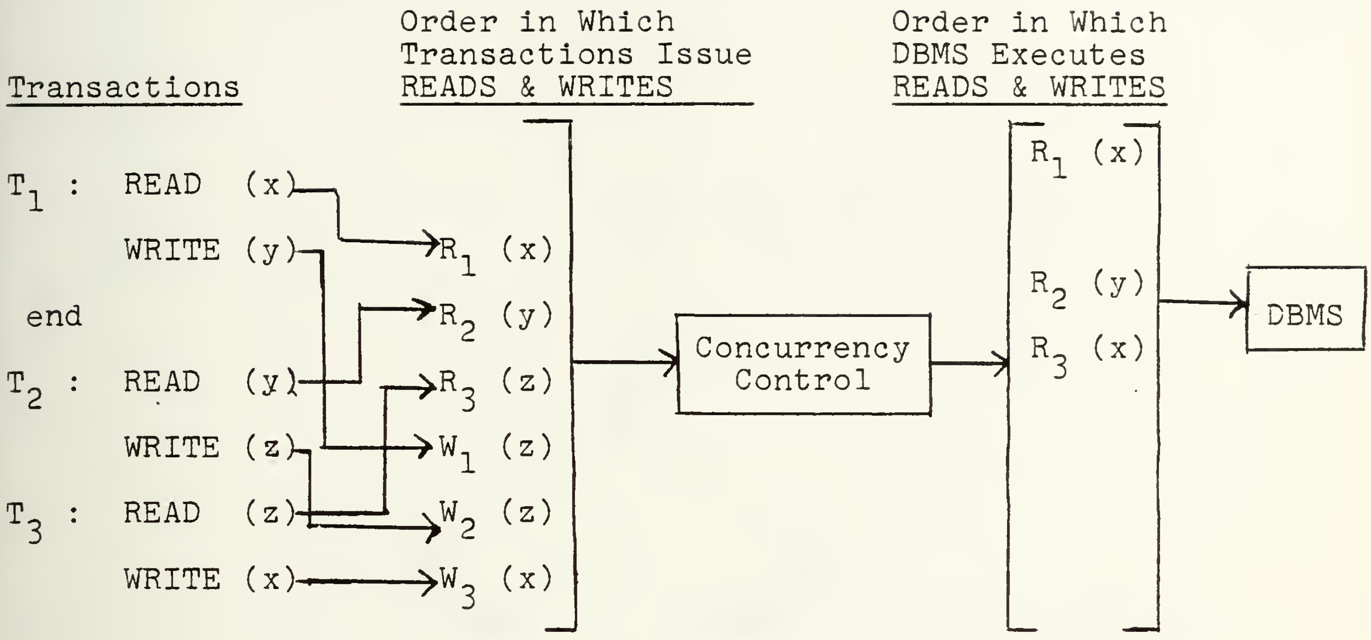
(1) Distributed locking mechanisms.

(a) Local (central) locking. This mechanism is the most widely used in concurrency control. Locking synchronizes transactions by explicitly detecting and preventing conflicts at local levels when transaction issues a READ or WRITE command. The DBMS attempts to "set a lock" on the desired data item; the lock is "granted" only if no other transaction holds a conflicting lock. If the lock is not granted, the requesting transaction waits until the lock is

available and can be granted. DBMS is responsible to generate lock requests for each transaction issued at the local node. Since transactions are made to wait for locks, the possibility of deadlock exists (see Figure 3).

By using a deadlock graph in the DBMS, deadlocks can be detected. There is a deadlock in the system if, and only if, the deadlock graph has a cycle (see Figure 4). If a deadlock exists some transaction in the cycle is backed out and restarted, but this may lead to cyclic restart. A simple way of avoiding this problem is to always abort the "youngest" transaction involved in the deadlock. Indefinite postponement can be prevented in a locking mechanism by processing lock requests on a first-come, first-served basis.

(b) Global locking. One site (node) of DDBMS may be designated a "primary site." It manages all synchronization for the whole system. When a transaction needs to access data at any node, a lock is requested from the primary site. Although locks are centralized at the primary site, the database is, of course, distributed. Once a transaction is granted a lock it may access data at whatever site has a copy. To maintain the data integrity in the case of updating data items that have many stored copies, all copies must be updated before the lock is released. Otherwise, another transaction can read a copy of the data item before the first update propagated there (inconsistency).



W_1 (y)	Cannot be scheduled because it conflicts with R_2 (y)
W_2 (z)	Cannot be scheduled because it conflicts with R_3 (z)
W_3 (x)	Cannot be scheduled because it conflicts with R_1 (x)

THEREFORE: DEADLOCK

Figure 3

DEADLOCK[1]

[1]Reference 23

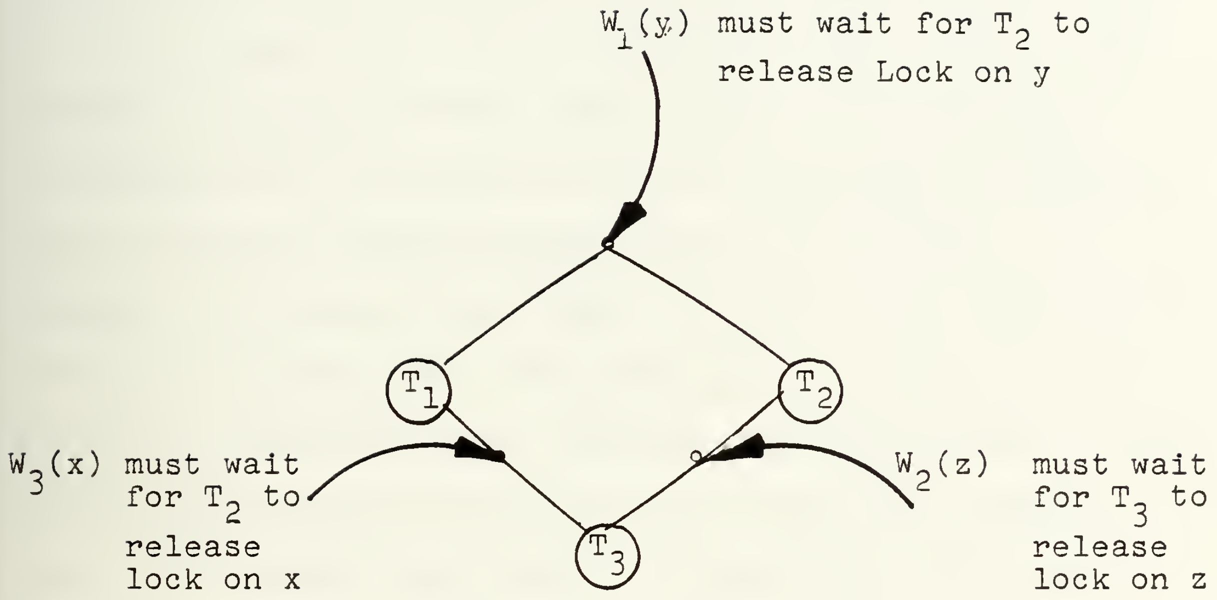

$W_1(y)$ must wait for T_2 to release Lock on y
T_1
T_2
T_3
$W_3(x)$ must wait for T_2 to release lock on x
$W_2(z)$ must wait for T_3 to release lock on z

Figure 4

DEADLOCK GRAPH

FOR FIGURE 3

The principal drawback of primary site locking is that the primary site tends to be a bottleneck. The capacity of the primary site to process locks binds the capacity of the entire distributed system.

(c) Redundant primary locking. If for each logical data item there is a copy in each site, there will be no single site that is primary in any sense. This approach is called primary copy locking. It eliminates the primary site bottleneck, but this mechanism introduces a new problem of deadlock detection. The solution is to designate one site of the DDBMS as the "deadlock detector." Periodically each other site sends it a list of newly granted or released locks, and newly pending requests. The deadlock detector then operates as in the Local Locking case. To maintain the integrity of database, if a transaction is written into a data item, all copies must be updated before the lock is released.

(2) Conflict-Driven Restart Mechanism. This mechanism is used as a model of transaction execution in which each transaction is active at only one site at a time. It moves from site to site during its execution. When a transaction wants to access a data item, a site must test whether it conflicts with a previous access made by an in-progress transaction. If it does conflict, one of three actions is possible: it waits, it is restarted, or another transaction is restarted. If the system responds to

conflict by making the requesting transaction wait, deadlock is possible. To avoid deadlock, Rosenkrantz, et. al. [Ref. 24] proposed two mechanisms that substitute restarts for waiting. Both mechanisms require that transactions be assigned unique "timestamps" when they are submitted. Intuitively, timestamps correspond to the time a transaction was submitted. They have two important properties: timestamps assigned at different sites must be different, and timestamps are used to resolve conflicts such as the following. In one mechanism, called the Wait-Die System, the requesting transaction waits if it has a smaller time-stamp (i.e., is older), or else it is restarted. In the second mechanism, called the Wound-Wait System, the requesting transaction waits if it has a larger timestamp (i.e., is younger), or else the transaction is restarted.

(3) Majority Consensus Mechanism. This is one of the first distributed concurrency control mechanisms proposed by Thomas [Ref. 25]. The majority concensus algorithm assumes a fully redundant database. A transaction executes at one site. The READ command accesses stored data at its site and does so without locking or any other synchronization. Whenever the transaction issues a WRITE command, the name of the data item being updated and its new value are recorded in an update list. The database itself is not modified at this time. When the transaction is completed, the update list is sent to all sites and each

site "votes" on it. If a majority of the sites vote, "Yes," the transaction is accepted and the updates are installed at all sites; otherwise the transaction is restarted. The origin of the algorithm is the rule that determines how each site votes. A site votes, "Yes," on transaction T if:

- The data items read by T have not been modified since T read them (the algorithm requires that a data item must be read before it can be written).

- T does not conflict with any transaction T' that is pending at the site (T' is pending if the site has voted, "Yes," but T' has not yet been accepted or rejected systemwide).

In order to meet condition (1), the algorithm uses a timestamping technique. Transactions are assigned timestamps as in "Conflict Driven Start" and each stored data item is tagged with the timestamp of the most recent transaction that has updated it. Also, update lists are augmented to include the name of each data item read by the transaction and its timestamp. When a site receives an update list it can compare timestamps to determine whether Condition (1) holds. Since augmented updated lists specify transaction READ-sets and WRITE-sets, Condition (2) is easily checked as well.

If Condition (1) is not satisfied, the site "votes" the transaction and it is restarted. If (1) is satisfied but (2) is not, the site cannot vote on this

transaction until the pending one is resolved. Since different sites receive update lists in different orders, they vote in different orders and deadlock could result. To avoid deadlock, the sites votes, "No," if (1) holds, (2) does not hold, and the transaction has a larger timestamp (i.e., is younger) than the pending one. If a majority of sites vote, "No," the transaction is restarted.

The voting rules ensure that two conflicting transactions are both accepted only if one has read the other's output. Since both transactions received a majority of "Yes" votes, some site, say S, must have voted "Yes" on both transactions. Since they conflict, S must have installed one before voting on the other. This guarantees that the second read the first one's output; otherwise S would not have voted, "Yes." This is sufficient to guarantee serializability and to preserve distributed database consistency.

D. COMMUNICATION STRATEGIES

1. Distributed Loop Data Base System (DLDBS)

Another strategy for communication is the DDLCN approach which was proposed by Liu [Ref. 26]. The approach is simple to implement; also it is robust with respect to failures of communication links and hosts. Moreover, the approach has good performance (high throughput and low delay). Discussion of the reliability of such an approach

in site crash will follow the definition and the implementations issues of DDLCN.

a. Definition

DDLCN is designed as a fault-tolerant distributed system that midi, mini, and micro computes through careful integration of hardware, software, and communication.

b. Implementation

DDLCN is a local network using a loop topology. It has two communication loops to transmit messages in opposite directions. Each host is connected to the network by a microprocessor-based loop interface unit (LIU) which has its own RAM, ROM and sufficient computer power to work as a front-end processor for the host. The LIU design is unique in that it incorporates tri-state control logic, thereby enabling the network to become fault-tolerant in instances of link failures by dynamically reconfigurating the logical direction of message flow. In designing distributed loop data base systems (DLDDBS) for DDLCN two types of nodes should be considered [Ref. 27]:

(1) Loop Request Nodes (LRNS). This is where users can make requests to DDB.

(2) Loop Data Nodes (LDNS). These contain the physical data and the DBMS needed to satisfy the requests.

It is assumed that when a user tries to access DLDBS by sending a transaction, a user process is created

in an LDN. After some integrity checking is done and the transaction is considered valid, the user process send this transaction (in case of an update transaction) to LDCS.

c. Operation of the Algorithm

Briefly the algorithm is assumed working on types of communication subsystems which have reliable end-to-end protocols. In <u>normal cases</u> (no site 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 (2) transaction messages from a node are delivered in the order in which they were sent.

The distributed software residing at each LDN to enforce mutual consistency among database copies is called the consistency enforcer. It is a component of the inter-database control software. Each DBMS at LDN 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 LDNS. User processes may be either processes representing some remote on-line users or processes on behalf of application programs.

In <u>abnormal cases</u> (site crashes and communication link failures), the robustness of the algorithm maintained the following:

- The system will continue operating in spite of site crashes and communication link failures.

- A transaction message is put in execution waiting a cue EWB of either every site or no site.

- If a transaction message is put into EWB, it will be dispatched and executed to completion sooner or later; all transactions are eventually dispatched in a total ordering according to their priority.

d. DDB Integrity Using DLDBS

(1) <u>Communication Link Failures</u>. The algorithm requires that each node has (direct and indirect) paths to every other node. Therefore, as long as no site is partitioned from the network communication, link failures do not create any difficulty to the algorithms. When the network becomes partitioned, the partition which has a majority of nodes in the network still can continue operating and it treats the nodes in the other partition the same as crashed sites. Only one partition is allowed to operate; otherwise, inconsistency among DDB in different partitions may occur. Using the recovery technique for site crashes, the network can return to a consistent state after the partitions are repaired. However DDLCN network partition is rare due to the tri-state control mechanism built into the interface.

(2) <u>Site Crashes</u>. The algorithm can continue operating in the case of one or more site crashes. The DDB will recover from anomalies and lead to a consistent state

when a crashed node has been repaired. Without going into further detail, the algorithm in this case needs "a reliable broadcast" facility [Ref. 28] which guarantees that a broadcast message will reach either every destination or no destination when the sender crashes during the broadcasting. Moreover, the algorithm needs a recovery algorithm to facilitate the withdrawal of a crashed site from the whole site.

2. Distributed Semaphore Method

Distributed semaphore is another approach of communication strategy for ensuring the consistency of a multiple copy database. Discussion in this section involves the definition of distributed semaphore, implementation issues, and how such a method operates in distributed database environments.

a. Definition

A distributed semaphore is designed so that for every P operation that is completed by a process, an associated V operation has been performed [Ref. 29]. This type of semaphore was originally developed to facilitate the solution of synchronization problems in distributed systems.

b. Implementation

Implementation of distributed semaphore according to Schneider [Ref. 30] needs certain assumptions regarding the communication network:

(1) Assumptions

- Broadcast Assumption. If a site broadcasts a message that message will be received by every other site.

- Message Order. All messages that originate at a given site are received by other sites in the order in which they were broadcast.

- Timestamp. A timestamp is associated with each message m and it is assumed that the timestamps are consistent with causality. In other words, timestamp of V_1 is less than the timestamp of V_2 if V_1 can affect V_2.

- A Message Queue. For each distributed semaphore implemented, a message queue is maintained. At each site this queue will contain the received messages arranged in ascending order by timestamp.

- Acknowledged Message. When a message is received at a site an acknowledgement message is sent to all other sites.

- A Fully Acknowledge Message. This message is sent by the originating site when the message has been received by every site in the system.

- V# (ds_i,x). The identification number of "V semaphore ds_i" messages with a timestamp is less than or equal to time 'x'.

- P#(ds_i,x). The identification number of P semaphore ds_i messages.

(2) Operations Implementation

P and V operations in distributed semaphore are implemented as follows:

$V(ds_i)$ Broadcast message "V semaphore ds_i"

$P(ds_i)$ Broadcast message "P semaphroe ds_i"

Let tc denote the timestamp on this message.

Then wait until any message m' concerning ds_i is received and fully acknowledged

$V\#(ds_i, ts(m')) \geq P\#(ds_i,tc)$.

It is not necessary to store the entire message queue for each semaphore at every site. Instead, the relevant information from the message queue can be coded in a few integer variables. Due to the message m' order assumption, after a message m is fully acknowledged at site L, no message m' where $ts(m') < ts(m)$ will be received at L. Furthermore, the implementation of distributed semaphores outlined above requires only $V\#(ds_i,x)$ and $P\#(ds_i,tc)$ [Ref. 31].

The initial portion of the message queue can be stored in two integer variables: P# and V#. As messages are received, they are put in a bound message queue. The capacity of that queue need not exceed the number of sites in the system. P# and V# are updated by increments of one and then the message is deleted.

c. Operation of the Method

In the situation where there are multiple copies of some of the entities in the database, i.e., partially or fully redundant distributed database, all copies should have the same change when any transaction updates one of these copies. The transaction need only deal with one copy of the database in order to update the other's copies. Thus the transaction has to broadcast to all these sites a timestamped message containing the entry and its new value. Upon receipt of such messages, a site must broadcast an acknowledgement message to all other sites. The update on the database at site M may not be executed until site M receives a fully acknowledged message from the other sites. This is because prior to that time other messages may be received which carry updates to the semaphore for the database. Since the message order holds for all messages, then both the update and distributed semaphore messages will use the same communication network. This implies that when a transaction is executed, the local copy for every node has its own value. This is because prior to accessing an entity, a P operator on a semaphore associated with that entity is perfomed resulting in the broadcast of a message that must be fully acknowledged for the P to complete. This serves to "flush" all update messages to that site from the communication network.

d. DDB Integrity Using Distributed Semaphore Method

This type of communication strategy is well suited to maintain the integrity of distributed database

(fully or partially redundant) by putting all the sites in full communication with each other. However this impact of distributed semaphore needs to be developed more fully in order to avoid the problem of site crashes, since a full acknowledgement requires the participation of all the sites.

One characteristic of this strategy is that it can develop solutions which are applicable in a broad range of systems.

V. CONCLUSION

Maintaining the distributed database integrity is not a trivial problem. Much progress still needs to be made in several areas, especially regarding link failures, dead-locking, and integrity constraint monitoring.

We have presented several approaches to preserve the integrity of a distributed database, and the obvious question is, "Which one is the best method?" or, "How many of these approaches need to be considered in one system?" There are no clear answers to such questions since each system has its own characteristics and environment.

However, integrity of the database system has to be provided at many levels. The initial concerns regarding integrity must start at the design level, which has to use the preservation of data integrity as one of the design objectives; followed by the management and operations levels, which must allocate resources to large numbers of users and resolve their process conflicts; and then followed in the communication systems, which have to manage multiway message traffic between nodes.

A strategy of regular monitoring of a database is essential during the maintenance phase. Monitoring is possible on two levels: internal monitoring which can be carried out by DBMS, and external monitoring which can be carried out by the user. The latter requires the user to provide assertions

regarding data relationships. Finally, more research in this area is still needed especially regarding time consistency, reconstruction of consistent global states in the DDB, and distributed database communication.

LIST OF REFERENCES

1. Katzan, H., An Introduction to Distributed Data Processing, Petrocelli Books, Inc., 1978, p. 135.

2. Davenport, R. A., Integrity in Distributed Database Systems, European COMP. '78, p. 767.

3. Martin, J., Design and Strategy for Distributed Data Processing, Prentice-Hall, 1981, pp. 282-283.

4. Schneider, F. B., Synchronization in Distributed Environment, Technical Report 79-391, Computer Science Department, Cornell University, 1979.

5. Date, C. J., An Introduction to Database Systems, Addison-Wesley, 1975, pp. 301-303.

6. Bernstein, P. A., D. S. Shipman, J. B. Rothnie, N. Goedman, The Currency Control Mechanism of SDP-1: A System for Distributed Database, Computer Corporation of America, TR CCS-77-09, 1977.

7. Thomas, R. H., A Solution to the Concurrency Control Problem for Multiple Copy Data Bases, Digest of Papers, COMPCON 1978.

8. Champine, G. A., Four Approaches to a Data Base Computer, Datamation, Vol. 24, No. 13, December 1978, pp. 100-106.

9. Endres, A. B., An Analysis of Errors and Their Causes in Systems Programs, IEEE Transactions in Software Engineering, June 1975, pp. 140-149.

10. Schneidewind, N. F., and J. Hoffman, Experiment in Software Error, Data Collection and Analysis, IEEE Transactions in Software Engineering, May 1979, pp. 276-286.

11. Lorin, H., Aspects of Distributed Computer Systems, Wiley, 1980, p. 63.

12. Lamport, L., Time, Clocks and the Ordering of Events in Distributed Data Storage Systems, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

13. Badal, D. Z., On the Degree of Concurrency Provided by Concurrency Control Mechanisms for Distributed Databases, IFIP Working Conference on Distributed Databases, Inria, 1980, p. 35.

14. Champine, op. cit.

15. McGlynn, D. R., Distributed Processing and Data Communication Cost, Wiley, 1978.

16. Bhart, B., Software Reliability in Real-Time Systems, AFIPS Proceedings, 1981.

17. Ramamoorthy, C. B., et. al., A Systematic Approach to the Development and Validation of Critical Software for Nuclear Power Plants, 4th International Conference on Software Engineering, September 1979.

18. Bhart, op. cit.

19. Herbert, S. M., An Overview of the Administration of Data Base, The Information Technology Series, Vol. 1, DBMS, 1979, p. 11.

20. Hammer, M. M. and S. K. Sarin, Efficient Monitoring of Database Assertions, ACM/SIGMOD International Conference on Management of Data, Dallas, 1978.

21. Gardarin, G. and M. Melkanoff, Proving Consistency of Database Transactions, 5th Very Large Database, Rio, 1978.

22. Davenport, op. cit.

23. Bernstein, P. A. and N. Goodman, Approaches to Concurrency Control in Distributed Data Base Systems, AFIPS Proceedings, 1979.

24. Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis, System Level Concurrency Control for Distributed Database Systems ACM Transactions on Database Systems, Vol. 3, No. 2.

25. Thomas, R. H., A Solution to the Update Problem for Multiple Copy Databases Which Use Distributed Control, BBN Report No. 3340, Belt, Beranek & Newman, Cambridge, MA, July 1975.

26. Liu, M. T., et. al., System Design of the Distributed Double Loop Computer Network (DDLCN), Proceedings, First International Conference on Distributed Computer Systems, 1979.

27. Chou, C. and M. T. Liu, A Concurrency Control Mechanism and Crash Recovery for a Distributed Database System (DLDBS), IFIP Working Conference on Distributed Data Bases, Inria, 1980, p. 201.

28. Rothnie, J. B. and N. Goodman, A Survey of Research and Development in Distributed Database Management, Proceedings of Very Large Data Bases, 1977, pp. 48-62.

29. Schneider, F.B., Ensuring Consistency in a Distributed Database System by Use of Distributed Semaphores, IFIP Working Conference on Distributed Databases, Inria, 1980, p. 183.

30. Schneider, F. B., Synchronization in a Distributed Environment, Technical Report 79-391, Computer Science Department, Cornell University, September 1979.

31. Ibid.

32. Fernandez, E. B., R. C. Summers and C. D. Coleman, An Authorization Model for a Shared Database, Proceedings 1975 SIGMOD International Conference, ACM, New York, 1975, pp. 23-31.

APPENDIX A[1]

SOFTWARE ERRORS IN REAL-TIME SOFTWARE

Software errors and their frequency of occurrence in real-time software. The types of errors can be grouped into the following major classes:

1. Computation errors: errors in or resulting from coded equations, equations that produced values directly from the physical problem being solved, and equations used in bookkeeping sense. Typical errors are mathematical modeling, index, conversion, and mixed-mode arithmetic.

2. Logic errors: incorrect logic code, missing condition test, flag not tested, etc.

3. Data input errors: format errors, input read from incorect data file, invalid input read from correct data file, etc.

4. Data output errors: format errors, data written on wrong file, incomplete or

[1]Reference 16.

	missing output, output field size too small, etc.
5. Data-handling errors:	errors made in reading, writing, moving, storing, and modifying data, etc.
6. Interface errors:	routine/routine interface errors, routine/system software interface errors, wrong routine called, and incompatibilities between database and using routines, etc.
7. Definition errors:	errors in specification of global variables and constants, data not properly defined/ dimensioned, etc.
8. Present database errors:	data not initialized, initialized to wrong values, incorrect data units, etc.
9. Documentation errors:	errors in design and operational documents.

10.	Operation errors:	wrong database used, wrong tapes used, configuration, control errors, etc.
11.	Others:	time limits exceeded, storage limits exceeded, etc.

APPENDIX B

MODEL FOR DECENTRALIZED AUTHORIZATION

This model is based on the one defined in Reference 32 and adapted to handle the decentralization of administration. The named items in the database are called database object types. We make the distinction between object type (or category) and instances (occurrences) of an object. A data class, D, is a set of database object occurrences. A subclass, D1, is a subset of the object occurrences of a data class, and can be defined in terms of the class D and an arbitrary predicate P:

$$D1 = D : P$$

Generally, we refer to data classes and subclasses as "classes." Classes are the units of delegation of administration and can either be disjoint (no common occurrences) or overlapping. (This is in general, later we will only allow subclasses to be overlapping.) The structuring of classes can be described by a class structure graph, CSG, where nodes represent classes and a directed arc from node i to node j indicates that class j is a member of class i. The CSG is always a tree.

Security policies are represented by authorization rules. An authorization rule is the tuple (s,O,t,p,f), which specifies that subject s has authorization of type t to those

occurrences of object type O for which predicate p is true. In general, user s can grant the access right defined by O,t, and p if the copy flag f is true. The combination of (O,t,p) of a rule is called an authorization right.

For this environment, there are two types of rules. Access rules which are rules controlling database access, where s is a user; O is a database object type; t is an access type such as READ, DELETE, or UPDATE; p can depend on database values or system variables; and f will be false since only administrators are able to delegate their rights.

The second type of authorization rule is the administrative rule, where s is a DBA identifier, O is a data class, t an administrative access type, p is always true, and f can be true or false depending on the administrator being authorized to delegate this right or not.

Administrative rights refer to the ability to control the database access actions, as opposed to the ability to access the database (some examples of administrative access types are shown in Figure 1). As we are mainly interested in administration aspects we will write administrative rules as (s, O,t,f) for simplicity.

DBA's delegate administrative rights by means of commands, expressed in some suitable syntax. From these commands in the system extracts an administrative request, which is a tuple (s' , P' , t' , f'), where s' is the DBA entering the command, O' is the object of the command to

which administrative access of type t' applies, and f' indicates if the access right of s is being delegated. A similar tuple is extracted by the system when a user requests access to a database object. (We call that tuple an access request.)

Validation of an administrative request (or an access request) implies finding a rule where s, O and t match the corresponding parts of the request, and f=true if f'=true. If such a rule is not found the request is not accepted and an enforcement procedures, such as logging the illegal request, is invoked.

It is useful in some situations (for example, when different DBA's administer classes containing common objects) to have a context or environment for the requests issued by the users of the system. In our case a useful context is provided by data classes, i.e., users make requests in the context of a class. An access rule becomes now (s,O,t,p,f,D), where D indicates the context (D is a data class name).

MECHANISM FOR AUTHORIZATION

Using the model discussed above we now propose a mechanism to implement these concepts. For concreteness we assume a multilevel relational database system[2] where the conceptual schema is composed of base relations. The allowed access types are assumed to be READ, DELETE, UPDATE and INSERT. Classes are restricted to be sets of relations. A basic class is a set of base relations. For example, suppose D1 is composed of three relations, then:

D1 = (R1,R2,R3)

If R1,R2,R3 are base relations then D1 is a basic class.

A subclass D2 of D1 is defined as

D2 = (R1',R2',. . .RN;)

where the relations R1' to RN' are projections, restrictions and joins of the base relations. As an example consider a simplified banking database containing account and customer information. Assuming joint accounts are allowed the database might contain the following three base relations:

R1: (ACCOUNT #, BRANCH #, ACCOUNT_DETAIL)

R2: (ACCOUNT #, CUST #)

R3: (CUST #, CUST_DETAIL)

Subclasses containing information relevant to each bank branch may then be defined. For example the subclass for branch B1 would contain the following three relations (informally defined)

R1' = (R1: WHERE BRANCH # = B1)

R2' = (R2: WHERE ACCOUNT # = R1' . ACCOUNT #)

R3' = (R3: WHERE CUST # = R2' .CUST #)

Administrative responsibility for these subclasses would then be delegated to DBA's in the local branches. Notice that in general subclasses are not disjoint, i.e., a a customer may have accounts in different branches.

Administrative access types which apply to a basic class D are listed in Figure 1. The set of types a1 - a6 are jointly referred to as A_B and are the types automatically

$$A_B \left\{ \begin{array}{l} \text{a1 - right to create, delete and modify} \\ \quad \text{objects in D;} \\ A_S \left\{ \begin{array}{l} \text{a2 - right to redefine and delete D;} \\ \text{a3 - right to authorize READ access to} \\ \quad \text{objects in D;} \\ \text{a4 - right to authorize DELETE access to} \\ \quad \text{objects in D;} \\ \text{a5 - right to authorize UPDATE access to} \\ \quad \text{objects in D;} \\ \text{a6 - right to authorize INSERT access to} \\ \quad \text{objects in D;} \end{array} \right. \\ \text{a7 - right to recall a delegated right for D.} \end{array} \right.$$

Figure 1

ADMINISTRATIVE ACCESS TYPES

given by the system to the definer of a new basic class. The type a7 is given by the system to a delegator only after the class has been delegated. The same types with the exception of a1 can apply to a subclass and are jointly known as A_S. Access type a1 includes among others, the ability to redefine a base relation (for example by adding a new column), delete a base relation from the conceptual schema and define semantic integrity constraints for a base relation.

As different DBA's can administer different basic classes and as the administration of a basic class is associated with the ability to redefine the underlying data object types, basic classes must be disjoint (i.e., they must possess no common objects) in order to avoid conflicts. In contrast, subclasses can be overlapping because no object types can be created, deleted, or redefined through subclass rights.

Class administrators may delegate some or all of their rights to other DBA's if the corresponding delegation flag is true. When they define a subclass, say D1, from a class D, they obtain for D1 the same set of administrative rights that they had for D . The DBA's also authorize user access to objects within a class (such as attributes), or to application views which are constructed using relational operators on the relation comprising the class. In a multilevel system, access rules pertaining to a view should be consistent with the access rules for the underlying objects. We consider that the conceptual level for DBA's consists of

the set of base relations comprising the classes which they administer.

We restrict the administration of a class to a single DBA and allow a class to be delegated only once. This avoids the situation where an administrator receives rights to a class from more than one delegator. Revocation is therefore simplified and time stamping is not required.

If a DBA delegates the administration of a class then any access rules that had been authorized in the class previous to the delegation become the responsibility of the delegatee.

As administration and database access are separate functions, a reorganization of the administration function should not mean that some users of the system can no longer access the database. Only administrative rights are recalled when a delegated class is therefore recalled. Access rules authorized by the DBA's whose administrative rights were recalled are not deleted but become the responsibility of the recalling DBA. The recalling DBA can then review the acquired rules and delete or modify them on an individual basis.

A simple example illustrates the principles of authorization and revocation. Figure 2 shows a sequence of authorizations (d1,d2,...d5) with each arc representing a delegation or authorization and each node a set of authorization rules. We call this type of directed graph an _authorization graph_.

$$\begin{tikzcd}
 & (\mathrm{DBA1},\ \mathrm{D1},\ A_B,\ \mathrm{true}) \arrow[dl, "\mathrm{d1}"'] \arrow[dr, "\mathrm{d2}"] & \\
(\mathrm{DBA2},\ \mathrm{D2},\ A_S,\ \mathrm{true}) \arrow[d, "\mathrm{d3}"'] \arrow[drr, "\mathrm{d4}"] & & (\mathrm{DBA3},\ \mathrm{D3},\ A_S,\ \mathrm{false}) \\
(\mathrm{DBA4},\ \mathrm{D4},\ \mathrm{a3},\ \mathrm{false}) \arrow[d, "\mathrm{d5}"'] & & (\mathrm{U1},\ \mathrm{V1},\ \begin{array}{l}\mathrm{READ}\\ \mathrm{DELETE}\\ \mathrm{UPDATE}\\ \mathrm{INSERT}\end{array},\ p,\ \mathrm{false},\ \mathrm{D2}) \\
(\mathrm{U2},\ \mathrm{O1},\ \mathrm{READ},\ p,\ \mathrm{false},\ \mathrm{D4}) & &
\end{tikzcd}$$

Figure 2

AUTHORIZATION GRAPH BEFORE RECALL

(It is used for illustration purposes only and need not be stored explicitly in the system.)

We assume DBA1 is authorized to define relations and hence basic classes. Initially DBA1 has the set of administration rights, A_B, for the basic class D1. Classes D2 and D3 are defined by DBA1 as subclasses of D1 and are delegated to DBA2 and DBA3 respectively (d1 and d2). Both DBA's are given the set of administrative rights, A_S, associated with a subclass, but only DBA2 may further delegate these rights. DBA2 defines D4 as a subclass of D2 and delegates the right to authorize read access to objects in D4 to DBA4 (d3). DBA2 also defines an application view V1 which is a relation constructed from the objects in class D2 and authorizes user U1 to have all access rights to it (d4). DBA4 grants U2 read access to object O1 in class D4 (d5). The associated class structure graph is shown in Figure 3. IF DBA1 recalls all delegated rights for D2 then the class structure subgraph for D2 is traversed and all administrative rules associated with the nodes of the tree are revoked from the relevant DBAs and given to DBA1. The situation after revocation is shown in Figure 4 and is logically equivalent to DBA1 having authorized all the access rules. Notice users U1 and U2 are still authorized to access the database.

$$\begin{tikzcd}
& D1 \arrow[dl] \arrow[dr] & \\
D2 \arrow[d] & & D3 \\
D4 & &
\end{tikzcd}$$

Figure 3

CLASS STRUCTURE GRAPH

$$\begin{tikzcd}
& (\text{DBA1, D1, } A_B\text{, true}) \arrow[dl, "d5"'] \arrow[dd, "d4"'] \arrow[dr, "d2"] & \\
(\text{U2, O1, READ, p, false, D4}) & & (\text{DBA3, D2, } A_S\text{, false}) \\
& (\text{U1, V1, } \begin{matrix}\text{READ}\\ \text{DELETE}\\ \text{UPDATE}\\ \text{INSERT}\end{matrix} \text{ ,p, false, D2} &
\end{tikzcd}$$

Figure 4

AUTHORIZATION GRAPH AFTER RECALL OF CLASS D

ALGORITHMS FOR DELEGATION AND RECALL

In this section we present high-level algorithms for delegation and revocation. Other necessary algorithms, such as those for defining classes and authorizing access are straightforward and have therefore not been included. We assume that the control information is a set of relations. In particular, authorization rules are contained in the relation AUTH defined as

AUTH (<u>s</u>,<u>O</u>,<u>t</u>,f),

where the column names are as previously defined and the underlines indicate the identifier of the tuples. Only administration rules are used by the algorithms described in this section. The following algorithms are written in a pseduo ALGOL and describe procedures which are invoked by some suitable language used by the DBA for authorization. The notation

RELATION_NAME.col_namel[col_name_2=val,...]

indicates a selection of tuples based on the criteria specified in brackets, followed by a projection onto col_name_1. For example,

AUTH.t [s=s', O=O']

selects those tuples in relation AUTH for which the subject is s' and the object is O' and then projects out their access types. Tuples are explicitly inserted and deleted. For example the following statement inserts a tuple (s' , O' , t' , f') into AUTH:

```
    insert (s -- s' , O -- O' , t -- t' , f -- f' ) into AUTH
```

The statement:

```
    delete AUTH [ s=s', O=O' ]
```

deletes the set of tuples from AUTH that have subject s' and object O'.

We consider first the procedure CHECK_RIGHTS which is invoked by all the subsequently defined procedures before any access to the data control relations is allowed.

CHECK_RIGHTS (s' ,O' ,A' ,f') procedure

(This procedure checks that the subject s' has the set of administrative access types A' for object O'. If the boolean variable f' is true the delegation flag must also be true for each access type in A'; f' false indicates a "don't care; condition.)

```
    begin
    if $a_i$   AUTH.t [s=s' , O=O' , fVf' = true]
          for all $a_i$   A'
    then return;
    else call ENFORCEMENT:
    end
```

If the set of access types A' is to be delegated, the flag f' must be true for all the rights in A'. If any of the checked rules is not found, a system-defined enforcement procedure (ENFORCEMENT) is invoked which, for instance, may notify a security operator of the illegal access.

DELEG_WITH_RECALL is the procedure used for delegating administrative rights for a class from one administrator to another while retaining the right to recall these rights.

DELEG_WITH_RECALL (s' , s'' , D' , A_F) procedure (This procedure is invoked when administrator s' delegates rights for class D' to administrator s''. A_F is a set of ordered pairs a,f supplied by the administrator s' , which represent the set of access type, delegation flag pairs delegated.)

```
begin
call CHECK_RIGHTS (s' ,D' , A_F. a, true);
for all pairs   $a_i$, $f_i$      A_F
    begin
    insert (s -- s'' ,O --D' , t -- $a_i$, f --$f_i$)
         insert AUTH:       .
end
```

The CHECK_RIGHTS procedure is invoked to validate that the delegator does have the administrative rights being delegated and that the delegation flag is true for each of them. The delegated rights for class D' are then inserted into AUTH on behalf of the delegatee. All the delegator's rights to class D' are deleted. Finally the delegator is given the right to recall the delegated administrative rights for the class. Note we do not allow this right to be delegated.

The delegation policy allows an administrator to delegate the rights for a class to only one administrator. If it is desirable to have multiple administrators for some set of objects, overlapping classes must be defined and separately delegated. This avoids the situation where an administrator receives administrative access to a class from two different delegators. Although the existing access rules associated with the objects in class D' are now logically the responsibility of the delegatee, no physical alteration of the rules is necessary.

Upon recall of a class the administrative rights that were initially delegated for that class are restored to the delegator and removed from the delegatee. However, there may now be a number of delegatees from whom administrative rights must be removed. This is because the initial delegatee may also have delegated the class. Furthermore, it is not sufficient just to remove all administrative rules from AUTH which are associated with the recalled class because subclasses may have also been defined. Thus, the CSG for the delegated class, which we assume is a by-product of the procedure for class definition must be examined and the administrative rules associated with each class corresponding to a node in the tree must be deleted. The recall procedure is defined as follows:

RECALL (s' ,D') <u>procedure</u>

(This procedure is invoked when s' recalls all delegated administrative rights for class D'. SAVE_ACC is a variable set by procedure PROP_DOWN containing the set of delegated access rights.)

```
     begin
          call        CHECK_RIGHTS (s' ,D' ,a7, false);
          call        PROP_DOWN (D');
          for all $a_i$  SAVE_ACC insert
                  (s -- s' , O -- D' ,t -- $a_i$, f -- true)
                                                into AUTH:
          delete AUTH [s=s' ,O=D' ,t=a7];
     end
```

The procedure PROP_DOWN is defined as:

```
     PROP_DOWN (D) procedure
```

(SAVE is a function which inserts access rights into the variable SAVE_ACC. CHILD is a function which provides the children of a node in the CSG.)

```
     begin
          SAVE(AUTH.t[O=D]);
          delete AUTH[O=D];
          for all $D_i$   CHILD (D) call PROP_DOWN ($D_i$);
     end
```

CHECK_RIGHTS validates that s' has recall rights for class D'. The PROP_DOWN procedures is used by RECALL to delete the administration rules associated with the classes specified in the call parameter. The function CHILD(D')

provides the subclasses which are the immediate children of class D'. This function is used recursively to identify all the nodes of the CSG for D'. Before deleting the rules for the subclasses of D' the access types are saved by the function SAVE into the set SAVE_ACC. The recall procedure then restores the administrative rights of s' for class D' using the administrative access types stored in SAVE_ACC. Finally the right for s' to recall class D' is deleted. Notice that since the access rules (indicating regular database access) do not indicate who is the administrator that wrote them, there is no need to modify them when a recall has occurred.

APPENDIX C[1]

RECOVERY METHODS

Recovery of the database when the basic configuration is dumping plus logging can employ a number of recovery methods. Which particular method will depend on the particular recovery situation and the recovery points provided. Recovery must take place to a consistent state of the relevant part of the database. The recovery points may be of two types: transaction recovery points which lie either on transaction or integrity unit boundaries; and system recovery points, which are checkpoints. Therefore there are two general types of recovery. Forward recovery is used where physical damage has occurred to the storage media. The other type of recovery is backward recovery which may be divided into off-line backward recovery and quick (or dynamic) backward recovery. In either of these cases the storage media are not damaged; what is desired is to reverse the changes made by partially completed transactions.

For all methods of recovery in a transaction oriented environment, it is advisable that the log records transactions. If transactions are not recorded then ambiguities may occur

[1]Reference 2

on restart as users may be unsure of which transactions completed successfully. The integrity of the database is maintained by the system but it may be destroyed by the terminal operator. The terminal operator may assume that processing of a previous transaction has completed successfully when it has not and he therefore does not re-input the corresponding input data. Similarly, assumption of non-completion of a transaction that has completed successfully leads to retransmission of the input data and double updating of records.

1. Roll Forward

In this method of recovery the procedure is the following:

(i) Restore the database or a particular area of the database from a dump copy.

(ii) Align the log file containing after images to a system recovery point (checkpoint) corresponding to the restored state of the database.

(iii) Apply after images until a nominated system recovery point is reached. This would normally be the last checkpoint before failure.

(iv) Restart processing of transactions from the nominated recovery point by receiving terminal inputs.

A search is made of the log file between the last checkpoint and the point of failure and only those transactions on the log file which do not have a corresponding end of transaction

indicator transmit output messages. The transactions which did complete successfully are rerun but message output is suppressed.

If an orderly termination of processing was not possible due to the type of failure, then it may be necessary before restarting to write an end-of-file (EOF) manually on the log file.

If duplicate output messages are to be suppressed, then stage i), ii) and iii) are as before followed by:

(iv) Search log file betwen last checkpoint and failure.

(v) Reprocess all transactions on log but suppress output messages for completed transactions.

(vi) Restart processing of input from terminals.

2. <u>Roll-Forward With Roll-Back</u>

The procedure for this method is the following:

(i) Restore the database or a particular area of the database for the dump copy.

(ii) Align the log file containing after images to a system recovery point (checkpoint) corresponding to the resorted state of the database.

(iii) Apply after images until the end of the log.

(iv) Apply before images back to the last system recovery point in order to achieve a consistent state of the database.

3. Re-Run

The procedure is the following:

(i) Restore the database or a particular area of the database for the dump copy.

(ii) Align the log, which only contains transactions records, to a point corresponding to the restored state of the database.

(iii) Reprocess all transactions until the end of the log file.

(iv) Restart processing of terminal input.

If end of transaction indicators are written on the log file, then output messages for those transactions can be suppressed so as to prevent output message duplication. Enquiry only transactions may be ignored if so chosen as these have no affect on the database.

Roll-forward, roll forward with roll-back and re-run are examples of forward recovery whereas the following method is a method of backward-recovery.

4. Roll-Back

The procedure is the following:

(i) Apply before images back to either:

(a) start of the failed command

(b) start of an integrity unit

(c) start of transaction

(d) system recovery point (checkpoint)

(ii) The log file is realigned to a point corresponding to the roll-back.

(iii) Processing of transactions is restarted.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Commander of the RSNF Naval Forces Headquarters Riyadh, Saudi Arabia	2
5.	Director of Planning and Operations Department Naval Forces Headquarters Riyadh, Saudi Arabia	1
6.	Director of Informations Department Ministry of Defense and Aviation Riyadh, Saudi Arabia	1
7.	Professor Norman F. Schneidewind Code 54Ss Administrative Sciences Department and Computer Science Department Naval Postgraduate School Monterey, California 93940	1
8.	Lieutenant Fahad A. Al-Lahaidan Post Office Box 818 Taif, Saudi Arabia	3

(duplicate)
Thesis
A37635 Al- 197987

Thesis 197987
A37635 Al-Lahaidan
c.1 Maintaining the integrity of distributed database.

Thesis 197987
A37635 Al-Lahaidan
c.1 Maintaining the integrity of distributed database.