Distributed Databases: A Review of Problems and Solutions

by

Paul M. Burega

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Computer Science
in
Computer Science

Winnipeg, Manitoba

DISTRIBUTED DATABASES: A REVIEW OF PROBLEMS AND SOLUTIONS

BY

PAUL M. BUREGA

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Paul M. Burega

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Paul M. Burega

ABSTRACT

The manipulation of large quantities of data is an increasingly important topic as more and more applications are computerized. The techniques used to manage a distributed database are crucial to the success of most scientific and business applications. In this thesis, we examine the techniques used to manipulate data that are distributed among or between several computers.

The problems incurred by connecting together geographically dispersed computers are examined and some solutions given. Problems such as data integrity and security over the transmission medium are discussed. A look is taken at a general network model, the International Standards Organization Open System Interconnect model, along with a look at some commercial networks.

There are many problems with distributed databases, some of which occur because of the distributed environment. Conflicts arise between concurrently executing transactions. If the conflicts are not handled correctly, they can offset any performance gains made by the distributed environment. The two main methods for controlling concurrent execution, locking and timestamping, are compared and contrasted, along with some new hybrid techniques.

We also examine some current distributed database systems in detail. Major emphasis is placed on the methods used for concurrency control, as this is a major problem in a distributed database management system.

ACKNOWLEDGEMENTS

I would like to thank Dr. A. N. Arnason and Dr. D. H. Scuse who helped motivate this thesis, and who provided guidance and direction when it was most needed. I would also like to thank Dr. W. J. Davidson for his many welcomed contributions towards the content of this thesis. Special thanks to Dr. C. M. Laucht for his many suggestions on polishing this thesis.

Finally, I would like to acknowledge the support given to this work by the University of Manitoba and the National Sciences and Engineering Research Council of Canada.

CONTENTS

LIST OF FIGURES

Chapter I

INTRODUCTION

The manipulation of large quantities of data is an increasingly important topic as more and more applications are computerized. The techniques used to store, retrieve, and modify data are crucial to the success of most scientific and business applications. In this thesis, we examine the techniques used to manipulate data that are distributed among or between several computers.

The most important reason for distributing computing power is the relative price of computing power versus the price of communications facilities (cheaper memory, faster processing) [Marti79b]. The price of computing power has fallen drastically within the last decade, due partly to the introduction of the micro-computer. This means that it is often more economical for a company to put a computer at a remote location, rather than pay the cost for high speed communication lines to support terminals. It is cheaper to process or analyze the data at the place it is generated, and only send occasional summaries to the main computer centre. This reduces communications cost, which now represents a much larger percentage of the total cost than it used to. By using a remote computer, it is possible to use communications lines during off-peak hours to realize further savings.

There are two methods of storing collections of related data on computers. The simplest of these is to store data in standard files.

This technique is adequate if only a few programs access the data, and there is no restriction on the length of time needed to process a query. These programs must be concerned about the physical layout of the data. If the record format changes, then all the programs must be re-compiled. Furthermore, each program can see every part of all records. This is not desireable. As the number of users with access to a given set of data increases, so does the need to restrict access to certain fields to specified users. To eliminate these and other problems, a database system should be used. This allows application programmers to access data logically and independently from the physical format of the data. A database system also provides the required integrity and security for the data [Atre80, Carde79, Date83, Marti77].

Once a database is in use, new problems arise as more users require access to the data. The increasing numbers of accesses require more processing. If these users are online, their requests should be completed in a brief period of time, e.g. five seconds. To accommodate this time requirement, a fast computer must be used; however, such a computer is not always available. Therefore several computers may be required to share the processing to meet the access requirements.

With an increasing number of users accessing a database, it is inevitable that some of these will not be situated near the computer complex. They may be across the street, at the opposite end of the city, or continent, and in extreme cases, on the other side of the world. If terminals are placed at remote locations, it is usually necessary to have one communication line per terminal. As the number of terminals in one location increases, so does the cost of the

communication lines. Communication lines delay data transfer and increase the apparent response time to the end user [Marti81a]. To improve performance and lower the cost, a computer system may be installed at the remote site to handle the terminal queries. This decreases response time, increases availability due to fewer communication lines, and allows data to be stored near where they are used. As an example, take the case of a firm with offices downtown and a warehouse on the outskirts of town. If an inventory system is run on a computer in the warehouse instead of on the computer in the business office, then not only do the warehouse terminals get better response, but so does the downtown office computer, since it no longer needs to process the inventory program. But what if a user on one computer system requires data from the other?

In order to share data between or among computers, the computers must be linked together. If the two computers are physically close, it is possible to have a high-bandwidth channel connecting the two computers. This method of connection allows a transfer rate in excess of one megabyte per second and each computer looks like an input/output device to the other computer. With this method, both computers need not be general purpose computers. One of the computers could be a dedicated database processor specially designed to perform database operations. The other computer would then be the host to which terminals would be connected. The host would perform general computing and pass special database requests to the database processor. This offloads expensive database transactions from the host and allows a more suitable machine to do the transactions. There may be large transfers of data between

the two computers, and hence a need for a high-bandwidth communications channel.

Where the two computers are separated by a larger distance, for example in different buildings or across town, then communication lines must be used. Typically, the greater the distance, the lower the bandwidth available for reliable communications [Marti79b]. When both computers reside on the same property, it is possible to run wires between the two machines. If, however, the computers are on separate pieces of property, then common carrier transmission lines must be used (telephone company lines). These lines may be specially designed to give lower error rates and higher bandwidth than normal telephone lines, but the maximum bandwidth would typically be around fifty thousand bits per second (higher rates are available, but may be beyond economic reach of many companies [Pooch83]). The telephone company may also use microwave and satellites to complete part of the communications link [Marti78].

As soon as there is any distance between the two computers, the maximum bandwidth available economically drops drastically and the error rate increases. In order for distributed databases to function, there must be network hardware and software to compensate for the problems created by distance. This implies sending as little data as possible and employing error detection and correction mechanisms to make the communications facilities useable but transparent to the end user.

It also makes sense to store the data where they are required. Data frequently used at one location may rarely be needed at another, but

they must remain accessible. To transmit all of these data to another computer may be unnecessarily complex and expensive. The justification for local data is that everything works much more smoothly when the accuracy, privacy, and security of the data are a local responsibility [Marti79a]. This also increases reliability. If one computer is not functioning and the others are, at least some work may still be done.

In other cases, there is no natural distribution to the data, but the organization or user may want to keep important data at two or more sites for increased safety and reliability. For example, a bank whose head office is located in an earthquake zone or floodplain may want to keep a duplicate of its database in a distant city. In some applications such as the military, even short periods of data unavailability due to hardware failures are completely intolerable; hence the need for redundancy.

Yet another reason for distributing data is sheer size. It is not feasible to attach an arbitrary number of disk drives to a computer. Beyond some finite limit, multiple machines are needed to handle the large amounts of storage. To allow incremental growth, it is better to have a collection of smaller systems than one huge one. With a single central system, it is possible to reach a point where adding one more disk drive is impossible because of the finite number of devices that can be connected to one computer.

A related reason for having multiple computers is the need for a high transaction rate. With present computing technology, a throughput of a few hundred transactions per second per central processing unit (CPU) is

considered excellent [Marti79b]. To achieve a better rate, additional CPUs are needed. However, if all the disks are attached to one CPU, at some point the interrupt load will saturate that CPU. This upper limit on the load will force the database to be distributed among the other CPU's.

A distributed database may have many different forms. There is no need for all the computers which process the database to be the same, nor for the database program to be the same. Each computer may have unique data or, for efficiency and reliability, it is possible for multiple computers to have copies of the same data. Thus the two extreme types of distributed databases are: fully partitioned and fully replicated. A fully partitioned distributed database is one where each unique data item is stored at exactly one location, whereas a fully replicated distributed database is one where each unique data item is stored at all locations. Typically, a compromise between the two is chosen, as a replicated database is faster to access, but more complicated to update. Thus, if updates are rare, then replication is very attractive. As updates become more frequent, replication quickly loses this attractiveness. Replication gives faster response for queries, and if one computer malfunctions, then the other will still have current copies of the data.

While a distributed database solves some problems, it can create several new ones. The problems of integrity and security become much more difficult. Maintaining the integrity of the database is a problem because the database is distributed over several computers, some of which may contain identical copies of the same data. The difficulty

arises in keeping those copies identical, even though a computer may become disconnected from the others because of problems. Security also becomes more of a problem, because a network as a whole is more vulnerable to attack. To enable computers to communicate with each other, there must be a network connecting the computers. This connection could be via phone lines, microwave, satellite, or some combination of the three. Once unauthorized access is gained at any point, access is allowed to the whole network without much further difficulty. A further problem with distributed database is the need to allow different machines which may have data in different formats, to talk to on another. The rules, conventions, and procedures which permit computers to communicate with each other are referred to as communication protocols. The definition and implementation of these protocols has a fundamental impact on the speed of communication as well as on the integrity and security of the data being passed from one computer to another.

Supporting the parallel processing of requests is also a problem in a distributed database system. It is normal for queries to be initiated from multiple hosts simultaneously. Consequently, multiple queries may attempt to read, or even worse, update the same data item at the same time. Allowing unrestricted concurrent access is intolerable because it can lead to undetected semantic errors in the database. Take as an example, a banking database in which two transactions are both about to update the same account record. One transaction will deposit $100.00 to the account, and the other transaction will debit $100.00 from the account. If both transactions are run concurrently, then both

transactions will read the same initial balance. One of the transactions will complete before the other, and install its updated balance first. When the other transaction completes, it will write over the results of the first one. Since neither transaction saw the effects of the other, the final balance is clearly incorrect. The balance should have remained the same as the deposit and debit should cancel each other. This is what would happen if the transactions had not run concurrently.

While allowing unrestricted concurrent access is intolerable, allowing only a single transaction to run at a time eliminates the concurrency errors but at the price of greatly reducing the performance of the system. Most transactions do not interfere with one another and could be run concurrently without danger.

An important research issue is the design of concurrency control algorithms which maximize the amount of parallel activity, while maintaining the semantic integrity or correctness of the database. This problem is greatly amplified in a distributed environment where multiple copies of data exist. It is possible for two transactions to simultaneously update two copies of a data item. This problem occurs because there is a noticeable time delay through the network for update information to pass from one host to another. The distributed database system must be able to cope with this situation.

Another problem which is crucial in a distributed database system is crash recovery. Although modern computers are reasonably reliable, failures or crashes do occur. The more components there are in a

computer system or network, the greater the chance that one or more components will fail at some time. In addition, machines must be taken offline for preventive maintenance, making backups, or other purposes. In many networks, users expect the database system to continue to operate even though a few hosts are down. Furthermore, when a missing host comes back online, it must be able to resynchronize itself by applying all new update transactions, without causing deadlock or consistency errors. (Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each waiting for one of the others to release a lock before it can proceed.)

Distributed databases are most useful when the data are collected in widely separated places, when the database is large, or when the transaction rate is high. Distributed databases have some additional complications not found in a centralized database. Among these problems are where to put the data (and how many copies to make), where to do the query processing, how to perform concurrent accesses and multiple-copy updates efficiently and without deadlock, and how to keep multiple copies of the database in synchronization in the face of system crashes. In the remainder of this thesis, we examine the problem of manipulating databases that are distributed over multiple computer systems and indicate how current and proposed distributed database systems handle the problems involved in distributed databases.

Chapter II

DISTRIBUTED DATABASE ENVIRONMENT

In this chapter, we will discuss computer systems in general and show a growth path from a single computer to a distributed environment. This distributed environment will then be discussed in some detail, with a look at the general hardware and software needed to support a distributed environment.

Given a distributed environment, the services provided by a communications network and the impact they have on a distributed database system are examined. A need is shown for text compression and encryption in a distributed environment, and a closer look is taken at how these two options can increase the speed and security of a distributed database management system.

Finally, a look is taken at some of the existing commercial networks. As well, an indepth look is taken at RelNet, a network designed especially for distributed databases. RelNet is able to offload some of the problems of operating in a distributed environment from the distributed database management system by providing a more reliable network communications system than a typical commercial network.

2.1 COMPUTER SYSTEMS

Until the mid-1970's, most computing was carried out by (third-generation) systems employing a large, centralized computer. The CPU had a diverse collection of relatively simple machines connected to it, some of them connected via telecommunications links. These simple machines consisted of card readers and printers, as well as 'dumb' terminals. By 'dumb', we mean that the terminals could only display text, and did not have any fancy functions, such as block transmission, field validation, etc. They would even have problems if the data arrived too fast. For certain applications, networks were built into the computer system allowing large numbers of simple terminals to be connected to this central computer system. In some installations, there were even two processors for reasons of reliability, but the processing of each transaction was done by one large computer [Marti79b].

The 1970's were the era of first the minicomputer, and later the microcomputer. With large scale integration (LSI) circuits, the cost of building a processor dropped steadily until it became clear that one computer system could be made up of many processors if this were useful. There was a concomitant change in the perception of how computers should be used. The concept of an isolated, factorylike machine room processing batches of data for many users gave way to users wanting their own terminals and processing capability. In some cases, the users' local processing machines were connected to a distant, larger machine which maintained a database and provided extra processing power if needed.

By the mid-1970's, requirements for a new type of system architecture had become clear. This new (fourth-generation) architecture must provide a stable foundation for on-line, transaction-driven database applications. Unlike the architecture designed for batch processing, it must be highly reliable because on-line users become very frustrated if their systems have frequent periods of failure. This architecture must meet a diverse range of processing requirements and, as technology is changing very rapidly, the architecture must facilitate the introduction of new technology without a major system disruption. Large-scale integration has given way to very large-scale integration (VLSI).

Figure 2.1 illustrates a common configuration of a computing installation from the 1970's. Economies of scale in computing lead to centralization and all work is funnelled into centralized, factorylike, data-processing shops. The computer is capable of running multiple programs concurrently. These programs are usually refered to as processes or tasks. Each process has a specific function, such as a database management system or interactive terminal editor. Now, micro and mini-computers have the power of these mainframe computers.

The reason for the growth of microcomputers is the use of VLSI circuits which are mass-produced. Not only can small machines be mass-produced, but also their development cycle is much shorter than that of large machines. Therefore, they tend to use later technology which is cheaper because the technology is dropping in cost. Costs can only drop so far, and then instead of the cost decreasing further, the product becomes smaller and faster. The price/performance ratio on all computers will continue to drop, but it will drop much more rapidly on

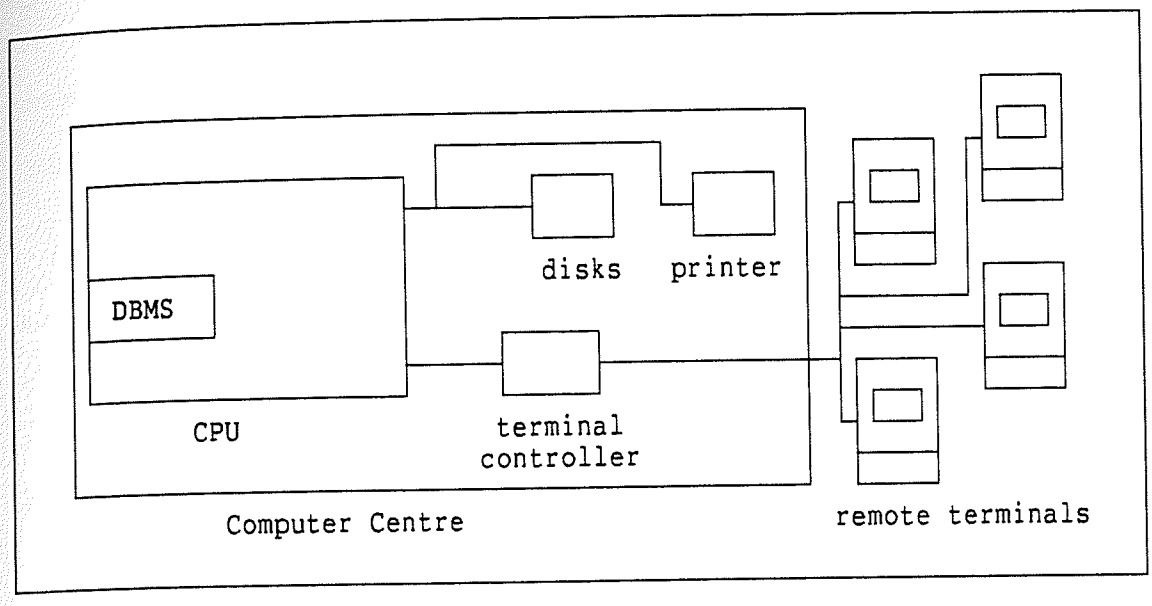

Figure 2.1: A typical centralized Computer Organization

tiny, mass-produced machines than on machines costing hundreds of thousands of dollars. It will thus be much more economical to have thousands of small computers than one large computer, with the combined computing power of the small computers much larger than that of the fastest large computer.

However, there will still be a need for large computers. To meet the requirements of the vast computing resources needed to serve large numbers of people, computer systems will interconnect many processors, both large and small. The computers will often be interconnected over large distances with computer networks. The term distributed processing implies multiple processors, usually interconnected by a telecommunications system (figure 2.2).

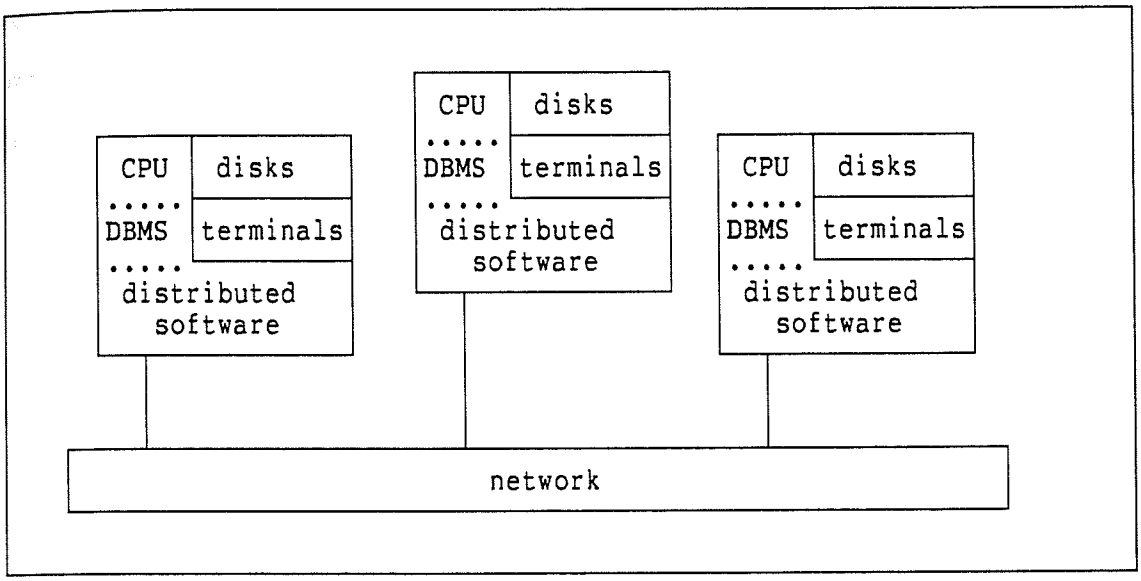

Figure 2.2: A Distributed Computer System

While many computers can be connected together over large distances, a user will typically just interact with the computer to which his terminal is connected. When the user enters a transaction, he does not care if the data are found on, and the computations carried out on, his local machine or on a remote computer in the network. The user of a distributed system should not be aware that there are multiple processors; the system should appear as one large, virtual processor to the user.

There are essentially two reasons why a transaction is sent to a remote machine as opposed to processing it on the local machine [Marti79b]. The first is that the local machine has insufficient power. Typically the local machines will be small, while the central processor will have large number-crunching power. The second is that the transaction needs data which are stored elsewhere. Most transactions do

not need much computer power, and hence these transactions could be processed on an 'intelligent' terminal or on the controller to which dumb terminals are connected.

Formerly the term teleprocessing was used to imply the use of telecommunications facilities for accessing processing power. Now calculators and minicomputers are cheap and what was originally done by terminals connected by telecommunications facilities to remote machines can be done on a local machine. The local machine may itself be connected by telecommunications links to other machines, and a transaction may then be processed either on the local machine or on a remote one. The main reason for teleprocessing then becomes to obtain data, not to obtain processing power.

While teleprocessing allows for the access of remote data, the speed of access is orders of magnitude slower than that of a computer's own local data. For infrequently accessed data, this performance degradation is hardly noticable, and is much better than not being able to access the data at all. For frequently accessed data, the slow speed of telecommunications degrades performance of the computer significantly. The computer performs almost no computation, but instead waits for data transmission from remote computers.

If many of the computers are awaiting data transmission of remote data, then some computers will be under-utilized while others may have processing backlogs. Full use is not being made of the computing resources in the network. The computers have been distributed by the network, but the processing loads have not.

To distribute the processing loads among the computers, it is usually necessary to distribute copies of the data among the distributed computers. The location and design of this data are particularly important in a distributed system. Data placed in the wrong spot can cause performance degradation and unbalanced processing loads, while data placed correctly causes balanced processing loads which thereby increases system performance.

When data are used at multiple locations, it is often desirable to access the data by a database management system (DBMS). The DBMS manages local data, and provides security and integrity for the local computer user. Data should not be arbitrarily distributed, as the economies of scale in storage systems are different from those in processors. The cost per bit stored on very large storage units is orders of magnitude lower than on small storage units [Marti79a]. While it is more cost effective to have many small computers than one large one, it is not cost effective for each of these small computers to have large data storage units.

When data are stored in multiple locations, each copy may have different changes made to it by its computer. Multiple copies of data are only useful if they remain consistent. If each copy has different changes applied to it, then the copies are no longer true copies, and become inconsistent with one another. To keep consistency, changes made to one copy of the data must eventually be reflected in the other copies of the data.

To co-ordinate these changes, a reliable telecommunications facility must be used. The facility must be reliable at all costs. For data to be consistent, and hence of value, update messages from one computer to another must arrive without being lost or garbled beyond recognition. Even a small change in the update message will cause a loss of consistency, defeating the purpose of having replicated data. Any performance enhancement must ensure reliable results. If the results are questionable, then performance has been lost, not gained. A facility which allows reliable telecommunications between computers is called a network.

2.2 COMPUTER NETWORKS

A network consists of the hardware and software that supervise the transmission of data between two or more computers. Depending on the type of network, the hardware for interconnection can range from a passive cable to a collection of special purpose computers whose job is to route the data from the sender to the receiver.

Once computers are connected together electrically, there must exist a standard set of software protocols to allow the computers on the network to converse with each other. These protocols are necessary to ensure reliable communications (no lost or damaged data), as well as to enable one computer to interpret the data it receives from another. If the computers use different codes for transmission, then neither will understand the data they have received. These network protocols must also be able to detect a failed computer, and report this status to other computers in order that they no longer send messages to, or await replies from, that failed computer.

Once reliable communications are established, precautions may also be necessary to safeguard the security of data transmission. The transmission cables are often public, such as telephone lines and cables, and it is possible for an unauthorized computer system to tap in and extract confidential data. Not only must data be safeguarded against passive intruders (intruders who just listen to the data transmissions), but they must be safeguarded from active intruders who are trying to pass themselves off as an authorized network computer.

These problems must be overcome to establish and maintain an environment suitable for a distributed database management system. In this section, various general solutions to these problems are given. Later sections of this thesis will detail specific examples of systems using these solutions.

2.2.1 Components

In any network there exists a collection of machines intended for running user or application programs. One of the first major networks was ARPANET, and much network terminology comes from this network. The collections of machines were called hosts, and they are connected by the communications subnet, or subnet for short. The job of the subnet is to carry messages from host to host. By separating the pure communications aspects of the network (the subnet) from the application aspects (the hosts), the complete network design is greatly simplified (figure 2.3).

In order to communicate, the two computers must have rules for data transfer. There are three possibilities: data can travel in one

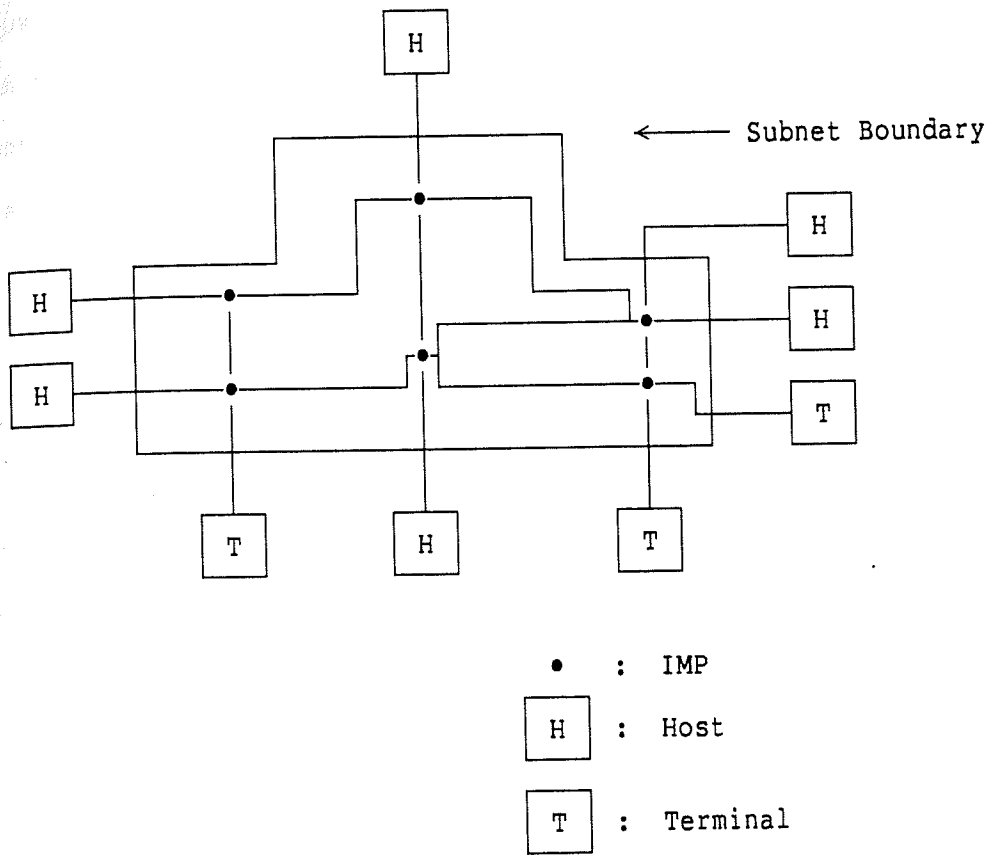

Figure 2.3: A Typical Point-to-Point Long Haul Network
(modified from [Marti81b])

direction; data can travel in both directions, but not simultaneously; or data can travel in both directions simultaneously. The first of these possibilities is called simplex communication, while the other two methods are called half-duplex and full-duplex respectively.

Simplex communication requires two transmission cables; one for incoming data and one for outgoing data. Each cable may be fully utilized whenever needed. Half-duplex communication requires only one cable (which is cheaper), but the cable transmits in only one direction at once (either send or receive). To reverse directions, the sender

must notify the receiver, and after a suitable time delay, transmission in the opposite direction can occur. Note that a receiver must wait until a sender notifies him that he can now send. Urgent messages cannot leave until all incoming transmissions are completed.

Full-duplex allows simultaneous reception and transmission on one cable. The one draw-back is that the sum of the send and receive data rates must be less than the bandwidth of the cable. If the bandwidth of the cable is greater than twice the send or receive rate, then no problems occur. However, if the bandwidth of the cable is low, then the send and or receive data rates must be low (it is not necessary for both rates to be the same). If a higher bandwidth is needed, it is best to use two simplex cables.

While a network allows the physical connection of multiple computers, this does not necessarily allow the computers to understand one another. Consider the example of an English speaking person on the phone to a French speaking person. Even though there is a physical means of communication between the two, they are unable to understand one another. To enable the increasing variety of computers to communicate with one another, there must be a set of rigorously defined protocols. Along with the definition of a protocol, the formats of the control messages and the headers and trailers of the data message must likewise be rigorously defined.

The protocol must also determine how many logical channels the connection corresponds to, and what their priorities are. Descriptions of data and of the relationships between data are of two forms: logical

or physical. Physical descriptions refer to the manner in which data are recorded physically on the hardware. Logical descriptions refer to the manner in which data are presented to the user of the data. Many networks provide at least two logical channels per connection, one for normal data, and one for urgent data.

When numerous networks have been previously set up, it is often useful to allow two or more networks to communicate with one another. Interconnection of two networks is not a simple task. Different networks may have different packet, addressing, interface, protocol, error-control, protection, resource management, accounting, and other structures and algorithms. To connect two nonsimilar networks, translation at one or more levels is required to effect interconnection. The collection of hardware interfaces, software interfaces, and services necessary to effect network interconnection, is called a gateway. The gateway appears to each network as a normal node of that network. The gateway takes the data in the format of one network and converts it to the format expected by the other network. This allows non-similar network architectures to be connected transparently, at least in theory.

Networks can be interconnected at any level where equivalent services exist within each network. However, equivalent services may not always exist. To achieve equivalent services at the gateway interface, the services of one or both networks to be interconnected may be augmented by a layer of functions within the gateway or within gateways and hosts [Benha83, Hinde83]. Problems occur with finding matching sets of services at all levels, the main problem being that there could be an enormous number of services when all layers of the network are considered. They must all be mimicked for network interconnection.

2.2.2 Communications Subnets

In nearly all networks, the subnet consists of two basic components: switching elements and transmission lines. The switching elements are generally specialized computers called IMPs (Interface Message Processors) or nodes. Transmission lines are often called circuits or channels. Each host in the network is connected to the subnet via one or more IMPs, and all traffic to or from a host goes via its IMP. Several hosts may share one IMP.

There are two main types of communications in computer computer subnets: packet-switching and circuit-switching [Tanen81a].

A packet-switching subnet divides the data traffic into blocks, called packets, which have a given minimum and maximum length. Each packet of user data travels in a data 'envelope' which includes the destination address of the packet plus some control information. Each node in the subnet reads the packet, examines the address, and by using knowledge about network conditions, sends the packet on its way to another node. The packet eventually reaches its destination, at which point the control information is removed and the original message is assembled from the packets.

A circuit-switching subnet establishes a physical circuit between two machines. The circuit is rapidly set up and disconnected for each burst of data. Thus each message has exclusive use of a circuit for the time needed to transmit the message. When the message has been transmitted, then the circuit is released so that it may be used by another message. This is analogous to a copper wire, directly connected for brief periods

of time between the communicating machines. In fact the path is not a simple circuit because time-division switching is used in which many streams of bits flow through an electronic switch, all interleaved with one another.

Computer networks are usually packet switched, but occasionally are circuit switched [Tanen81a]. With packet-switching, no one user can monopolize the transmission line for more than a small fraction of a second (the time needed to transmit the largest size packet). This packet can be sent on its way as soon as it arrives at an IMP, thereby unburdening the IMP, and improving network throughput. With circuit-switching, it is possible for one user to tie up a line for a relatively long time. While this line is in use, no other data can get in or out of the IMPs in the path. This can lead to delays, which decreases network throughput.

There are basically two types of designs for communications subnets: point-to-point channels and broadcast channels [Tanen81a]. In a point-to-point subnet, the network contains numerous cables or leased telephone lines, each one connecting a pair of IMPs. If two IMPs that do not share a cable wish to communicate, they must do so via other IMPs. When a message is sent from one IMP to another via one or more intermediate IMPs, the message is received at each intermediate IMP in its entirety, stored there until the required output line is free, and then forwarded. Thus a point-to-point subnet is sometimes referred to as a store-and-forward subnet. The number of times a packet is stored and forwarded is directly proportional to the length of time needed to deliver the packet.

The second type of communications subnet uses broadcasting. In a broadcast subnet, there is a single communications channel shared by all IMPs. A message sent by one IMP is broadcasted to all other IMPs on the communications cable. In order for the message to reach its intended destination, something in the message must specify for which host the message is intended. After receiving a message not intended for itself, an IMP discards it. A broadcast subnet is more efficient for sending packets that must go to all hosts. Only one packet need be sent, and all the hosts receive it. With a point-to-point subnet, one packet must be explicitly addressed to each intended recipient.

The layout or topology of a broadcast subnet is usually either a bus or a ring. A bus topology is one in which there is one cable, and it is connected to each host once.

In a bus topology, only one machine can transmit at any one time instant. This machine is called the bus master. Since all other machines are required to refrain from sending data when the bus master is sending, there must be some arbitration mechanism on the bus to resolve conflicts when two or more IMPs wish to transmit simultaneously. This arbitration mechanism may be centralized or distributed.

Another possibility for a bus topology is to use a satellite or ground radio system. Each IMP then has an antenna through which it can send and receive data. All IMPs can hear the output from the satellite.

The second broadcast topology is a circular topology: that of a ring or loop. A ring is basically a bus, with the two ends joined together. Each bit propagates around the ring on its own. The bit does not wait

for the rest of the message to which it belongs. As an IMP receives a bit, it immediately sends it back out. Each bit will typically circumnavigate the entire ring within a few bit times, often before the message has been completely transmitted.

In a loop, each message is not retransmitted by the next IMP until the entire message has been received, as in store-and-forward. In a loop, each line might have a different message on it, whereas in a ring this situation is unlikely unless the messages are extremely short.

Broadcast subnets are also subdivided into type types depending on how the channel is allocated. The two types are static and dynamic. A static allocation algorithm divides time into discrete intervals. Each IMP can broadcast only when its specific time interval comes up. Channel capacity is wasted when an IMP has nothing to say during its allocated slot. To improve performance, some systems attempt to allocate the channel dynamically, on demand.

Dynamic allocation methods are either centralized or decentralized [Tanen81a]. In a centralized channel allocation method, there is a single entity which determines who goes next. This might be done by accepting requests and making a decision according to some algorithm. In the decentralized allocation method, there is no central entity; each IMP must decide for itself whether to transmit or not.

User machines connected to a packet-switching network must observe a rigorous set of rules for communicating with the network. Because of this, there is a high degree of international agreement on the protocols for public packet-switching networks, centering around recommendation

X.25 of the Comité Consultatif International de Télégraphique et Téléphonique (CCITT).

X.25 defines the interface between the host (computer or remote terminal), called a DTE (data terminal equipment) by CCITT, and the carrier's equipment (modem), called a DCE (Data Circuit-terminating Equipment) by CCITT (figure 2.4). An Interface Message Processor (IMP) or node is known as a DSE (Data Switching Exchange). This CCITT terminology is in widespread use in public network circles. X.25 defines the format and meaning of the information exchanged across the DTE-DCE interface. Since this interface separates the carrier's equipment (the DCE) from the user's equipment (the DTE), it is important that the interface be very carefully defined.

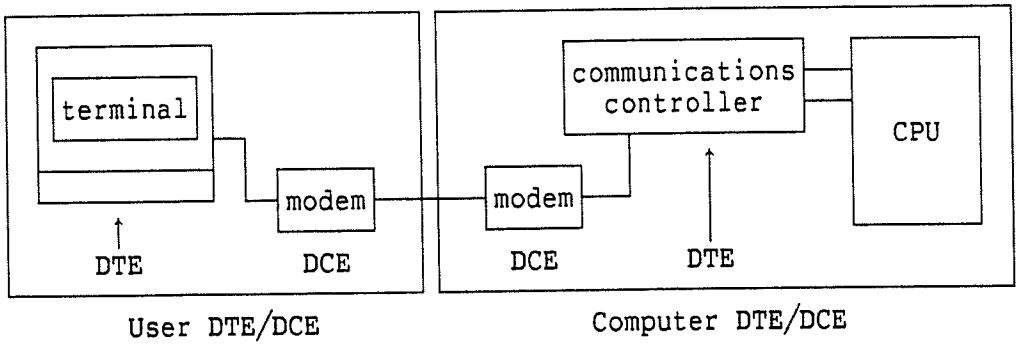

Figure 2.4: Relationship of DTE/DCE

The X.25 standard defines three layers (levels) of communication: the physical layer, the frame layer, and the packet layer. The physical layer deals with how zeros and ones are represented, how contact is established with the network, timing aspects, etc. The frame layer

ensures reliable communication between DTE's and DCE's, even though they may be connected by a noisy telephone line. The third, or packet layer is concerned with the format and meaning of the data field contained within each frame. The packet layer provides for routing and 'virtual circuit' management. These three layers defined in the X.25 model are essentially the same as the bottom three layers of the ISO model which is described later in this chapter.

Under X.25, when a DTE wants to communicate with another DTE, it must first set up a virtual circuit between them. To do this, the DTE builds a CALL REQUEST packet and passes it to its DCE. The communications subnetwork then delivers the packet to the destination DCE, which then gives it to the destination DTE. If the destination DTE wishes to accept the call, it sends a CALL ACCEPTED packet back. When the originating DTE receives the CALL ACCEPTED packet, the virtual circuit is established. AT this point both DTEs may use the full-duplex virtual circuit to exchange data packets. When either side has sent or acquired the necessary data, it sends a CLEAR REQUEST packet to the other side, which then sends a CLEAR CONFIRMATION packet back as an acknowledgement.

The originating DTE may choose any idle virtual circuit number for the conversation. If this virtual circuit number is in use at the destination DTE, the destination DCE must replace it by an idle one before delivering the packet. Thus the choice of circuit number on outgoing calls is determined by the DTE, and on incoming calls by the DCE. It could happen that both simultaneously choose the same one, leading to a call collision. X.25 specifies that, in the event of a call collision, the outgoing call is put through and the incoming call is put through shortly thereafter using a different virtual circuit.

In addition to these virtual calls, X.25 also provides for permanent virtual circuits. These are analogous to leased lines in that they always connect two fixed DTEs and do not need to be set up.

2.2.3 Layered Protocols

Most networks are organized as a series of layers or levels, to reduce their complexity. In this manner, each layer builds upon its predecessor. The number of layers, and the name and function of each layer differ in different networks. In all such networks, the purpose of each layer is to offer certain services to the higher layers. At the same time each layer is shielded from the details of how the offered services are actually implemented in the layers below it. If any changes are made to the hardware or software of a layer, no or relatively few effects should be felt in any of the other layers.

For two machines to converse, layer '**N**' on the first machine carries on a conversation with layer '**N**' on the second machine. The entities comprising the corresponding layers on different machines are called peer processes. No data are actually transfered from layer **N** on one machine to layer **N** on another machine, except below the lowest level. Instead, each layer passes data and control information to the layer immediately below it. This continues for each layer, until the lowest layer is reached. Below the lowest layer there is direct physical communication with another machine, as opposed to the virtual communication used between the higher layers (figure 2.5).

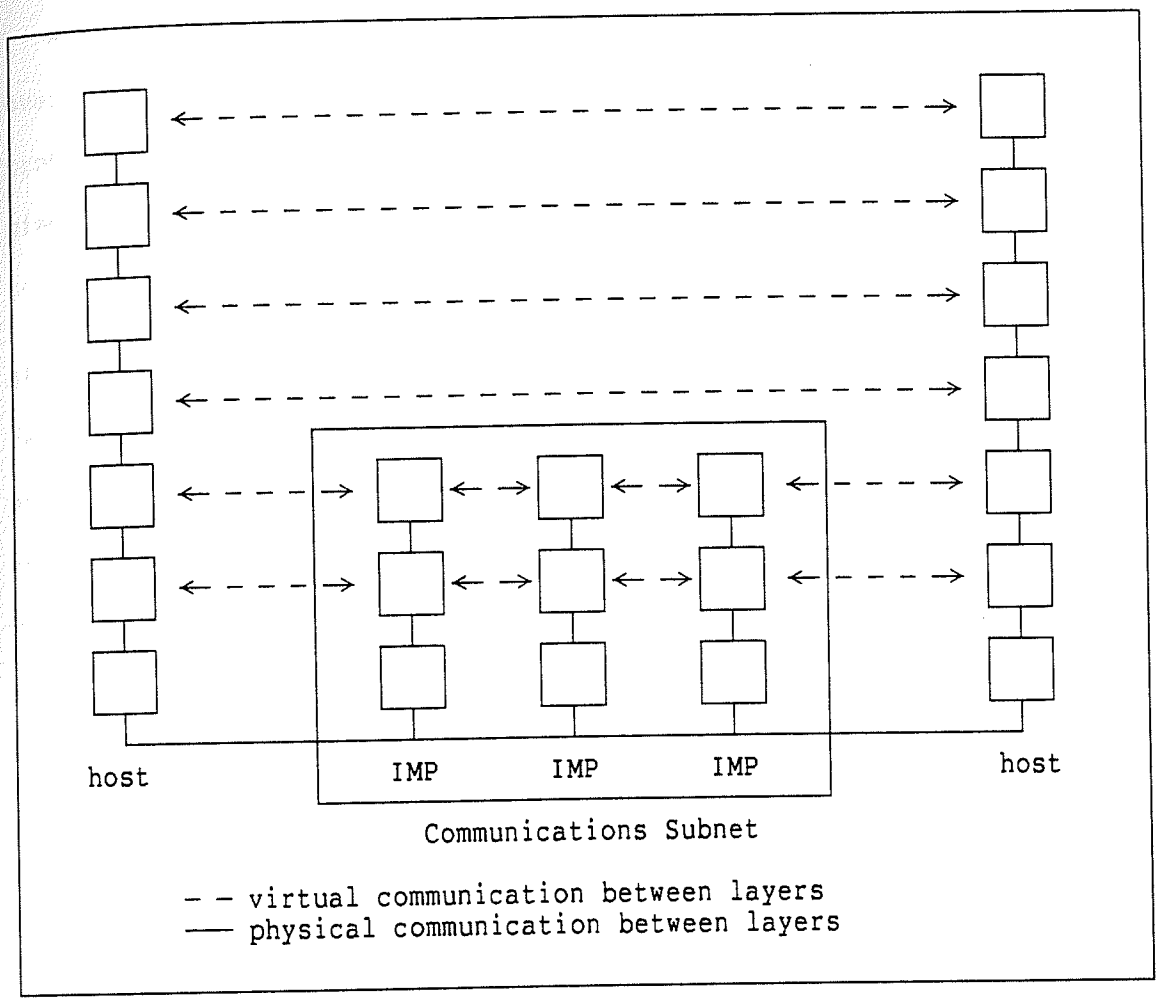

Figure 2.5: Process Communication Between Layers

Since the physical communications circuits are never perfect, there is a great need for error control. Many error-detecting and error-correcting codes are known [Tanen81a, Marti81b], but both ends of the connection must agree on which one is being used. There must also be communication between the receiver and the sender to inform the sender whether or not the messages are arriving correctly, or with errors. If there was an uncorrectable error, then the message must be resent.

Not all communication channels preserve the ordering of messages sent over them. It is possible for messages to arrive out of order, and the protocol must make explicit provision for the receiver to allow the pieces to be put back together in the order the sender intended them.

Since different computers may be connected to a network, and different networks connected together, a problem occurs with the inability of all processes to accept arbitrarily long messages. This problem leads to the necessity for mechanisms for disassembling, transmitting, and then reassembling messages. On the other hand, if messages are very short, then it may be more efficient to gather several short messages together, and put them in a single large message. The large message could then be split up into its original small messages at the receiver. This optimization only works if the combined messages had been destined for the same receiver.

When it is inconvenient or expensive to set up a separate connection for each pair of communication processes, the underlying layer may decide to use the same connection for multiple, unrelated conversations. As long as this multiplexing and demultiplexing is done transparently, it can be used by any layer. Multiplexing is needed at the lowest level when all the traffic for all connections has to be sent over only one, or two physical circuits.

2.2.4 Open System Interconnection Model

A model of network architecture which is widely known is the International Standards Organization (ISO) Open Systems Interconnection

(OSI) reference model [Tanen81a, Tanen81b]. This model logically groups the functions and protocols necessary to establish and conduct communications between two or more parties into seven layers (figure 2.6). An IMP contains only the bottom three layers, while a host contains all seven layers (figure 2.5).

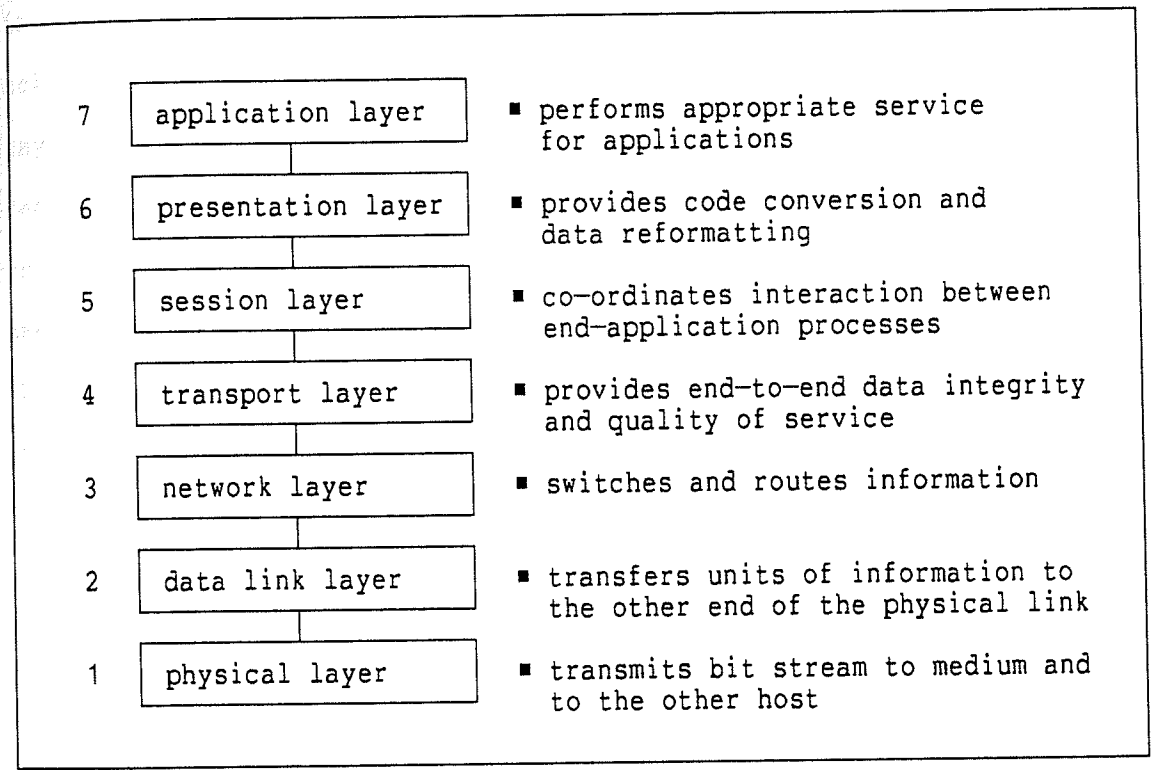

Figure 2.6: ISO/OSI Model of Network Organization

The bottom or lowest level layer is the physical layer. This layer relates to the setting up of a physical circuit so that bits can be moved over it. This layer is concerned with the physical, electrical, functional, and procedural characteristics to establish, maintain, and disconnect the physical link. This layer consists of hardware to attach

to a communications cable. This cable links up the physical layers of various nodes.

The function of the second lowest layer, the data link layer, is to take the raw bit transmission facility offered by the physical layer and use it to create a communications line that appears free of transmission errors to the network layer. To do this, the input data is broken down into data frames, the frames are transmitted sequentially, and acknowledgement frames sent back by the receiver are processed. Since layer one merely accepts and transmits a stream of bits without any regard to meaning or structure, it is up to the data link layer to create and recognize frame boundaries. This can be done by attaching special bit patterns to the beginning and end of each frame. However these bit patterns may also occur in the data, and care must be taken so that the frame boundaries can be found.

A burst of noise on the communications line can destroy a frame completely. When data is lost, the data link layer software on the source machine must retransmit the frame. However, if multiple transmissions of the same frame occur, then it is possible for duplicate frames to arrive at the destination. This happens if the acknowledgement frame from the receiver back to the sender is destroyed. The data link layer must solve the problems caused by damaged, lost, and duplicate frames, so that the network layer can assume it is working with an error-free (virtual) line.

Besides lost and duplicate frames, another problem which arises at layer 2 and higher is how to keep a fast transmitter from saturating a

slow receiver with data. The solution to this problem is usually integrated with the error handling for lost and damaged packets. One solution is for the received packets to be discarded, and eventually the sender will retransmit them because no acknowledgement was received from the receiver. This, however, is very wasteful of the communications bandwidth, and of the network in general.

The network layer is sometimes called the communications subnet layer. The function of this layer is to control the operation of the subnet, and to shape the characteristics of the IMP-host interface. The network layer also determines how packets, the units of information exchange in the network layer, are routed within the subnet. This layer accepts messages from the source host, converts them to packets, and insures that the packets get directed toward the destination.

A key design issue for a network is how the route is determined; it could be based on static tables which are hardcoded into the network and rarely changed, or it could be determined at the start of each conversation, or it could even be highly dynamic and determined anew for each packet according to the current network load.

When too many packets are present in the subnet at the same time, they will get into one anothers' way, causing bottlenecks. Hence, the network layer is responsible for the control of such congestion and is also responsible for generating accounting information to bill the users of the network. This level must at least keep track of how many packets, characters, or bits are sent by each customer in order to produce billing information. This information can also be useful for

tuning. By analyzing which traffic routes are highly used, and for what reason, duplicate copies of data can be installed to reduce the communications load.

The transport layer is known as the host-host layer. This layer is present only in the hosts, and is not present in the IMPs. The function of the transport layer is to accept data from the session layer, and split the data up into smaller units if need be, before passing them to the network layer. The network layer must also ensure that the pieces arrive at the other end. This must be done efficiently, and in such a way that the transport layer is isolated from the session layer.

The transport layer usually creates a distinct network connection for each transport connection required by the session layer. If the transport connection requires a high throughput, the transport layer might create multiple network connections, dividing the data among the network connections to improve throughput. This is only useful if multiple physical paths exist out of the host, or if different routes will be used for each connection. But, if creating or maintaining a network connection is expensive, the transport layer might multiplex several transport connections onto the same network connection to reduce costs. In all of these cases, the transport layer must ensure that its multiplexing or multiple connections are transparent to the session layer.

The transport layer also determines what type of service to provide the session layer. The most popular type of transport connection is an error-free (virtual) point-to-point channel that delivers messages in

the order in which they were sent [Tanen81a]. However, other possible kinds of transport service are the transport of isolated messages with no guarantee about the order of delivery and the broadcasting of messages to multiple destinations. This first type of service is called a virtual circuit, while the second is called a datagram. The type of service is determined when the connection is established and a network will usually support only one of the two types of service.

The session layer is the user's interface to the network. Using this layer, the user must negotiate to establish a connection with a process on another machine. The session layer can manage the conversation between the two hosts in an orderly manner once this connection is established,

A connection between two user processes is usually called a session. A session might allow a user to log into a remote time-sharing system or to transfer a data file between two machines.

The session layer manages the session once it has been set up. If the transport connections are unreliable, then the session layer may be required to attempt to recover (transparently) from broken transport connections. In a database management system, it is crucial that transactions against the database never be aborted halfway, as doing so would leave the database in an inconsistent state. The session layer often provides a facility by which a group of messages can either all be delivered, or none will be delivered if there were problems. This mechanism ensures that a hardware or software failure within the subnet will not cause a transaction to be aborted halfway through. If the

transport layer does not order messages, then the session layer can do this.

The presentation layer's role is to handle the representation of information which applications wish to exchange or manipulate. The presentation layer performs functions that are requested sufficiently often to warrant finding a general solution for them, rather than letting each user solve the problems. A typical example of a transformation service that can be performed is text compression. The presentation layer could be designed to accept character strings as input and produce compressed bit patterns as output. As well, other services are encryption for security, and conversion between data types used on different computers.

The application layer is the layer user applications interface to. A DBMS would communicate to the network using the application layer. This would guarantee the DBMS that all of its messages would be received at the intended host, and the DBMS would be notified of any host that is unable to accept its messages.

2.2.5 Network Types

Networks can be classified into two types based on the area covered by the network. Networks which cover a small area are called local area networks, and are characterized by a relatively high bandwidth for data transmission.

Networks which cover a large area are called long haul networks. Long haul networks are characterized by their ability to span cities,

countries, and even continents. In this section, we will discuss the benefits and drawbacks of each of these two network types.

2.2.5.1 Long Haul Networks

At the present time, many nations of the world have public packet-switching networks which span large geographical distances. These networks are often connected into multi-national networks so that packets can travel around the world. Canada has the Telecom Canada network called Datapac, while the United States has Tymnet and Telenet [Marti81b]. Most network users are linked to the network by common carrier lines going directly to a switching node or else to a concentrator (a concentrator allows several communication lines to share one physical line by multiplexing the data over that one line, then reconstructing the individual lines at the other end). This usually restricts their maximum data rate to that of a telephone line: 9600 bits per second (bps). Some users can have higher-speed digital links into their premises if they are willing to pay the price.

An example of a long-haul network is Datapac. Datapac is Telecom Canada's public, Canada-wide, packet-switched network. The network began in 1977 and has 19 nodes interconnected via 56 Kbps (56,000 bps) digital transmission facilities and provides service to 60 Datapac serving areas across Canada. [Unsoi81].

Long-haul networks permit computers which are separated by long distances to share data and processing capabilities. While throughput rates are not very high, they are sufficient to allow computers on the

network to exchange sizeable amounts of data. However, if care is not taken to minimize the amount of data transfered over the network, then long delays will occur while the data are transferred. Communications bandwidth is a precious resource in a long-haul network, and it must be conserved to achieve performance.

2.2.5.2 Local Area Networks

Networks which span large distances can have poor performance due to low data rates and large amounts of congestion. Performance does not need to be sacrificed if the need arises to connect several computers in the same or adjacent buildings. This network could then be entirely controlled by its users, and could have better performance than a public, long-haul network. This type of private, small network is called a local area network.

A local area network (LAN) is a specialized type of network which has three distinct features. The first of these is that a LAN has a diameter of not more than a few kilometers. Hence, a LAN cannot span the globe or a continent, but at most a few large buildings. Secondly, the data rate of a LAN exceeds 1 million bits per second. Thirdly, a LAN is owned by a single organization, and is thus a private network [Tanen81a].

One reason for using a LAN is to tie together multiple machines in the same or adjacent buildings. This allows all of the machines to communicate with one another, but may also allow each of them to access a remote host or another network via a gateway. A second reason to use

a LAN is to exploit the advantages of functionally distributed computing. This approach involves machines dedicated to certain tasks, such as file storage or database management.

LANs differ from long-haul networks in several ways. The big difference is that long-haul networks are forced to use the public facilities provided by the common carrier network (for economic or legal reasons). In contrast, nothing prevents the designers of a LAN from stringing their own high-bandwidth cables. This means that in a LAN, bandwidth is no longer as precious a resource as it is in the long-haul case.

While a long-haul network uses IMPs as an interface from the host to the network, the host in a LAN has its own interface to tap into the physical medium. Several hosts can use one IMP in a long-haul network, and the IMPs are provided throughout the network to give the network intelligence to perform such duties as routing of messages. In a LAN, each host is connected to all the other hosts via a host interface, and the communications medium is totally passive. Thus, each message in a LAN is broadcast to all other active hosts. The store-and-forward techniques offered by IMPs only occur in point-to-point networks, and not in the broadcast LAN. Instead, one host on a LAN channel grabs control of the channel when no other host is transmitting, and transmits a packet (broadcast subnet). Every other host listening on that channel examines the packet to see if it is the intended recipient; if not, it discards the rest of the packet. If a host has a packet ready to be sent, and there is a packet currently on the channel, then the host holds off until the channel is free, and then transmits it. Collisions

of two or more transmissions are handled in different ways in different types of LANs [Tanen81a].

As the bandwidth of the LAN is high compared to that of a long-haul network, it does not matter that only one host can be active at any instant. As well, the error rates in a LAN are relatively low as compared to a long-haul network. LANs can also be connected together by having each LAN have a gateway to a long-haul network [Schne83].

2.2.6 Network Performance Criteria

Both long-haul networks and local area networks must meet several criteria if they are to perform well. Five major performance criteria for networks are: throughput, delay, availability/accessibility, reliability, and accuracy/error.

In packet switched networks, a packet is the standard unit of information transfer [Marti81b], and the efficiency of the resource utilization is usually expressed in terms of packet throughput. This number is highly variable according to the packet length, amount of piggybacking of acknowledgements, error and retransmission rates, priority control, network overhead, number of logical channels, and various flow control mechanisms. Network overhead encompasses header and trailer bits for packets, called setup packets, as well as network activity and congestion statistics passed around by the nodes in the network.

Network overhead causes delays in the delivery of packets. There are two types of delay in all networks: component delays and network

transmit delays. Component delays are caused by the node hardware. Each node must fully receive a packet before it can be retransmitted. As well, there is some processing time involved before this turnaround can occur. Then, outgoing packets are queued for transmission and may be held up due to higher priority packets, scheduling policies, errors or trunk protocol. Network transit delay is defined as the elapsed time between the correct receipt of the last bit of a packet at the source line processor and the correct delivery of the last bit of the same packet to a transmit queue of a local link at the destination line processor. Datapac attempts to keep this delay below one second for 90% of the packets. Overall, Datapac attempts to keep the node-to-node communication path operational 99.85% of the time [Unsoi81].

Due to errors and lost packets, there is a certain amount of packet retransmission. Thus, the network attempts to increase the accuracy of packet communication by detecting transmission errors, and retransmitting the packet. However, no error detection scheme is perfect, and some errors are passed on. Datapac's objective is an undetected packet error rate of less than one error in 10^{10} packets [Unsoi81].

Errors are usually caused by physical processes such as lightning, surges on power lines, and electromechanical devices at older switching offices. Whenever two cables carrying signals are in close proximity, they can cause interference with one another. Errors tend to come in bursts rather than singly. Error detection is accomplished by adding some number of check bits to the end of each packet. If the packet is received and the check bits are correct for the received data, then it is assumed that the packet was transmitted without error.

A method in widespread use for error detection is a polynomial code (also known as a cyclic redundancy code or CRC code) [Tanen81a]. The basic idea is to append a checksum to the end of the message in such a way that the polynomial represented by the checksummed message is divisible by some generator polynomial **G**(x). When the receiver gets the checksummed message, it tries dividing the message by **G**(x). If there is a remainder, there has been a transmission error.

The transformation performed on the input data is parameterized by a polynomial of degree **k**. The exact transformation process is described by Tanenbaum in [Tanen81a]. The cost of performing this transformation is proportional to **k**. A large **k** will detect a large percentage of errors, but at the same time increase the size of every packet by **k** bits. The amount of computation required also increases with **k**. These two arguments support using a polynomial with a small degree. There is a trade off between undetected errors and the costs of computation. A 16-bit checksum in common use is CRC-CCITT:

$$\text{CRC-CCITT} = x^{16}+x^{12}+x^{5}+1$$

This generator polynomial creates a 16-bit checksum, catching all single and double bit errors, all errors involving an odd number of bits, all burst errors of length 16 or less, 99.997% of 17-bit error bursts, and 99.998% of 18-bit and longer bursts [Tanen81a]. There is thus a slight possibility for a packet with transmission errors in it to arrive with the errors undetected. Because of the millions of packets that travel through a network daily, it is possible for undetected errors to occur once every day or longer, depending on the polynomial used. It is left to the higher levels of software to detect when these

errors occur. This could be done by applying another different checksum in the user's software, or assuming that the packet will be so badly damaged, that the software will reject the packet.

The performance evaluation of an operating network can be accomplished by the use of several techniques, two of these being analytical and simulation models. Most of the techniques require performance measurement data collected from the network. Datapac's nodes continuously generate performance statistics about their components' behaviour and these statistics are reported to the Network Control Centre every 15 minutes. Also, network component failures and recoveries are signalled by transmission of alarm records to the network control centre. Thus the network generates feedback about changing network conditions. This allows the network to change routing parameters dynamically to avoid congested and failed areas. Note, however, that all the performance information is sent over the network via packets, thus adding to congestion, and slowing response.

2.2.7 Enhancements to Basic Network Services

When several computers are connected to a network, they have the facilities for potentially robust, reliable communications between each other. An application such as a DBMS transaction may make substantial use of the network to transfer data between the various computers. The network may even be the bottleneck which slows down the response time for transactions.

While the network guarantees to deliver your data; it does not care how the data are represented. The DBMS (or any other application) may choose to compress its data before transmitting, and expand its data upon receiving. Doing so means fewer bits are transmitted through the network. This has two benefits. First, less data means fewer and shorter packets, and this means the network can deliver the packets in less time with less chance of congestion occuring. Secondly, since users are typically charged for the data volume transmitted, fewer and shorter packets mean lower cost. The major advantage is that of speed. With the delivery time of data being the slowest part in transaction processing, lessening the amount of data traffic directly decreases the real time needed for processing.

A side benefit of data compression is that the data are now unintelligible to all that do not know what methods were used for compression. This is not a very secure method for keeping data secret, but it means that the intruder must take some effort to expand the data.

A more secure method is needed for data security; a way to encrypt data so that an unauthorized individual will never be able to decode the original information. In this section, we will examine some methods used for data compression, to reduce data traffic, and data encryption, to keep data secure from unauthorized users.

2.2.7.1 Text Compression

In many applications, messages sent through a network consist of just numbers, or just alphabetic characters. If a data item can hold a

number with twelve digits, and a two digit number is stored in that field, it is possible to just transmit the two digits, ignoring the leading zeroes. As most fields are defined to hold worst case values, a great data transmission savings can be had by not transmitting redundant information.

If the field were alphabetic, and the field size were 100 characters, a savings could be had by not transmitting leading and/or trailing blanks. A count of the number of blanks could be sent instead. In many applications, a good part of the data consists of blanks or zeroes. As long as some indication of what was compressed out is implied or sent with the data, then the data can be faithfully reconstructed at the receiving end.

The data sent over a transport connection can be viewed as a sequence of symbols. These symbols are drawn from some (possibly infinite) set of symbols. Text compression may be done in three general ways based on [Tanen81a]:

1. the finiteness of the set of symbols
2. the relative frequencies with which the symbols are used
3. the context in which a symbol appears

If all alphabetic data stored are in upper case, then it is possible to achieve a savings by not transmitting all the bits used to represent every character. Usually alphabetic data are transmitted in 8 bits (1 byte ASCII). However, it is possible to express all upper case letters, numbers, and some special symbols with just 6 bits (64 different characters). If only upper case letters are needed, then 5 bits (32 different characters) can be used.

While each field of data can be compressed via individual algorithms, the resulting data usually still have some global pattern in it. For example, in English text, certain letters occur more frequently than others. For example, each of the vowels occur more often in this thesis than the letter 'z'. It should then be possible to achieve a savings in space by representing frequently used characters with a small number of bits, and infrequently used characters with a larger number of bits.

A code compression technique which does this type of compression is called Huffman coding [Tanen81a]. Huffman codes express the most frequent symbol by using only one bit. The second most frequent symbol would be expressed in two bits. The less frequent the symbol, the longer the bit string needed to transmit the symbol. Huffman codes can usually result in a saving of over thirty percent of the total number of bits required to transmit a message.

Larger saving can be made by first removing leading and trailing blanks and zeroes, and then huffman encoding the result. While the data transmission savings are large, a price is paid in the expense of doing the compression and expansion. With the decreasing costs of VLSI, it should be possible to do compression and expansion in external hardware, rather than using the computing power of the host.

2.2.7.2 Data Security and Privacy

Data compression can provide some security for the transmitted data as long as the compression algorithm is kept secret. This is not really feasible, as there are only a few good techniques for data compression,

and it would not take long for someone to try all possible expansion algorithms until the data became intelligible.

In the days before distributed processing, achieving security was quite easy. As long as only authorized personnel were allowed access to the computer, no breach of security could occur without the full knowledge of someone. However, when remote terminals were added to computer systems, security became more difficult. No longer was a guard sufficient to enforce who was allowed access to the computer. Anyone who could gain access to a remote terminal could attempt to wreck havoc with the computer system.

With distributed processing and networking, the situation becomes worse. The millions of bits which flow through a network cannot be policed by anyone. They are vulnerable at many points. No longer must someone break into a computer to steal information. Anyone with a satellite or microwave antenna can pick up these transmission, which are commonly used for long-haul networks [Voydo83]. Telephone company cables can be easily tapped on the user's premises.

With so much data available with such easy access, computer break-ins are probably much more frequent than reported, as most places fear to admit to computer security lapses or the ease with which security can be violated [Parke84, Perry84]. The best and only safe way to protect telecommunications from wiretapping is to use some type of encryption (also called encipherment) to ensure that the data are unintelligible to all but the intended recipient.

The enciphering and deciphering of messages is called cryptography, and the art of breaking ciphers (encrypted text) is called cryptanalysis. The messages to be encrypted are known as plaintext. To encrypt the plaintext, it must be transformed by some function that is parameterized by a key. This key can usually take on a very large number of possibilities, making it almost impossible to guess.

The data to be transmitted through the network are the output of the encryption process. This output is known as cipher text or a cryptogram.

There are only a few known excellent methods of encryption. Because of the small number, one can normally assume that the cryptanalyst knows the general algorithm used to encrypt the text he wishes decrypted. It is usually impractical to keep the algorithm secret because of the vast amounts of effort needed to invent, test, and install a new algorithm.

Instead, it is much easier to keep secret a key. Only the sender and the receiver need know the key. The correct recipient of the transmission can decrypt the data through the use of this correct key.

Without knowing the correct key, an intruder can still intercept and record the encrypted transmission. With some effort, he may eventually be able to guess the proper key, at which point all the messages he recorded can be decrypted with ease.

Some intruders may only wish to listen to the network communications, while others may wish to play back older messages, insert new messages, or modify legitimate message before they arrive at the intended

recipient. The first type of intruder is a passive intruder, and his only goal is to obtain information. The second type of intruder, an active intruder, has many goals. By modifying and inserting his own messages, he is able to obtain data which may not normally be transmitted over the network.

The encryption techniques must prevent all of these security breaches. The general algorithm used for encryption must be stable and secure. This is to prevent someone from discovering an easy way to decrypt the cipher. If this occurs, then important data which were thought to be secure are no longer.

If the algorithm is stable, then it must be parameterized by an easily changed key. This would allow a new key to be chosen any time security was thought to be breached. The key may be changed as often as required, whereas it is not practical to frequently change the general encryption algorithm. Just as there are many different keys to fit a few different designs of locks, there are many possible keys for the few encryption algorithms.

There are two main requirements of encryption algorithms. First, only the intended recipient should be able to decrypt the transmission successfully. Secondly, only the rightful sender should be able to encrypt the transmission so that the receiver can decrypt the data. This is to prevent some other sender, other than the rightful one, from creating or altering messages. This may seem unimportant at first, but it prevents someone from creating or altering encrypted messages.

Several criteria are needed for encryption to work. There needs to be a high degree of secrecy. For example, in a system which employs a key that is used only once, absolute theoretical secrecy is possible. If the key is guessed, it can only be used to decrypt a small portion of the message. In order to do this however, the key size must match the data size. This is impractical for large volumes of data. The main problem is how to generate, store, and transport this large key.

To maintain secrecy, the key needs to be changed fairly often. To do this, both the sender and receiver are required to change their keys simultaneously. This requirement requires that the key be small.

For encryption to be useable, the encryption/decryption process needs to be fast; it should neither delay the transmission nor consume an excessive amount of computing resources. At the same time, the size of the message should not be increased. Some algorithms increase the size of a message in order to invalidate the use of statistical techniques in deciphering. When data volumes are large to begin with, the encryption process should not make them larger.

Unfortunately, these criteria do not work well with each other. A balance must be found between the various criteria. With the decreased computing costs in using VLSI, it is now possible to put an encryption algorithm on a chip. This decreases the load on the main system, but since the algorithm is now in hardware, it is more difficult to change. As well, a chip can easily be purchased or duplicated.

Encryption has been used since the days of Julius Caesar. In those days, the military used encryption for battle plans. One of the

earliest types of ciphers used was one in which every letter is replaced with a different letter. This cipher can easily be broken if one knows the relative frequencies of the letters. For instance, the letter 'e' is the most common letter in the English language, hence the most common letter in the encrypted text probably represents an 'e' (assuming the text is in English). A space is also very common in any kind of text, and spaces often occur in multiples, such as in paragraph indentation, and for separating fields in columns.

While these early simple encryption techniques worked well for centuries, these ciphers were easily broken with the advent of the high-speed computer. What took hours or days to decipher by hand, can now be done in seconds. Modern cryptography uses the same basic ideas as traditional cryptography, except that the emphasis is different. The object now is to make the encryption algorithm so complex and convoluted, that a cryptanalyst will never be able to make sense out of it (never is assumed to be a length of time far greater than the length of the time the encrypted data are useful, which could be two weeks or hundreds of years).

Even with an excellent encryption algorithm, there is a problem in that the keys are chosen by the end users. People tend to choose a key which is small and easy to remember, such as 'SALLY' or 'FIDO'. This makes it much easier for the cryptanalyst to break the cipher since he may try many possible keys until one works. He may try various names, pet names, and street names trying to guess the correct key. Hackers with persistence and ingenuity can usually find a way to break an uncrackable code.

The solution to this problem is to have long keys. With long keys, there can be a selection from many different possible keys. Another solution is to intersperse nonsense data among the real data. This is a waste of bandwidth, but produces a code which is difficult to break because the position of the real characters and messages is a carefully guarded secret and is changed whenever the key is changed.

Encryption may be done at any level in the network hierarchy, using hardware at each end of the communications lines to encrypt/decrypt the data, to encryption done at the highest layer, by software, or special hardware [Kak83]. This second method is not transparent to the software, as the software must then process encrypted data.

These two methods of data encryption differ in several aspects. When encryption is done at the lowest level, the installation management chooses the encryption method. If encryption is done at a higher level, no user is forced to live with an algorithm he feels is too weak. When done at a high level, a user can also change his encryption technique whenever he likes, and can thwart attempts to decipher his data. Another problem with encryption done at the lowest level is that, if a hardware error causes a message to go to a wrong destination, it will arrive decrypted. If someone gains access to one node, he may use that node to decrypt transmissions with very little effort. However, if each user is using a different encryption algorithm, then the misdirected information will be received garbled and no harm is done.

In the ISO-OSI environment, encryption can be done between points in the subnet, at entry/exit to the subnet, or at the presentation level.

Of course, more than one of these can be present at the same time [Davie81].

If carried out in the right way, encipherment at the bit level could conceal whether or not traffic is passing over the line. Encipherment that is part of the OSI link layer does not reveal network sources and destinations, but it reveals the traffic level and the sizes of the frames, which might be useful information to an enemy in exceptional circumstances.

Within the switches, multiplexers, or concentrators, the information is not encrypted, so data security depends on the physical security of these functions, on the trustworthiness of their operators, and on the accuracy and degree of protection of the software. This might be acceptable for a private network in which the interests of the users and the subnet operator coincide, but it would not be acceptable when a public data network is being used.

In a network employing distributed processing, layer one or two encipherment would not give efficient security. It does have a use as a method of concealing traffic flow, which would be a useful service feature provided to all users of a public network.

End-to-end encipherment can be incorporated at any layer above the network layer: transport, session, or presentation. The closer the encipherment is to the end user, the more it is under his control, providing the most security for the data being carried. At the same time, it tends to reveal to a line-tapper how much data is carried and how it is being used. Thus, since traffic information can best be

concealed at the lowest level, but data are more secure being encrypted at a higher level, a combination of encryption at both levels is best.

Node-to-node encipherment employs a different key for each line entering and leaving a node. The packet headers, in which routing decisions are made, are transmitted in unenciphered form. Using this information, the switching node can make its routing decision. The packet data are then deciphered and re-enciphered in a physically protected security module that contains a key for each physical line into a node. Thus, each path a packet takes enciphers the packet differently.

In a private network where complete trust exists between users of a subnet and the operator of the subnet, encryption at any level is possible. When a public data communications network is used, some form of end-to-end encipherment is needed for the user's safety, and at the same time, the network could well provide encipherment at layers one or two, particularly over such vulnerable links as microwave radio or satellite transmission paths. Encipherment at layers one, two, four, and six are being considered for international standardization [Davie81].

One way of encrypting data is to use the Data Encryption Standard (DES) [FIPS77]. This method uses a 56 bit key, but has the problem that it requires the receiver of a message to use the same key for decrypting the message as the sender used for encrypting the message. Many believe this key is too small, and DES in its present form will be obsolete by 1990 [Davie81], as VLSI will allow cracking DES encrypted data in a

short period of time (within a day or less) [Diffi77]. There is also a problem in distributing the key securely. Traditionally, keys were generated at a central source and distributed by courier. For many receivers, this method is cumbersome and unsatisfactory, especially if security requirements dictate changing the keys daily. It would be much more convenient to distribute the keys over the network; but to do so would mean that the keys themselves require encryption to prevent unauthorized access to them.

A more secure method for encryption, though more costly in terms of processing power required to encrypt/decrypt, is public-key cryptography [Diffi76]. Cryptographers generally assumed that both the encryption and decryption keys had to be kept secret. Public-key cryptography uses an encryption algorithm '**E**' and a decryption algorithm '**D**' with '**D**' and '**E**' chosen in such a way that deriving '**D**' given a complete description of '**E**' would be effectively impossible. Therefore, '**E**' can be made public. Any person or organization wanting to receive secret messages first devises two algorithms, '**E**' and '**D**'. The encryption algorithm or key is then made public, and hence the name public-key cryptography. While the key for public-key cryptography is larger than a DES key, the key is still small compared to the amount of data in one paragraph of text.

The only problem with public-key cryptography is the need for algorithms that satisfy all the requirements. One method developed at M.I.T. by Rivest et al. [Rives78a], uses two prime numbers, each greater than 100 digits long. The product of the two numbers is found and it is made public. The security of the method is based on the difficulty of

factoring large numbers and the ease of finding large (100 digit) primes to multiply together. Factoring a 200 digit number requires 4 billion years of computer time.[1] Rivest in [Rives78b] comments on the difficulty of attacking the M.I.T. public-key cryptosystem, and ends up saying that it is almost a waste of time, as the chances of success are extremely small ($1/10^{90}$).

Other methods exist for public-key cryptography. One such method is described by Merkle and Hellmen in [Merkl78], in which trapdoor knapsacks are used. The knapsack problem is believed to be extremely difficult to solve in the general case, belonging to a class of problems that are thought not to be solveable in polynomial time on any deterministic computer. Shamir and Zippel in [Shami80] comment on the resistance to cryptanalytic attack of this method, and they give some enhancements to the method to make it even more secure.

Muller-Schloer gives the details for a hybrid system which uses both the quick DES method with the more secure public-key method in [Mulle83]. To start a transmission, the sender generates a random DES key, and uses the public-key method to securely send it to the recipient. The rest of the message is encrypted with the DES method using this random key. DES is used for the bulk of the work because it is fast, inexpensive, and easy to use. DES chips are available on the marketplace, the only problem in using them being that all receivers need to know the correct key to decrypt incoming messages. To keep DES

[1] Assuming the best known algorithm and a 1 micro-second instruction time. Increasing the speed of computers by several orders of magnitude will not make much difference, for even an increase in speed of a billion times will still require 4 years, at which point larger numbers could be used.

secure, the key must be changed frequently. The only easy way to do this is to place the DES key in the transmission, but to protect it, it must be encrypted; hence the choice for public-key cryptosystems for this job. This hybrid system combines the best of both cryptosystems, while eliminating some of the drawbacks of each.

It should be obvious that even if computers become much faster, data will remain secure for a long period. However, no one has yet proven the absence or presence of a method that would allow the cipher to be broken quickly. No matter how good the encryption, the security of the data are still at risk through human carelessness or desire [Rushb83]. Some people search through garbage cans for computer printouts looking for passwords and encryption keys. Others can find out passwords by looking over someone's shoulder, or by tricking them into revealing the key. Ciphers, of whatever kind, do not produce absolute security. In each case they require a secret key. Therefore, security always depends on the physical integrity of a device as well as on correct system design and software.

2.3 EXISTING COMPUTER NETWORKS

In this section, we examine several of the popular systems for linking distributed machines. These networks are the backbone of the distributed database systems discussed in the following chapters.

One of the earliest networks to be designed was SNA. IBM developed SNA before any standards existed for networks. DECnet was developed later, but was designed for DEC computers.

A network designed especially for distributed database management systems is RelNet. RelNet is not a full network, but instead fits on top of an existing network and provides some ancillary functions vital to the successful operation of a distributed database system.

2.3.1 Systems Network Architecture

A network which has been in use since 1974 is IBM's Systems Network Architecture (SNA). SNA is itself a packet switched network and runs in conjunction with the Terminal Control Access Method (TCAM) or with Virtual Terminal Access Method (VTAM) in the IBM-compatible host and the Network Control Program (NCP) in the 3705 communications controller [Susse78], [Sunds80].

An SNA network is made up of nodes connected by data links. Each node contains a path control element for routing, as many data link control (DLC) elements (to schedule transmission) as there are link connections to adjacent nodes, and a Physical Unit (PU) to activate and control the links. There are a variable number of Logical Units (LU) which act as ports into the network for end users. Communications controller nodes perform useful network routing and control functions without necessarily containing LUs, but most SNA nodes contain one or more LUs.

An SNA network may optionally implement a Systems Services Control Point (SSCP). These control points provide two kinds of services to the network. First, they connect the network operator(s) to the PUs in the network. The connection between a control point and a PU is called a

SSCP-to-PU session. This session allows activation, deactivation, and status monitoring of the resources of the network from network operator sites. Secondly, the control points co-ordinate the creation of sessions between LUs. Two of the services provided are: the resolution of LU names (used in login requests from network users) to LU addresses (so that the network users are not sensitive to changes in network configuration) and allocation of access to LUs that are serially reusable [Schul81]. (Serially reusable refers to a resource that cannot be shared concurrently, but can be reused by another process when the first process has finished with the resource.)

When multiple computer systems are connected into SNA networks, processing can be distributed in several ways. When the system contains two or more Job Entry Subsystem 2 (JES2) components, a network job entry application is available to the remote job entry (RJE) work station. This allows jobs to be submitted at a system with JES2 for execution at any JES2 system.

A transaction from a terminal on one system can be processed on another system, or the data required for that transaction can be passed to the originating system. Thus, the SNA architecture permits transaction routing and access to remote data, but it also allows arbitrarily distributed computations. By dividing an application into two or more pieces that run attached to separate LUs, the ability to run the application on two or more separate processors is created.

SNA supports multiple links operating concurrently between the same two adjacent communications controller nodes. These parallel links

allow increased bandwidth whenever it is required and provide increased availability and reliability through redundancy. SNA allows parallel links to be logically grouped, and provides a transmission group protocol, which automatically distributes traffic across the links of a group. This protocol compensates for degradation resulting from errors on any of the links in a transmission group; sessions using a transmission group are disrupted only if the last remaining link in the group fails. The receiving transmission group component reorders traffic that may have arrived out of order because of messages having different lengths, or as a consequence of retransmissions due to errors.

Another reason for additional links is tariff considerations which make multiple slower-speed links less expensive than a single high-speed link or because the highest available speed of a single link does not provide sufficient bandwidth. With multiple links, a single link failure is not disruptive to sessions using the transmission group. Session traffic is automatically routed over the remaining links in the transmission group.

Network availability can also be increased by providing multiple routes between the same two points, so that disrupted sessions can be reconnected and traffic rerouted to avoid failing intermediate nodes as well as failing links. Multiple routes can also be useful for traffic load-leveling. SNA uses a form of routing called explicit routing, which when activated in a session, is assigned to a particular route. Explicit routing allows the user to have control over the selection of the physical path used for session traffic between nodes. For example, low-delay routes can be chosen for better response and secure routes for

sensitive data (because of tariffs, the shortest route is not necessarily the cheapest). Vancouver to Toronto may be cheaper by using an American network with connections to a Canadian network in Vancouver and Toronto (this may not be legal). If both networks operate at the same speed, the American route is slower, but in most cases the cost advantage will outweigh the speed degradation. Routing schemes that allow routing decisions on individual messages, based solely on congestion conditions, do not provide the same measure of control.

Transmission groups are used to move data between adjacent nodes; explicit routes provide routing between two, not necessarily adjacent, nodes. In both instances, traffic is handled in first-in, first-out (FIFO) order.

Any network has a maximum throughput limit, which cannot be exceeded even if the network traffic is unbounded. Due to cost considerations, commercial networks are normally designed so that peak network traffic loads occasionally exceed storage, cycle, and bandwidth capabilities of nodes and links within the network. SNA seeks to prevent significant network throughput degradation and to prevent network deadlock conditions as network load increases through the use of flow control mechanisms [Schul81].

To maintain network throughput as network load increases, SNA provides global and local flow control mechanisms. Global flow control is accomplished through the use of virtual-route pacing. This means that independent sessions have traffic within the network contending for the same storage, cycle, and bandwidth resources. When the virtual

route is prevented from sending by its pacing algorithm, it queues session traffic until a virtual-route pacing response arrives indicating that adequate resources are available within the network to transport traffic across the virtual routes. The queuing for entry to the transit network ensures that it will not become overloaded.

Session pacing has characteristics similar to route pacing, but the purpose of session pacing is to prevent a fast sender from swamping the receiving session.

Session traffic flows through the network at one of three transmission priority levels. Traffic at a higher priority is queued ahead of any lower-priority traffic at each transmission-group send-queue. Within each priority level, traffic is queued FIFO for delivery to an adjacent node. The FIFO queues are aged to ensure that lower-priority traffic is not completely stopped. The old, low-priority packets have their priority increased proportionally to the amount of time they sit in the queue. Eventually, a low-priority message has its priority raised to be higher than all the other packets in the queue; the packet is then transmitted through the network. Because virtual-route pacing responses are so critical to network performance, they are transmitted at a fourth priority - ahead of all other virtual-route traffic, thus ensuring that heavy traffic in one direction will not interfere with the flow of virtual-route pacing responses in the other direction; if such interference were allowed, it would decrease network throughput under heavy loads.

Transmission priority is useful for ensuring continued good response time to favoured applications during periods of network overload. And, since the network will displace low-priority traffic with higher-priority traffic, bulk data-transfer applications can be run continuously instead of being scheduled for specific slack periods. These applications utilize spare network capacity when it is not needed.

Distribution of data in a network requires the transportation of data from sources to destinations. In an SNA network, the method of transportation of data between nodes can be tailored to the needs of an application. Link data rates can run from 600 bps to 230.4 Kbps on SDLC lines, or channels can be used as links. These channels operate at peak data rates from 10 kilobytes per second for a small controller up to hundreds of kilobytes per second for the IBM 3705 communications controllers.

A service available to SNA sessions is that of encryption of all user data or of selected packets of user data. Security for the session can also be enhanced by selection of a class of service that causes the session to be assigned to physically secure routes. For instance, the nodes might be in secure locations while the lines along the secure routes might employ transmission level cryptographic techniques. SNA uses an 8-byte key and follows the DES encryption standard.

SNA allows the use of start-stop terminals by running the Network Terminal Option (NTO) along with NCP in the 3705. These terminals appear to the rest of the network as SNA peripheral nodes.

2.3.2 DECnet

Digital Equipment Corporation's (DEC) Digital Network Architecture (DNA) is the standard structure for DECnet network products, which support the flexible interconnection of Digital's families of computers while providing an easy-to-use interface. DNA defines the interfaces, structures, and protocols that comprise the design of the network intercomputer communication mechanism.

DNA was designed to create a communications mechanism supporting a wide range of user applications, host computer systems, and interconnect technologies. DNA architecture supports communication between hosts, independent of the physical structure of the underlying data transport network. The overall operation of the network is not adversely affected by the failure of a topologically noncritical node and/or channel. Critical functions such as message routing, communications establishment, and network maintenance should use distributed algorithms [Green82].

The communication mechanism of DNA creates a sequential, full-duplex, error-free, message-oriented communications path connecting processes in the network. This path is independent of the underlying network topology and characteristics of the individual communication channels.

To initiate communications between two processes, one process requests that a logical link be created between itself and a remote process. The network requests communication with the remote process and, assuming no conflicts and an acceptance by the remote process, the logical link is created. The two processes are then free to send and receive messages sequentially over the link.

Flow control functions allow the data receiver to control the rate of transfer over the link to match buffer availability. Data may be sent in either short segments (part of a message) or longer message blocks. The network divides these longer message blocks into smaller segments for transmission, reassembling them at the destination.

In addition to the normal logical link data path, there is an interrupt data path over which short, high-priority messages may be sent to notify the remote process of special conditions and events occuring within the application. This interrupt data bypasses the normal data flow control mechanism and, in some implementations, actually causes a program interrupt to the receiving process. When communication is complete, either process may disconnect and terminate the logical link.

An important component of logical link operation is the addressing of the communicating processes within the network. Objects are referenced via two-component addresses. The first part is the address of the system within which the object resides, the node address; the second part is the address of the object within that system, the object address. Although DECnet does not provide for global addressing of objects without knowing their node address, this function can be easily added by creating a global network directory and resource manager which would be accessed to map global name references to specific 'node, object' pairs.

Since DNA was designed prior to the ISO OSI standard, DNA is divided up into six functional layers: physical link layer, data link layer, transport layer, network services layer, session control layer and

application layer. However, DNA structure corresponds very closely to the ISO architectural model but differs in the names and functions of some of the layers. The DNA transport layer corresponds most closely to the ISO network layer, and the DNA network services layer corresponds to the ISO transport layer.

The transport layer of DNA creates a network pathway among the nodes of the network via a routing function. By using the data link layer for transmission of message blocks over individual channels, the transport layer routes messages among the network channels, connecting them into a path between a source and destination node. The path is not maintained on a per user pathway basis, as in circuit-switched systems, but on a node-addressed basis by having the transport layer at each intermediate node examine the transport header of the routed message and determine the outgoing channel that forms the best path to the destination node based on its routing table. Each message given to the transport layer is treated individually. The routing algorithm and table determine whether all messages to a given destination follow the same route or whether that route changes based on the occurrence of specified events, such as operator demand, channel failures, or queue delays.

The transport layer makes an effort to deliver all messages presented to it, but it does not guarantee delivery, sequential delivery, or destruction (deletion) of messages in a bounded amount of time. This service requires higher levels of the architecture to use a message numbering, acknowledgement, and retransmission mechanism to recover from lost messages, and to be concerned with old duplicates caused by their own retransmissions.

The distributed routing algorithm used by DNA is based on the premise that the best total path from a source node to a destination node, calculated in a distributed fashion, is the sum or concatenation of the many individual node-to-node best paths. Each node individually maintains a list of its best next hop (outgoing channel) to each destination. Messages to be routed are transmitted via that best next hop. The next node does the same, thus building a total best path from the source node to the destination node. This determination of best path is based on a cost function. Each outgoing channel from a node is assigned a cost to route a message through that node over that outgoing channel. The better the route, the lower the cost. Cost is usually based on line quality characteristics such as delay, throughput, or error rate, but may also include characteristics of the switching node such as buffer resource availability and processing capacity. These cost values are assigned by an offline algorithm and can be changed by an operator or program. If the costs of all channels are set to the same value, the path chosen will be the one with the minimum number of hops.

2.3.3 RelNet

The Reliable Network (RelNet) consists of a set of facilities intended to ensure reliable communication and co-ordination among related processes operating at sites connected by means of a communications network. In a distributed system, a function will in general be realized by means of a number of processes, executing in parallel at distinct sites of a network. As these processes execute, they will find occasion to communicate and synchronize with each other.

Individual sites and processes can fail at any time, and each site must be prepared to recognize and react to the failures of its cohorts; the sites with which it co-operates and interacts. One approach would be to embed this responsibility in the application logic and code of each cohort. A better approach would be to factor out this logic and code and thereby provide the application program with a view of the environment which exhibits a degree of reliability that simplifies the program. This is the approach implemented by RelNet. RelNet provides each process running in the system with a set of facilities for reliable communication and interaction with other processes; these facilities can be utilized by invoking a set of procedure calls. RelNet is used instead of whatever communications facilities are provided by the actual communications network connecting the sites in the distributed system. Thus RelNet functions on top of the real network, and enhances its operation.

The basic function of any network is to allow for inter-site communication. RelNet can be thought of as a virtual network that provides several additional capabilities. The network contains a single global clock that any site can access. This clock's function is to impose a uniform and consistent ordering on events occuring at different sites in a distributed system.

Every network site is at any one time in one of two states, UP or DOWN. The UP state is characterized by correct operation and by timely response to messages sent it by other sites; a site in the DOWN state is not operational. Transitions between these two states occur instantaneously with respect to the global clock. Any process has the

ability to ascertain the correct status of any site in the network, and to request that it be informed when that site changes its state.

The reliable communications service offered by RelNet makes two guarantees [Hamme80]. The first is that messages sent from one site to another are received in the same order as they are sent. The second is that a message can be sent to a site that is DOWN, by requesting guaranteed delivery. RelNet will then guarantee that the message will be received by that site upon its recovery. Receipt takes place even if the sending site is DOWN at the time the destination site recovers.

Consider the situation which occurs when a sender sends a message to a site that is DOWN. If the message is not marked for guaranteed delivery, then it is discarded. If the message is marked for guaranteed delivery, then the message will be delivered some time after the DOWN site comes UP. If a second message is sent to this site from the same originating site after the crashed site recovers, then it is possible in general, for this second message to arrive before the first guaranteed message. In RelNet however, the first guaranteed message will arrive before the second message. However, RelNet cannot guarantee that any message, including one marked for guaranteed delivery, is certain to be received, since the destination may never recover from a failure.

Guaranteed delivery of messages is accomplished with a mechanism called a reliable buffer. There is one such buffer for each destination site in the network. Messages destined for a DOWN site are routed to the site's reliable buffer instead (which is at an UP site). When the site recovers, it requests RelNet to provide it with all the messages in the reliable buffer.

For purposes of robustness, the reliable buffer is replicated at a number of different sites, each replication being called a spooler. Multiple sites are necessary as one site might fail, thus cancelling out the backup mechanism of the spooler. RelNet assumes that when a destination is DOWN, at least one of its spoolers is UP. If not, then a RelNet catastrophe occurs.

If a sender wishes to buffer a message reliably, RelNet will send a copy of that message to all spoolers associated with the destination site. When all of the spoolers have acknowledged receipt, the messages are considered reliably buffered. When the recipient recovers, it issues a request to any one of its spoolers to obtain its buffered messages.

If a spooler should crash while it is being emptied, then the recovering receiver should switch to a new spooler. To prevent duplicate messages, an acknowledgement-vector is maintained by the receiver. This array indicates, for every sender site, the timestamp of the last message from that site that the receiver has received and acknowledged.

If a spooler should crash while the receiver is DOWN, and remains DOWN until the receiver has recovered and emptied some other spooler, no problems can arise. However, if a spooler does crash and subsequently recovers while the receiver is still DOWN, that spooler's message queue will reflect a gap during which it received no messages. To signify that messages may be missing, upon recovery the spooler marks the gap in its queue during which it was DOWN, and lets the receiver fill the gap

from messages held in other spoolers. Thus as long as one spooler was UP at any time period, it is possible to recover all lost messages by getting some from each spooler.

RelNet also provides a facility for distributed transaction control. This allows a process running at one site to co-ordinate the activities of a number of distributed processes which are seeking to realize a global activity. The main feature of this facility is a global abort/commit capability, which enables a controlling process to cancel a transaction at any point instantaneously or to signal its successful completion and cause the results to take effect uniformly at all involved sites.

Since a network is constructed out of many discrete components - sites and communications lines - each of which is subject to failure, reliable communications must still occur in the presence of some failures. RelNet is designed to be resilient to the failure of some of its parts, and to function correctly as long as enough of the components behave correctly. If too many failures occur, then a catastrophe results. Under catastrophe situations, RelNet is not guaranteed to work correctly; either some services will not work at all, or they may operate in unanticipated and unpredictable ways [Hamme80]. Some catastrophe situations can be automatically detected by RelNet, but others can only be observed from outside the system. In either case, manual intervention by a system administrator is necessary to rectify the situation.

With the design of RelNet, it is possible to make a catastrophe arbitrarily unlikely by the increased replication of reliability mechanisms. The price to be paid for increased reliability is increased overhead.

RelNet assumes that the basic communications network will look after the communications link between two sites and will employ others to send messages between them if one fails. RelNet also assumes that the network remains connected at all times, and no part of the network disconnects itself from the rest of the network, thus leaving two functioning entities. Should this assumption be violated, the result is a catastrophe.

RelNet thus provides a base for a DDBMS in which some of the DDBMS problems are reduced or solved by the network. RelNet fits between the real network and the network user. Thus there is no need for the DDBMS to ensure that the communications are reliable.

2.4 DISTRIBUTED DATABASES

Given a network, it is possible to install a distributed database over the network. According to Date [Date83], a distributed database is "a database that is not stored in its entirety at a single location, but rather is spread across a network of locations that are geographically dispersed and connected via communications links". A distributed database management system (DDBMS) consists of a collection of sites or nodes, connected together into a network. Each site has its own database management system, which may or may not be the same as the

database management systems in the other nodes. Distribution does not affect the user's view of the database, known as the logical view; but it does however affect concurrency control, recovery, and physical database design. The major advantages of distributing rather than centralizing a database are many:

- potential for improved performance and economics
- increased reliability
- easier incremental growth

Performance may improve because of reduced communications volumes, smaller transmission delays and less congestion. By having the database spread over several computers, reliability may be increased because the entire database is not rendered inaccessible when one of the networked computers fails. Incremental growth may be easier because a new node can be added without incurring excessive down times. In a centralized system, it is often difficult to upgrade without major service disruption and conversion costs. A distributed database is particularly useful to applications that involve extensive processing in different locations.

In this section, we will discuss some of the capabilities of a distributed database management system. We will also look at some of the techniques used to provide these capabilities and some of the problems and benefits which these techniques produce.

There are five basic capabilities which a distributed database system must provide [Allen82]:

1. co-ordination of the DDBMS with the data transmission network such that reliable delivery of messages can be ensured;
2. decomposition of transactions into atomic parts, selection of nodes to execute those parts, and control of any movement of data between sites necessary to process transactions;
3. synchronization of logically related updates and retrievals that are processed at different nodes;
4. detection and resolution of conditions where a part of the database becomes inaccessible due to node or line failure;
5. management of metadata describing the distributed database and environment.

One of the capabilities of a DDBMS is that the data can be distributed redundantly among the nodes of the network. Freely distributed data are needed in many circumstances: when a highly distributed user community exists; when there is a need for high availability - the data must remain available when one or more copies are inaccessable; when there is a need for survivability - the data must remain available after destruction of multiple system nodes; when there is a need for fast response - access to local data are faster than access to distant, highly shared data. At times, there is a need for data to be moved to different nodes as usage patterns change; data heavily used in one geographic region can be stored in that region. When traffic volumes are too high for a single storage system, the only way to increase performance is to have multiple machines each with its own copy of the data (especially with very large storage systems).

Another reason for redundant data is that it permits more flexibility in increasing database capacity to support very large databases. Portions of the database which are frequently accessed can be stored at many small sites using relatively fast secondary storage. Other portions of the database that are needed only occasionally could be stored at an archival site on tertiary storage which is slow but inexpensive. Moreover the redundant approach allows additional database sites to be added to accommodate increases in database activity, whereas in a non-redundant system, increases in activity against a selected subset of a large database could require an upgrade of the site at which that subset was stored.

Without redundant data, the reliability goals of distributed data management can only be partially met. Without redundant data, the failure of a particular database site must cause the failure of all applications that require data stored there. Even though many other nodes are still working, these nodes are of no value to applications requiring data from the failed component. Thus, from the point of view of these applications, the DDBMS will have suffered a total failure from the failure of a single component.

Another capability of a DDBMS is the ability to exploit the parallel processing capabilities of the network. This means that instead of a transaction being executed on only one computer, parts of that transaction can be executing concurrently on several different computers in the network. Theoretically, if three computers are used to process one transaction instead of just one computer, then the transaction should complete in one-third of the time. This usually cannot be

achieved, as the computers must gather data from the other hosts in the network, and then transmit their results back to where the result is needed. This extra data transmission can increase the time needed for the transaction to complete. If there is only one copy of the data that all three computers need, then they may cause some network congestion as the network attempts to deliver that data.

To increase parallelism in a DDBMS, multiple copies of data can be used. There are two methods for storing data in a database [Marti77, Carde79]. The first method is to store the data in a hierarchy or tree structure. This was the first method to be developed, and is used in systems such as IMS. The second method of storing data is to use a two-dimensional table. The table, as in figure 2.7, is referred to as a relation. A database constructed using relations is called a relational database. All of the databases developed recently are relational (see INGRES and SDD-1 which are discussed later in this thesis). Relational databases are prefered over hierarchical as in a relational database, all operations may be described through mathematical rules. Queries may be expressed in terms of either a relational algebra or calculus, whereas in the case of a hierarchical database, queries may only be expressed in terms of a tree search.

A distributed database can be allocated among the nodes of a network according to various criteria. There are two methods for distributing a database: a replicated database consists of overlapping subsets (replication); and a partitioned database consists of nonoverlapping subsets.

prov	name	age	salary	sin
Ont	Jones	27	12324	375
Ont	Smith	54	34953	057
Ont	Kane	35	27548	245
Man	Bones	22	9000	893
Man	Clark	59	234567	002

Figure 2.7: Data Stored In a Table

Replication may enhance the availability and locality of the database, but it requires the DDBMS to provide more sophisticated concurrency and recovery procedures. Each machine may have fragments from different data. For example, sites 1, 3, and 5 could have fragments for employee data, while sites 1, 2, and 4 could have fragments for inventory data.

prov name age salary sin

Ont	Jones	27	12324	375
Ont	Smith	54	34953	057
Ont	Kane	35	27548	245

Site 1

prov name age salary sin

Man	Bones	22	9000	893
Man	Clark	59	234567	002

Site 2

Figure 2.8: Horizontal Partitioning

sin salary prov

375	12324	Ont
057	34953	Ont
245	27548	Ont
893	9000	Man
002	234567	Man

Site 1

sin age name

375	27	Jones
057	54	Smith
245	35	Kane
893	22	Bones
002	59	Clark

Site 2

Figure 2.9: Vertical Partitioning

Distributed data may be partitioned in several ways: horizontal and vertical. With horizontal partitioning, each site has only a subset of all the records; but all the information in each record is at the one site. For example, if data were stored about people working in different provinces, the site in each province would contain the records pertaining to the employees in that province (figure 2.8).

With vertical partitioning, only parts of each record are stored at any one site; but each site contains all the records. Figure 2.9 shows data which has been split into two vertical partitions. Each site has records for all employees, but not all information about each employee. Vertical partitioning occurs when only certain fields are useful at one site, while other information about the same items is useful at another site.

While data could be distributed horizontally or vertically, it is more common to see a mix of both, perhaps with some overlap of the partitions. This distribution of fragments of a database may lead to faster processing, but it can also decrease processing speed depending on how the data are organized, and what type of retrieval is needed. In order to find the data for a retrieval or update, a sophisticated data dictionary or directory is needed in order for the DBMS to find the data. The requirements of the distributed data dictionary are examined later in this chapter.

It is much simpler to retrieve and search a fragmented table than a fragmented tree. The only way to find a particular leaf in a tree is to traverse the entire tree. If the tree is arbitrarily fragmented, then

each path through the tree may involve an arbitrary number of nodes to search. Each switch to a different node can only be done by communication through the network. This may involve large amounts of communication which will drastically increase the retrieval speed. Hierarchical database systems usually do not permit fragmentation because of the difficultly of retrieval.

Relational tables pose no such problems, and may be arbitrarily partitioned and replicated. Several commercial systems will be discussed in chapter 4. One of these systems is hierarchical and allows no fragmentation, and two of these systems are relational and they allow varying degrees of fragmentation.

The major problems in the development of good techniques for managing a distributed database are due to communication volumes and delays. While a distributed system is designed to exploit parallel processing, in reality it may severely hamper it instead. Communication requirements may cause some centralized database managers to be inefficient in a distributed environment. For example, the traditional use of locks (discussed later in this section) by a centralized DBMS may cause excessive delays in the distributed environment due to the transmission of lock messages throughout the network. Parallel processing has the potential to increase throughput, but requires complex controls to synchronize concurrent reading and writing activities at dispersed sites. The DDBMS must ensure that operations executed in parallel have the same net effect on the database as an equivalent set of serially executing transactions.

There are two design approaches for multiple copies of data: centralized and decentralized. The normal and most easily-controlled approach is to have a single, secure master copy of the data. The other replicated copies are regarded as secondary to the master copy. The system is designed so that if the master copy is destroyed, it can easily be reconstructed. Different data each have a single master copy, and these could be stored in different locations.

The other approach is for data to be stored in multiple nodes, no one of which is of higher status than the others. All updates go to every copy of the data. Often they cannot be updated simultaneously, so the protocols are designed so that the system converges to a state in which every copy is the same. No one copy is designated as the master copy.

This latter, horizontal approach needs elaborate, carefully thought out protocols to deal with the various failures, update interference, and deadlock conditions that can occur.

When a master copy of data is used, it may or may not be updated in real time. Two approaches are practical:

1. All transactions immediately update the master. The master issues new copies of changed records to the other processors periodically.
2. Transactions update a non-master file. All transactions are saved for periodic updating of the master, and when the master is updated, new copies of changed records are sent to other processors which use them.

However, when a READ is done, depending on whether the master copy or another copy of the data is read, different values can be obtained. Normally, a local copy will be read to reduce transmission cost and improve speed, but there is no guarantee that this local copy is up to date.

After a failure of part of the system, resynchronization is achieved by issuing new copies of any changed records in the master to the processors which keep them. If the master itself fails, then copies of the transactions must be kept until the master is recovered so that the master copy can be updated, and then in turn issue copies of the changed records to other processors which store the redundant data.

There are two operations which apply to the data in the database: READ and WRITE. With replicated data, a READ operation is simpler than a WRITE operation. The READ operation can read any copy of the data, while the WRITE must update all copies of the data it refers to. A trade-off must be made between a large amount of replication to support READ operations, and a small amount of replication for WRITE operations. In practice, it is more reasonable to keep two copies rather than many copies of data which are frequently updated. This approach is commonly used on hierarchical database management systems.

The distributed system must maintain consistency of all copies of the data. It must organize recovery after a copy of the data has been down for a period of time. Furthermore, it must prevent one update operation from interfering with another. To do this with a high update rate and not incur excessive system overhead requires tight and rather complex protocols.

A processor, processing a transaction, can retrieve the data it requires from more than one place. The different copies of data should therefore all be up to date; when an update is made, it should be applied immediately to all copies of that data. It is not practical to update all copies exactly simultaneously. The system should be designed so that all copies converge quickly to a consistent state. If one copy is inaccessible for a period due to a failure, then a recovery action should follow, bringing it up to date as quickly as possible. As previously discussed, two transactions attempting to update the same data at the same time can interfere with one another and write invalid data. Similarly, a READ could occur while data are in the process of being updated and could give invalid results. One method commonly used in centralized database systems, and in distributed systems, to ensure database consistency, is to require that all transactions observe a locking protocol: a set of rules that require transactions to lock data they access or modify. This ensures that another transaction cannot see or modify data that the lock-holding transaction is manipulating.

Locking protocols involve some system overhead. It is necessary for the lock manager of the database system to see if the lock can be granted and then either grant the lock request or put the requesting transaction in a queue. To minimize this overhead, it is possible to associate a relatively large subset of the data with each lock. But, if transactions lock data they do not use, then concurrency may be reduced. This means it is better to associate a lock with a small subset of the data, as the finer the locking granularity, the greater the concurrency; the coarser, the fewer locks to be set and tested and the lower the overhead [Gray75].

In a distributed database system with a high level of updates, locking the data can cause substantial performance degradation. The node processing a transaction is often distant from the node containing the data. The data have to be locked for the time taken to transmit the data to the processing node, prepare the update, transmit the data back to the data node, and write the new data. Because transmission times are lengthy, data are locked for a much longer time than with a centralized system at one location. To compound the problem, the distributed system may have several (perhaps many) copies of the data in different nodes. Each has to be locked during the updating process.

A variety of locking schemes have been devised for distributed database systems [Berns81b]. Some of these schemes are discussed in the next chapter. Some require a primary site for updating while others avoid requiring that one site have primary authority. All involve substantial transmission overhead if all data being updated have to be locked and there is a high update rate. The best way to reduce the overhead is to structure the updating protocols to avoid time-consuming locks whenever possible. A technique which does this is used in SDD-1 (System for Distributed Databases), which will be discussed later.

While redundant data has many advantages, there are some problems to be overcome in order to realize the advantages. The multiple computers in the DBMS are attached to a network. Certain data items may be contained in each computer, and each computer is required to update its local data. This is a complex situation which can be handled with complex distributed storage protocols. Most commercial software is not designed to operate in a free-for-all distributed data environment.

Instead, one machine must tightly control each type of data. To achieve data integrity, one machine must control the updating of the others, even though the data reside, and the requests to change these data originate, in multiple machines.

When there is more than one copy of data, and sometimes with only one copy of distributed data, inconsistent information can be obtained when reading the data due to time delays introduced through the network. This problem can be overcome with appropriate locks or protocols. These locking mechanisms will be discussed in chapter 3.

When locks are used in a distributed environment to prevent update interference, it is possible for deadlock to occur. Deadlock is a state where two or more transactions are in a wait state, each transaction waiting for the other to complete. Deadlock is discussed in more detail in chapter 3. To circumvent deadlock, fairly complex protocols must be used. Unfortunately, unless the protocols are carefully thought out, the protocols to prevent invalid updates, inconsistent reads, and deadlock can cause excessive overhead, especially when there are multiply-replicated copies of the data in the network. The protocols to prevent invalid updates and inconsistent reads as well as the problems caused by these protocols are discussed in detail in chapter 3.

Another problem which arises with distributed data is interference between updating transactions. Two transactions may be updating the same data item on a remote storage unit and they can interfere with one another causing the stored data to be incorrect (concurrent update problem). This problem can also be prevented by appropriate locks or protocols.

Difficulties in supporting parallel processing are complicated by, but not solely due to, the existance of multiple copies in the distributed system. A similar problem can occur in a single computer when two executing programs both attempt to update the same area in memory. On large machines, a small cache or high-speed buffer is inserted between the fast central processor and the relatively slower large main memory. The hardware attempts to keep the appropriate memory locations in the cache to increase the speed of memory accesses. If a value is read and then changed, the change will be reflected in the cache, but will not necessarily be immediately reflected in main-memory. This is fine for processor-only accesses to memory, since the processor 'sees' the cache contents thinking it is main-memory contents. However, if an additional processor, such as an I/O processor, reads the same memory locations, then the unchanged value will be given to it. Thus, there may be problems with multiple copies of data even when controlled by hardware in very close proximity.

There are systems in which the same data are stored in many locations and the updates originate in many locations. It is difficult and complex to control the correctness of the updating unless the updates are applied to a single copy of the data. However, a single copy implies lack of redundancy, which is one of the reasons for using a DDBMS.

2.4.1 Data Dictionary

A system whose data are scattered geographically must have some means of determining where any piece of data is stored. As with other aspects of distributed databases, location of data can vary from very simple to extremely complex.

A simple method is for the user to specify the location of data when he makes a request to use them. His transaction may then be transmitted to a computer at the location of the data, or alternatively the data may be transmitted to a location where they can be processed, possibly the location where the transaction originated. Note that this is not very practical, for the user must know where all the data he wishes to access are, and he must be notified whenever his data moves.

A slightly more complicated approach is for the user to specify information about the data from which their location can be simply determined; e.g. A bank account number, which has the branch specified by the first digits.

Locating the data becomes more complex when the user does not know where the data are located in a distributed database with horizontal distribution. The system must contain some form of catalog or directory, which permits the data to be found. The directory may exist in one particular computer in the network, the request for data being passed to this computer and the location of the data established. Alternatively, every computer in the network may have a complete directory, listing each data field and indicating its physical location.

An important characteristic of modern database management systems is that they employ a directory or data dictionary to define the database being managed. This frees the user or application program from the need to supply this information and it simplifies many types of database structural changes. A directory typically contains 4 types of information [Allen82, Marti79a]:

1. logical structure definition
2. physical structure definition
3. file statistics
4. accounting data

For a distributed database, there must be an additional category of information: the location of each piece of the database in the network.

The DDBMS must have access to this information in order to parse user requests, choose and execute an accessing strategy, and account for the resources used. The problem then is where to store the directory.

There are several solutions to this problem, varying with the database and the particular accessing pattern. Not all of the data in the directory are accessed as frequently as others. In practice, there are two basic categories of directory management schemes. First, each directory entry is stored in only one location (non-redundant schemes). Secondly, each directory entry may appear in several places (redundant schemes). Thus, for non-redundant schemes, the following alternatives exist [Rothn77]:

- centralization - The complete directory is stored at one specific site. This requires access to the directory site for every retrieval or update to the directory.
- distribution - Each site has the directory entries for the data stored at that site. Completely local access can proceed using the local directory. Any request requiring access to remote data must be broadcast so that other sites can determine whether or not they have relevant data.
- combinations - An intermediate approach is to partition the network, and employ a centralized directory in each part.

The redundant approaches have various alternatives:

- centralization - A redundant but centralized approach with a combination of the centralized and distributed non-redundant alternatives. Each site has the entries for its local data but a central site has the complete directory. In this way, non-local references can query a single site to determine the location of the remote data.
- distribution - Each site has the complete directory stored locally. All user transactions can be executed without a remote directory reference, but all directory updates must be posted to every site.
- combinations - Each site is allowed to have an arbitrary subset of the directory. This permits a great deal of flexibility but it requires a more complicated 'directory directory' to tell the system where the various pieces of the directory are stored.

Some disadvantages to having a directory in each computer are the storage space required for such directories and the work of keeping them up-to-date. In addition, small local computers may not be well equipped for searching the directories as rapidly as a specialized or larger computer.

In many systems, most of the transactions received by a local computer relate to the data kept at that location, whereas a few transactions are for data in other locations. In this situation, each computer could have a directory of its own data only, and pass other requests to another computer if it relates to data it does not have. The directory problem is a distributed data problem scaled down. For efficiency the data in the directories should be replicated, but to what extent? The problem is then to update the replicated copies and maintain them in a consistent state.

The directory can be treated just like any other data by the system or it may be specially treated for improved efficiency and safety. If the directory is treated like any other data, then there is no special problem with fragmentation and replication.

However, the system must always know where the directory itself is stored, otherwise it will not be able to find any data, as the system cannot look up the location of the directory in the directory, unless it knows where the directory, or fragments of it are stored. The information about the directory must be at every node.

Whenever a request is processed by a computer which does not have the data it requires, there are two possibilities: to move the data to the

processing computer; or move the transaction and process it where the data are.

Which method is used is affected by many parameters: by the machines in use and their software, and by what would have to be transmitted and how frequently. It is expensive to move large amounts of data too often, and it is slow. It is usually cheaper and faster to move the transaction to the site where the data is, than to move the data to the site where the transaction is.

As with all distributed data, a high frequency of read access encourages local, redundant directories. Since every user transaction requires a directory retrieval, directory redundancy is virtually essential. If there is a high update rate to the directory, this discourages redundancy as each copy must then be updated. However, high reliability requires redundancy and distribution.

2.4.2 Recovery

One of the key motivations for distributed database systems is a requirement for high database availability; that is, a need to ensure that a database is nearly always accessible. Distributed database systems seem to offer this characteristic since availability is not limited by the reliability of any single component but rather by the reliability of combinations of components (processing nodes and communication links in the network). These combinations can be configured to achieve arbitrarily high availability. In order to achieve this availability potential, it is necessary that the

distributed system be able to cope with the failures of individual components and continue operation.

The central problem in reliable operation of a distributed database system is maintaining database consistency in the presence of failures during update transactions. Such failures threaten to destroy the 'atomicness' (indivisibility) of these transactions by causing the update to be only partially accomplished in the database. Consider an example in which transaction T_1 updates portions of a database stored at nodes N_1 and N_2. If a failure occurs during the execution of T_1 which prevents the update from being recorded at N_2 then the database has been made inconsistent. A later transaction which reads the results of T_1 will see an anomalous database with unpredictable results. The system updating algorithms must therefore be aware of the possibility of component failure and avoid these partial results.

Recovery after a failure needs to be controlled so that updates are not accidently lost or double-processed. Suppose we consider a transaction which withdraws money from one bank account and transfers it to another account. If a failure occurs before the transaction completes, and the transaction is redone at a later time, there is a possibility for twice as much money to be withdrawn as was deposited, or for twice as much to be transfered.

When multiple copies of data exist, they may be in different states of update after a period of failure. They must all be brought back to the same state, but problems may arise as real-time transactions are being processed during the recovery period. For example, consider a

situation where one copy of a data item has not yet been updated because its host was down. If a user attempts to update that same field again, the update must not be applied until after the data item is updated from the previous updates which are still pending.

The avoidance of database inconsistencies of this type is the major objective of failure-handling mechanisms but it is clear that there are other considerations which are important to the design of these algorithms.

The most important consideration is that of efficiency. People desire good responsiveness and throughput of the system during normal operation and in the presence of failure. This is important because it precludes using a very simple algorithm which preserves consistency but introduces intolerable delay when faced with a failure. The simple algorithm just waits indefinitely until the failed node recovers. While this does indeed guarantee consistency, it can lead to indefinite delays, which is unacceptable. An effective DDBMS must employ a different type of mechanism, one which attempts to execute to completion any transaction which can proceed without a resource which is exclusively available at the failed node.

This problem of failed sites can be partitioned into several subproblems: reliable broadcast, operation with missing or inoperable nodes, restarting and re-integrating a node, failure detection, and partitioning.

Reliable broadcast refers to the fact that if a transaction **T** is updating fragments of a database stored on nodes $\mathbf{N}_2$, $\mathbf{N}_3$, and $\mathbf{N}_4$, then it

is possible for a message to be sent to N_2 to complete the update, and then for a failure to occur before a message has been sent to N_4. If this occurs, then reliable broadcast has not been achieved. This is handled in various ways in different systems. Some systems attempt to send out all the messages at once, thus lessening the possibility a failure will occur between the messages. SDD-1 tries to ensure that reliable broadcast is achieved by requiring that the first few recipients of the message send it on to the other recipients. This requires that all recipients be capable of rejecting duplicate messages. In addition, each node is warned when a reliable broadcast is about to be sent and any node which does not receive the message in a reasonable period will ask the other addresses if they have received it. Thus in SDD-1, the likelihood of a breakdown in the reliable broadcast is considerably reduced.

A second subproblem in failure handling is continuing operation when one or more nodes in the network are known to be inoperable. The objective is to avoid having the absence of these nodes cause transactions to be delayed until the missing nodes are recovered. All existing system designs try to accomplish this by acting (almost) as if the missing nodes never existed since non-existent nodes of course cannot cause any delay. A serious problem arises if unique data are stored on the failed node(s). All transactions requiring this data must either wait for the node to recover, or abort themselves. If the node is inoperable for a long time, it would be best to abort the transaction to avoid tieing up resources. This problem is more severe if the failed node contains unique data required for the DDBMS to operate, such as a system directory. In this case, the entire DDBMS ceases to operate.

The third sub-problem is re-integrating a node into the system after the cause of a node's failure has been corrected. Since the fragment of the database stored at the node may be out of date, it is necessary for the node to find out what happened during its absence prior to continuing its active participation in the system. The solution to this relies on persistent communication: guaranteeing the delivery of a message to a node even if the addressee is down when the message is sent and even if the sender is down when the addressee recovers. This recovery method is very clean, as the restarting node simply acts on its old messages in almost the same way it does in normal operation.

For a system to function with missing nodes, and to restart nodes, there must be some way of knowing when a node has failed. The most common method of detecting that a node is dead is a time-honoured technique called time-out. Under this scheme, a node which is suspected of having failed is probed with a message to which an active node will respond. If the response does not arrive within some prescribed time period the probed site is assumed to have failed.

The final sub-problem is a serious one for which no adequate solution has yet been devised. This problem arises when communication failures break all connections between two or more active segments of the network partitioning them into separate pieces. When this occurs each piece of the network continues its operation, including processing updates, but there is no way for the separate pieces to co-ordinate their activities. Hence the fragments of the database in the separate pieces will become inconsistent. When reconnected, it may be impossible to carry out all of the updates because some of the transactions may not be correct in the context of the whole network.

2.5 DISTRIBUTED QUERY PROCESSING

Accessing data that are stored at separate computers in a distributed system differs in two important ways from accessing data from a centralized computer. First, necessary movement of data over communications lines introduces substantial time delays. Secondly, the distributed system has the ability to process and move data in parallel at separate points in the network. The DDBMS must determine a strategy for processing queries that takes into account these two facts. Rothnie and Goodman [Rothn77] have shown great variations in communications costs among feasible distributed query processing strategies.

In a relational database, the data are stored in a set of tables (called relations) and are manipulated by means of high-level operations. There has been much work in optimizing queries in a non-distributed system, and Sagiv [Sagiv81] discusses many approaches outlining their benefits and problems.

Single relation queries can be processed very easily on a DDBMS. When the relation is geographically distributed, the query can be sent to a node that has a copy of the relation and be processed there. The results after processing are sent back to the originating node. Communications needed to transmit the results is much less than the communications needed to transmit the entire relation [Wah81].

The processing of a multi-relation query is more complicated. When multiple relations are accessed by the same query, the relations usually have to reside at a common location before the query can be processed. Substantial communication overhead may be involved if these relations

are geographically distributed and a copy of each relation has to be transferred to a common location. It is therefore necessary to decompose the query into sub-queries so that each sub-query accesses a single relation. This technique is used in Distributed-INGRES and in SDD-1.

The technique consists of two steps. The first step is to select a site with the minimum amount of data movements before the query can be processed. This is used as a starting point for the second step of the algorithm which determines the sequence of moves that results in the minimum cost.

The distributed query problem is one of transforming a distributed query into a local query by means of a combination of local processing and data movements. Local processing reduces the size of relations, which then decreases the cost of moving the relation. Data movement brings the data together to allow local processing. The final data movement must move the data to the result node for final processing.

In determining the most effective access strategy, it usually can be assumed that local processing costs are insignificant in comparison with data movement costs. Thus, before any data are moved, processing should be done to reduce the total amount of data needed to be moved. Hevner and Yao [Hevne78] show that minimizing the cost of data movement is equivalent to minimizing the amount of data moved. In SDD-1, Bernstein et al. [Berns81b] assume network bandwidth to be the system bottleneck, and SDD-1 seeks to minimize use of this resource.

Using the data dictionary, the query processor locates the data fragments required, and then determines a good set of moves and computations to give the desired result. In many cases, it is possible to overlap data movement and processing, and have concurrent processing at multiple sites. By using the resources of the network, it is possible to obtain better performance from a DDBMS than from a DBMS. However, if the query processor chooses a bad move, it is possible for the DDBMS to be orders of magnitude slower than a centralized DBMS.

Chapter III

CONFLICT ANALYSIS

In this chapter, we will examine some of the conflicts which arise in a DDBMS and discuss some methods of eliminating these conflicts. These conflicts occur during concurrent execution of different transactions.

One major conflict is the synchronization of update transactions. Transactions which read data may conflict with transactions wishing to update the same data. It is necessary to control concurrent transactions in order that database consistency is preserved while excessive overhead in propagating control information among the nodes of the DDBMS is avoided.

Since data items may be stored redundantly at multiple database sites, whatever update synchronization methodology is used by the DDBMS must also ensure that all redundant copies of data are updated correctly and consistently. This methodology is termed concurrency control, and much research has been done in this area.[2]

While there are solutions to the update synchronization problem, some of these solutions themselves cause new problems: that of global deadlock. Two or more hosts in the network are waiting for data the other have locked. Some host or hosts in the network must be able to

[2] [Adiba81, Badal78, Badal79, Badal80, Berns78a, Berns78b, Berns80a, Berns80b, Berns80c, Berns80d, Berns81b, Berns83, Garci79, Garci82, Korth80, Kung81, Lim79, Meras78, Milen78, Minou79, Ries79, Rosen78, Schla81, Stone79].

detect this problem and resolve it in an orderly manner.

When global deadlock occurs, it is not really possible to tell if the deadlock is real, or was a false deadlock caused by the delays in the communications network. In order to check the validity of the deadlock, it is necessary for some host to confirm the messages which were sent out. If they are still valid, then a deadlock occurs, but if the transactions have completed processing, then a false deadlock occured, and there is no need to act upon it. Deadlock is usually solved by aborting or restarting one of the effecting transactions.

3.1 CONCURRENCY CONTROL

Concurrency control is the activity of co-ordinating concurrent accesses to a database in a multiuser DBMS. Concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. The concurrency control problem is greater in a distributed DBMS than in a DBMS because in a DDBMS a transaction can access data stored in many different computers, and a concurrency control mechanism at one computer cannot instantly know about interactions at other computers. The primary purpose of concurrency control is to exploit parallel execution facilities while preserving database integrity. An example of a loss in integrity would be two airline clerks simultaneously discovering that one seat remains free on a flight and then both selling that same seat.

The purpose of the concurrency component of a database system is then to interleave the steps of transactions in such a way that the order of steps within a transaction are preserved and that each transaction sees a consistent database. If a transaction must violate consistency constraints, these violations should be hidden from other transactions.

In centralized systems, locking is the primary mechanism used for concurrency control. In a distributed system, an extension of locking techniques could be extremely costly, especially if there are replicated data, for then all copies would have to be locked whenever one of the copies is updated [Ries79]. Setting remote locks in a network is several orders of magnitude more expensive than setting locks locally because of the communication delays, lost messages, and site failures. In order to permit concurrency control, there is a need for reliable communication in the network.

Given a correct state of the database as input, an individually correct transaction will produce a correct state of the database as output if executed in isolation. Read-only transactions do not change the database, hence the state does not change. With multiple copies of data, a READ can obtain any copy of the data, while a WRITE must update all copies. It is not necessary for all copies to be written at the exact same instant; instead all copies must converge to the same value.

Even if all transactions are individually correct in this sense, it is possible in a multiuser system for transactions that execute concurrently to interfere with one another in such a way as to produce an overall result that is not correct. One problem is the lost-update problem:

```
Transaction A          Transaction B
   read X                    .
     .                       .
     .                    read X
   write X+1                 .
     .                    write X+2
```

When **B** completes, **A**'s update has become 'lost'. If either of these transactions ran individually, the final value would be correct. However, **B** overwrites **A**'s update without even seeing it. In a multiuser environment some sort of concurrency control mechanism is needed in order to avoid this type of problem.

Another problem is one of inconsistent retrievals. Consider a transaction which transfers funds between two accounts, while another transaction is computing the sum of both accounts:

```
Transaction C          Transaction D
   read X                    .
   write X-1000              .
     .                    read Y
     .                    read X
   read Y                    .
   write Y+1000              .
     .                    print X+Y
```

In this case, **C** writes the correct value into the database, however **D** has printed out an inconsistent result which is short $1,000.00. Some sort of concurrency control is also needed to prevent this type of interference.

In the first example, the problem is that **A** and **B** are both updating X on the basis of the initial value of that field - that is, neither one is seeing the output of the other. To prevent this situation, there are three basic things a concurrency control mechanism might do [Date83]. First, it can prevent **B** from accessing X on the grounds that **A** already

has it and may be going to update it. Secondly, it could prevent **A**'s update on the grounds that **B** has already gotten a copy of X before the update. Thirdly, it could prevent **B**'s update on the grounds that **A** has updated X and therefore **B**'s update is based on a now obsolete value.

The first two solutions can be handled by a concurrency control technique known as locking, the last solution by a technique known as timestamping. Of these two techniques, locking is more commonly used in centralized DBMS, while timestamping is more commonly used in DDBMS [Date83].

One must understand how a concurrency control algorithm fits into an overall DDBMS before one can understand how a concurrency control algorithm works. A simplified view, as adopted by [Berns81b], of a DDBMS that permits analysis of concurrency control algorithms will now be presented. A DDBMS is a collection of sites interconnected by a network; each site is a computer running one or both of the following software modules: a transaction manager (TM) or a data manager (DM). TMs supervise interactions between users and the DDBMS while DMs manage the actual database. The network is assumed to be a reliable communications system. If a message is sent from one site to another, the receiving site is guaranteed to receive the message without error.

A database consists of a collection of logical data items, denoted **X**, **Y**, **Z**. A logical data item may be stored at any DM in the system or redundantly at several DMs.

To access the data, users interact with the DDBMS by executing transactions. Transactions are either queries from terminals, or

application programs. The concurrency control algorithms presented in later sections pay no attention to the computations performed by the transactions. Instead, these concurrency control algorithms make all of their decisions on the basis of the data items a transaction reads and writes. A transaction is assumed to be an entity which takes as input a database, and some data, and modifies the database according to the input data the transaction received. When the transaction completes, the database will have changed, but the database should still be consistent. The readset of a transaction is the set of logical data items the transaction reads, while the writeset is the stored data items that a transaction writes.

Whether a concurrency control algorithm works correctly is based on a user's expectations about a transaction. Users expect their transactions to be executed within a reasonable time frame. They also expect their transactions to work consistently, no matter what other transactions are concurrently being processed. Concurrency control tries to meet these two expectations.

A DDBMS contains four components: transactions, TMs, DMs, and data (figure 3.1). Transactions communicate with TMs, TMs communicate with DMs, and DMs manage the data.

TMs supervise transactions, and each transaction executed in a DDBMS is supervised by a single TM. The TM manages any distributed computation that is needed to execute the transaction.

Two operations are possible at the transaction-TM interface. A READ operation returns the value of a logical data item in the current

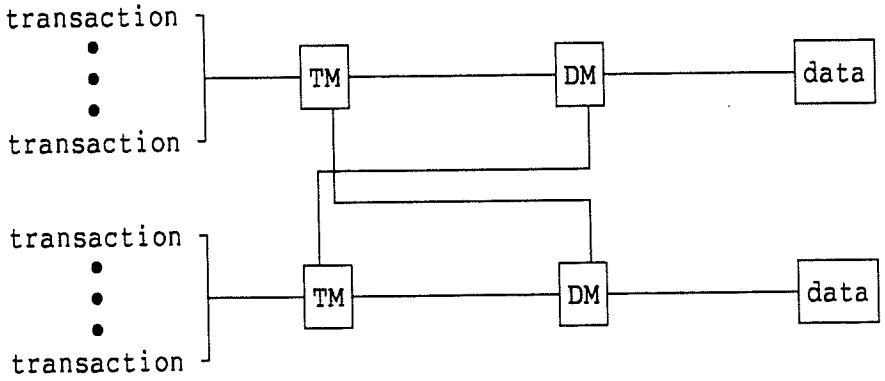

Figure 3.1: DDBMS System Architecture

logical database. A WRITE operation creates a new logical database where some data item has a specified new value.

DMs manage the stored database, functioning as database processors. In response to commands from transactions, TMs issue commands to DMs specifying stored data items to be read or written. The TM communicates this to a DM, and the DM transfers the appropriate data to the TM. In a centralized system, there is only one TM and one DM, and they communicate through memory. In a distributed system, there are many TMs and DMs, and the movement of data between a TM and a DM can be very expensive. To reduce this cost, many DDBMSs employ query optimization which regulates, and if possible reduces, the flow of data between sites.

Two operations conflict if they operate on the same data item and one of the operations is a write. The order in which operations execute is computationally significant if and only if the operations conflict. If one of the operations is a read, and the other is a write, then the conflict situation is known as read-write (rw) conflict. If both

operations are writes, then the conflict situation is known as a write-write (ww) conflict.

When a TM executes a transaction, it never knows whether it will ever complete. At any time the transaction may be backed out due to a conflict or even canceled due to a failure. Furthermore, it may be a very long time before the transaction actually completes because some nodes where the transaction has to perform actions may be inaccessible. It is desirable if at some point in time the transaction could know if it was going to complete. This point, called the commit point, occurs when the transaction can guarantee that it will never be backed out or cancelled. The commit point is sometimes defined as the time when the values produced by a transaction first become available to other transactions [Garci82].

Bernstein and Goodman [Berns81b] state that, after studying a large number of concurrency control algorithms, they find that all the algorithms are compositions of only a few subalgorithms. Bernstein and Goodman go on to state that "the subalgorithms used by all practical DDBMS concurrency control algorithms are variations of just two basic techniques: two-phase locking and timestamp ordering".

These two practical techniques are examined in this section, along with a theoretical technique, optimistic concurrency control. As well, some combination techniques are described; techniques which use both locking and timestamps.

3.1.1 Locking

The simplest method for synchronizing distributed updates is to lock those portions of the database being read or written by an active transaction. Locking is the usual mechanism employed for update synchronization in conventional, non-distributed DBMSs. However, in a distributed database environment, an appreciable and often intolerable delay is introduced as locking information is propagated to the many computers in the database network.

Consider a locking algorithm with the following steps:

1. send out a lock request message
2. a lock grant message is returned
3. the update is transmitted
4. the update acknowledgement is returned
5. a lock release message is sent

If there are **N** copies of data, then 5**N** messages must pass through the network. Given that the network delay could be up to one second per message, the total delay in a distributed DBMS is 2 to 3 orders of magnitude greater than of a delay typically encountered when setting locks in a centralized system.

For this reason, the straightforward locking approach is inadequate for a general purpose distributed DBMS and other methods must be sought. There are several solutions to this problem. Many methods try to reduce the number of messages sent from 5**N** to about 2**N** [Rothn77, Milen81]. One method piggybacks the update message with the lock request and the lock

grant message with the update acknowledgement. This only reduces communications volume if the update transmission messages are short or if most lock requests are granted. While the savings of these methods are substantial, in many instances they still require a large amount of inter-computer communication in order to perform updates. Such methods will not perform satisfactorily in networks containing large numbers of sites and high transaction volumes.

The amount of overhead message traffic does not, by itself, determine the effects of concurrency control or the overall performance. Processing loads, network loads, and the types and sizes of transactions all affect performance as well.

Locking is a technique for regulating concurrent access to shared objects, such as records in a database. A transaction can obtain a lock on a record by issuing a request to a system component called the Lock Manager. If transaction **T** holds a lock for a record **R**, then **T** will be guaranteed that no concurrent transaction will be able to update **R** until **T** releases its lock. The lock is released by means of another request to the Lock Manager. The exact nature of the guarantee depends on the lock type. The most common type of lock is the exclusive lock. If a transaction **T** holds an exclusive lock on some object, then no other transaction can acquire a lock of any type on that object until **T** releases its lock.

If all update requests incorporate an exclusive lock on the initial read, then lost updates can be avoided. If some transaction already has an exclusive lock on the data item, then the current request waits for

the first lock to be released. The transaction will eventually come out of the wait state, unless the transaction caused deadlock, which will be discussed later. A simple technique to ensure that transactions will eventually come out of the wait state is to service all lock requests for a given object in first-come/first-served order. If the system does not provide such a guarantee, then it is possible for a transaction to wait forever for some lock. This condition is called livelock.

A given interleaved execution of some set of transactions is said to be serializable if and only if it produces the same result as a serial execution of those same transactions.

Serialized transactions still execute concurrently, but some concurrency control mechanism is used to synchronize the transactions as necessary. Serializability is widely accepted as a formal criterion for correctness. A given interleaved execution sequence will be considered correct if and only if it is serializable.

While the exclusive locking protocol can be used to solve the lost update problem, it can also create worse problems. One of these problems is deadlock. Deadlock is a situation in which two or more transactions are in simultaneous waitstate, each waiting for one of the others to release a lock before it can proceed.

Transaction **A**	Transaction **B**
lock X_1	
.	
.	
	lock X_2
	.
	.
	.
lock X_2	
wait	
.	
.	
.	
.	
	lock X_1
	wait
	.
	.

The problem of deadlock is not just confined to databases. Various deadlock avoidance protocols have been defined [Carde79]. One scheme is to impose a total ordering on all lockable objects, and then refuse a lock request for an object **Y** if the requesting transaction already holds a lock for any object **Z** that appears later than **Y** in that ordering.

Such deadlock-avoidance protocols are generally inapplicable in a database environment. One reason is the set of lockable objects is very large and highly dynamic. These objects are not addressed by name but by content, so it cannot be determined until execution time whether or not two distinct requests are for the same object. Finally, the precise locking scope for a given transaction is usually determined dynamically; that is, one record's contents may cause another record also to be locked.

A system must be prepared to detect the occurrence of deadlocks and to resolve them when they occur. Detecting deadlock is basically a matter of detecting a cycle in a wait-for graph; that is, a graph of who is waiting for whom. In this graph, the nodes represent executing transactions, and the edges represent waits. An edge is drawn from node **Ti** to node **Tj** when transaction **Ti** requires a lock on an object that is held by **Tj**, and erases that edge when **Tj** releases its lock. Thus, if there are edges from T_1 to T_2, T_2 to T_3, and T_3 to T_1, then the transactions T_1, T_2, and T_3 are deadlocked (figure 3.2).

Checking for deadlock can be done at two times: whenever a lock request causes a wait; or on some periodic time basis. Checking on every wait allows deadlocks to be detected as soon as they occur, but

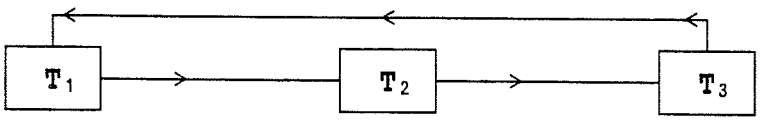

Figure 3.2: A Wait-For Graph Showing a Cycle

may impose too much overhead as most checks will not find deadlock. Checking less often reduces overhead but may mean that some deadlocks are not detected as soon as they occur. The system may detect deadlock by using a timeout mechanism, and simply assume deadlock if no work is done in a given time period.

Breaking a deadlock involves choosing a victim, one of the deadlocked transactions, and rolling it back (undoing it). The victim is not necessarily the transaction that actually caused the deadlock; it may be the one most recently started, or the one holding the fewest locks. The rollback process involves not only terminating the transaction and undoing all its updates, but also releasing all its locks, so that the resources can now be allocated to other transactions.

This backing out of deadlock can be clumsy and complex in a distributed environment. It is better to prevent this occuring by timestamping the transactions and ensuring that the data operations are performed in timestamp sequence. A method that prevents many deadlock situations is to force the transaction to issue all locks at once and not permit the transaction to proceed if all locks cannot be satisfied.

A transaction may need to keep data locked even if it is not updating that data item. For example, if the sum of several records is being

added then that sum will be inconsistent if one of the records is updated midway through. Using an exclusive lock would solve the problem but at the same time reduces concurrency unnecessarily. Thus a need arises for a shared lock, one in which other transactions may also obtain shared status, but not exclusive.

In many transactions, it is not known if an update will occur until the data used by the transaction are read from the database. The need arises for an update lock, a lock which indicates an update may occur. This lock can then be upgraded to an exclusive lock if need be.

Date [Date83] gives the following theorem:

> If all transactions obey the following rules:
>
> a) before operating on any object the transaction first acquires a lock on that object; and
> b) after releasing a lock the transaction never acquires any more locks;
>
> then all interleaved executions of those transactions are serializable.

A transaction that obeys rules (a) and (b) are said to be 'two-phase', or to satisfy the two-phase locking protocol (2PL). The two phases are a growing phase, where locks are acquired, and a shrinking phase, during which they are released. Thus, the theorem may be restated as: if all transactions are two-phase, then all executions are serializable. The condition that all transactions be two-phase is sufficient but not necessary to guarantee serializability. But 2PL provides guidelines for safe transactions. An implementation of 2PL requires building a 2PL scheduler, a software module that receives lock requests and lock releases and processes them according to the 2PL

specification. The basic way to implement 2PL in a distributed database is to distribute the 2PL schedulers along the with database, placing the scheduler for data item **x** at the DM where **x** is stored. If a transaction requests a lock, and the lock cannot be granted, the request is placed on a waiting queue. This can produce deadlock.

A transaction may read any copy of redundant data and need only obtain a readlock on the copy read. However, an update transaction must obtain writelocks on every copy of the data item.

There are several methods used to implement 2PL: primary copy 2PL, voting 2PL, and centralized 2PL [Berns81b]. In primary copy 2PL, one copy of each logical data item is designated the primary copy; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy [Stone79]. This technique is used in Distributed-INGRES, which is discussed in the next chapter. Voting 2PL also exploits data redundancy. The TM asks all the DMs containing **x** if it can lock **x**. If a majority of the DMs respond positively, then the lock is granted. Otherwise, the TM must try again. A detraction from this method is the concomitant high communication cost and delay.

Instead of distributing the 2PL schedulers, it is possible to centralize the scheduler at one site. Before accessing data at any site, appropriate locks must be obtained from the central 2PL scheduler. This method also requires a large amount of intersite communications and is prone to failure of the central site.

Centralized systems tend to allow deadlocks to occur rather than avoiding them as avoiding deadlock is generally more costly than it is

worth. There are four techniques used to avoid deadlock: transaction scheduling, request rejection, transaction retry, and timestamping.

Transaction scheduling involves scheduling transactions for execution in such a way that two transactions whose data requirements conflict are not run concurrently. As this requires that each transaction's data requirements be known prior to execution time, this method tends to be rather pessimistic. In many cases, it is not known what data will be used until some of the data are examined.

Request rejection is the second method to prevent deadlock. Any request that would immediately cause deadlock is simply rejected, and the transaction can either wait and retry, or tidy up and abort. Deadlock only occurs when the last edge is installed in the wait-for graph, causing a cycle.

Transaction retry was proposed by Rosenkrantz et al. [Rosen78]; it was designed for distributed systems, whereas all the other methods had been originally designed for centralized systems. The technique has two versions; 'wait-die' and 'wound-wait'. The basic idea is to avoid the creation of cycles by a suitable protocol that makes such cycles impossible. There is no need to construct a graph, it is only necessary to be able to tell whether a given record is locked and by which transaction. Each transaction is given a unique timestamp. When transaction **A** requests a lock on a record held by transaction **B**, one of two things can happen. In wait-die, **A** waits if it is older than **B**, otherwise **A** is rolled back and effectively dies. In wound-wait, **A** waits if it is younger than **B**, otherwise **B** is rolled back and automatically

retried; thus **B** is wounded. In all cases, a transaction retains its original timestamp. There is no possible way for a cycle to occur. Rosenkrantz et al. prove that every transaction is guaranteed to terminate (livelock cannot occur), and that transactions can only be rolled back and retried a finite number of times.

The problem of concurrency control is less complex in a centralized system because the one site has all the knowledge about the transaction. In a distributed system, all the sites must communicate with one another in order to have the same amount of information available as in the centralized case. Locking generates a large amount of message traffic in a distributed environment. The communication delays can cause the time taken to perform an update to easily be several orders of magnitude greater than in a centralized system. A centralized lock manager is one solution. All locks would be granted through that one site. However, it is probable that this site will become a system bottleneck and if this site fails, the entire system also fails. Even local transactions (transactions that just access data at their originating site) must send lock requests to the primary site. This adds an unnecessary delay to the local transactions, of which there are probably many.

Garcia-Molina [Garci79] recommends a centralized locking controller for each replicated fragment of data as a centralized control strategy is superior to a distributed one. A better approach is to have a primary copy of each data item. The lock manager at the site containing the primary copy of **x** will handle all lock requests on **x**. Typically, the primary copies of different objects will be at different sites. This method overcomes the drawbacks of the single locking site scheme

but it can lead to global deadlock which is not possible in the single lock manager. Hence, locking is always susceptible to deadlock, while timestamping is not.

3.1.2 Timestamps

Timestamping techniques are all based on the premise that transactions can be assigned a unique identifier, which can be thought of as the transaction's start time. In a distributed system, each node is given a unique identifier, and a timestamp is then generated by concatenating the local time with the local identifier.

The main distinction between timestamping and locking techniques is that locking synchronizes the interleaved execution of a set of transactions in such a way that it is equivalent to some serial execution of those transactions, while timestamping synchronizes that interleaved execution in such a way that is equivalent to a specific serial execution, the execution defined by the chronological order of timestamps.

By definition, there are no locks available to disallow a transaction from reading uncommited changes. It is necessary to defer all physical updates to commit time. If any physical update cannot be performed for a transaction, then none of that transaction's physical updates are performed. The transaction is assigned a new timestamp and is restarted.

A conflict occurs if a transaction attempts to access a record updated by a younger transaction or if a transaction attempts to update

a record that already has been accessed or updated by a younger transaction. Conflicts are resolved by restarting the requesting transaction. As physical updates are not written until commit time, transaction restart never requires any physical rollback. Thus timestamping is deadlock free, but the advantage is gained by restarting transactions instead of letting them wait. Restart tends to be more costly, as all operations performed on the database must be undone.

With conservative timestamping, no database operation is ever performed until that operation can be guaranteed not to cause a conflict at some time in the future [Berns80c]. When a request for a database operation is received, that request is delayed until the system knows that no transaction with a smaller timestamp can arrive. If a site is not sending out any transactions, then the delayed transaction could wait for a long time. Each site is required to send null transactions so that the site's latest timestamp is known at other sites.

Disadvantages to this are that this requires a lot of intersite communication. If there are **N** sites, then **N**(**N**-1) messages are needed to inform every site of the status of every other site. This is infeasible in large networks. This method is very conservative and it eliminates conflicts by actually serializing all operations at each site, not just those that would otherwise conflict.

3.1.3 Optimistic Concurrency Control

In section 3.1.1, locking was discussed as a method of concurrency control. Locking imposes extra overhead on most transactions. For

example, if the number of records is large compared to the number of currently executing transactions, few conflicts will result. The same is true if the number of read transactions is large compared to the number of update transactions.

While locking allows multiple transactions to execute concurrently while preserving database integrity, it can lead to deadlock. Research directed at finding deadlock-free locking protocols usually attempts to lower the expense of concurrency control by permitting less concurrency. By processing operations without concurrency control overhead, a transaction is not delayed during its execution [Schla81].

Distributed concurrency control methods based on the assumption that conflicts are rare are called optimistic (in the sense that they rely on the hope that conflicts between transactions will not occur). An optimistic method discussed by Kung and Robinson in [Kung81] eliminates locking from concurrency controls. Since locks are not used, this method is deadlock free, however care must be taken to prevent starvation. (Starvation is a case where a transaction continues to be restarted, never being allowed to complete, because some of the resources it requires are in use.)

The idea behind the optimistic approach is quite simple. Instead of suspending or rejecting conflicting operations, as in two-phase-locking and timestamping, a transaction is always executed to completion. However, the write operations issued by transactions are performed on local copies of the data. Only at the end of the transaction, if a validation test is passed by the transaction, are the writes applied to

the database. If the validation test is not passed, the transaction is restarted. The validation test verifies if the execution of the transaction is serializable. In order to perform the test, some information about the execution of the transaction must be retained until the validation is performed.

Any transaction consists of three phases: a read phase, a validation phase, and a write phase [Ceri84]. During the read phase, a transaction reads data items from the database, performs computations, and determines new values for the data items of its write-set. These values are not written into the database at this point. During the validation phase a test is performed to see whether the application of the updates to the database which have been computed by the transaction would cause a loss of consistency or not. During the write phase, the updates are applied to the database if the validation phase returned a positive result. If validation fails, the transaction is backed up and started over again as a new transaction. The write phase is needed for a read transaction that displays its read results on a terminal to ensure that the results are consistent.

Starvation occurs when the same transaction is continually started over again as a new transaction. It is prevented by keeping a count of the number of times a transaction is started over, and once this count reaches a threshold, by locking out the entire database so the transaction can finally complete.

During the read phase, all the updates for the transaction, (if any), are written into an update list. The validation phase consists of checking that the updates can be applied at all sites.

During the validation phase, the update list is sent to every site. Voting by each site as to whether or not an update list is valid occurs as follows. Each site votes on whether the update list is valid and sends its vote back to the site of origin. If there is a majority of yes votes, then the transaction is committed, otherwise the update list is discarded and the transaction is restarted. In both cases, the result is communicated back to all sites.

Each site compares the timestamp of each data item of the read-set of the update list to be validated with the corresponding timestamp of the data items which are stored in its local database. If they are equal, the site votes yes; otherwise, it votes no as some conflicting update has occured.

In a locking approach, transactions are controlled by having them wait at certain points, while in an optimistic approach, transactions are controlled by backing them up. Since reading a value can never cause a loss of integrity, reads need not be run with concurrency control overhead. However, if the result of a READ is to be displayed at a terminal, then it is subject to validation just as a WRITE is. If the READ were not subject to validation, then update transactions may have caused inconsistent values to be read. These inconsistent values would then be displayed on the terminal. Kung and Robinson [Kung81] believe the optimistic approach is superior to locking methods in systems where transaction conflict is highly unlikely, such as in query-dominant systems. The optimistic method avoids locking overhead and can better use the parallel processing of a distributed system.

3.1.4 Combination Techniques

The concurrency control techniques of locking and timestamping each have their own problems. Locking is prone to creating deadlock, while timestamping can be overly conservative. It is possible to combine the two techniques to create a better concurrency control algorithm.

The major difficultly in constructing methods that combine two-phase locking and timestamping lies in developing an interface between the two techniques. The problem is to ensure a consistent serialization order between the locking and the timestamping [Berns81b].

The serialization order induced by two-phase locking is determined by the locking points of the transactions that have been synchronized. The serialization order induced by timestamping is determined by the timestamps of the synchronized transactions. To combine both two-phase locking and timestamping, it is necessary to use locked points to induce timestamps [Berns80d].

Associated with each data item is a lock timestamp, L-ts(**x**). When a transaction **T** sets a lock of **x**, it simultaneously retrieves L-ts(**x**). When **T** reaches its locked point it is assigned a timestamp, ts(**T**), greater than any L-ts it retrieved. When **T** releases its lock on **x**, it updates L-ts(**x**) to be max(L-ts(**x**), ts(**T**)). Timestamps generated in this way are consistent with the serialization order induced by 2PL [Berns81b].

One advantage of using a combination technique is that restarts are needed only to prevent or break deadlocks caused by write-write

synchronization; read-write conflicts never cause restarts. This property cannot be attained by a pure two-phase locking method. It can be attained by pure timestamp ordering methods, but only if conservative timestamping is used; in many cases conservative timestamping introduces excessive delay or is otherwise infeasible.

With a combination technique, queries set no writelocks, and the timestamp generation rule does not apply to them. Hence the system is free to assign any timestamp it wishes to a query. It may assign a small timestamp, in which case the query will read old data but is unlikely to be delayed by transactions requesting writes; or it may assign a large timestamp, in which case the query will read current data but is more likely to be delayed. No matter which timestamp is selected, a query can never cause an update to be rejected. This property cannot be easily attained by any pure two-phase locking or timestamp ordering method.

3.2 GLOBAL DEADLOCK

While concurrency control techniques solve the problems of conflicting transactions, they create a brand new problem; that of global deadlock. A global deadlock occurs when there is a deadlock between two or more sites. This occurs when transactions at two sites each require data locked by the other. Since the lock managers are local, no apparent cycle in the local wait-for graph appears to exist. The detection of a distributed deadlock is a distributed task, which requires the exchange of information between different sites. The cycle will appear if the two (or more) wait-for graphs are joined together.

Global deadlock detection incurs further communication overhead because of this need to join the local graphs. Figure 3.3 shows a wait-for graph in a centralized system with obvious cycles. Figure 3.4 shows one way in which the transactions could be split into three distributed sites. Each individual wait-for graph shows no apparent cycles, yet if the three graphs are connected together, then they will show the same cycles as the graph for a centralized system.

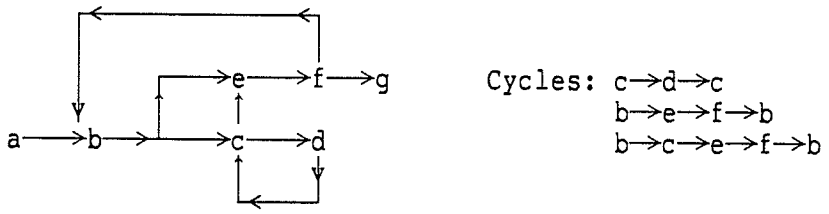

Figure 3.3: Wait-For Graph in a Centralized System

Deadlock may be detected either through a centralized controller, or through a distributed controller. With a centralized controller, one site is chosen at which the centralized deadlock detector is run. The centralized deadlock detector has the responsibility of building the distributed wait-for graph and of discovering cycles in it. In order to do this, the deadlock detector must received information from all sites. The global deadlock detector collects these messages, builds a distributed wait-for graph, checks for cycles, and selects transactions to be aborted. This operation can be performed either periodically or every time there is a change in the situation of potential deadlock

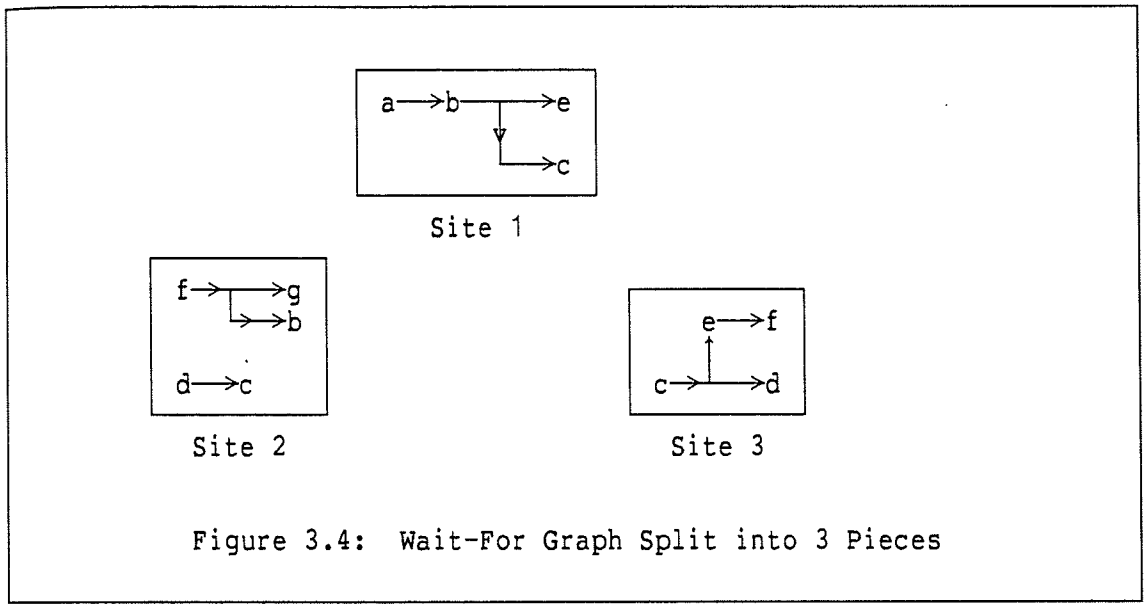

Figure 3.4: Wait-For Graph Split into 3 Pieces

cycles. There is a trade-off between the cost of the detection process and the cost of determining deadlocks late.

Centralized deadlock detection is simple, but it has two main drawbacks. First, it is vulnerable to failures of the site where the centralized detector runs. Secondly, it may require large communication costs, because the centralized detector may be located very far from some other site in the network.

Much theoretical work has been done with the problems of deadlock in both centralized and distributed systems. Recently, Agrawal et al. [Agraw83], have devised what they determine to be a 'cheap' deadlock detection algorithm. By cheap, they mean an algorithm that does not involve an excessive amount of overhead for each transaction.

The basic idea of their algorithm is that whenever a transaction **Ti** requests a lock owned by another transaction, **Tj**, one must test to see

if **Tj** is waiting for **Ti**. Since all the locking information is stored in a wait-for graph, this test is performed by taking a directed walk starting from **Tj** to the root of the tree. A deadlock occurs only if the root corresponds to **Ti**.

While a tree search is expensive for large trees, Agrawal et al. say that the probability of a transaction deadlocked in a cycle of length more than two is very small. Hence, by assuming small cycles, they give a tree traversal algorithm which has a space complexity of O(N).

To improve the performance of their deadlock detection algorithm, Agrawal et al. go on to suggest that the check need not be done continuously. Instead of checking for a cycle before adding an edge to the wait-for graph, the edges are added to the graph without any test and the graph is only periodically examined for cycles. The exact algorithm is given in [Agraw83].

Since this technique is a centralized deadlock detection algorithm, it suffers from the problems of vulnerability and the high communication overhead. As well, a centralized system cannot utilize the full processing power of the network, as all systems must wait for the one centralized system to inform them as to whether they can proceed or abort.

One of the main features of a distributed system is its resiliency from the failure of a few sites. This effect is nullified by using a centralized algorithm, such as a centralized deadlock detection algorithm. If the site running the centralized algorithm fails, then the entire distributed system is unable to function, so it has

effectively failed. Instead of having a centralized deadlock detector, it is possible to use a distributed deadlock detector.

In a distributed deadlock detection mechanism, each site has responsibility for detecting deadlock. Sites exchange information about waiting transactions in order to determine global deadlocks.

One such method is described by Obermarck. Obermarck [Oberm82] presents a distributed algorithm for deadlock detection which works by introducing a special node **EX** into the graph, representing all agents at all other sites. A global deadlock potentially exists whenever a path exists from **EX** through various nodes to another **EX**. On observation of this, one site will transmit a copy of the graph to the site for which it is waiting. This site can then add this additional information to its wait-for graph, and check for cycles in the expanded graph. Obermarck proves that if global deadlock really does exist, then his procedure will cause a cycle to appear at some site, at which time the deadlock can be detected and broken.

The main difference between distributed and centralized deadlock detection is that in centralized deadlock detection all potential deadlock cycles are sent to one designated site, while in distributed deadlock detection there is no such site. Therefore, in distributed deadlock detection, the local deadlock detectors need a rule for determining to which site potential deadlock cycles are transmitted. This rule must assure that global deadlocks are eventually detected and must attempt to minimize the amount of transmitted information.

3.3 FALSE DEADLOCKS

There is a delay associated with the transmission of messages through the network which transfer information for deadlock detection. This delay can cause the detection of false deadlocks. For example, suppose that the deadlock detector receives information that a transaction **Ti** is waiting for transaction **Tj** After some time, **Tj** releases the resource which was requested by **Ti** and requests a resource held by **Ti**. Before receiving the information that **Ti** is not blocked by **Tj** any more, if the deadlock detector received the information that **Tj** requests a resource held by **Ti**, a false deadlock cycle of length 2 is detected.

False deadlock can also occur when a transaction **Tj**, which blocks a transaction **Ti**, aborts for some reason not related to deadlock detection, while at almost the same time **Ti** requires a resource which was held by **Tj**. It is possible that the message informing the deadlock detector of the latter request arrives earlier than the message informing it of the abort; thus the deadlock detector would determine a false deadlock.

Two solutions to false deadlocks exist. First, they can be treated as real deadlocks. This approach is acceptable if the number of false deadlocks is low. The second approach is to validate the deadlock cycle. This requires collecting the information on the presumed cycle a second time, thereby causing more network traffic. If the deadlock were real, it would still be present and therefore be detected again, otherwise it would not be detected again. Distributed protocols are more vulnerable to the occurance of false deadlocks than centralized

ones, due to the delays added to all the additional messages which must be carried over the network.

Chapter IV

DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

In this chapter we will examine some of the currently available distributed database management systems and discuss some of the services they provide. There are several basic capabilities for a DDBMS, as have been discussed earlier in this thesis in section 2.4. These capabilities have to do with controlling the network environment, management of data, data replication and fragmentation, and concurrency control. In this chapter we will give an overview of some of the available DDBMSs, and then examine them with respect to these points.

The available DDBMSs fall into three basic categories: systems that allow no data redundancy; systems which allow limited data redundancy; and systems which are designed to be robust in the wake of multiple failures.

Systems which fall in the first category are IBM's Multiple Systems Coupling and Tandem's ENCOMPAS. These systems were centralized DBMSs which had been modified and enhanced to support access of remote data. In this chapter, we will take a look at Multiple Systems Coupling as a representative system from this area.

Systems which allow limited data redundancy include IBM's R* and Distributed-INGRES. R* is a research product of IBM's San Jose Research Labs and is a distributed version of the commercial product SQL/DS. R*

uses many state of the art techniques for query optimization and concurrency control, but does not support fragmentation of data. Distributed-INGRES is a research system developed at the University of California at Berkeley. It is a distributed version of the INGRES DBMS. A look at Distributed-INGRES is taken later in this chapter.

Systems which fall into the third category include SDD-1 and DDM, both by the Computer Corporation of America. Both systems are designed to give very high reliability by allowing many replicated copies of data and a conservative concurrency control technique. SDD-1 allows both horizontal and vertical fragmentation while DDM allows only horizontal. As SDD-1 was the first operational system to use some of the techniques discussed in previous chapters, we will use it as our representative example in this category.

4.1 IBM'S MULTIPLE SYSTEMS COUPLING

The Multiple Systems Coupling (MSC) feature of IMS allows two or more IMS systems to be interconnected in such a way that an end-user or program on one of those systems can invoke a program on another [Gray79, IBM]. An end-user can enter an input message to invoke a transaction, and that message will be placed on the input queue at the site of entry, exactly as in the nondistributed case. IMS will then examine a local catalogue to see whether the program to be executed for this transaction resides at a remote site. If it does, the input message will then be transmitted to that remote site, where it will be processed just as if it had originally been entered directly at that site. Any output messages will be transmitted back to the original site.

MSC provides location transparency in a limited form. An end-user does not have to know where data or programs reside, but can invoke a transaction from any site. Programs can only access local data. While they do not need to know the precise location of remote data, they do have to know when data are remote and they do have to know the identity of the remote programs involved. To this extent knowledge of the data distribution is built into the application logic. There is no replication transparency; any data replication is user-controlled, not system controlled.

The MSC feature supports four types of physical links between systems [IBM]:

1. binary synchronous communication (BSC) line, using BTAM
2. channel-to-channel (CTC) adapter
3. main storage-to-main storage (MTM)
4. synchronous Data Link Control (SDLC), using VTAM and SNA

Only the BSC line, VTAM, and CTC adapter represent actual hardware links. The MTM link is a software link between IMS systems running in the same processor, and is intended primarily for backup and testing purposes. IF BSC is chosen for the physical link, one side must be the master and the other side must be the slave.

MSC provides a way to extend the throughput of an IMS system beyond the capacity of a single processor. This is possible if the IMS applications can be partitioned among systems such that either:

1. applications execute in more that one system with database contents split between systems (horizontal partitioning)
2. applications execute in one system with complete database that they reference attached to that system (vertical partitioning); the transactions can originate in any system.

Message routing is accomplished by logical destinations. A destination is either a logical terminal or a transaction code. A destination is considered a local destination if it resides in the local system and a remote destination if it resides in a remote system. Each system knows by way of system definition tables or catalogue all local destinations and all remote destinations that may be referenced by that system.

Major design considerations stress defining the environment so that as many transactions as possible are processed locally and to use physical links which go directly from local to remote systems with no intermediate systems and if possible, to use CTC links. The amount of communication between systems is large, so inter-system communication must be minimized for performance. The workload should be distributed in such a way that it avoids excessively high utilization of any one processor. This is done by distributing the applications and their associated transactions and terminals between the available processors.

Each system in the network uses the full recovery capabilities of IMS. These capabilities assure that messages are not lost or duplicated within the single system as long as no cold start or emergency restart from an earlier checkpoint is performed and as long as no log records

are lost. ACF/VTAM provides message integrity for SDLC links in each IMS system in addition to the MSC control functions. This is accomplished by logging information about a transaction in both the sending and receiving systems. This information is restored during restart and exchanged between the systems once the link is started. The sending system can then dequeue a message that was received by the receiving system but for which the acknowledgement was lost because of a link or system failure. The sending system can also resend a message that was sent but not enqueued by the receiving system because of a failure in the receiving system. If a system in the network fails to recover, the messages for which it has recovery responsibility are lost.

Multiple Systems Coupling does not really support distributed transaction processing, instead it supports transaction routing. This does not create a DDBMS, but instead allows some remote access to data. No redundancy is allowed, and if a host fails, then some data is unavailable for access. There is no need for concurrency control or distributed deadlock detection, since multiple copies of data are not supported.

4.2 DISTRIBUTED-INGRES

Distributed-INGRES was developed at the University of California at Berkeley as a distributed version of the relational database system INGRES. Distributed-INGRES was designed to connect machines running the UNIX operating system using local area networks and/or long-haul networks.

Distributed-INGRES was an add-on to INGRES, and as such is not a full-fledged DDBMS, but supports a distributed environment better than does a system such as IMS. Distributed-INGRES does not allow data to be arbitrarily split between nodes, but instead only supports horizontal fragmentation and duplicate copies of relations. Vertical fragmentation is not allowed [Stone77]. This simplifies the DDBMS in some ways, as there is no need to transfer data between nodes during the execution of a transaction to join several vertical fragments to form one record. Fragments can be duplicated at various nodes; but if they are, then one copy is designated a primary copy. This primary copy is used by transaction management, concurrency control, and reliability algorithms.

Query processing in Distributed-INGRES has two optimization procedures. First, processing is distributed for exploiting parallelism. The more sites that are involved, the greater the parallelism, and the less time that is needed to process the query. This optimization leads to redistributing fragments of relations at the execution sites of operations, with the purpose of equalizing fragments (having fragments of about the same size and therefore requiring about the same processing time) [Ceri84]. Equalization is used especially with LANs, where communication bandwidth is high. Equalization conflicts with locality of processing, since equalization requires distributing a relation even when processing would be local.

Secondly, since optimization is done during execution, if intermediate results are unexpectedly large, the backtracking of a tactic can be done. An additional optimization can then be done in order to change the query processing strategy.

Catalogues for data on each machine are stored on that machine. For non-local data, a cache is maintained of the most recently used fragments and where they are located. If the location is not in the cache, then the entire network is searched to find the data.

Distributed-INGRES uses two-phase locking for concurrency control (section 3.1.1). All locks must be granted at once; if not, then all locks must be released and the transaction must try again. Deadlocks are detected and resolved with a centralized deadlock detector. All wait-for information is sent to a single process, called **snoop** [Stone79]. This method has the disadvantage of heavy communications and requires solving the reliability problems of failures and recoveries of the site where the **snoop** is located.

Updates are first applied to the primary copy of the data, and then to the non-primary copies. In the commitment of transactions, each participant responds 'ready to commit' to the co-ordinator after having updated the primary copy, and waits for the decision from the co-ordinator. In the case of successful commitment, each participant generates a process which co-ordinates several 'copy' agents for applying the deferred updates to nonprimary copies. This protocol is not resilient to a failure of participants which occurs after having performed the local commit but before having generated the 'copy' agents. As well, mutual consistency of different fragments could be lost when the same update transaction operates on different fragments; then only part of them would be correctly updated. To make this system more reliable, it is possible to incorporate nonprimary copies in the two-phase-commitment and queue messages for crashed sites to at least **k**

other sites (for having **k**-resiliency). As mentioned in section 3.1.1, the concurrency control method generates a large amount of message traffic for highly replicated data. This is not a problem in a fast LAN environment, but in a slow long-haul network, delays quickly build up. If only a few copies of each relation exist, the expense of concurrency control is low, even with a long-haul network.

4.3 SDD-1

SDD-1 is a distributed database management system currently being developed by Computer Corporation of America [Rothn80]. SDD-1 manages databases whose storage is distributed over a network of computers. Users interact with SDD-1 precisely as if it were a nondistributed database system because SDD-1 handles all issues arising from the distribution of data such as distributed concurrency control, resiliency to component failure, and distributed directory management. SDD-1 was designed to permit a large amount of replication of data. This is desirable for lessening transmissions by a potentially worldwide system, and for increasing availability and survivability of the information, especially when under military attack. SDD-1 is designed to use long-haul networks. No optimization is possible when using a LAN.

An SDD-1 database consists of logical relations. Each SDD-1 relation is partitioned into subrelations called logical fragments, which are the units of data distribution, meaning that each may be stored at any one or several sites in the system. Logical fragments are defined and the assignment of fragments to sites is made when the database is designed. User transactions are unaware of data distribution or redundancy. They

reference only relations, not fragments. It is SDD-1's responsibility to translate from relations to logical fragments, and then to select the stored fragments to access in processing any given transaction.

SDD-1 uses a local data manager and a separate network data manager. The local data manager has the functions of a conventional single-site DBMS, and does not worry about distribution problems. The network data manager does not access the data itself, but determines the access strategy for handling each distributed data operation efficiently. It requests the local data managers at any site to perform local processing and/or move portions of the data from one site to another.

SDD-1 is a collection of three types of virtual machines: Transaction Modules (TMs); Data Modules (DMs); and a Reliable Network (RelNet). All data managed by SDD-1 are stored by Data Modules. DMs respond to commands from Transaction Modules. DMs respond to four types of commands [Rothn80]:

1. read part of the DM's database into a local workspace at that DM;
2. move part of a local workspace from this DM to another DM;
3. manipulate data in a local workspace at the DM;
4. write part of the local workspace into the permanent database stored at the DM.

Transaction Modules plan and control the distributed execution of transactions. Each transaction processed by SDD-1 is supervised by a TM which performs several tasks. First, the TM translates queries on relations into queries on logical fragments and decides which instances of stored fragments to access. The TM then synchronizes the transaction

with all other active transactions in the system. Next, the TM compiles the transaction into a parallel program which can be executed co-operatively by several DMs. Finally the TM co-ordinates execution of the compiled access plan, exploiting parallelism whenever possible.

The Reliable Network interconnects TMs and DMs in a robust fashion and provides four services: guaranteed delivery, transaction control, site monitoring, and a network clock [Hamme80]. Guaranteed delivery allows messages to be delivered even if the recipient is down at the time the message is sent, and even if the sender and receiver are never up simultaneously. Transaction control is a mechanism for posting updates at multiple DMs, guaranteeing that either all DMs post the update, or none do. Site monitoring keeps track of which sites have failed, and informs sites impacted by failures. A network clock is a virtual clock kept approximately synchronized at all sites.

The overall SDD-1 design is simplified by having the architecture divide the DDBMS problem into three pieces: database management, management of distributed transactions, and DDBMS availability. Each of these pieces is implemented as a self-contained virtual machine.

There are three problem areas in a distributed DBMS: concurrency control, distributed query processing, and reliable posting of updates. SDD-1 has a three-phase processing procedure, with each phase handling one of the three problems of READ, EXECUTE, and WRITE. By handling each of the problems separately, overall complexity is reduced [Rothn80].

The READ phase exists to control concurrency. The TM that is supervising a transaction **T** analyzes it to determine its read-set, the

portion of the logical database it reads. In addition, to obtain that data, the TM decides which stored fragments to access. The TM then issues READ commands to the DM's that house those fragments, instructing each DM to set aside a private copy of that fragment for use during subsequent processing phases.

These private copies obtained by the READ phase are guaranteed to be consistent even though the copies reside at distributed sites. Since the data are consistent when read, and since the copies are private, subsequent phases can operate freely on these data without fear of interference from other transactions.

No data are actually transferred between sites during the READ phase, the data are set aside in a workspace at the DM. The organization of data at a DM is in fragments called pages. A page is a unit of logical storage. Pages are referenced through a page map, a function which associates a physical storage location with each page. With this type of storage organization, it is possible to make a copy of data by just copying the page map. Thus no physical data are copied. Since pages are never updated in place, a copy of the page map is a copy of the file. If a page is to be updated, a new block of secondary storage is allocated, and the modified page is written there. This scheme has been used very successfully in the System R DBMS [Astra76]. Only the updating transaction can thus reference the new page because only its page map is updated. Other transactions are unaffected because their page maps remain unchanged. When the entire transaction is completed, then the master page map is updated. In this way, if a transaction is aborted or fails, there is no need to undo any of its actions on the data.

The second phase is the EXECUTE phase. This phase implements the distributed query processing. At this time, the TM compiles the transaction **T** into a distributed program that takes as input the distributed workspace created by the READ phase. The compiled program is supervised by the TM to ensure that the commands are sent to the DMs in the correct order. As well, the TM must handle run-time errors which occur.

The output of this compiled program is a list of data items to be written into the database (in the case of update transactions) or displayed to the user (in the case of retrievals). This output list is produced in a workspace at one DM, and is not yet reflected in the permanent database. Consequently, problems of concurrency control and reliable writing are irrelevant during this phase.

The WRITE phase is the final phase in the execution of a transaction. This phase installs data modified by a transaction **T** into the permanent database and/or displays data retrieved by **T** to the user. For each entry in the output list, the TM determines which DM(s) contain copies of that data item. The TM orders the final DM that holds the output list to send the appropriate entries of the output list to each DM; it then issues WRITE commands to each of these DMs thereby causing the new values to be installed into the database. Special techniques must be used during the WRITE phase to ensure that partial results are not installed even if multiple sites or communication links fail midstream.

SDD-1 attempts to optimize the execution of a query command through an optimization procedure called access planning. Access planning

minimizes the transaction's intersite communication needs while maximizing its parallelism [Berns81a].

The simplest way to execute a transaction **T** is to move all of **T**'s read-set to a single DM, and then execute **T** at that DM. While this approach produces the correct answer, it has two major drawbacks: **T**'s read-set might be large, thus moving it between sites would be extremely expensive; and little use is made of the parallel processing capabilities of the network.

The access planner overcomes these drawbacks by eliminating as much data from **T**'s read-set as is economically feasible without changing **T**'s answer. Then, during the final processing, the reduced read-set is moved to some designated DM where **T** is executed. The effect of this execution is to create a temporary file to be written to the database or to be displayed for the user.

To complete the transaction processing, the temporary file at the final DM must be installed in the permanent database. Since the database is distributed, the temporary file must be split into a set of temporary files that list the updates at each DM.

Each of these temporary files must be transferred to the appropriate DM by a WRITE command. The problem which results is to ensure that all DMs install the updates. There are two types of failures which can occur: failure of a receiving DM, and failure of the sender. The first is handled by reliable delivery and the second by transaction control.

Reliable delivery is accomplished by the communications network where duplicate messages, missing messages, and damaged messages are detected and corrected. There exists a problem if both the sender and receiver are not up simultaneously. To overcome this problem, RelNet uses the spooling facility described earlier in section 2.3.3 (A spooler is a process with access to secondary storage that serves as a first in, first out message queue for a failed site. Any message destined for a failed DM is delivered to its spooler instead [Hamme80].) Each spooler uses DBMS reliability techniques to guarantee the integrity of its messages. To protect against spooler failure, multiple spoolers are used. As long as one spooler is running, messages can be reliably stored.

To keep track of all the data, SDD-1 maintains directories containing data fragment definitions, locations, and usage statistics. [Rothn80]. Efficient and flexible directory management is important to SDD-1 as TMs use directories for every transaction. SDD-1 treats the directory as ordinary user data, hence the directories can be fragmented, distributed with arbitrary redundancy, and updated from arbitrary DMs.

With this flexibility in directory management comes the possibility of performance degradation since every directory access incurs general transaction overhead, and every access to remotely stored directories incurs communication delays. For example, it is possible, though inefficient, for a local object to have its directory entry at a remote site. This problem is avoided in SDD-1 by caching recently referenced directory fragments at each TM, discarding them if rendered obsolete by directory updates. Directories tend to be relatively static, so this does not incur a large performance degradation.

The second problem is that a directory is needed to tell where each directory fragment is stored. SDD-1 calls this directory a directory locator, and stores a copy at each DM. The directory locators are relatively small and static in nature.

4.3.1 Concurrency Control

To obtain maximum transaction concurrency, SDD-1 uses several of the previously described concurrency control algorithms. Transactions are pre-analyzed to fit into a set of classes. Conflict graphs are used to determine if two transaction classes will conflict at run time. Timestamps are used on the stored data items to order READs and WRITEs. This combination of techniques was used to avoid the overly conservative aspects of locking. In a system that uses locking, each transaction locks data before accessing them, so conflicting transactions never run concurrently (section 3.1.1). However, not all conflicts violate serializability, and some conflicting transactions can safely be run concurrently. Greater concurrency can be attained by checking whether or not a given conflict is troublesome, and only synchronizing those that are. Conflict graph analysis is a technique used by SDD-1 for doing this [Berns78b,Berns80a].

The nodes of a conflict graph represent the read-sets and write-sets of transactions, and the edges represent conflicts among these sets. Different kinds of edges require different levels of synchronization. Blocking is a strong synchronization and is only required for edges that participate in cycles. Interclass conflicts that cannot cause nonserializable behavior do not need to be blocked.

Too much intersite communication would be required to exchange information about conflicts at run time. Instead, transaction classes, that is; named groups of commonly executed transactions, are defined by the database administrator at database design time. Each class is defined by its name, a read-set, a write-set, and the TM at which it runs. Conflict graph analysis is performed on the transaction classes, and not on individual transactions. The output of the analysis is a table which tells, for each class, which other classes that class conflicts with, and for each conflict, how much synchronization is required to ensure serializability [Berns80b]. Classes are exploited by conservative timestamp ordering schedulers [Berns81b].

To synchronize two transactions that conflict dangerously, one must be run first, and the other delayed until it can safely proceed. In systems which use locking, the execution order is determined by the order in which transactions request conflicting locks. In SDD-1, the order is determined by a total ordering of transactions induced by timestamps (section 3.1.2). Each transaction in SDD-1 is assigned a unique timestamp by its TM. The values of local clocks used for timestamps are kept synchronized by advancing the local time if a message with a later timestamp is received. The timestamp of each transaction is sent along with each update message.

When a DM receives a READ command, it defers the command until it has processed all earlier WRITE commands, and no later WRITE commands from the specified classes. The DM can determine how long to wait because of a DM-TM communication called piping.

Piping requires that each TM send its WRITE commands to DMs in timestamp order. In addition, the Reliable Network guarantees that messages are received in the order sent.

Thus when a DM receives a WRITE from a TM with a certain timestamp, the DM knows it received all WRITE commands from that TM with smaller timestamps. To process a READ command with a timestamp **TSi**, the DM first processes all WRITE commands from that class's TM up to, but not beyond, **TSi**. If the DM has already processed a WRITE command with timestamp beyond **TSi**, the READ is rejected.

To avoid excessive delays in waiting for WRITE commands, idle TMs periodically send null timestamped WRITE commands; also an impatient DM can explicitly request a null WRITE from a TM that is slow in sending them.

To complete transaction processing, the temporary site at the final DM must be installed in the permanent database and/or displayed to the user. Since the database is distributed, the temporary file must be split into files for each DM.

Each of the temporary files is transmitted to the appropriate DM as a WRITE command. The problem is to ensure that failures cannot cause some DMs to install updates while causing others not to. Failure of a receiving DM is handled by reliable delivery, while failure of the sender is handled by transaction control.

Transaction control addresses failures of the final DM that occur during the WRITE phase. If the final DM fails after sending some

updates, then the database will be inconsistent as some DMs will reflect the results of the transaction, and some will not. Transaction control ensures that inconsistencies of this type are rectified in a timely fashion.

The basic technique used for transaction control is a variant of two-phase commit [Rothn80], discussed in section 3.1.1. In SDD-1, the final DM transmits the data for the WRITEs during phase 1, but the receiving DMs do not install them yet. During phase 2, the final DM sends commit messages to the DMs, and those DMs install the data. If some DM has received its data but not a commit, and the final DM has failed, this DM consults the other DMs involved in the transaction. If one has received a commit, the DM does the installation; if no DM received a commit, no DM does the installation, thereby aborting the transaction. While this offers complete protection against failures of the final DM, this technique is susceptible to multisite failures.

Instead of processing WRITE messages in timestamp order, each data item has a timestamp associated with it. This timestamp is the timestamp of the last WRITE message that updated it.

Each DM processes WRITEs by only updating a data item if the data item's timestamp is less than the WRITE message's timestamp. The WRITE command contains the new value for the data item. If the timestamp of the WRITE message exceeds the timestamp of the stored data item, then the new value of the data item in the WRITE message is written into the stored data item also with the new timestamp. Otherwise, the update is not performed on that stored data item. This is a data item by data

item check, and some data items in the WRITE message may result in update operations, while others may not.

The data items which do not result in update operations are not errors. It is simply the way SDD-1 reorders updates to occur in the same order that their generating transactions executed. The net effect is the same as if they were updated in timestamp order. The advantage to this though, is that a WRITE message can be processed as soon as it is received, thereby avoiding artificial queuing delays at the DMs. However, since later WRITEs may be processed before earlier ones, a database copy may be temporarily inconsistent. Concurrency control never permits a transaction to read an inconsistent state if this could lead to incorrect results.

All timestamp-related mechanisms in SDD-1 will operate correctly with unsynchronized clocks, but for reasons of efficiency, it is necessary to assume that clock values in different TMs are reasonably close.

One problem with timestamped data items is the cost of storing the timestamps. If the timestamp of a data item is earlier than the timestamp of any transaction whose WRITE messages have not been processed, then the data item's timestamp is effectively zero. Thus it is only necessary to cache the timestamps of recently updated items, and after a period of time, the data item's timestamp can be assumed to be zero.

The SDD-1 concurrency control mechanism fully distributes the concurrency control [Berns80a]. While each transaction is controlled from a single site, different sites are concurrently supervising the

synchronization of many different transactions. No one site is in charge of any system-wide activity. The main advantage of this full distribution is enhanced reliability. A site failure only effects those transactions executing and/or using data at that site.

This concurrency control method is generally inefficient for unpredictable transactions as it requires transactions to preclaim their resources. Although based on an original theoretical approach, practical performance of this method for concurrency control is conditional upon several assumptions whose validity remains to be demonstrated. The SDD-1 approach works well if several conditions hold true. First, transactions can be grouped into disjoint classes. Secondly, transactions can preclaim their resource requirements. Thirdly, several transactions belonging to different classes should be simultaneously available for execution so as to provide a good mix. This method will not work well if the majority of transactions are in the same class, as they will conflict. This method works poorly if the transactions cannot be classed into disjoint classes.

Chapter V

CONCLUSIONS

In this thesis, we have examined the reasons for using a DDBMS. Some of these reasons are the possibility for increased performance and reliability through the use of redundant data and multiple parallel execution units.

A close look was taken at the hardware and software needed to interconnect computers. Some general standards were examined for networks, along with the International Standards Organization's Open System Interconnect model for networks. One problem with operating a DDBMS in a distributed environment is the high cost of transmitting data, and the lack of security of this transmitted data. Various ways of compressing data were examined, along with some ways of enciphering the data to make it more secure. Long-haul networks were also compared to local area networks. Several performance criteria were examined, and a look at network overhead showed that with proper tuning, a network can be very efficient. Local area networks can only span a small area, but have a speed advantage over long-haul networks.

Several commercial networks were then compared and contrasted. SNA and DECnet were shown to have some deficiencies when operating in a DDBMS environment. RelNet attempts to overcome those deficiencies by operating on top of a network, and providing some services which are beneficial to a DDBMS.

While a distributed database can lead to faster and more reliable database operation, the extra hardware and software create new problems. These problems deal with keeping the multiple copies of data consistent through the use of concurrency control methods.

With distributed data, there is also a problem of how to best access the replicated, fragmented data. An algorithm which decides to move a large amount of data over the network will take hours instead of seconds to process a query. To be able to retrieve the dispersed data, each system must be able to first locate the data. This is done through a data dictionary, which is a database in its own right.

Various locking techniques were discussed as concurrency control methods. Many locking techniques involve some sort of centralized controller. The centralization of any function undermines some of the very rationale for data distribution - reliability, accessibility, and availability of the system. Other problems include the possibility of congestion of the central node and unnecessary serialization of all updates.

Timestamping techniques eliminate the problems of locking, but can be overly conservative causing transactions to wait needlessly. Hybrid techniques which use both locking and timestamping seem to combine the best of both techniques, without the drawbacks of either. These remain to be tested in a real system.

A major concurrency control side effect is that of global deadlock. Because of the time delays encountered in the network, false deadlocks can appear. Additional overhead is needed to check if the deadlock is real, and if so, to cancel the transaction.

In the final chapter, we examined several commercial DDBMS. IBM's Multiple System Coupling was shown to lack most of the essential features of a true DDBMS. Distributed-INGRES supports most of the features of a DDBMS. However it does use a centralized deadlock detector. Distributed-INGRES uses a primary copy two-phase locking technique. Problems arise if the updates are not applied to the non-primary copies or if the primary site fails.

SDD-1 utilizes the assumption that probability of interference among concurrent transactions is low by preanalyzing transactions in order to determine which transactions could possibly affect each other. Transactions are grouped into different classes based on their read and write sets. Transactions whose classes do not intersect may be safely scheduled for parallel execution, because they cannot pose conflicting resource requests. Conflicting transactions are scheduled for serial execution.

Further research is required to determine the benefits of the various concurrency control algorithms, and to demonstrate the conditions under which various methods work well and under which they work poorly. No one has yet devised a way to survive network partitions. It seems that there is no one best method for concurrency control, but it may be shown that one of the hybrid techniques is indeed superior to either timestamping or locking.

One problem not discussed is that of tuning a distributed database. Not enough information is known on what information is needed to tune a system and where the tradeoffs are between the benefits and costs of

multiple copies of data. Statistics are gathered by both the DBMS, and the network, but these statistics must be related to one another to get a true picture of what needs to be tuned. As the use of distributed data grows, there will be more need to solve these problems.

BIBLIOGRAPHY

[Adiba81] Adiba, Michel, and Juan Andrade, "Update Consistency and Parallelism in Distributed Databases", Proceedings 2nd International Conference on Distributed Computing Systems, Computer Society Press, Paris, France, 1981, pp. 180-187.

[Agraw83] Agrawal, Rakesh et al., "Deadlock Detection is Cheap", ACM SIGMOD Record, volume 13, number 2, January 1983, pp. 19-34.

[Akl83] Akl, Selim G. "Digital Signatures: A Tutorial Survey", IEEE Computer, February, 1983, volume 16, nummber 2, pp. 15-24.

[Allen82] Allen, Frank W., Mary E.S. Loomis, Michael V. Mannino, "The Integrated Dictionary/Directory System", ACM Computing Surveys, volume 14, number 2, June, 1982, pp. 245-275.

[Astra76] Astrahan, M. M., "System R: A relational Approach to Database Management", ACM Transactions on Database Systems, volume 1, number 3, 1976.

[Atre80] Atre, S., Data Base: Structured Techniques for Design, Performance, and Management, John Wiley & Sons, Inc., 1980.

[Badal78] Badal, D. Z., and G. J. Popek, "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems", Proceedings of the Third Berkeley Conference on Distributed Data Management and Computer Networks, August 1978, pp. 273-288.

[Badal79] Badal, Dusan Zdenek, Semantic Integrity, Consistency and Concurrency Control in Distributed Databases, University of California, Los Angeles, Ph.D. Thesis, 1979.

[Badal80] Badal, D. Z., "Concurrency Control Overhead or a Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, Berkeley, CA., February, 1980, pp. 85-104.

[Benha83] Benhamou, Eric, and Judy Estrin, "Multilevel Internetworking Gateways: Architecture and Applications", IEEE Computer, volumer 16, number 9, September 1983, pp. 27-34.

[Berns78a] Bernstein, Philip A., and David W. Shipman, "A Formal Model of Concurrency Control Mechanisms for Database Systems", Proceedings of the Third Berkeley Conference on Distributed Data Management and Computer Networks, August 1978, pp. 189-205.

[Berns78b] Bernstein, P.A. , N. Goodman, J. B. Rothnie, and C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Transactions on Software Engeneering, volume SE-4, number 3, May 1978, pp. 154-168.

[Berns80a] Bernstein, Philip A., David W. Shipman, and James B. Rothnie, Jr, "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, March 1980, volume 5, number 1, pp. 18-51.

[Berns80b] Bernstein, Philip A., and David W. Shipman, "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, March 1980, volume 5, number 1, pp. 52-68.

[Berns80c] Bernstein, P. A., and N. Goodman, "Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems", Proceedings 6th International Conference Very Large Data Bases, October 1980.

[Berns80d] Bernstein, Philip A., Nathan Goodman, and Ming-Yu Lai, "Too Part Proof Schema for Database Concurrency Control", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, Berkeley, CA., February, 1980, pp. 71-84.

[Berns81a] Bernstein, Philip A., D. W. Shipman, and J. B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems volume 6, number 4, December 1981, pp. 602-625.

[Berns81b] Bernstein, Philip A., and Nathan Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, volume 13, number 2, June 1981, pp. 185-221.

[Berns83] Bernstein, Philip A, and Nathan Goodman, "Multiversion Concurrency Control - Theory and Algorithms", ACM Transactions on Database Systems, volume 8, number 4, December, 1983, pp. 465-489.

[Carde79] Cardenas, Alfonso F., Data Base Management Systems, Allyn and Bacon, Inc., Boston, Mass., 1979.

[Ceri84] Ceri, Stefano, and Guiseppe Pelagatti, Distributed Databases Principles & Systems, McGraw-Hill Book Company, 1984.

[Date83] Date, C.J., An Introduction to Database Systems, volume II, Addison-Wesley Publishing Company, 1983.

[Davie81] Davies, Donald W., "Ciphers and the Application to the Data Encryption Standard", IEEE Tutorial: The Security of Data in Networks, New York, N.Y., 1981 pp. 3-16.

[Deato80] Deaton, G. A. and D. J. Franse, "Analyzing IBM's 3270 Performance Over Satellite Links", Data Communications, October, 1980.

[Delob80] Delobel, C., and W. Litwin (ed.), Distributed Databases, North-Holland, New York, 1980.

[Diffi76] Diffie, W., and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, November 1976, volume IT-22, number 6, pp. 644-654.

[Diffi77] Diffie, Whitfield and Martin E. Hellman, "Exhaustive Cryptanalysis of the NBS Data Encryption Standard", IEEE Computer, volume 10, number 6, June 1977, pp. 74-84.

[FIPS77] "Specifications for the Data Encryption Standard", Federal Information Processing Standards Publication 46, National Bureau of Standards, January 15, 1977.

[Garci79] Garcia-Molina, Hector, "A Concurrency Control Mechanism for Distributed Databases Which Use Centralized Locking Controllers", Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks, August, 1979, pp. 113-124.

[Garci82] Garcia-Molina, Hector, "Reliability Issues for Fully Replicated Distributed Databases", IEEE Computer, volume 15, number 9, September 1982, pp. 34-42.

[Gray75] Gray, J. N., R. A. Lorie, and G. R. Putzolu, "Granularity of Locks in a Large Shared Data Base", Proceedings 1st International Conference on Very Large Data Bases, September, 1975.

[Gray79] Gray, J. P. and T. B. McNeil, "SNA Multiple-system Networking", IBM Systems Journal, volume 18, number 2, 1979, pp. 263-297.

[Green82] Green, Paul E., Jr, (Ed), Computer Network Architectures and Protocols, Plenum Press, New York, 1982.

[Hamme80] Hammer, Michael, and David Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases", ACM Transactions on Database Systems, December, 1980, volume 5, number 4, pp. 431-466.

[Hevne78] Hevner, Alan R., and S. Bing Yao, "Query Processing on a Distributed Database", Tutorial: Distributed Database Management, IEEE Computer Society, 1978, pp. 69-85.

[Hinde83] Hinden, Robert, Jack Haverty, and Alan Sheltzer, "The DARPA Internet: Interconnecting Heterogeneous Computer Networks with Gateways", IEEE Computer, volume 16, number 9, September 1983, pp. 38-48.

[IBM] IBM Corporation, IMS/VS System Administration Guide, IBM Form number SH20-9178.

[Kak83] Kak, Subhask C., "Data Security in Computer Networks", IEEE Computer, February 1983, volume 16, number 2, pp. 8-10.

[Korth80] Korth, Henry F., "A Deadlock-Free, Variable Granularity Locking Protocol", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, Berkeley, CA., February, 1980, pp. 105-121.

[Kung81] Kung, H. T., and J. T. Robinson, "Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, volume 6, number 2, June 1981, pp. 213-226.

[Lamps81] Lampson, B.W., M. Paul, and H. J. Siegut, Distributed Systems - Architecture and Implementation, Springer-Verlag, New York, 1981.

[Lim79] Lim, Wen-Te K., "Concurrency Control in a Multiple Copy Distributed Database System", Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks, August, 1979, pp. 207-220.

[Marti77] Martin, James, Computer Data-Base Organization, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

[Marti78] Martin, James, Communications Satellite Systems, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

[Marti79a] Martin, James, Distributed File and Data Base Design: Tools and Techniques, Savant Institute, August, 1979.

[Marti79b] Martin, James, Distributed Data Processing: The Opportunity and the Challenge, Savant Institute, September, 1979.

[Marti81a] Martin, James, Design and Strategy for Distributed Data Processing, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[Marti81b] Martin, James, Computer Networks and Distributed Processing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[McLea81] McLean Jr., Gordon, "Comments on SDD-1 Concurrency Control Mechanisms", ACM Transactions on Database Systems, volume 6, number 2, June 1981, pp. 347-350.

[Meras78] Merasce, Daniel A., and Richard R. Muntz, "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Conference on Distributed Data Management and Computer Networks, August 1978, pp. 215-234.

[Merkl78] Merkle, Ralph C., and Martin E. Hellman, "Hiding Information and Signatures in Trapdoor Knapsacks", IEEE Transactions on Information Theory, volume IT-24, number 5, September 1978, pp. 525-530.

[Milen81] Milenković, Milan, Update Synchronization in Multiaccess Systems, UMI Research Press, Ann Arbor, Michagin, 1981.

[Minou79] Minoura, Toshimi, "A New Concurrency Control Algorithm for Distributed Database Systems", Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks, August, 1979, pp. 221-234.

[Mulle83] Muller-Schloer, Christian, "A Microprocessor-based Cryptoprocessor", IEEE Micro, volume 3, number 5, pp. 5-15.

[Oberm81] Obermarck, R., and C. Beeri, "A Resource Class Independent Deadlock Detection Algorithm", Proceedings 7th. International Conference on Very Large Databases, Sept. 1981.

[Oberm82] Obermarck, Ron, "Distributed Deadlock Detection Algorithm", ACM Transactions on Database Systems, June, 1982, volume 7, number 2, pp. 187-208.

[Parke84] Parker, Donn B., "The Many Faces of Data Vulnerability", IEEE Spectrum, May, 1984, pp. 46-49.

[Perry84] Perry, Tekla S., and Paul Wallich, Associate Editors, "Can Computer Crime Be Stopped?", IEEE Spectrum, May, 1984, pp. 34-45.

[Pooch83] Pooch, Udo W., William H. Greene, and Gary G. Moss, Telecommunications and Networking Little, Brown & Company (Canada) Limited, 1983.

[Ries79] Ries, D., "The Effects of Concurrency Control on the Performance of a Distributed Data Management System", Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks, August, 1979, pp. 75-112.

[Rives78a] Rivest, R. L., A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, volumes 21, number 2, February, 1978, pp. 120-126.

[Rives78b] Rivest, Ronald L., "Remarks on a Proposed Cryptanalytic Attack on the M.I.T. Public-Key Cryptosystem", Cryptologia, volumes 2, number 1, January, 1978, pp. 62-65.

[Rosen78] Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis, "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems, volume 3, number 2, June 1978.

[Rothn77] Rothnie, James B., and Nathan Goodman, "A Survey of Research and Development in Distributed Database Management", IEEE Tutorial: Distributed Database Management, pp. 30-41.

[Rothn80] Rothnie, J. B. Jr., et. al., "Introduction to a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, volume 5, number 1, March 1980, pp. 1-17.

[Rushb83] Rushby, John, and Brian Randell, "A Distributed Secure System", IEEE Computer, volume 16, number 7, July, 1983, pp. 55-68.

[Sagiv81] Sagiv, Yehoshua C., Optimization of Queries in Relational Databases, UMI Research Press, Ann Arbor, Michigan, 1981.

[Schla81] Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems", Proceedings 7th International Conference on Very Large Databases, Sept. 1981.

[Schne83] Schneidewind, Norman, "Interconnecting Local Networks to Long-Distance Networks", IEEE Computer, volume 16, number 9, September, 1983, pp. 15-24.

[Schul81] Schultz, Gary D., "An Anatomy of SNA", Computerworld, March 18, 1981, pp. 35-41.

[Shami80] Shamir, Adi, and Richard E. Zippel, "On the Security of the Merkle-Hellman Cryptographic Scheme", IEEE Transactions on Information Theory, volume IT-26, number 3, May, 1980, pp. 339-340.

[Stone77] Stonebraker, M., and Neuhold, E., "A Distributed Database Version of INGRES", Proceedings Second Berkeley Workshop Distributed Data Management and Computer Networks, May 1977, pp. 19-36.

[Stone79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Transactions on Software Engineering, volume SE-5, number 3, May 1979, pp. 188-194.

[Sunds80] Sundstrom, R. J. and G. D. Schultz, "SNA's First Six Years: 1974-1980", Proceedings of the Fifth International Conference on Computer Communication, Atlanta, Georgia, October 27-30, 1980, pp. 578-585.

[Susse78] Sussenguth, E. H., "Systems Network Architecture: A Perspective", Proceedings of the Fourth International Conference on Computer Communication, Kyoto, Japan, September 26-29, 1978, pp. 353-358.

[Tanen81a] Tanenbaum, Andrew S., Computer Networks, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Tanen81b] Tanenbaum, Andrew S., "Network Protocols", ACM Computing Surveys, volume 13, number 4, December 1981, pp. 453-489.

[Unsoi81] Unsoi, M. S., "Performance Monitoring and Evaluation of Datapac Network", NTC, pp. f6.5/1-5, 1981.

[Voydo83] Voydock, Victor L., and Stephen T. Kent, "Security Mechanisms in High-Level Network Protocols", ACM Computing Surveys, volume 15, number 2, June 1983, pp. 135-171.

[Wah81] Wah, Benjamin W., Data Management on Distributed Databases, UMI Research Press, Ann Arbor, Michigan, 1981.

LIST OF REFERENCES

Adiba81 98
Agraw83 123-124
Allen82 . . . . . 73, 87
Astra76 138
Atre80 2

Badal78 98
Badal79 98
Badal80 98
Benha83 21
Berns78a 98
Berns78b 98, 142
Berns80a 98, 142, 146
Berns80b 98, 143
Berns80c 98, 116
Berns80d 98, 120
Berns81a 96, 140
Berns81b
. . 83, 98, 105, 112, 120, 143
Berns83 98

Carde79 . . . . . 2, 76, 109
Ceri84 . . . . . 118, 133

Date83 . . . 2, 72, 101-102, 111
Davie81 53-54
Diffi76 55
Diffi77 54

FIPS77 54

Garci79 . . . . . 98, 114
Garci82 . . . . . 98, 105
Gray75 82
Gray79 129
Green82 64

Hamme80 . . . . . 69, 71, 137, 141
Hevne78 96
Hinde83 21

IBM 129-130

Kak83 52
Korth80 98
Kung81 . . . . . 98, 117, 119

Lim79 98

Marti77 . . . . . 2, 76
Marti78 4
Marti79a 5, 16, 87
Marti79b 4-5, 11, 14
Marti81a 3
Marti81b 29, 37, 40
Meras78 98
Merkl78 56
Milen78 98
Milen81 106
Minou79 98
Mulle83 56

Oberm82 125

Parke84 47
Perry84 47
Pooch83 4

Ries79 . . . . . 98, 100
Rives78a 55
Rives78b 56
Rosen78 . . . . . 98, 113
Rothn77 . . . . . 87, 95, 106
Rothn80 . . . . 135-137, 141, 145
Rushb83 57

Sagiv81 95
Schla81 . . . . . 98, 117
Schne83 40
Schul81 . . . . . 59, 61
Shami80 56
Stone77 133
Stone79 . . . . . 98, 112, 134
Sunds80 58
Susse78 58

Tanen81a 22-23, 25
29-30, 34, 38, 40, 42, 45-46
Tanen81b 30

Unsoi81 . . . . . 37, 41

Voydo83 47

Wah81 95

LIST OF DEFINITIONS