PROCEEDINGS of the

SIXTH BERKELEY WORKSHOP on DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS

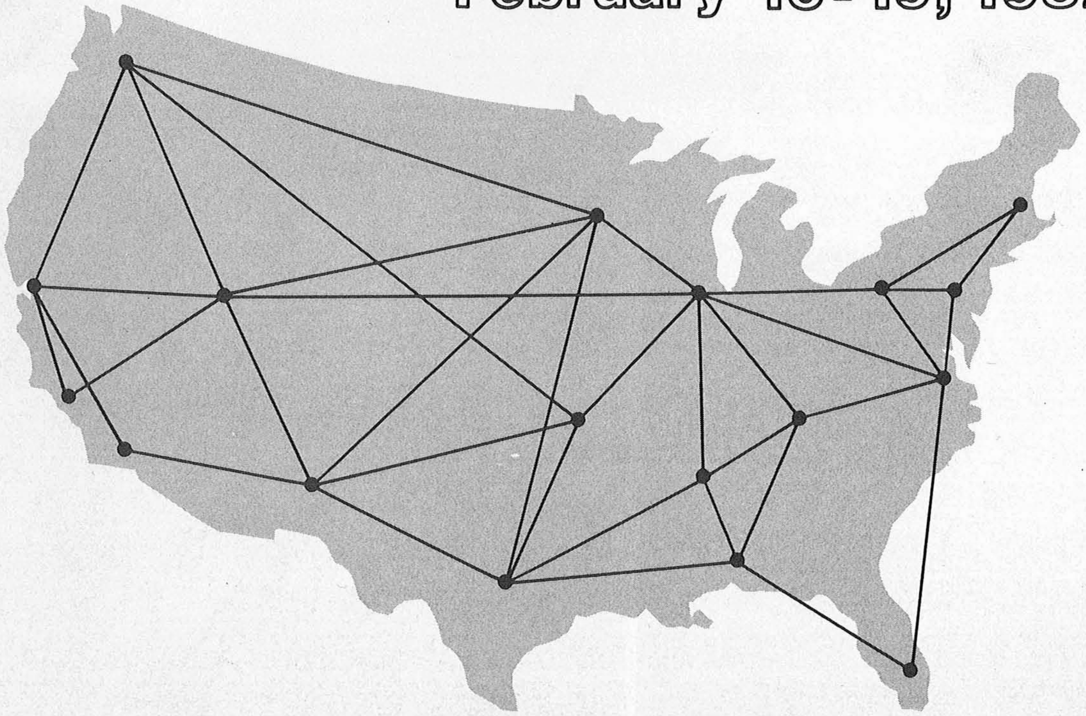
LBL-13452
UC-32
CONF-820201

RECEIVED
LAWRENCE
BERKELEY LABORATORY

FEB 22 1982

LIBRARY AND
DOCUMENTS SECTION

Asilomar
February 16-19, 1982



LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA, BERKELEY

For Reference

Not to be taken from this room

LBL-13452

PREPARED FOR THE U.S. DEPARTMENT OF ENERGY UNDER CONTRACT W-7405-ENG-48

LEGAL NOTICE

This book was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price Code: A17

Lawrence Berkeley Laboratory is an equal opportunity employer.

PROCEEDINGS OF THE SIXTH BERKELEY CONFERENCE ON DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORK

Sponsored by

Computer Science & Mathematics
Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Applied Mathematical Sciences
Research Program
Office of Energy Research
U.S. Department of Energy
Washington, D.C. 20585

General Chairperson:	Paula Hawthorn, Lawrence Berkeley Laboratory
Program Chairperson:	David Dewitt, University of Wisconsin-Madison
Program Committee:	D. Z. Badal, Naval Postgraduate School
	Kenneth Biba, Sytek Incorporated
	Greg Chesson, Bell Laboratories
	Lynn A. DeNoia, Bentley College
	Susan Eggers, Lawrence Berkeley Laboratory
	Frank Germano, Digital Equipment Corporation
	Peter Kreps, Lawrence Berkeley Laboratory
	Daniel Ries, Computer Corporation of America
	Lawrence A. Rowe, U.C., Berkeley
	Fred Schneider, Cornell University
	Robert H. Thomas, Bolt Beranek and Newman, Inc.
	Kevin Wilkinson, Bell Laboratories

ACKNOWLEDGMENT

We would like to thank Dr. Donald Austin of the Office of Energy Research, U.S. Department of Energy, for his continued support of the workshop, under Contract No. W-7405-ENG-48.

CONTENTS

GEMINI - A RELIABLE LOCAL NETWORK

ABSTRACT

Syntrex has developed a local network called Gemini for word-processing terminals. Gemini's unique redundant architecture ensures that no single failure will keep the system from operating. The development of a distributed message switched operating system allowed the network to be easily constructed and expanded. Users can migrate to the network environment without being forced to learn new operating procedures, since the word-processing software that runs in a stand-alone terminal is the same software that runs on Gemini.

Neil B. Cohen

Charles B. Haley

Scott E. Henderson

Chak L. Won

Syntrex Inc.
Industrial Way West,
Eatontown, N.J. 07724

GEMINI - A RELIABLE LOCAL NETWORK

1. INTRODUCTION

Syntrex has developed a local network called Gemini for word-processing terminals. The system is fully redundant, so that single failures don't force the users to stop work and wait for repairs. The network is designed to avoid problems of "perpetually open files" that can occur if a terminal fails before closing all its files. The development of a distributed message switched operating system allowed the network to be easily constructed and expanded.

Section 2 gives an overview of the architecture of the local network. Section 3 describes the operating system and the features that are used specifically for distributing the control. Section 4 describes the hardware and software architecture of the local network. Section 5 provides details of some of the interesting problems that were solved as the system was being built. Section 6 discusses potential plans for expanding the system.

2. SYNTREX NETWORK ARCHITECTURE

2.1. Background

One of Syntrex's major goals is to provide highly reliable, easy to use office automation equipment. The redundancy designed into Gemini provides reliability, since all the hardware is duplicated. The local network system is easy to use because the software that runs on a stand-alone word processor is the same software that runs on Gemini.

Our user community is extremely non-technical. Procedures such as backing up disks are foreign to normal office operation, and are often avoided or forgotten by users of automated systems. Gemini's automatic real-time backup alleviates this problem, protecting users against the loss of information and at the same time providing a measure of protection against lost work time due to a down system.

It should be noted that the system is **not** designed to keep a fully redundant system operating at all costs, or to connect Gemini units together to provide load sharing or further redundancy (a la Tandem[5,7]). Rather, the system is designed to continue operating without backup in the face of any **single failure** of hardware or software. This allows the customer to continue to do useful work until such time as a service person can get to the site to repair and restore the damaged half of the system.

2.2. The Terminal

The Aquarius is a stand-alone intelligent word processing terminal produced and marketed by Syntrex. It provides text editing and formatting capabilities, as well as access to printers, a spelling checker, and other features. Data and programs are stored on two 5 1/4 inch floppy diskettes, each of which can store up to 400 kilobytes of information. The Aquarius connects to several standard electronic typewriters, and fits on the typing extension of a secretary's desk.

2.3. Network Topology

Figure 2-1 shows a sample Gemini configuration. This star network can have up to 14 Aquarius workstations and printers attached. Gemini itself acts as a completely redundant file server, allowing users to share files and printers, send mail to one another, and have access to up to 600 megabytes of information.

Figure 2-2 shows how Gemini units can be clustered into a larger distributed system. This network has no central controller, and each part (individual Gemini) will continue to operate correctly if a neighboring Gemini is cut out of the system.

Figure 2-3 shows how Gemini clusters can communicate over a connection to a Public Data Network.

3. Network System Software

3.1. Operating System

The Syntrex Operating System (SOS) was developed for the Aquarius. SOS is a sophisticated message switched operating system. It was structured in such a way that the extension to a network environment required a minimal amount of changes. The same basic operating system that runs in the Aquarius also runs in the disk controller of Gemini. Applications programs (text editor, print programs, etc.) were moved to the network environment with no significant modifications. The remainder of this section will describe the operating system, with emphasis on the elements of the kernel that allow a networking environment to be easily created.

We assume that the reader is familiar with standard operating system concepts such as kernels, messages, processes, tasks, etc. These will not be defined in the text that follows. The process and interrupt structure of SOS is similar to that of Thoth[3], where system control relies upon teams of processes to perform specific functions. We have also relied heavily on the concept of a **link**[1,8], which is used for inter-process communication, and will be described in detail below.

Processes communicate by sending **messages** to one another. The messages are routed from process to process by the SOS kernel(s). A message consists of a message header, followed by an (optional) message body. The header contains information about the link on which the message was sent, an (optional) reply link (see below), the message type, and the length of the message body. Certain pre-defined message types exist for use with well-known processes (such as open a file, write a record, create a new process, etc.)

Messages are sent over **links**. A link is a capability for one-way communication over a channel between two processes. It is important to note that a link is unidirectional.

When the kernel is told to send a message on a link, it compares its own machine id with the machine id of the destination process. The Aquarius terminal is given a machine id at the time the operating system is started. In a stand-alone configuration, the machine id is set to zero. If the Aquarius is connected to Gemini, then the machine id is set by the Gemini during the startup phase of operation. If the destination process is located on the same machine, then the message is queued directly to the destination. If the machine id is different, then the kernel places the message on the queue to Gemini. The communication line protocol will transmit the message to Gemini, where it will be routed to its destination machine. The kernel in the destination machine then queues the message to the destination process.

When a process creates a new link, it is said to **own** the link. Ownership of a link may not be transferred. The owner of a link will receive any message sent on the link. Until given away, the owner process also **holds** the link. The holder of a link can send a message on the link. Subject to having the appropriate permissions, the holder of a link may give the link away to another process, thus allowing that process to send a message to the link owner. The holder of a link can not determine the identity of the owner. This helps to ensure that there are no hidden dependencies between processes which would make networking difficult.

Links have certain associated permissions, or attributes. These include the ability to duplicate a link, give away a link, destroy a link after sending a message on it (one time use), inform the owner when the link is destroyed or duplicated, etc.

When a message is sent to a process, provision is made for including a **reply link** in the message header. This is a link held (but not necessarily owned) by the original sender, which the receiver can now use to send a response message back to the link owner.

At the time a process is created, it is given a link to the Process Manager process. The process manager passes arguments and a link to the system file manager to the newly created process. This "well-known" link to the file manager is used to access all files in the file system. The following example (see also figure 3-1) shows how a file on the floppy disk is accessed:

1) The process sends an "open" request to the file manager on the well-known link (marked "a" in the diagrams). Included in the open request is a reply link (marked "b"). The reply link is coded for one-time use.

2) The file manager opens the file and sends a reply to that effect to the process. Included with the reply is a reply link (the "file" link - marked "c"). The file link is owned by the file manager, and has attributes such that the file manager will be informed when the link is duplicated or destroyed. All further requests for action on the newly opened file must be sent on the file link. If the file manager had failed to open the file (nonexistent file, file busy, etc.), an error message would have been returned to the user, and the file link would not be created.

3) The process sends read (or write) requests on the file link. Each request may contain a reply link for the file manager to use to send back an ack/nack. If the process does not wish to receive such a response, no reply link is sent with the read/write request, and the file manager can not make a response. If a reply link is specified by the process it is coded for one time use.

4) When the process is through with the file, it destroys the file link (i.e. closes the file). The file manager is notified of the destruction of the link, and then closes the file. The well-known link between the process and the file manager still exists, so that other files may be opened. Of course, several files may be opened simultaneously. Each open file has its own file link.

It should be obvious that the above mechanism can be extended to multiple machines in a straightforward manner. In particular, suppose that the file manager in a terminal has a "well-known" link to a file manager in another terminal (see figure 3-2). Then suppose an open request arrives on the well-known link from a local process. The file manager attempts to access the indicated file and fails. Now it is a simple matter to "forward" the message to the remote file manager, supplying as a reply link the original reply link provided by the user process. The remote file manager accesses the file (or fails to; it makes no difference), and sends its reply on the link provided by the original process and forwarded by the local file manager. The message is routed by the kernel(s) directly. The local file manager does not see the reply, and if the file is opened successfully the local file manager will not be involved in any further message forwarding. Furthermore, the program that asked to open the file remains unaware that the file has been opened on a remote machine. It is this fact that allows SOS to extend to a network environment with no changes in the application software.

4. GEMINI

4.1. Overview

Gemini was designed to meet a number of different goals. Primary among them were automatic information backup and minimal customer down-time.

Gemini is a completely redundant system, in which the user's information is continuously backed up on the **slave** half of the unit (see section 4.3.4 for a more complete description of master/slave concept in Gemini). When a failure occurs, half the system may be disabled, but the remainder can continue processing the user's information. Information about the failure (location of the system, the part that failed, recovery procedures, if known) is sent automatically via the phone network to Syntrex. In this manner, the customer can continue working until the service person arrives to fix the broken half of the system. During the time that half the system is broken, no further backup of the user information is done.

Secondary goals included an increase in available storage from 400 kilobytes on a single floppy disk, to as much as 600 megabytes on Gemini, as well as resource sharing, which allows users to conveniently share documents and data base information.

4.2. Hardware Description

The Gemini hardware consists of two major components, the Aquarius Interface (AI) and the Disk Controller (DC). Gemini is a redundant system. It contains two identical halves, each with its own AI and DC, as well as duplicated power supplies, battery backups and cables. Figure 4-1 is a block diagram showing the major hardware components of Gemini.

The AI performs all communications between Gemini and the Aquarii. The main features of the AI are:

1) Up to fourteen synchronous, serial, half duplex communication lines for the connection of Aquarii.

2) One 8 bit parallel, full duplex port for communication between the two AI's.

3) Shared memory for communication with the DC.

4) An 8088 CPU and local ROM and RAM storage.

The DC provides file storage and file management for the Aquarii. The main features of the DC are:

1) One to four disk drives, each with capacities ranging from 10 to 150 megabytes.

2) An RS232-C port for the connection of a modem and auto-dialer.

3) Shared memory for communication with the AI.

4) An 8086 CPU and local ROM and RAM storage.

4.3. Software Description

The software can be broken into two broad classifications: the disk controller software and the Aquarius Interface software.

The DC software will not be discussed here. It contains essentially the same kernel software that runs in a stand-alone Aquarius and manages files in the manner that was described in section 3.1. It is possible to extend the file manager to know about clusters of Gemini units, and to use a straight-forward routing algorithm to access files on different nodes of the network.

The AI software subsystems will be described in some detail in the sections that follow.

4.3.1. AI Software Architecture

The software in the AI consists of interrupt routines that handle the clock, communication hardware, interfaces to shared memory and the other AI. There are diagnostic routines that run at regular intervals as well as during failure conditions. Finally, there is a "scanner" routine that continuously looks for and processes incoming/outgoing traffic on the communication lines.

The scanner is implemented as a finite state machine. It examines each terminal in turn, checking for events that initiate state transitions. These events include the arrival of an HDLC frame from a terminal, a timer expiration, or a synchronization message from the other AI.

The AI software can be further subdivided into the following categories:

Interface to the Aquarius terminal

Interface to the disk controller

Communications between the two AI's

Self-testing procedures

4.3.2. Interface to Aquarius

Each Aquarius terminal is linked to the Gemini by means of a high speed (300+ Kbps) synchronous line. The line operates in half-duplex mode, using a subset of the ISO HDLC[6] link protocol. HDLC was chosen because it is simple to implement, and symmetric, so the same code could be run in both the terminal and the AI.

4.3.3. Interface to Disk Controller

The AI and the DC communicate by the exchange of message buffers in shared memory. Access to the various data structures in shared memory is synchronized by means of semaphores. The hardware provides a read-modify-write (or test and set) instruction, which allows either the AI or the DC to test a semaphore, and to lock it if possible. Shared memory can be viewed as a very high speed communications channel.

4.3.4. AI-AI Communications

Gemini is designed so that **the system will continue to operate normally** (as far as the user is concerned) **even though part of the system has completely failed.** To accomplish this, each independent half of the system must continually monitor the well-being of its partner. The heart of this monitoring is the AI-AI communications procedures.

When both halves of Gemini are operating normally one side is said to be "master" and the other side is the "slave." If a serious error occurs (disk breaks, memory parity errors, etc.) on one side, the bad side is powered off, and the good side continues to run. If the good side was the slave, it becomes master at the time of the switch-over.

The only difference in the work performed by the master and the slave is that the master actually transmits data to the Aquarius terminals, and the slave does not. Anything else done by the master is done by the slave. This includes reading data from the Aquarius and passing it on to the disk controller, receiving data from the disk controller and preparing it to be sent to the Aquarius, running on-line diagnostics, logging errors, etc. If an error occurs on the master and the slave takes over, the slave is in a position to continue talking to the terminals without interruption, or loss of information.

The AI's communicate with one another at different times for the following reasons:

1) Startup testing and synchronization

When Gemini is started, the AI's talk to each other to make sure the link between them is operating. If they fail to establish communications, then the slave side shuts itself off, and the master runs in simplex mode. It is possible for the slave to detect that the master is not operating (as opposed to the link between the AI's being broken), in which case the slave will become master and run in simplex mode.

2) Synchronization during processing of Aquarius information

Before a message from an Aquarius can be passed on to the disk, it must be correctly received by both AI's. It is possible, though unlikely, for a frame to come in correctly on one side and have a CRC error on the other. The AI's exchange information on the status of each incoming message to ensure that synchronization is maintained.

Similarly, before a frame is sent to the terminal, the AI's exchange information to ensure that the same type of frame is being sent by both sides. See section 5.1 for more detail on the AI-AI synchronization procedures.

3) Error detection and recovery

When an error occurs either the bad side informs the good side about the fault or it stops talking to the other AI altogether. In either case, the information is logged on the good disk controller, the bad side is powered down, and the system continues to run in simplex mode. The Syntrex Service Genie™ calls the nearest Syntrex service center to report the error. In the meantime, the customer can continue working without interruption.

As long as both sides are active, the two disks contain exactly the same information with respect to the user's documents. If the system breaks a service person will come out to repair it. Between the time of the failure and the arrival of the service person, the customer can continue working on the simplex system. During this time there is no further backup of information. Also, the information on the two disks is no longer the same; the broken disk is now out of date. When the Gemini is repaired after one side has broken, the information on the "broken" side must be brought up to date. A recovery program copies all the data from the "good" disk to the side that needs updating. Once the disks are identical, the customer can begin working on the system again.

4.4. Self Testing Procedures

While the system is active, it periodically runs tests on different parts of the hardware and software. These tests include auditing the message buffers, checking for the existence and accuracy of the clock, running memory tests, sanity checks on the DC, and testing the port between the AI's. If any test fails, the system logs an error, informs the other AI (if possible), and forces the other side to perform a switch to simplex mode. The good side then powers off the side with the error, and the Service Genie reports the error to the nearest service center.

5. SPECIAL PROBLEMS

A number of significant problems were encountered and solved during the course of building Gemini. Some of the more interesting ones are described in this section.

5.1. Synchronization

Synchronization of the master and slave presented a major challenge during the system development. In order to guarantee that no data was lost during any transaction, and to be sure that the slave was prepared to take control at any time, it was decided that all incoming and outgoing frames must be synchronized between the AI's prior to being processed. On input, this meant that the AI's exchanged information about the CRC of each incoming frame. If both sides received the frame the same way (either with a correct or an incorrect CRC), then the message was processed. If the AI's disagreed (one side had a CRC error while the other did not), the frame was processed as if a CRC error had occurred on both AI's. A duplicate data frame or a frame with a CRC error is discarded. Valid data frames are passed on to the disk controller. Processing in this manner ensures that both AI's are strictly synchronized on input from any one Aquarius.

No attempt was made to synchronize processing in the disk controller. Transactions processed by the DC will take varying amounts of time due to disk retries, physical defects on the disk, seek optimization processing, etc. As a result, when one disk controller sends a message to the AI to be passed on to an Aquarius there is no guarantee that the message is ready to be sent from the other AI.

We do not allow any frame to be sent to the terminal until both AI's agree to send the same frame. When one AI has a frame to send to the Aquarius, it sends a message to the other AI, describing the type of packet it wants to send. It then waits until a similar message is received from the other AI. If they are the same (in other words, if both sides want to send the same type of frame), then the master side transmits the packet to the Aquarius. The slave side listens for the master to complete the transmission. If the two sides don't agree on the type of frame to send, then a "lowest common denominator" frame is selected and sent. One common case where the two sides won't agree on the frame to send occurs when one AI wants to send a data packet that it has received from the disk controller, and the other wants to send only an acknowledgement (an HDLC RR frame), perhaps because a disk error is forcing a retry and the data has not yet been read. In this case, the lowest common denominator would be an RR frame. The next time around, if both sides have the data packet, it will be sent to the Aquarius. Another example occurs when one side wants to acknowledge a frame with an RR, and the other side wants to initiate flow control (an HDLC RNR frame). The lowest common denominator in this case is the RNR frame. There are safeguards in the system to avoid infinite loops (such as a disk controller that does continuous retries without realizing that it is broken).

5.2. Timeouts

When a timeout occurs in the standard implementation of HDLC, the side that times out retransmits its most recent message. In our implementation, if the Aquarius times out, it retransmits. If the Gemini times out, it places itself in receive mode, and waits for the Aquarius to time out and retransmit. This was done because the clocks on the two halves of Gemini do not run at exactly the same speed. We found that one side of the system would occasionally time out and try to start a re-transmission, while the other side was still expecting a message from the terminal. Rather than attempt to untangle the states of the two AI's, we decided that the Gemini would never retransmit a message. The major consequence of this was to lengthen the Aquarius timeout value, to make sure that both sides of Gemini time out before the terminal retransmits.

We have found that when the system is working normally there are very few re-transmissions necessary, so the longer timeout value does not impair the performance of the system.

5.3. Resource Identification

The dynamic allocation of global resources must be carefully managed in a redundant system. Otherwise, the independent halves of the system may create different names for the same resource. The result is then confusion if one is lucky, and disaster otherwise. This problem occurs in Gemini with respect to the allocation of links in the disk controller.

Links are identified by a serial number, which is assigned at the time the link is created. The following scenario describes the problem as it could occur:

1) The Aquarius sends a file open request to Gemini (see section 3.1 for a detailed example of the operations required to open a file).

2) Both the master and slave DC open the file, and both allocate a "file link", as described in section 3.1. However, since the DC's are not completely synchronized, they could assign **different** serial numbers to the file link.

3) The master AI sends the reply to the open back to the Aquarius. The file link belongs to the "master" side of the disk controller.

4) The Aquarius sends a read (or write) request on the file link. The message is passed through the AI to the disk controller. On the master side, there is no problem; the operation takes place as requested. However, on the slave, the link does not have the correct serial number, so the operation fails!

There are several ways to avoid this problem. One would be to require that the operation of the disk controllers be completely synchronized. However, this may adversely affect the performance of the system. Our approach is less restrictive. We set up a link translate table in the AI, and map all ambiguous link serial numbers to unique serial numbers as the links pass through the AI. Since the AI is synchronized on both input and output (on a per terminal basis), the required mapping function is easy to create and maintain, and the performance impact is minimal.

Mapping names to remove ambiguity is extensible from a single system to a network of systems. We expect to use a similar mapping scheme to handle the names of links in a cluster of Gemini units. The mapping may take place in a gateway station between Geminis, or it may occur in the Gemini unit itself.

5.4. Deadlock

Gemini has a limited amount of shared memory for use by the AI and the disk controller. Without careful management of this resource, several forms of deadly embrace are possible.

The first type of deadlock occurs when the disk needs a shared memory buffer to send something to a terminal, and all the buffers are currently filled with data being sent to the disk, or waiting for messages from the terminals. The system is effectively dead, since the disk will not process the next transaction until the current one has been put into shared memory, and shared memory can't be cleared because all the buffers contain information destined for the disk. This form of deadlock can be avoided by never allowing the AI to use the last free buffer in shared memory. In this way, the disk can always queue information to the terminal, and the AI and disk controller won't lock up.

This solution is not sufficient for Gemini. It solves the disk/AI deadlock, but not the master disk/slave disk deadlock that can also occur. This second form of deadlock is a generalization of the first case. Consider the following scenario:

1) The two disk controllers receive requests from terminals A and B that require information to be sent back to the terminal.

2) The master processes request A and sends it to the AI. This uses its last buffer in shared memory (which is reserved for use by the disk controller).

3) The slave processes request B and sends it to the AI, also using the last available free buffer.

4) The AI now attempts to send the two requests back to the respective terminals. However, both AI's must agree that they have the same data before it can be sent. Unfortunately, this is not the case. There are no free buffers in the master to complete terminal B's request, and no free buffers in the slave to complete terminal A's request, so the system is deadlocked. Gemini would eventually solve this problem by shutting off the slave half of the system, assuming that the slave disk is bad and is unable to read the requested data.

The problem can be avoided by extending the number of buffers that are reserved for use by the disk controller. In general, (number of terminals/2) + 1 buffers are required to stay out of a deadlock situation. This guarantees that regardless of the order in which transactions are processed by the two disk controllers, there will be enough shared memory to ensure that data from the disk controller will be sent back to the terminal.

6. CONCLUSIONS

Gemini is a real, commercially available local network. It employs state-of-the-art hardware, and several interesting software protocols (Aquarius-AI, AI-AI, and AI-DC).

The Gemini system is "Always Up™." Its unique redundant architecture is designed to ensure that no single failure will stop the system. The design of the operating system allows application software to make full use of the network environment without modification.

It is possible to connect several Gemini units together to form a larger network (see figures 2-2 and 2-3). In both cases, the file manager in the disk controller would have to be aware of the existence of other Gemini units, and would have to be able to forward messages to the file managers in those units (via well-known links, as described in section 3.1). Provision must be made for avoiding infinite loops in the network, as messages are forwarded from one file manager to another. For an Ethernet™[4]1 connection, very little would have to change in the Gemini, aside from the level 2 software required to talk to the Ethernet itself. In the case of a Public Data Network, an X.25 level 3 interface would be required to get information to and from the Public Network.

Studies are underway to analyze the performance of the system. Final results are not yet available.

7. REFERENCES

[1] Basset F., Howard J., Montague J., **Task Communication in Demos**, Proceedings of the Sixth Sigops, November, 1977, pp. 23-31.

[2] **CCITT Recommendation X.25**, CCITT Gray Book, Geneva Switz., ITU 1979.

[3] Cheriton et. al., **Thoth, A Portable Real-Time Operating System**, University of Waterloo, Department of Computer Science Report CS-77-11, Oct. 1977.

[4] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, **The Ethernet, A Local Area Network**, Version 1.0, September 30, 1980.

[5] Highleyman, W., **Survivable Systems**, Computerworld - In Depth, Computerworld, 1980.

[6] International Standards Organization, **High Level Data Link Control Proposal** Doc. No. 1005-ISO TC97/SC6.

[7] **Non-Stop™2 Systems - Tandem 16 Introduction**, Tandem Computers.

[8] Solomon M., Finkel R., **The ROSCOE Distributed Operating System**, Proceedings of the Seventh Sigops Principles, December, 1979, pp. 108-114.

1Ethernet is a trademark of Xerox Corporation.

2 Non-Stop is a trademark of Tandem Computer Incorporated.

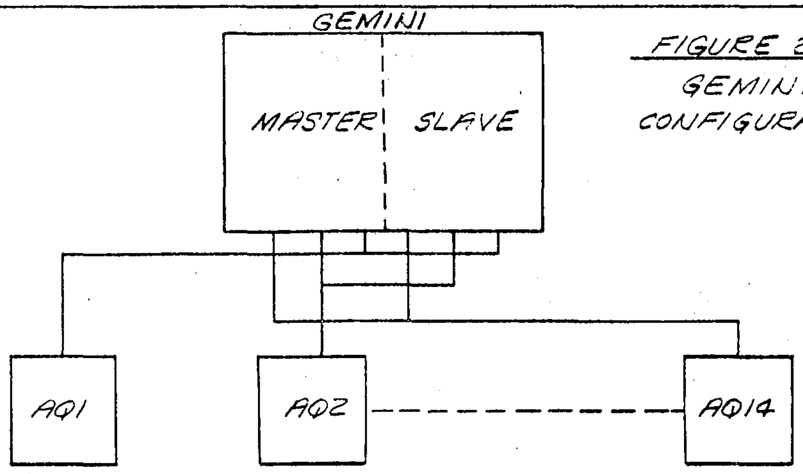


FIGURE 2-1
GEMINI CONFIGURATION

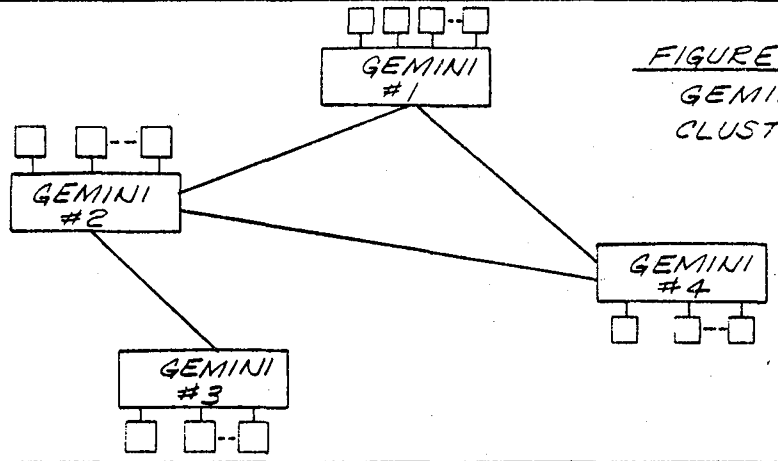


FIGURE 2-2
GEMINI CLUSTER

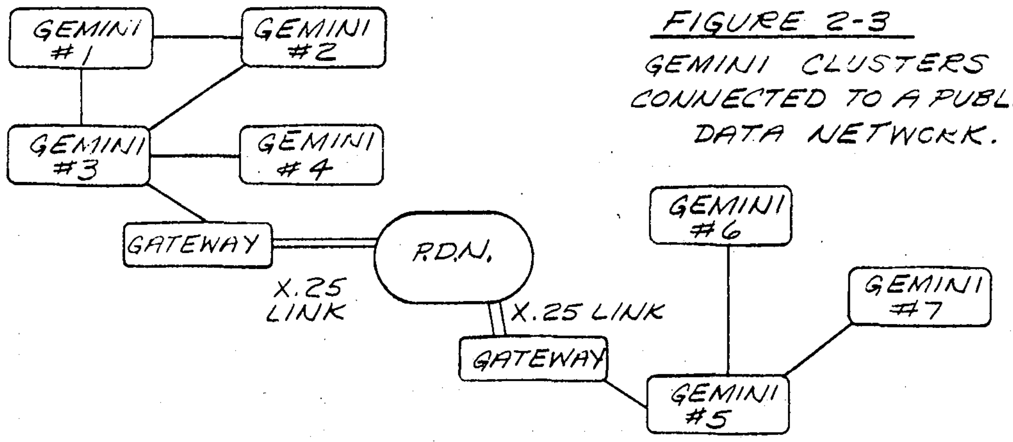
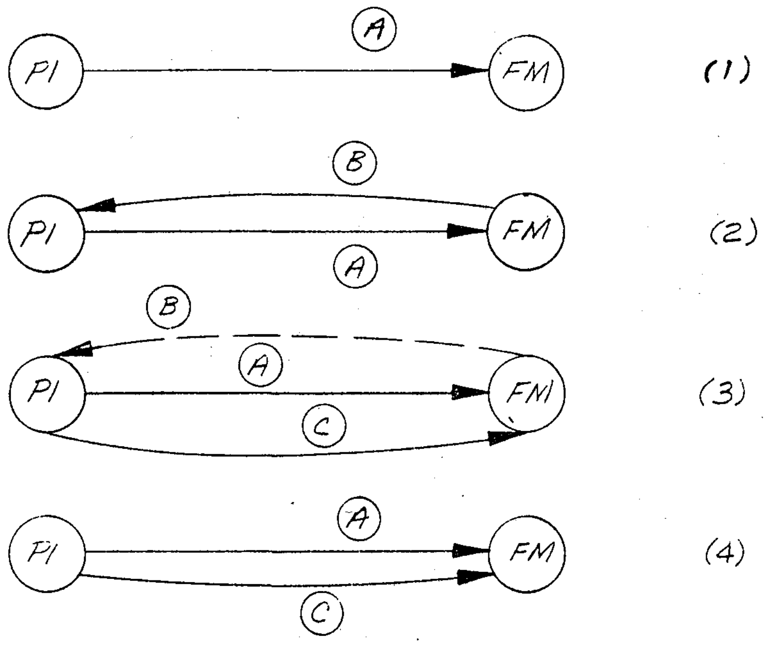


FIGURE 2-3
GEMINI CLUSTERS CONNECTED TO A PUBLIC DATA NETWORK.



P1 - USER PROCESS
FM - FILE MANAGER
Ⓧ→Ⓨ - ACTIVE LINK - HELD BY Ⓧ, OWNED BY Ⓨ
- - -→ - DESTROYED LINK

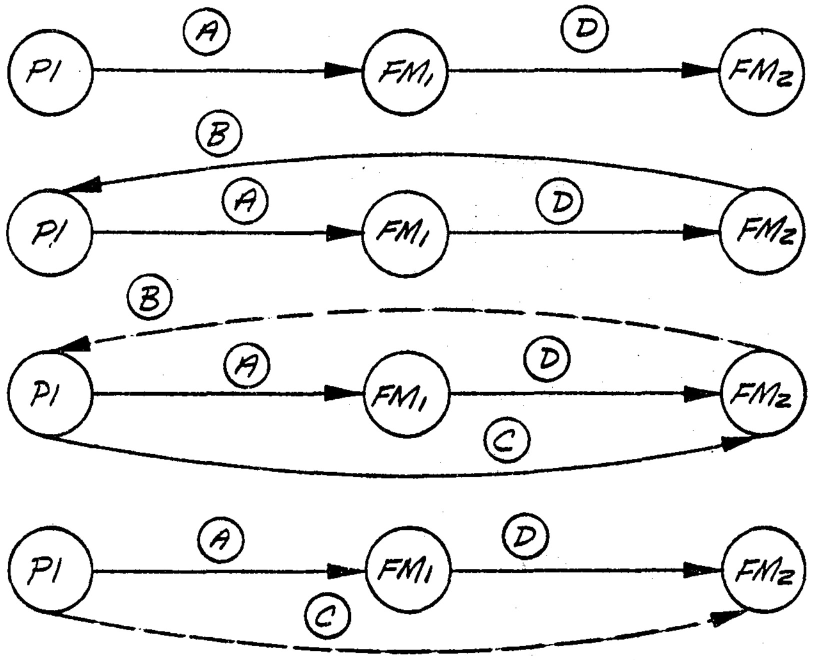
Ⓐ 'WELL-KNOWN' LINK

Ⓑ REPLY TO OPEN REQUEST (ONE TIME USE)

Ⓒ 'FILE LINK'

FIGURE 3-1

ACCESSING FILE USING LINKS



P1 - USER PROCESS
FM1 - LOCAL FILE MANAGER
FM2 - REMOTE FILE MANAGER

$\text{Ⓧ} \rightarrow \text{Ⓨ}$ - ACTIVE LINK - HELD BY Ⓧ, OWNED BY Ⓨ

$-\,-\,-\rightarrow$ - DESTROYED LINK

Ⓐ - 'WELL KNOWN' LINK

Ⓑ - REPLY TO OPEN REQUEST (ONE TIME USE)

Ⓒ - 'FILE LINK'

Ⓓ - 'WELL KNOWN' LINK BETWEEN FILE MANAGERS.

FIGURE 3-2
ACCESSING REMOTE FILES USING LINKS

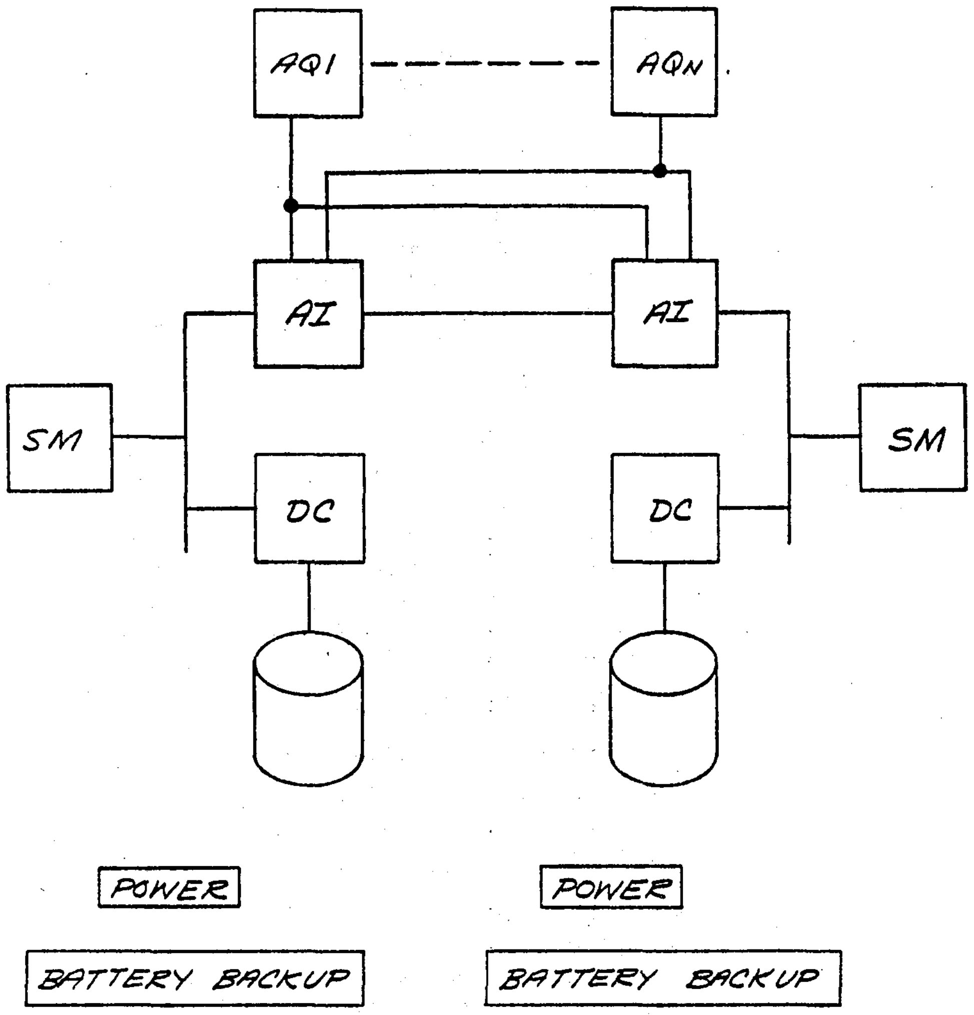


FIGURE 4-1
GEMINI HARDWARE COMPONENTS

The Resiliency of Fully Replicated Distributed Databases

Wing Kai Cheng*
ROLM Corporation
4900 Old Ironsides Dr.
Santa Clara, CA 95050

Geneva G. Belford
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

One of the well publicized advantages of a distributed database system is its availability. The increased availability is obtained in part by keeping multiple copies of data. In a fully redundant distributed database system, in which a copy of the data is kept at each node, the probability that a read transaction can be successfully processed is $(1-(1-a)^N)p_r$, where a is the probability that a node is available or "up" and p_r is the probability that the node remains up until the read transaciton is processed. In the case of multiple-copy distributed databases, the read resiliency is obviously better than the read resiliency of ap_r of a single-copy database. However, the update resiliency of a multiple-copy database is not necessarily better than a single-copy database because many of the update synchronization algorithms require that all nodes be "up" when an update is being processed. The resiliency of the system to process an update request depends on the update synchronization algorithm used since not all algorithms require the same degree of node availability. This paper presents an analysis of the availability of distributed database systems in relation to their update synchronization mechanisms and node availability.

*Part of the research was conducted when the author was with the University of Illinois.

1 Introduction

Reliability of a system is a measure of the success with which the system conforms to some authoritative specification of its behavior. Availability of a system, on the other hand, refers to the fraction of time that the system satisfies its specification. Reliability and availability are two of the most commonly noted advantages of distributed databases.

For a fully replicated database, the probability that a read request can be satisfied is $(1-(1-a)^N)p_r$, if a is the probability that a site is available at any given time and p_r is the probability that the read-site remains up until the read transaction is processed. The probability that an update can successfully be committed, however, is dependent on the update strategy and the system configuration. For example, an update strategy may require that all sites be available when an update request is executed, some require that only a majority of the nodes be available; furthermore, the length of time that the nodes have to be "up" differs among the update strategies. This paper will analyze the issues of availability of DDB systems in the light of update strategy and resiliency techniques. We are interested in exploring the probability that node failure will not cause the update to be aborted or blocked. In other words, we are interested in how different synchronization techniques and the availability of nodes may affect the probability that an update can be completed. We define read resiliency as the probability that a read request will not be rejected due to the unavailability of the nodes that have the data items. Read resiliency is equal to the probability that the nodes which are needed to process a read transaction are available ("up") when the transaction arrives multiplied by the probability that each node remains up until it completed its task as specified by the synchronization algorithm. We define update resiliency as the probability that the update can be committed; that is, the probability that the update will not be aborted due to the failure of nodes. Update resiliency is the probability that the nodes which are needed to execute the synchronization algorithm are "up" when the update transaction arrives multiplied by the probability that each node remains "up" until its task as specified by the algorithm is completed. Taking into consideration read resiliency and update resiliency, the resiliency of a distributed database system can be expressed as

$$(\text{fraction of updates})*R_u + (\text{fraction of read-only})*R_r$$

where (fraction of updates) + (fraction of read-only) = 1

R_u is the resiliency of the system for update processing, and
R_r is the resiliency of the system for read-only transaction processing.

Throughout this paper, we will assume that the database is fully replicated; namely, a copy of the database is stored at each node. In the following section, the concurrency problem and resiliency problem will be reviewed. We will then present some models that relate update resiliency and read resiliency to node availability and the amount of processing time required from each node. A comparison of the models can be found in Section 4.

2 Concurrency Control

The primary goal of concurrency control algorithms is the preservation of consistency of data. The problems that arise when multiple users access a shared database are of the following types: (1) If transaction T1 is reading a portion of the database while transaction T2 is updating it, T1 might read inconsistent or obsolete data. (2) If transactions T3 and T4 are both updating the database, race conditions can produce erroneous results. The objective of concurrency control is essentially to control the sequencing of user-specified operations so as to preserve the illusion that each transaction is a simple, complete, atomic action. The issue is to ensure serializability (Bernstein et al., 80) (Bernstein et al., 79); that is, even if the transactions are running concurrently and in an interleaved manner, the overall effect must be the same as if the transactions were run in some sequential order.

In a distributed database system, the availability of data is increased by keeping multiple copies of data. The presence of multiple-copies of data, the possibility of failure, the variance in transmission delay, and the fact that a site cannot know instantaneously the activities at the other sites have all contributed to complicating the updating process. Several algorithms have been proposed for updating replicated databases. The majority of them use either the techniques of locking or timestamping.

Algorithms using locking can have either centralized control (Bernstein and Goodman, 80b) (Bunch, 75) (Garcia-Molina, 79) (Cheng and Belford, 80b), or distributed control (Bernstein and Goodman, 80b) (Ellis, 77). Consistency of data is preserved by locking the data before access. The theories behind locking as a means to control concurrent access are fairly well developed (Eswaran et al., 76).

For the algorithms using timestamps, consistency is preserved because timestamps induce a unique execution order for the transactions. In a decentralized system, a timestamp typically consists of a pair (c,s) generated locally at each node, where c is the physical clock time (or logical clock time--i.e., counter value) and s is the unique identification of the node where the update request originated. A timestamp T1=(c1,s1) is said to precede (or be older than) T2=(c2,s2) if either (c1<c2) or (c1=c2 and s1<s2). Two general approaches have been used to update distributed databases using timestamps. One of them is to allow updates to be applied only when it is sure that the read-set of the transaction in consideration is up-to-date and is not made obsolete by another update at this or other nodes. The other approach is to apply the updates and later undo the updates when conflict is detected. In a distributed environment where failure may occur, or messages may be delayed, it is not easy for a site to know when it is "safe" to apply an update. Examples of synchronization algorithms that use timestamping technique include the Majority Consensus Algorithm (Thomas, 79), the SDD-1 algorithm (Bernstein, 78), and others (Cheng and Belford, 80a,c) (Bernstein and Goodman, 80a) (Kaneko et al., 79). As we will see in the next section, different concurrency control schemes require different degrees of node availability.

To guard against the failures of nodes and communication networks (such as partitioning) from disrupting the consistency of data, some additional restrictions have to be imposed. To insure that an update is applied to every copy (or atomicity of update), some algorithms require that an update be committed only if each node that has a copy is able to update its copy. Preserving transaction atomicity in the single site case is a well understood problem (Gray, 79). Basically, at some time during its execution, a commit point is reached where the site decides to commit or to abort the transaction. A commit is an unconditional guarantee to execute the transaction to completion, even in the event of failures. An abort is an unconditional guarantee to back out the transaction. The problem of guaranteeing transaction atomicity in a distributed system is that of insuring that all the sites either unanimously abort or unanimously commit. This is accomplished by using multi-phase update protocols. With the use of persistent communication (i.e., storing update messages and broadcasting to the failed nodes at a later time), the number of nodes that have to be "up" can be relaxed a little.

Communication failure can partition a network into a number of disjoint sub-networks that are unable to communicate with each other. Some algorithms only allow the majority partition to perform updates.

If one allows every group of sites in a partitioned network to perform new updates, the databases of the groups will diverge. When the groups are to be re-united, some updates will have to be undone. Coordinating the undoing of updates is a very difficult task. Update activities must therefore be restricted in order to facilitate merging of the updates when the distributed system is recovering from having been partitioned. The strategy is usually to allow only one of the sub-networks to run update transactions. For example, one may restrict updating to the partition with a majority of nodes in it. In some situations, different weighting factors assigned to each site or to each copy of data and used to compute a weighted majority may be more appropriate. Application-specific knowledge may be used to help determine weights for each node. For an inventory-like database, where the operations are commutative, one can perhaps apportion a percentage of updates to each site during partitioning (Hammer and Shipman, 79). For a system where the primary copies of different parts of the database are at different sites (e.g. as in distributed INGRES (Stonebraker, 79)), one may adopt the policy of allowing an update to be executed only if the primary copies of all the data items in the read-set and write-set are in the partition. As one may expect, partitioning considerations also affect the degree of node availability required when applying updates.

3 Update Synchronization Techniques

Numerous algorithms for consistently updating distributed databases have been proposed; in this paper, we consider examples of the following types:

1) Locking all copies.

2) Centralized Locking

3) Centralized locking with backup nodes,

4) Linear Majority Voting with Timestamps, and

5) Broadcast Majority with Timestamps.

Before our discussion of these algorithms, we shall briefly review the architecture of the distributed database system. We assume that the system consists of N nodes, each node with the same hardware availability. The mean time between failure is given by the parameter MTBF and the mean time to repair is MTTR. The availability of a node is

$a = MTBF/(MTBF+MTTR)$

Guaranteed delivery of messages is assumed; that is, messages sent from A to B will eventually be received by B.

In the following sections, we shall focus our attention on update resiliency. The probability of no blocking for read-only transactions is assumed to be the same for all of the concurrency control algorithms discussed here. Given that the database is fully replicated, the resiliency R_r for read processing is $(1-(1-a)^N)p_r$ since the data can be obtained from any of the nodes. The term $(1-(1-a)^N)$ represents the probability that at least one of the nodes is up when the transaction arrives and p_r represents the probability that the node remains up until the read is completed. We assume the probability that a site would fail in the next r seconds is Poisson distributed; the probability that it would not fail is p_r, where

$$p_r = e^{-r/MTBF} \tag{1}$$

The parameter r represents, for example, the amount of time to obtain shared locks and read the items at the read-site.

3.1 Locking All Copies

A simple distributed locking algorithm may require that all copies be successfully locked before data are updated. That is, messages are broadcast to lock and update the items in a tentative mode; if these can be done at all nodes, the update is finalized. An example of such an algorithm might work as follows:

Algorithm LAC1

1) Update transaction T arrives at site q.

2) Issue LOCK_REQUEST for the read-set to the lock manager at q.

3) When LOCK_REQUEST is granted, read the read-set and compute the update.

4) Broadcast INTEND_TO_UPDATE messages (including the write-set and new values) to every node.

5) Each node stores the message on stable storage, obtains the locks and performs the updates in tentative mode, and then sends an AGREE_INTEND_TO_UPDATE message to the controlling

module (at site q).

6) At site q,

   a) when AGREE_INTEND_TO_UPDATE is received from each node, send COMMIT and RELEASE_LOCK messages to every node; otherwise,

   b) if AGREE_INTEND_TO_UPDATE is not received from some of the nodes before the timeout, ABORT is sent to every node to abort the update and release the locks.

The above protocol requires that all nodes be available to receive and store the INTEND_TO_UPDATE message, and to send an acknowledgment back to the controlling process. After this point, failure of any of the nodes (other than the controlling node) will not delay the progress of the update. Assuming guaranteed delivery of messages to the failed node, we can show that the database will be consistent upon recovery.

This model requires that all N nodes in the system be up when the update transaction arrives; i.e.,

$$A = a^N$$

With this model, if any of the nodes fails before it finishes its tasks (Step 5 for cohorts and Steps 1 to 6 for controlling module) for the pending update, the update will be blocked or aborted. We can compute the probability that an update will be blocked. First assume that the occurrence of a site failure is a Poisson process. Essentially, this means that we assume that the probability that a failure occurs in any time interval is proportional to the length of the interval, with constant of proportionality the failure rate, or 1/MTBF. This assumption, which seems reasonable in the absence of detailed information on the failure behavior of specific machines, leads to the well known exponential distribution for time intervals between failures. That is, the probability that a node i will not fail within g seconds of any initial time 0 is given by

$$p_i=e^{-g_i/MTBF} \tag{2}$$

If g_i is the response time of a node to a task requested by some transaction, p_i then represents the probability that the task will be completed. The parameter g_i can be decomposed into the term W_i, which represents the mean wait time, and the term X_i, which represents the mean processing time; that is,

$$g_i = W_i + X_i$$

(3)

X_i denotes the processing at site i; the amount of processing is mainly dependent on whether i is the site where update originated. For the site of origin, g_i is essentially the time needed to complete the update. For the cohorts, g_i is the duration from the time the transaction arrives until the COMMIT message is processed. The resiliency of the system for update processing is the probability that all N nodes are up when the transaction arrives multiplied by the probability that each of the N nodes remains up until it finishes its respective task:

$$R_u = Ap_o p^{N-1} = a^N p_o p^{N-1}$$

p_o is obtained from Equation 2 for the site of origin and p is computed using Equation 2 for the N-1 cohorts, assuming that they all behave identically.

The resiliency of a system for update processing when this kind of synchronization mechanism is used is lower than that of a centralized database system (where the complete database is located at one central site). The resiliency of a centralized database system for processing updates is ap_o. If the number of update requests is greater than the number of read requests, the resiliency of the distributed database system is no longer better than that of a centralized database system. Higher update resiliency can be achieved with some modifications to Algorithm LAC1; we shall refer to the more resilient algorithm as LAC2. Algorithm LAC2 improves update resiliency by allowing a partition to apply updates when at least a majority of the nodes are "up." In Algorithm LAC2, it is necessary to check whether at least $\lceil (N+1)/2 \rceil$ nodes are up and record these nodes in the list_of_available_nodes before Step 2. In Step 6 of the algorithm, AGREE_INTEND_TO_UPDATE messages must be received from every nodes in the list_of_available_nodes before an update is allowed to commit. The update resiliency of Algorithm LAC2 should be quite similar to the algorithm in (Cheng and Belford, 80a). In brief, the update resiliency of LAC2 is

$$R_u = \sum_{\lceil \frac{N+1}{2} \rceil}^{N} \binom{N}{i} a^i (1-a)^{N-i} p_o p^{i-1}$$

The update protocols in this section consist essentially of two phases (Gray, 79) (Lampson and Sturgis, 79) Algorithms with three (Skeen, 81), four (Hammer and Shipman, 79), or more phases have also been proposed.

3.2 Centralized Locking

The centralized locking algorithm is appealing because of its conceptual simplicity. Consider a network of N nodes: a central and N-1 secondary nodes. Unlike the secondary nodes, the central has the additional responsibility of preserving the consistency of data and resolving conflict. Basically, locks are requested at the central before an item is read or updated. The steps that an update transaction has to go through may look like:

Algorithm CLA1

1) Update transaction is issued at a node q.

2) Lock requests are sent to the central.

3) If the locks could be granted, a GRANT message is forwarded to q.

4) At q, the read-set is read and the update is computed. COMMIT_UPDATE request is sent to each node to update the local write-set.

5) Each site applies the update and sends ACK to q.

6) When all the ACKs are received by q, it sends a RELEASE_LOCK message to the central.

This model requires that the central and the N-1 secondary nodes must be up when the transaction arrives. The update resiliency of the above algorithm is thus given by

$$R_u = a^N p_o p_c p^{N-2} \tag{4}$$

where all p, p_c, and p_o have the form

$$p = e^{-g/MTBF} \tag{5}$$

We use the subscript o and c to denote the site of origin and the central respectively; parameters (p and g) without subscripts refer to those of the cohorts. Although the equations for update resiliency of CLA1 and LAC1 have the same form, they are not the same because the amount of time that each node has to be "up" are different. Since g, g_c, and g_o in the centralized algorithm CLA1 are less than their counterparts (g and g_o) in the Lock All Copies Algorithm, p and p_o are larger for CLA1, resulting in slightly higher R_u for CLA1 when compared to LAC1. A more resilient algorithm is to omit the ACK and use sequence numbers like the algorithm below:

Algorithm CLA2

1) Update transaction is issued at a node q.

2) Lock requests are sent to the central.

3) If the locks could be granted, a GRANT message and a sequence number are forwarded to q.

4) Site q waits for requests with lower sequence numbers to be completed at q. The read-set is read and the update is computed.

5) COMMIT-UPDATE request and the sequence number are sent to each node to update the local write-set. A RELEASE_LOCK request is sent to the central.

6) Each site applies the update according to the sequence number.

The condition for the system to be available to process updates is that the central and the site where the update originated must be available for a certain period of time. The update resiliency is

$$R_u=a(1-(1-a)^{N-1})e^{-g_c/MTBF}\ e^{-g_o/MTBF} \tag{6}$$

where g_c is the time needed for the central to grant the locks (Step 3) and g_o is the length of time the site of origin spends on the update request. (This assumes that other sites will eventually get the update message and process the update, even if they fail during this intial processing.) The factor $a(1-(1-a)^{N-1})$ represents the probability that the central and one of the secondary nodes to which the transaction is to be submitted are up upon the arrival of the transaction. If the transaction can be submitted to the central, it is not necessary for a secondary node to be up before processing of the transaction can begin; the resiliency is simply

$$R_u=ae^{-g_c/MTBF}$$

In case of a partitioning, only the partition with the central is allowed to process update transactions. There is only one partition that contains the central node; thus, only one partition will perform updates.

Proposed algorithms that belong to this category include those of Garcia-Molina (Garcia-Molina, 79) and Bunch (Bunch, 75).

3.3 Centralized Locking with One Backup

The centralized locking algorithm, although quite appealing for its conceptual simplicity, does not provide high resiliency. All updating has to be halted when the central fails. Researchers in distributed databases have suggested improving the resiliency of the centralized locking algorithm by keeping backups for the central (Alsberg et al., 76). Alsberg and co-workers have also shown the sufficiency of one backup for most applications.

The centralized locking algorithm with one backup is assumed to consist of a central node, which takes care of locking and resolving conflicts, a backup, and N-2 secondary nodes. Essentially, the backup is kept informed of the state of the central and, when the central fails, the backup assumes the role of the central, and a new backup is elected from the secondary nodes. When the backup fails, but not the central, a new backup is likewise elected from the secondary nodes. All failed nodes will be repaired and restored as secondary nodes. If partitioning occurs, we assume that only the partition with the central can make updates.

The centralized locking algorithm with one backup (CLOB) would work as follows:

Algorithm CLOB1

1) An update transaction T arrives at a node q.

2) Node q requests from the central node locks for all the items referenced by the update.

3) As soon as the central receives the lock request (from Step 2), it transmits a LOCK_REQUEST_COOP message to the backup to request cooperation. The message includes a list of the items to be locked.

4) When the backup receives the LOCK_REQUEST_COOP, it stores the message on stable storage (Lampson and Sturgis, 79).

5) When the central decides that the lock request can be granted, it assigns a sequence number to the request and sends a GRANT_COOP message along with the sequence number to the backup. The central marks the items in the database as locked.

6) In response to the GRANT_COOP message, the backup will send a GRANT message and the sequence number to q, the node where the lock request originated.

7) Once node q gets the GRANT message, the items are read from the local database, and the update is computed.

8) After computing the update, q transmits an UPDATE message to the cohort. The message contains update information, such as the sequence number, names of items, and the new values.

9) Site q sends a message to release the locks at the central.

10) The central transmits RELEASE_LOCK_COOP to the backup and releases the locks.

Based on the centralized locking algorithm with one backup described above, an update transaction is blocked when one of the following sequences of events occur before the update is completed:

1) The central c fails. The backup b is promoted to be the new central c'. A new backup b' is elected, but the new central c' fails before the new backup b' is fully installed.

2) The backup b fails, but not the central c. A new backup b' has to be installed, but the central c fails before the new backup b' is fully installed.

The probability of update transaction not being able to be committed due to Cases (1) and (2) is equivalent to the probability that the failures of the central and the backup occur within k seconds of each other, where k is the mean time to install a new backup. In Case (1), the old backup fails within k seconds after the old central failed. In Case (2), the central fails within k seconds after the old backup failed.

The probability that a site will fail within the next t seconds is assumed to be Poisson distributed; namely,

$$1-e^{-t/MTBF}$$

or approximately t/MTBF.

The probability that update processing will be blocked due to Cases (1), and (2) is therefore approximately

$$q_2=(1-e^{-g_c/MTBF})(1-e^{-k/MTBF}) + (1-e^{-g_b/MTBF})(1-e^{-k/MTBF}) \tag{7}$$

where MTBF is the mean time between failure of a node, k is the mean time to install a backup, g_c is the duration for which the central has to be up to process the update, and g_b is the duration the backup has to be up. The first term represents the probability that the central fails before the update is completed, the backup becomes the new central, but it too fails before a new backup is installed. The second term represents the probability that the backup fails and the central subsequently fails before a new backup can be installed.

The resiliency of the system to process updates is then equal to

$$R_u= a^2(1-(1-a)^{N-2})(1-q_2)p_o,$$

where p_o is the probability that the site of origin remains available until COMMIT messages are sent to all cohorts:

$$p_o=e^{-g_o/MTBF} \tag{8}$$

The factor $a^2(1-(1-a)^{N-2})$ represent the probability that the central, the backup, and one of the secondary nodes to which the transaction is to be submitted are up when the update transaction arrives. The factor $(1-(1-a)^{N-2})$ can be removed if the transaction is submitted to the central or backup.

The resiliency can obviously be increased if the system is allowed to start processing update whenever the central is up (even if the backup is not); we refer to this more resilient algorithm as Algorithm CLOB2. The resiliency of CLOB2 can be approximated as follows: If the central and backup are available when the transaction arrives, the resiliency is given by Equation 7. When the central alone is available, the central must survive until the update is completed or a backup is installed. The probability that the central fails before the update is completed and before a backup is installed is approximately

$$q_1=(1-e^{-g_o/MTBF})(1-e^{-k/MTBF})$$

The update resiliency when the central alone is up is

$$(1-(1-a)^{N-2})a(1-a)(1-q_1)p_o \qquad (9)$$

In the case when the backup alone is up, the backup immediately becomes the central, and the resiliency is similar to that given by Equation 8. The update resiliency of CLOB2 is therefore the sum of Equations 7 and 9:

$$R_u=a^2(1-(1-a)^{N-2})(1-q_2)p_o + 2(1-(1-a)^{N-2})a(1-a)(1-q_1)p_o$$

3.4 Linear Majority Voting with Timestamps

The daisy chain model of Thomas' Majority Consensus Algorithm (Thomas, 79) is an example of a Linear Majority Algorithm. In the database of each node, each data item has a timestamp that reflects the time the item was assigned its current value. Updates are accepted only if the read-set used in computing the updates have not been made obsolete by another transaction. To determine whether the read-set is up to date, a vote request is passed from node to node to give each node a chance to decide.

In the daisy chain model of the Majority Consensus Algorithm (Thomas, 79) (in which the vote request is passed from one node to another, instead of broadcast), after the first node has voted, at least one of the remaining N-1 nodes must be available to receive the update request. The probability that all N-1 nodes fail is $(1-a)^{N-1}$; the probability that at least one of the N-1 is available is $1-(1-a)^{N-1}$. After the second node has voted, at least one of the remaining N-2 nodes must be available to receive the update request and vote on it. After the third node has voted, at least one of the remaining N-3 nodes must be available to receive the update request and vote on it, and so forth. The resiliency of the system for processing updates is therefore

$$(1-(1-a)^{N})p_o(1-(1-a)^{N-1})p(1-(1-a)^{N-2})p\ldots(1-(1-a)^{N-m+1})p$$

where m is the mean number of nodes that have to vote before a consensus is reached. p and p_o are given by

$$p = e^{-g/MTBF}$$

and

$$p_o = e^{-g_o/MTBF}$$

For p_o, the probability that the node of origin doesn't fail, g_o includes the time to read base-set, compute update, vote, and transmit the vote message to another available site. For p, the probability that one of the other nodes doesn't fail, g includes the time to receive the message, read timestamps and vote, and to transmit the vote message to another available site.

Other linear algorithms generally have lower resiliency, especially those in which the communication medium is a ring and all the sites must participate in the synchronization in a pre-determined order.

3.5 Broadcast Majority with Timestamps

One of the ways to facilitate recovery of a distributed database from failure due to partitioning is to allow only one of the partitions to process updates. This can be done by allowing only the partition with the majority of nodes to process updates, as in the distributed algorithm DM1 in (Cheng and Belford, 80a). The update resiliency (for algorithms requiring a majority such as DM1 (Cheng and Belford, 80a) and LAC2 referred to earlier) is the probability that an arriving transaction finds that a majority of nodes are "up" and that each of these survive through their critical period of duty.

$$\sum_{\lceil \frac{N+1}{2} \rceil}^{N} \binom{N}{i} a^{i}(1-a)^{N-i} \; p_o p^{i-1},$$

p and p_o again have the form

$$p = e^{-g/MTBF} \tag{10}$$

With this type of algorithm, before the update transaction T is processed, the concurrency control algorithm makes sure that a majority of nodes are available and records the names of these nodes in the list_of_available_nodes. If the update transaction T does not conflict with any of the pending transactions originated from any of the nodes in the list_of_available_nodes, T will be committed.

An algorithm that uses a weighted majority has been proposed by Gifford (Gifford, 79). With this weighted voting algorithm, every transaction has to collect a read quorum of R votes to read the database, and a write quorum of W votes to write.

4 A Comparison of Resiliency

Queueing models have been developed for the concurrency control models described in this paper in an effort to quantitatively compare the resiliency of the different models. The analytic models are derived in the following manner: The I/O service time requested from the I/O server depends on the type of request. The I/O server is modelled by an n-stage parallel server. By inspecting the synchronization algorithms, one can determine the moment of the service time for each, and their relative request frequencies. From this information on I/O requests, the first and second moments of the overall I/O service time can be computed. Knowing these moments, the mean wait time can be computed using the well-known mean-wait-time equation for M/G/1 queues. To compute g, g_o, g_c, etc., we simply determine the task each node has to perform for the update before the update can be committed and add up the delay (response time) incur at each step of the task. (Note that we assume that the network consists of a set of independent queues.) These g's are then used in the respective resiliency equation to obtain the resiliency of the concurrency control algorithm.

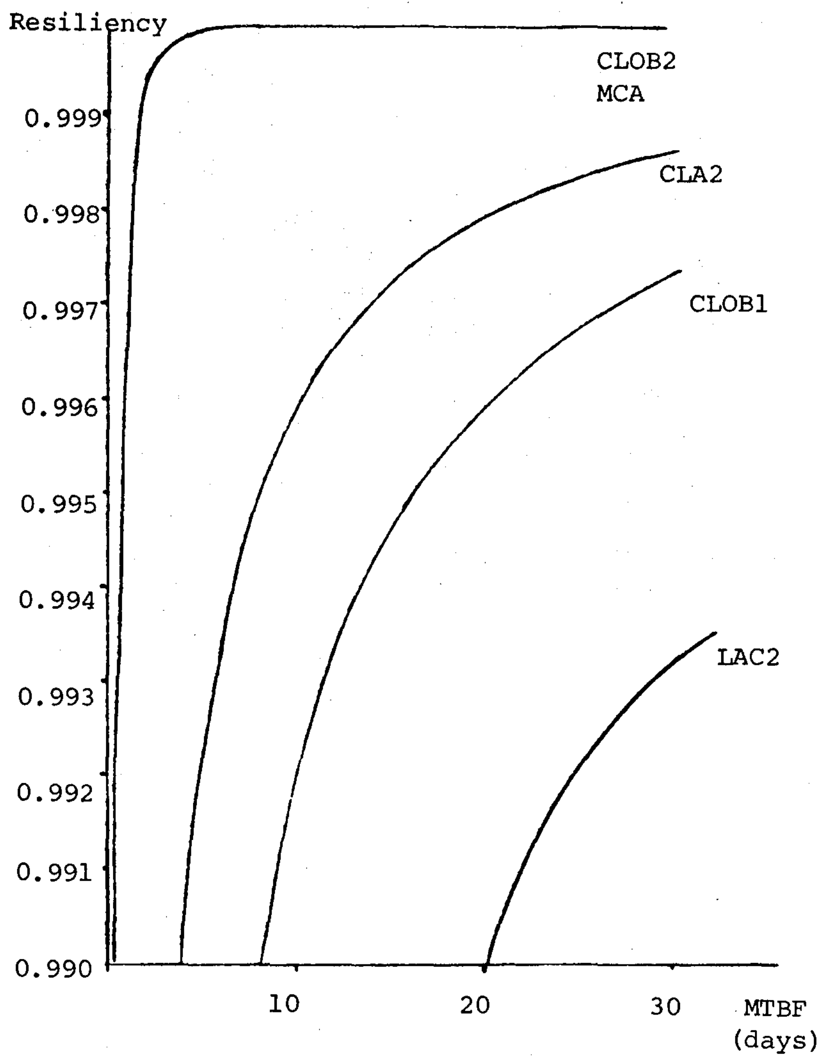
The parameters for the models and the values used to obtain the curves given in Figure 1 can be found in Table 1. The resiliency of the different models as a function of MTBF is given in Figure 1. Table 2 summarizes the characteristics of the algorithms compared in Figure 1.

Parameters	Definitions	Values
Ar	arrival rate per node	0.1 updates/sec
Bs	read-set plus write-set size	5 items
I/O	I/O time for accessing an item, a lock, a timestamp, etc.	0.025 sec
M	number of items in the database	5000
MTTR	mean time to repair	3480 sec
N	number of nodes	5
T	network delay	0.1 sec

Table 1. Parameters and their values.

<u>Algorithms</u>	<u>Characteristics</u>
CLOB1	Centralized Locking with One Backup. The central, backup, and a secondary node have to be "up" before processing of transaction can begin; thereafter, processing will not be interrupted as long as the secondary and either the central or the backup are up.
CLOB2	Centralized Locking with One Backup. The central (or backup) and a secondary node have to be up before processing of transaction can begin; thereafter, processing will not be interrupted as long as the secondary and either the central or the backup are up.
CLA2	Centralized Locking Algorithm. The central and a secondary have to be up before processing of transaction can begin.
MCA	Thomas' Majority Consensus Algorithm. Processing can begin as long as one of the nodes is up.
LAC2	Lock All Copies. More than a majority of the nodes have to be up before processing can begin.

Table 2. Summary of algorithms compared in Figure 1.
(Please refer to preceeding sections for more detail)



CLOB1--Centralized Locking with One Backup.
CLOB2--Centralized Locking with One Backup.
CLA2--Centralized Locking Algorithm.
MCA--Thomas' Majority Consensus Algorithm.
LAC2--Locking All Copies.

Figure 1: Resiliency as a function of MTBF

Figure 1 shows that algorithms requiring that a majority of nodes be "up" (such as LAC2) have relatively low resiliency. Models requiring that the central, the backup, and a secondary node be "up" (CLOB1) before an update can be started has higher resiliency than LAC2. Models requiring only the central and a secondary node be up (CLA2) has higher resiliency than CLOB1. The resiliency of CLOB2 and the resiliency of MCA are essentially identical for the range of MTBF and the parameters chosen. CLOB2 and the daisy chain Majority Consensus Algorithm appear to have the highest resiliency of all the models considered in this paper. With CLOB2, processing of the transaction can begin when the central or/and the backup and a secondary are "up;" furthermore, failure of the central does not disrupt the progress as long as the backup is still alive. With the Majority Consensus Algorithm, processing of the update transaction can begin as long as one of the node is up; processing of the transaction will not be blocked as long as one of the nodes that have not voted on the update is alive to receive the vote request and vote on the update. It is interesting to note that the Thomas' Majority Consensus Algorithm, albeit it has a relatively high response time (Garcia-Molina, 78), appears to have very high update resiliency. The success of the Majority Consensus Algorithm can be attributed to the fact that it only requires a majority of the nodes for synchronization; furthermore, not all of these nodes have to be "up" simultaneously.

The resiliency of Broadcast Majority with Timestamp algorithms is very similar to that of LAC2. The resiliency of CLA1 and LAC1 are not plotted on the graph in Figure 1; their resiliency are the lowest among the models considered in this paper. Although both CLA1 and LAC1 require that all the nodes be up, the update resiliency of CLA1 is better than LAC1 because CLA1 runs faster. Locks have to be stored at all nodes in the case of LAC1 whereas locks have to be stored at the central alone for CLA1; thus, W_i and g_i for CLA1 are lower, implying that the cohorts have to be up for a shorter time.

5 Summary

In this paper, we have used read resiliency and update resiliency to compare the resiliency of a few concurrency control algorithms and point out changes to the algorithms that allow update resiliency to be improved. We have only considered some aspect of reliability, availability, and resiliency. The modelling in this paper can be considered as modelling of the first order effect of availability, resiliency, and reliability. The problem of communication network failure has been temporarily ignored to make the analysis tractable.

Not considered in this paper is the problem of recovery. Although the Majority Consensus Algorithm performs well in terms of resiliency as defined in this paper, its recovery is far more complicated than that of the Centralized Locking with One Backup Algorithm. Work is in progress, in analyzing and comparing more carefully the behavior of the models described in this paper, and in developing more detailed models to allow us to better understand the relationships among availability, resiliency, and concurrency control.

6 References

(Alsberg, Belford, et al., 76) Alsberg, P. A., Belford, G. G., Day, J. D., and Grapa, E., "Multi-copy Resiliency Techniques," 1976 CAC document reprinted in Distributed Data Management (J. B. Rothnie, Jr., P. A. Bernstein, and D. W. Shipman, eds.) IEEE, 1978, pp 128-176.

(Bernstein et al., 78) Bernstein, P. A., Rothnie, J. B., Goodman, N., and Papadimitriou, C. A., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases," IEEE Trans. on Software Eng., Vol. SE-4, pp 154-168, May 1978.

(Bernstein et al., 79) Bernstein, P. A., Shipman, D. W., and Wong, W. S., "Formal Aspects of Serializability in Database Concurrency Control," IEEE Trans. Softw. Eng. SE-5, 3 (May 1979), pp 203-215.

(Bernstein and Goodman, 80a) Bernstein, P. A. and Goodman, N., "Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems," Proc. 6th Int. Conf. on Very Large Data Bases, Oct. 1980.

(Bernstein and Goodman, 80b) Bernstein, P. A. and Goodman, N. "Fundamental Algorithms for Concurrency Control in Distributed Database Systems," Tech Report CCA-80-05, Computer Corporation of America, Feb., 15, 1980.

(Bernstein et al., 80) Bernstein, P. A., Shipman, D. W., Rothnie, J. B., "Concurrency Control in A System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems, Vol 5, No. 1, pp 18-51, March 1980.

(Bunch, 75) Bunch, S. R., "Automated Backup," in Preliminary Research Study Report, CAC Doc. 162, Center for Advanced Computation, Univ. of Illinois at Urbana-Champaign, May 1975.

(Cheng and Belford, 80a) Cheng, W. K. and Belford, G. G., "Update Synchronization in Distributed Databases," Proc. Sixth Int. Conf. on Very Large Data Bases, Montreal, Oct. 1980.

(Cheng and Belford, 80b) Cheng, W. K. and Belford, G. G., "Analysis of Update Synchronization Schemes in Distributed Database," Proc. COMPCON Fall 80, Sept. 1980.

(Cheng and Belford, 80c) Cheng, W. K. and Belford, G. G., "A Clock Synchronization Algorithms for Update Synchronization in Distributed Databases on Local Broadcast Networks," Proc. of the 5th Conference on Local Computer Networks, Minneapolis, Oct. 1980.

(Ellis, 77) Ellis, C. A., "A Robust Algorithm for Updating Duplicate Databases," Proc. 2nd Berkeley Workshop in Distributed Databases and Computer Networks," May 1977.

(Eswaran et al., 76) Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in A Database System," Comm. ACM, vol 19, No. 11, pp 624-663, Nov. 1976.

(Garcia-Molina, 79) Garcia-Molina, H., "Centralized Control Update Algorithms for Fully Redundant Distributed Databases," Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp 699-705.

(Garcia-Molina, 78) Garcia-Molina, H., "Performance Comparison of Two Update Algorithms for Distributed Databases," Proc. Thrid Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 29-31, pp 108-199, 1978.

(Gifford, 79) Gifford, D. K., "Weighted Voting for Replicated Data," in Proc. 7th Symposium on Operating Systems Principles, ACM, Dec. 1979.

(Gray, 79) Gray, J. N., "Notes on Database Operatiing Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1979.

(Hammer and Shipman, 79) Hammer, M. H. and Shipman, D. W., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," Tech. Rep. CCA-79-05, Computer Corporation of America, Cambridge, MA, July 31, 1979)

(Kaneko et al., 79) Kaneko, A., Nishihara, Y., Tsuruoka, K., and Hattori, M., "Logical Clock Synchronization Method for Duplicated Database Control," Proc. 1st Int. Conf. Distributed Computing Systems, Oct. 1979, pp 601-611.

(Lampson and Sturgis, 79) Lampson, B. and Sturgis, H., "Crash Recovery in Distributed Data Storage System," Comp. Sci. Lab., Xerox Palo Alto Res. Center, Palo Alto, CA, unpubl. paper, 1979.

(Skeen, 81) Skeen, D., "Nonblocking Commit Protocols," Memorandum No. UCB/ERL M81/11, March 10, 1981.

(Skeen and Stonebraker, 81) Skeen, D. and M. Stonebraker, "A Formal Model of Crash Recovery in A Distributed System," Proc. of The Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Feb. 1981, pp 129-142.

(Stonebraker, 79) Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Trans.on Software Engineering, vol SE-5, No. 3, pp 188-194, May 1979.

(Thomas, 79) Thomas, R. H., "A Majority Consensus Approach to Concurrency Control," ACM Trans. on Database Systems, Vol. 4, No. 2, pp 180-209, 1979.

A ROBUST AND EFFICIENT PROTOCOL FOR CHECKING THE AVAILABILITY OF REMOTE SITES

Bernd Walter

Institut für Informatik
University of Stuttgart
Stuttgart
Fed. Rep. of Germany

Abstract: A robust and efficient protocol for checking the availability of remote sites is described. A remote site is said to be available if it has not crashed and if the communication facilities are able to transmit messages to and from this site. The presented protocol is robust against any number of site crashes and communication breakdowns including network partitioning. It is proven that the protocol is minimal in the given context. Some applications such as the recovery of multi-site-transactions are given.

Keywords: Availability, robust protocols, distributed processing, transaction processing, distributed databases.

1. Introduction

The protocol for checking the availability of remote sites in computer networks can be used in the context of arbitrary applications. However, since the original intention was to develop just a tool to support the recovery of multi-site-transactions in distributed database systems (DDBS), a DDBS context will be used in the following discussions.

Update-transactions in DDBS may be arbitrarily complex, such that several sites of the system are involved in the processing of just one transaction. To ensure consistency, update-transactions must be atomic, i.e. all its updates must be committed or none of them. Adequate recovery-mechanisms and so-called two-phase-commit protocols /1, 6/ are needed to guarantee this all-or-nothing property throughout normal and abnormal conditions. A lot of work has been invested in the development of commit-protocols and recovery-mechanisms for DDBS, however, only few investigations were made in order to develop suitable protocols for the detection of abnormal conditions (failures). In distributed systems those failures are of special interest which affect the availability of remote sites. Events which cause sites to become unavailable are called crashes, events which cause sites to become available again are called recoveries.

Assume a transaction T and two sites A and B such that some updates of T are to be performed at A and the others at B. Assume further, that A is waiting for some ready-message from B. Now if B crashes or becomes unavailable due to communication breakdowns, two different strategies are possible:

1 A waits until the awaited message arrives, whatever is happening. If B eventually recovers and remembers T, then this strategy will work. However serious drawbacks are associated with this strategy:
   -- Since none of the resources locked by the affected transaction may be released, these resources are not available for other transactions, i.e. other transactions may be blocked as well until B recovers.
   -- The cost of waiting increases proportional to the time a transaction has to wait /10/.
   -- There is no guaranty that B will recover within an acceptable amount of time. Sometimes, there is even no guaranty that B will recover at all.
2 A backs out the affected transaction T and releases all its resources. As soon as B recovers, a message that T has been backed out is sent to B. This strategy avoids the drawbacks of the first strategy, however, a protocol is needed to ensure fast and reliable detection of crashes and recoveries. Note that such a protocol needs not to detect crashes of single processes, since in this case the local supervisor remains intact and will be able to detect and to handle such events locally. Remote detection is only needed if the supervisor and thus the whole system becomes unavailable (see also application example 1).

It would also be possible to use a mixture of the two strategies, such that site crashes are detected whenever some site tries to send a message to a crashed site. However this non-systematic approach cannot guarantee that a site crash will be detected within an acceptable period of time, that means that the drawbacks of the first strategy are valid for this mixture as well.

Up to now SDD/1 is the only system which provides a facility for systematic detection of site failures and recoveries. However, the mechanisms of SDD/1's RELNET /4/ are not robust against network partitioning. Furthermore, as will be seen in the discussions at the end of this paper, the corresponding RELNET-protocols need more messages than our proposal.

In this paper a protocol will be described with the following characteristics:

- Each site of a network is provided with a table which presents this site's current view of the network, i.e. the table shows which remote sites are currently available for the owner of the table and which are not.
- The application of the protocol is not limited to the use in DDBS, but may be used in other distributed systems as well.
- Robustness is provided against any number of site crashes and communication breakdowns including network partitioning.
- During the execution of the protocol no data must be written onto stable storage, only during site start/restart one disc access will be needed.

- The protocol uses distributed control.
- The protocol is minimal in the given context, i.e. a minimal number of messages is needed during normal operations and a minimum of time is needed to propagate crashes and recoveries.

In the remainder of this paper first of all the basic communication facilities of the underlying network will be defined. Then a detailed description of the protocol for checking the availability of remote sites will be given including some proofs of the protocol's characteristics. The application of the protocol will be demonstrated in the context of transaction recovery, maintenance of redundant data, and compile-time checks. Finally the described protocol will be compared with the corresponding mechanisms of SDD/1's RELNET.

2. Basic Communication Facilities

The underlying computer network, which provides the basic message transmission facilities (ARPANET-like) is assumed to have the following characteristics:

- If A receives two messages from B, then they are received in the same order in which they were sent.
- If A did not fail between the receipt of two messages from B, then A also received any other messages sent to it by B between these two messages.
- It is not guaranteed that a message which is sent by one site, will actually be received by the intented receiver. If the receiver fails before the message arrives, then the message is lost. The only way for a sender to be sure that a message has been received is to require an acknowledgement from the receiver.
- If communication facilities fail such that A is able to communicate with B but not with C, then also B cannot communicate with C.
- If B is unreachable from A, then A cannot determine whether B has crashed or is working in isolation.
- Due to crashes of communication lines the network may be split into two or more independent subnetworks. In reality partitions will mostly consist of just one isolated site, this will be the case whenever the local communication facilities fail such that no messages can be sent to remote sites.

Furthermore it is assumed, that there are two constants LMAX and LMIN where LMAX reflects the maximum transmission time of a single message between two arbitrary nodes and LMIN the minimum transmission time. Note that for the sake of simplicity clock divergencies during LMAX and LMIN are not treated explicitly.

3. Availability of Remote Sites, Views and State Tables

A remote site B is available for a site A, only if the following holds:

1 B is able to process requests from A, i.e. B has not crashed.

2 The communication facilities are able to transmit messages between A and B.

3 A is aware that 1 and 2 hold, i.e. it is aware that it may send requests to B.

In order to enable a site A to determine whether 1 and 2 hold, B has to send messages to A. Such messages could be send on A's request or periodically. If A does not receive an awaited message within a certain period of time, then B is assumed to be unavailable for A.

If such messages are sent on request, then two messages (request + answer) and one timer (A waits for a period of length 2LMAX + AMAX where AMAX is the maximum processing time needed at site B to generate an answer to A's request) are needed to perform a single check of 1 and 2. One message and two timers are needed, if B periodically sends messages (HI-messages) to A. In order to have a minimum of messages per check, the second alternative will be used in the remainder of this paper. Periodical checks also assure that, whenever 1 and 2 hold, A will be made aware of this fact.

Periodical checks are performed as follows :
- Each time, B sends a HI-message to A, a timer TSENDER is started. When TSENDER expires, the next HI-message is sent etc.
- Each time, A receives a HI-message from B, a timer TCONTROL is restarted. If TCONTROL expires without A having received a HI-message, then B is assumed to be unavailable. As soon as a new HI-message from B is received, B is assumed to be available again.

Let SINV be the timeout period of TSENDER, then the timeout period of TCONTROL can be determined as EINV = SINV + LMAX - LMIN. This is the maximum period which may pass between two successive HI-messages, it reflects the worst case, i.e. a 'slow' message follows a 'fast' message.

This small protocol enables A to have a certain view of B. Two different views are possible:
- B is UP: No timeout has been reported by TCONTROL, thus B is assumed to be available for A.
- B is DOWN: A time-out was reported by TCONTROL and no HI-message has been received since that time, thus B is assumed to be unavailable.

Due to several reasons a view sometimes will not reflect the actual state of the system:
- State change (crash or recovery) since last check.
- HI-messages were lost.
- Message transmission time exceeded LMAX.

However, since checks are performed periodically a view eventually will reflect the true state (provided that the state will be stable (no crashes and recoveries) for a sufficient period of time).

A state table contains the current views of all remote sites as seen from the owner of this state table. Such a table simply consists of an entry for each site of the network. If CURRENT-STATE$_A$ is the state table of A, then CURRENT-STATE$_A$ [B] reflects A's view of B and thus contains one of the values UP or DOWN.

It is assumed that there are physical clocks at each site of the system. Physical clocks are needed to implement timers and to increment local logical clocks /4/. Logical clocks are used for the generation of timestamps. All the clocks in the network are synchronized in a way as described in /5/ (see also below).

4. The Protocol (RSC-Protocol)

In the following the protocol to be described will be called RSC-protocol (RSC for Reliable State Control). The RSC-protocol consists of two subprotocols, CONTROL and CHANGE. CONTROL is used to detect state changes and CHANGE is used to perform the corresponding updates in the CURRENT-STATE tables of the various sites. At first CONTROL and CHANGE will be described separately and the interactions between these two subprotocols will be defined. Then the behaviour of the RSC-protocol during site start/restart and in the case of a network partitioning will be discussed. Finally some proofs of the characteristics of CONTROL and CHANGE will be given. An algorithmic definition of the protocols by means of a PASCAL-like notation can be found in the appendix. Note that the complete RSC-protocol has also been specified formally and verified by means of predicate-transition-nets /3/ in combination with some temporal logic; however, due to space limitations the formal specifications are omitted in this paper; the interested reader is referred to /12/.

4.1 CONTROL

The purpose of CONTROL is to detect crashes and recoveries as well as differing views within a partition. Whenever such an event has been detected, CHANGE will be initiated.

First of all a virtual ring is defined over all sites of the network to be controlled. All sites are numbered along the path of the virtual ring, starting with 1 and ending with N (N is the number of sites in the network) such that site 2 follows site 1 follows site n etc. It should be clear, that the virtual ring does not reflect the physical structure of the network. A protocol using a virtual ring was also presented in /7/ as a solution to the mutual exclusion problem. However, our concept of the virtual ring is quite different. In /7/ the ring is used as a message path for a circulating token. In our protocol the virtual ring is used to obtain well defined control structures, control messages may also be transmitted along other ways.

During normal operations (each site is available for each other site) the basic control protocol presented in the last section is performed by every two adjacent sites along the virtual ring such that an arbitrary site K behaves as follows:
- K periodically sends HI-messages to site (K+1)modN. The periods are determined by means of TSENDER. Each HI-message contains an actual timestamp which is used for clock synchronization.
- K periodically receives HI-messages from site (K-1)modN1) and restarts

TCONTROL. If TCONTROL expires, subprotocol CHANGE is started, since (K-1)modN's state is assumed to have changed from UP to DOWN.

Of course, HI-messages could be substituted by normal messages, provided that actual timestamps are included in such messages as well. However, for the sake of simplicity in the following only explicit HI-messages will be considered.

In reality an arbitrary number of sites may have crashed and the net may be partitioned. If (K-1)modN is DOWN in K's view, then
- K has to monitor (K-1)modN to detect when it will be available again (due to site recovery or recovery of the corresponding communication paths).
- K has to monitor those sites which formerly had been controlled by (K-1)modN.

All these sites to be monitored by K make up K's AREA-OF-CONTROL. More exactly K's AREA-OF-CONTROL includes the following sites:
- The first site preceeding K along the virtual ring which is UP in K's view. Let this site be (K-J)modN where 0 < J < N.
- All sites (K-I)modN which preceed K and succeed (K-J)modN along the virtual ring (0 < I < J). All these sites are DOWN in K's view.

If K itself is the only available site in it's view, then K's AREA-OF-CONTROL does only include DOWN-sites.

If (K+1)modN is DOWN in K's view, then K has to send HI-messages to all potential controllers of itself (Remember that there may be some sites which have not crashed but which are not available for K because some communication lines have crashed). All these sites make up K's so-called BROADCAST-AREA. K's BROADCAST-AREA exactly contains the following sites:
- The first site (K+H)modN (0 < H < N) which follows K along the virtual ring and which is UP in K's view.
- All sites (K+I)modN which follow K and preceed (K+H)modN along the virtual ring, such that 0 < I < H. All these sites are DOWN in K's view.

For an arbitrary state of the network CONTROL can be defined as follows (for an arbitrary non-isolated site K):
- K periodically sends HI-messages to each site in its BROADCAST-AREA. Each time TSENDER generates a timeout signal HImessages are sent to all these sites at once and TSENDER is restarted.
- K periodically receives HI-messages from site (K-J)modN which is the only site of K's AREA-OF-CONTROL which is UP in K's view. Each time a HI-message is received from (K-J)modN, timer TCONTROL is restarted. If a timeout signal is received from TCONTROL, then subprotocol CHANGE is started. CHANGE is also started whenever a HI-message is received from any other site in K's AREA-OF-CONTROL (sites which are DOWN in K's view).

1) Let (K-I)modN be defined as follows: If K > I then (K-I)modN = K-I; if k = I then (K-I)modN = N; if K < I then (K-I)modN = K-I+N.

Fig.1 in the appendix shows the arrangement of the AREAs-OF-CONTROL and the BROADCAST-AREAs in a network consisting of 5 sites, 2 of which are down. Fig. 2 shows the corresponding arrangement for the same network, now fallen apart into 2 partitions such that partition 1 contains sites 1, 4, and 5 with site 4 being down and partition 2 contains sites 2 and 3.

So far the dynamic behaviour of CURRENT-STATE has not been considered. However, at each point of time CURRENT-STATE may be updated by some other site which has detected some crash or recovery and thus has started CHANGE. In this case a new version of CURRENT-STATE is broadcast by the updating site (see next chapter).

Since new versions may be broadcast by different sites concurrently (due to the detection of multiple crashes/recoveries), a version number is attached to each new version. A version number consists of an actual timestamp and the broadcasting site's identifier. Each site accepts a new version only if its version number is higher than that of the local occurrence of CURRENT-STATE. To ensure that all sites in a partition have identical views and to detect the loss of new versions, the version number of the local CURRENT-STATE table is attached to each HI-message. Now, differing views can be detected by the receiving site.

However, due to different message delays during the distribution of a new version of CURRENT-STATE temporary inconsistencies between the views of different sites in the same partition may occur. Since such temporary inconsistencies should be tolerated, CONTROL must be able to distinguish between inconsistencies caused by message delays and inconsistencies caused by failures.

If a new version of CURRENT-STATE is broadcast at time T, then it will be received by the other sites at a point of time between (T + LMIN) and (T + LMAX). If a HI-message is received containing a version number higher than the local table's version number, this cannot happen earlier than at time (T + LMIN + LMIN). Whenever this is happening, a timer TCHECK (timeout period: LMAX - LMIN - LMIN) is started at the receiving site and the received version number is stored in a variable CHECK-VERSION. If a new version of CURRENT-STATE with a version number at least as high as the value of CHECK-VERSION is received, then TCHECK is reset, else, if TCHECK expires, a failure must be assumed and CHANGE is started. If a further HI-message is received with version number higher than the value of CHECK--VERSION, then this new value is stored in CHECK-VERSION and TCHECK is restarted.

This procedure has to be applied only to HI-messages received from the UP-site in the receiver's AREA-OF-CONTROL and only if the received version number is higher than the local one (else each inconsistency between views will be detected twice and two CHANGE executions will be initialized instead of one).

So far five events may happen during the execution of CONTROL (as will be seen later there are two more):

- Receipt of a HI-message.
- Receipt of a new version of CURRENT-STATE (to be called STATE-TABLE-message).
- Time-out signal from TSENDER.
- Time-out signal from TCONTROL.
- Time-out signal from TCHECK.

To avoid misinterpretations of a site's status, time-out signals from TSENDER are processed at the highest priority. All other events are processed in a FIFO order.

4.2 CHANGE

The purpose of CHANGE is to propagate crashes and recoveries detected by CONTROL. The propagation should be performed within one execution of CHANGE however complex the detected changes had been. In the following a site initiating a CHANGE execution will be called coordinator, all other sites participating in this execution will be called cohorts. Since CHANGE may be executed at different sites concurrently, some sites may simultaneously be the coordinator in one incarnation of CHANGE as well as a cohort in some other incarnations.

Sometimes, CONTROL only detects a part of a complex change and the coordinator of the subsequent CHANGE execution can be provided only with incomplete information on the real situation.

> Example: Assume that K's AREA-OF-CONTROL consists of (K-1)modN and that (K-1)modN's AREA-OF-CONTROL consists of (K-2)modN. Now, if (K-1)modN and (K-2)modN crash nearly simultaneously, then CONTROL assures that K detects the crash of (K-1)modN. However, at this point of time CONTROL does not enable any site to detect the crash of (K-2)modN.

Since we do not want CONTROL to use additional messages (control messages should not decrease system throughput), it is CHANGE's task to collect the lacking information on the characteristics of the detected change.

When CHANGE is initiated, the coordinator's physical clock can be in one of the following states:
- The clock contains the value 0: A site has initiated CHANGE after its restart (see 4.4) and no timestamped message has been received between restart and initiation.
- The clock possibly diverges from other clocks in the system: A site has initiated CHANGE after having detected a state change from DOWN to UP (the change might have been caused by the physical recovery of a network partitioning).
- The clock is synchronized: All other cases.

Since the version number of the new version of CURRENT-STATE to be built up by CHANGE must contain an actual timestamp, it is also a task of CHANGE to synchronize the coordinator's clock.

CHANGE consists of two phases:
- Information-collection- and clock-synchronization-phase: At first the coordinator sends so-called REQ-STATE-messages containing the coordinator's identifier and an actual timestamp to all sites in the network and

then starts its local timer TCHANGE (timeout period: 2•LMAX + AMAX). A Cohort receiving a REQ-STATE message responds by sending a STATE-REPORT-message back to the coordinator. A STATE-REPORT-message contains the cohort's identifier, the timestamp of the corresponding REQ-STATE-message and an actual timestamp. This phase ends when STATE-REPORT-messages have been received from all cohorts; if any response is missed, this phase lasts until a timeout signal has been reported by TCHANGE.
- State-table-distribution-phase: Depending on the received STATE-REPORT-messages a new version of CURRENT-STATE is built up. Sites from which a STATE-REPORT-message was received are marked UP, non-responding sites are marked DOWN. This new version of the state-table is broadcast as part of a STATE-TABLE-message to all sites which are marked UP. A STATE-TABLE-message also contains an actual timestamp and the coordinator's identifier, which together make up the new version number. A cohort accepting a new version of CURRENT-STATE, immediately sends HI-messages to all sites in its new AREA-OF-CONTROL and restarts its timers TSENDER and TCONTROL. If no UP-site is contained in the new AREA-OF-CONTROL, TCONTROL is stopped (this can happen if a site has been isolated due to communication breakdowns).

As already mentioned, concurrent CHANGE-executions are synchronized by means of the version-number of the new state-table. If clocks fail such that a state-table is accepted which does not reflect an actual view of the network but does contain the highest timestamp in its version number, this failure will be detected by CONTROL. Therefore the RSC-protocol is also robust against diverging and/or failing clocks.

4.3 RSC = CONTROL + CHANGE

Since a CHANGE-coordinator remains a member of some other site's AREA-OF-CONTROL, a coordinator must keep sending HI-messages during its CHANGE-execution. The following events can occur during a CHANGE execution:
- Time-out signal from TSENDER.
- Time-out signal from TCHANGE.
- Receipt of a HI-message (will be ignored).
- Receipt of a REQ-STATE-message.
- Receipt of a STATE-REPORT-message.
- Receipt of a STATE-TABLE-message.

These events are processed depending on their priority. Timeout signals from TSENDER possess the highest priority, REQ-STATE-messages possess the second highest priority, all other events are processed in a FIFO order.

In addition to the events listed above, two further events may occur during the execution of CONTROL:
- Receipt of a REQ-STATE-message (generate response).
- Receipt of a STATE-REPORT-message, however, this can only happen due to failures (ignore).

During a site's start/restart this site just executes the CHANGE-protocol and then switches over to CONTROL, all further CHANGE-executions are initiated by CONTROL (for more details see below). If CHANGE and CONTROL are regarded as procedures then RSC can be interpreted as a procedure which consists of a call of CHANGE and a subsequent call of CONTROL. In a real environment, e.g. in the context of a distributed operating system kernel, all events which can happen during RSC can be interpreted as kernel events (interrupts) which may be processed in two modes, depending on whether CONTROL or CHANGE is currently executed.

An algorithmic specification of all constants, variables, timers, messages and procedures of the RSC-protocol can be found in the appendix.

4.4 Site start/restart

A site start/restart is done by executing the CHANGE protocol. The REQ-STATE-messages sent during this CHANGE-execution contain a zero-timestamp (assuming that during the site's down time the clock stopped too). With the receipt of the STATE-REPORT-messages the starting/restarting site's clock can be resynchronized. HI-messages are sent not before CURRENT-STATE has been reestablished; this is also the point of time at which TSENDER is started for the first time after the site's down-time.

The only information which has to be stored on stable storage is the description of the virtual ring, which is a simple binary relation, containing a pair of the form (physical address, position in the virtual ring) for each site in the network. All other structures need only be kept in volatile storage.

It should be clear that this procedure works even if all sites of a network have crashed. However, if the system-clock should be synchronized with the 'clock' of the real world, then real world's time has to be supplied from outside the system.

4.5 Network partitioning

If some communication lines crash such that the network is partitioned, then this will be detected since some sites will become unavailable for some other sites.

During the recovery of a partitioned network two requirements must be fulfilled:
- Resynchronization of the clocks, which may have diverged during the period of partitioning.
- Unification of the views of sites which formerly were included in different partitions.

The physical recovery of the crashed communication lines will be detected by means of HI-messages received from sites which are DOWN in the receivers view. The subsequent CHANGE execution fulfills both of the above requirements. Since an actual timestamp is contained in each message used by CHANGE, the clocks can be resynchronized. CHANGE also provides all sites in the available part of the network with a new version of CURRENT-STATE. Failures during a CHANGE execution will be detected by CONTROL.

Network partitioning is handled totally application-independent. The RSC-protocol provides mechanisms, not strategies.

4.6 RSC-Characteristics

In this section it is shown, that the RSC-subprotocols CONTROL and CHANGE are minimal in the sense that they use a minimum of messages (CONTROL) or a minimum of time (CHANGE).

Definition:

- A protocol is said to be message-minimal, if the total number of messages used by this protocol is the smallest possible number of messages in the given context.
- A protocol is said to be time-minimal if the critical message path is the smallest possible in the given context. A critical message path is the longest path of messages in the protocol, such that the sending of each message depends on the receipt of the preceeding message.
  Example: If a protocol consists of one message to be sent to a remote site, then the critical path has a length of 1. If an acknowledgement is requested, then the critical path has a length of 2, since the acknowledgement depends on the receipt of this message. If the message is sent to several sites in parallel, then the length of the critical path remains the same, since these messages do not depend on each other.

Assertion:
CONTROL is message-minimal during normal operations (each site available for each other).

Proof:
The context requires that checks in the network have to be performed periodically. Therefore it has to be proved, that the number of messages needed to perform one complete check of the network during normal operations is minimal.

CONTROL needs N messages to perform one complete check, one HI-message from each site. Now assume that there is another protocol which uses less then N messages. Then there must be at least one site which does not send a HI-message. However, a site which does not manifest itself by sending messages will be assumed of being not available. Since during normal operations all sites are available for all other sites such a protocol provides incorrect views; thus CONTROL is message-minimal.

Assertion:
The number of messages CONTROL needs in the general case depends only on the number of partitions. Let P be the number of partitions then P•N messages are needed per complete check.

The proof is straightforward and left to the reader.

Assertion:
CHANGE is time-minimal.

Proof (Outline):

CHANGE has a critical message path of length 3 (a STATE-REPORT-message is sent only after the receipt of a REQ-STATE-message, the sending of STATE-TABLE-messages depends on the receipt of the STATE-REPORT-messages). In order to prove the correctness of the assertion, it will be shown, that in CONTROL's context there can be no protocol with a path-length less then 3. It should be remembered that in CONTROL's context CHANGE has to complete the information provided by CONTROL and has also to synchronize the coordinator's clock before the new version is distributed.

Assume a protocol with a critical path of length one. The only thing such a protocol could do is to broadcast the information provided by CONTROL. However, CONTROL does not provide complete information on some kinds of multiple crashes (see example given in one of the preceeding chapters). The construction of crashes which cannot be completely propagated even in two subsequent executions of such a protocol is straightforward and left to the reader. Furthermore such a protocol is not able to synchronize the coordinator's clock.

Now assume a protocol with a critical path of length two. One message is needed to propagate the change and one message is left over to complete the coordinator's information on the state change to be propagated and to synchronize the coordinator's clock. However, additonal information is available only on request such that two messages would be needed to complete the information. Furthermore the synchronization of the clock would require the receipt of a timestamped message which again must be requested. Due to these reasons, in the given context there can be no such protocol which synchronizes the coordinator's clock, completes the state information and distributes a new version of CURRENT-STATE. Therefore, CHANGE is time-minimal under the given conditions.

Only under relaxed conditions an alternative protocol could be used in-instead of CHANGE. Assume that clocks never fail and that no clock divergences occur during network partitioning. Then the following protocol (to be called QUICKCHANGE) would work:
- At first the coordinator sends PREPARE-messages containing the version number of the new version of CURRENT-STATE to all sites of the network and starts its timer TCOORD (timeout period: LMAX + LMAX).
- Each site which has received such a message broadcasts a REPORT-message containing the new version number, starts its timer TQUICK (timeout period: LMAX + LMAX - LMIN) and then prepares the construction of a new version of CURRENT-STATE.
- Each site from which a REPORT-message is received, will be marked UP in the new version, else, if a timeout is reported by TQUICK or TCOORD all sites from which no REPORT-message has been received will be marked DOWN.

This protocol does not guarantee that state tables with identical version numbers do always reflect the same view (lost messages). In order to enable CONTROL to detect differing views even in this case, a HI-messages must contain a complete state table instead of a version numbers. However, even if clocks do not fail, CHANGE might be the better alternative, especially in large networks, where QUICKCHANGE needs much more messages than CHANGE.

5. Interface of the RSC-Layer

The RSC-layer contains the RSC-protocols and should be located on top of the basic network communication facilities. It could be implemented as part of a distributed operating system kernel such that all messages to remote sites must pass the RSC-layer. In the following a possible interface to the RSC-layer will be defined by describing the various procedure calls which must be issued to request RSC-services (this version of an interface has been designed to fulfill the needs of a distributed data base systems, other applications might require different services):

- SEND-MESSAGE (destination, message) RETURNS (status, ack).
  If the destination site is not available (marked DOWN in the local CURRENT-STATE table) then an error code is returned to the calling process. If the intented receiver is marked UP, then the message is sent and a timer (time-out period of length 2•LMAX + AMAX) is started. If the timer expires and no acknowledgement has been received for this message, then CURRENT-STATE is checked again. The message will be retransmitted if the receiver is still marked UP, else an error code is returned to the calling process. Of course it is application-dependent whether acknowledgements are required or not, therefore it might be appropriate to have also procedures for sending unacknowledged messages; however for the applications to be discussed in the next chapter the given interface will be sufficient.

- RECEIVE-MESSAGE (source) RETURNS (status, timestamp, message).
  If the site of the expected sender is DOWN (becomes DOWN) then an error code is returned to the calling process, else the message and its timestamp will be returned upon arrival. If a message is received from a remote site, then CURRENT-STATE has to be checked. The message will be ignored, if the sender is marked DOWN (note that there can't be a process waiting for such a message, since in this case an error code would have been returned such that this process might have already initialized some exception handling procedures). Of course, if there is no CURRENT-STATE table (during recovery) then incoming messages which are not RSC-related are ignored, too.

- SEND-ACK (destination, timestamp) RETURNS (status).
  An acknowledgement contains the timestamp of the message to be acknowledged. If the destination site is down, then an error is returned to the calling process, else an ok is returned in the status. In the environment of distributed data base systems acknowledgements sometimes are explicitly sent by the receiver after he has stabilized the information received in the corresponding message (e.g. logging of some information in stable storage).

- CHECK-STATE (site-number) RETURNS (state).
  The state of the site identified by this site-number is checked in CURRENT-STATE, the actual state is returned to the calling process.

- WATCH-UP (site-number) RETURNS (UP),
  WATCH-DOWN (site-number) RETURNS (DOWN).
  If the state of the site to be watched is already UP (DOWN) or as soon as its state changes from DOWN to UP (UP to DOWN), then UP (DOWN) is returned to the calling process.

- STOP-WATCH (site-number).
  The corresponding WATCH-UP or WATCH-DOWN formerly requested by the calling process is deleted.

- TIMESTAMP () RETURNS (timestamp).
  An actual and unique timestamp is delivered to the calling process.

To implement this interface two internal tables must be maintained, one table to contain the messages to be acknowledged, the other one to contain all WATCH-UP and WATCH-DOWN requests. Note that the RSC-layer has been given in an idealized form, in a real environment additional parameters, such as the identification of the calling process, would be needed.

6. Application of the RSC mechanisms

In this section three examples are given to demonstrate how the RSC-mechanisms can be applied to support
1 the recovery of distributed update-transactions
2 the maintenance of redundant data
3 the compile-time-checking of the availability of resources

Example 1: distributed update transactions

First of all the basic characteristics of transactions must be defined (see also /11/):
- From the user's point of view TAs possess the atomic property. In the case of update-TAs the system guarantees that either all of the updates are performed or none of them.
- TAs are static, i.e. all processing sites are predetermined at compile time.
- The processing of a TA is co-ordinated by its site-of-origin.
- A TA consists a set of subtransactions (STA).
- STAs are atomic actions at a lower level of abstraction.
- Each STA is processed at just one site.
- Over the set of STAs a precedence structure is defined, which determines the order in which the STAs must be processed. Parallel processing of STAs is possible too.

Updates may be arbitrarily complex, such that several sites are involved in the processing of such transactions. In principle update transactions could be processed as follows (TA- and STA-states must be kept in stable storage):

1 After the TA has been started by the user, STA-messages are sent to those sites which co-operate in the processing of this TA (TA enters state DO).
2 An STA enters the state DO after the STA-message has been received by the corresponding site.
3 STAs write their updates into intention lists /5,10/. When a STA has finished and its intention list has been written to disc, the STA-state changes to READY and a READY-message is sent to the site-of-origin. If an STA fails, then its state is converted to ERROR and an ERROR-message is issued.
4 After the site-of-origin has received READY- and/or ERROR-messages from all co-operating sites, the TA-state changes to COMMIT (all STAs were successful) or to BACKOUT (at least one STA failed) and COMMIT- (BACKOUT-)messages are sent to all sites co-operating in the processing of this TA. After the receipt of all acknowledgements the TA-state changes to UNKNOWN.
5 After the receipt of a COMMIT- (BACKOUT-)message the corresponding STA-state is changed to COMMIT (BACKOUT) and its intention list is executed (backed out).

Assume a TA in state DO after all STA-messages have been sent to the co-operating sites. In this case the site-of-origin is waiting for READY- or ERROR-messages, however, it cannot determine how long it has to wait. It even cannot determine whether there will be a response at all. In conventional systems this problem is handled by means of the positive-acknowledgement-or-retransmit technique, i.e. the STA-message will periodically be retransmitted until a READY- or an ERROR-message has been received. However, transactions may be arbitrarily complex and longlived /2/ such that an arbitrary number of messages is needed in order to process one TA. Furthermore this technique can only be applied if it is assured that co-operating sites do not crash forever (in this case retransmission must be performed forever). In the following it is shown how these problems can be avoided in using the RSC-protocol.

To assure the receipt of the STA-message, an acknowledgement must be required. Then the site-of-origin calls WATCH-DOWN for all co-operating sites during the TA is in the DO-state; each time a READY-message is received, a STOP-WATCH for the corresponding site is issued. If a site is reported to have crashed, then the TA can be backed out immediately (for corresponding protocols see /10/) and a WATCH-UP is called for the crashed site. When it is reported to be UP again, a BACKOUT-message is sent to this site.

A co-operating site calls WATCH-DOWN for the site-of-origin after having sent the READY-message. A STOP-WATCH is issued after the receipt of the corresponding COMMIT- (BACKOUT-)message. If a crash of the site-of-origin is reported, a request is sent to the other co-operating sites (for more details see /11/) to determine whether
- at least one of them has received a COMMIT- (BACKOUT-)message: all STAs

may convert to the COMMIT- (BACKOUT-)state.
- at least one of them is still in state DO: all STAs may convert to the BACKOUT-state.
- all of them are in state READY: all STAs have to wait for the recovery of the site-of-origin, since it is impossible to decide whether it would be correct to commit or to backout.

In each case the recovering site-of-origin will try to resume the processing of the TA. In the first case it finds the TA to be in state DO and unknown at all other sites, thus the TA was backed out. In the second case it finds the TA in state COMMIT (BACKOUT) and unknown on all other sites, thus the TA has been committed (backed out). In the third case processing can be resumed.

If only a single process fails on one of the co-operating sites, then this will not be reported by the WATCH-DOWN primitives. However, since process crashes do not affect the local supervisor in this case, the local system itself will be able to detect this crash. Since all STA-related informations are stored on stable storage it will also be able to initialize suitable recovery actions.

Example 2: Maintenance of redundant data

Assume, that copies of data are organized in a way, such that there is one primary copy and a collection of secondaries. Assume further that all updates must be performed via the primary. Most of the read-only-transactions will access secondary copies. However, some users will not be interested in retrieving data items which are not at the same level of actualization as the primary. Whenever the primary's site is not available from a secondary's site there might be updates which can only be propagated to the primary (network partitioning). To handle this problem a WATCH-DOWN is called for the primary's site. Whenever unavailability is reported the secondary will be marked to indicate that it may be not up-to-date and a WATCH-UP is called for the primary's site. When the primary's site is reported to be available again, then it can be checked whether some updates have been missed (perhaps by using version numbers).

Example 3: Compile-time-checks

For each multi-site-transaction it can easily be checked whether all needed remote sites are currently available by just calling the CHECK-STATE primitive. If one of the sites is not available, then the affected TA cannot enter it's processing phase.

7. Discussion

RSC keeps clocks synchronized and provides each site in the network with a complete view of all remote sites. To maintain these services RSC needs one message for each site and for each check. Similar services are provided by RELNET's so-called 'Global Time Layer'(GTL) /4/. However GTL needs more messages than RSC without providing the same level of robustness. Each RELNET-site has associated with it a collection of guardians to which it periodically issues so-called TIMESIGNAL messages. The mechanisms do not tolerate the crash of a site and all it's guardians. Therefore it would be desirable to have many guardians, however, more guardians require more TIMESIGNAL messages to be sent per check. Additionally so-called PROBE-messages are used to check the availability of remote sites, however, these checks are only performed on request such that two messages are needed per single check of one site (PROBE message plus response). Generally RELNET does not provide complete views of the network and thus cannot support the same set of applications as the RSC-protocol can. Although RELNET's GTL might be a very good solution for 'command and control' applications, we feel that RSC provides a more efficient and more robust solution for general applications.

The concept of having state tables was also suggested by other authors. However, the proposal in /8/ is based on centralized locking protocols and thus is strongly embedded into the context of certain strategies and not useable in more general contexts. The mechanisms described in /9/ are also embedded into a very special context, furthermore they can only be applied in environments with only a small amount of distributed transactions.

References

/1/ J.N. Gray, Notes on data base operating systems, in: R. Bayer, R.M. Graham, G. Seegmüller, eds., Lecture Notes in Computer Science 60: Operating Systems, an Advanced Course (Springer-Verlag, Heidelberg, 1978) 393 - 481.

/2/ J.N. Gray, The transaction concept: virtues and limitations, in: Proc. 7th Int. Conference on Very Large Data Bases (IEEE, 1980) 144 - 154.

/3/ H.J. Genrich, K. Lautenbach, The analysis of distributed database systems by means of predicate/transition-nets, in: G. Kahn, ed., Lecture Notes in Computer Science 70: The Semantics of Concurrent Computation (Springer-Verlag, Heidelberg, 1979) 123 - 146.

/4/ M. Hammer, D. Shipman, Reliability mechanisms for SDD/1: a system for distributed databases, ACM Transactions on Database Systems 5 (1980) 431 - 466.

/5/ L. Lamport, Time, clocks and the ordering of events in distributed system, Communications of the ACM 21 (1978) 558 - 565.

/6/ B. Lampson, H. Sturgis, Crash recovery in a distributed data storage system, Technical Report XEROX PARC, Palo Alto, Calif. (1979).

/7/ G. LeLann, Distributed systems - towards a formal approach, in: B. Gilchrist, ed., Information Processing 77 (North Holland Publishing Company, Amsterdam, 1977) 155 - 160.

/8/ D.A. Menasce, G.J. Popek, R.R. Muntz, A locking protocol for resource co-ordination in distributed databases, ACM TODS 5 (1980) 103 - 138.

/9/ M. Stonebraker, Concurrency control and consistency of multiple copies of data in distributed INGRES", IEEE Transactions on Software Engineering 5 (1979) 188 - 194.

/10/ B. Walter, Strategies for handling transactions in distributed data base systems during recovery, in: Proc. 6th Int. Conference on Very Large Data Bases (IEEE, 1980) 384 - 389.

/11/ B. Walter, Global recovery in a distributed data base system, to be published in: R.P. van de Riet, W. Litwin, eds., Distributed Data Sharing Systems (North Holland Publishing Company, Amsterdam, 1982).

/12/ B. Walter, Formale Spezifikation und Analyse des RSC-Protokolls, Technical Report, University of Stuttgart (1981).

Appendix 1

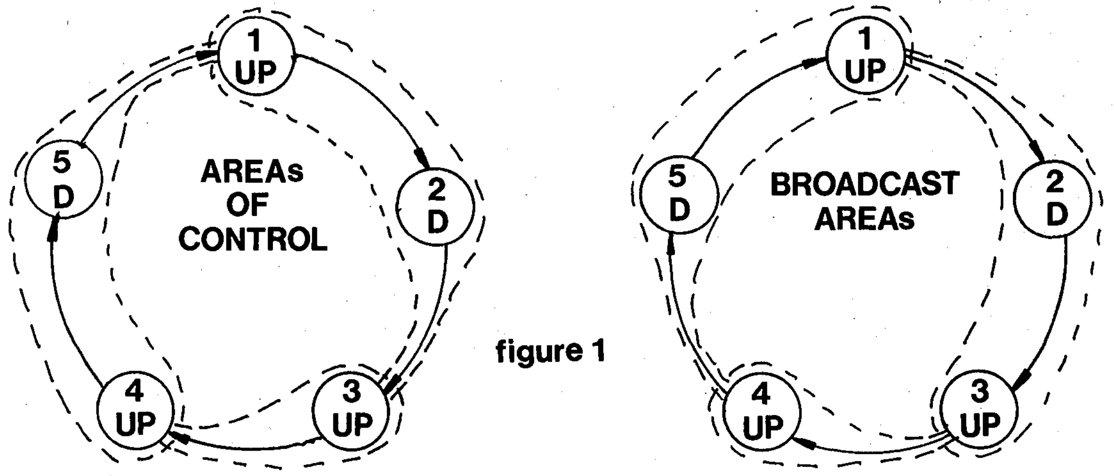


figure 1

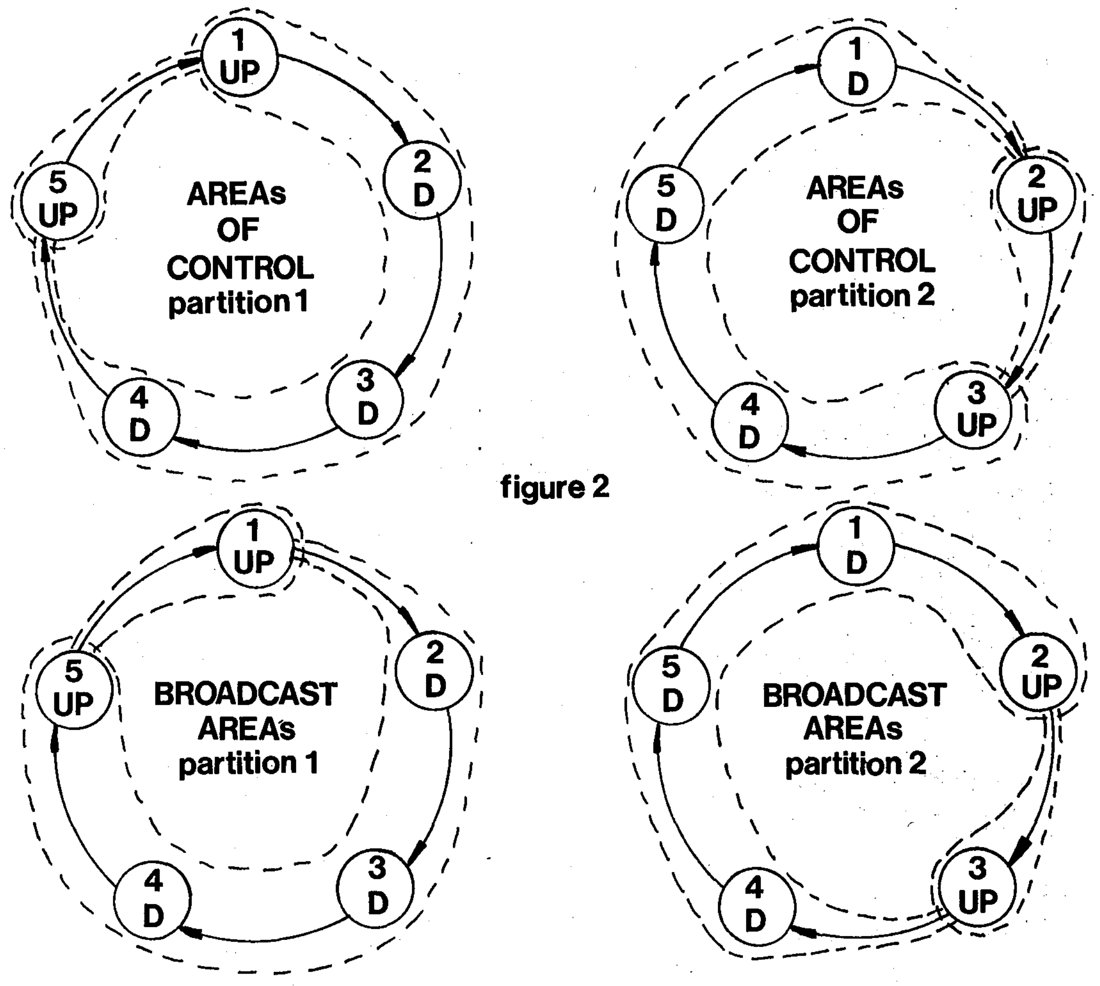

1 UP
2 D
3 D
4 D
5 UP
AREAs OF CONTROL partition 1
1 D
2 UP
3 UP
4 D
5 D
AREAs OF CONTROL partition 2
1 UP
2 D
3 D
4 D
5 UP
BROADCAST AREAs partition 1
1 D
2 UP
3 UP
4 D
5 D
BROADCAST AREAs partition 2

figure 2

Appendix 2: Algorithmic Description of the RSC-Protocol

In the following the algorithmic description of the complete RSC-protocol will be given, using a PASCAL-like notation. The protocol is presented in an idealized form and not embedded into a special environment like the kernel of a distributed operating system.

CONSTANTS

```
const LOCID = identifier of the local site;
      N = number of sites in the network;
      LMAX = maximum message delay between two arbitrary sites;
      LMIN = minimum message delay between two arbitrary sites;
      AMAX = maximum time a site needs to generate a response;
      SINV = time period between the sending two successive HIs;
```

DATA TYPES

```
type state = (UP,DOWN);
     area = array [1 .. N] of boolean;
     timestamp = to be specified system dependent;
     version = record TIME: timestamp; ID: integer end;
     statetable = record
                    VERSION: version;
                    TABLE: array [1 .. N] of state
                  end;
     event = (TCONTROL-OUT, TSENDER-OUT, TCHANGE-OUT, TCHECK-OUT, HI-IN,
              REQUEST-STATE-IN, STATE-REPORT-IN, STATETABLE-IN);
```

VARIABLES

```
var NEXTEVENT: event; CHECK-VERSION: version; EMPTY: boolean;
    AREA-OF-CONTROL, BROADCAST-AREA: area;
    CURRENT-STATE, NEW-STATE: statetable;
```

TIMERS are defined by their name and their individual timeout period (TX (INV) defines a timer TX with timeout period INV).

```
timer TCONTROL (SINV + LMAX - LMIN); TSENDER (SINV);
      TCHANGE (2•LMAX + AMAX); TCHECK (LMAX - 2•LMIN);
```

MESSAGES are defined by their name and the data-structure they contain. Messages may be interpreted as variables, the values of which are transferred between sites.

```
message
   HI = record
           SENDER : integer;              (Sender's identifier)
           MSG-TIME : timestamp;          (Actual timestamp)
           ST-VERSION : version           (Timestamp of sender's
        end;                                      CURRENT-STATE)
```

```
  REQUEST-STATE = record SENDER: integer;        (Sender's identifier)
                          MSG-TIME : timestamp  (Actual timestamp)
                        end;
 STATE-REPORT = record SENDER: integer;          (Sender's identifier)
                        REQ-TIME: timestamp;     (Timestamp of request)
                        MSG-TIME: timestamp      (Actual timestamp)
                      end;
 STATE-TABLE = record SENDER: integer;           (Sender's identifier)
                       MSG-TIME: timestamp;      (Actual timestamp)
                       STATE: statetable         (New state table)
                      end;
```

SYSTEM PROCEDURES AND INTERNAL RSC-PROCEDURES

```
procedure SYNCCLOCK (T: timestamp);
  (Sets clock to the maximum of its current value and the value of T)
function NEWTIMESTAMP: timestamp;
  (Generates an actual timestamp)
procedure START-TIMER (T: timer);
  (Start timer T. If T has already been started, then restart T)
procedure RESET-TIMER (T: timer);
  (Reset timer to its zero-position)
procedure AWAITEVENT (NEXTEVENT);
  (Returns event with highest priority)
procedure FORGETEVENT;
  (Delete all events except TSENDER-OUT and REQUEST-STATE-IN and reset
   TCONTROL and TCHECK to their zero-position)
procedure SEND-MESSAGE (M: message, DEST: integer);
  (Send message M to site DEST)
procedure GENAREAS;
   var I: integer;
   (Initialize AREA-OF-CONTROL)
   begin for I := 1 to N do AREA-OF-CONTROL [I] := false;
     I := LOCID; EMPTY := false;
     while CURRENT-STATE [I] = DOWN do begin
       if I = 1 then I := N else I := I - 1;
       AREA-OF-CONTROL [I] := true end;
     if I ≠ LOCID then AREA-OF-CONTROL [I] := true else EMPTY := true;
   (Initialize BROADCAST-AREA)
     for I := 1 to N do BROADCAST-AREA [I := false;
     I := LOCID;
     while CURRENT-STATE [I] = DOWN do begin
       if I = N then I := 1 else I := I + 1;
       BROADCAST-AREA [I] := true end;
     if I ≠ LOCID then BROADCAST-AREA [I] := true;
   (Restart TSENDER and broadcast HI)
     HI.SENDER := LOCID; HI.MSG-TIME := NEWTIMESTAMP;
     for I := 1 to N do if BROADCAST-AREA [I] then SEND-MESSAGE (HI, I);
     START-TIMER (TSENDER)
   (If AREA-OF-CONTROL contains an UP-site then restart TCONTROL)
     if ¬EMPTY then START-TIMER (TCONTROL)
   end;
```

CONTROL, CHANGE AND RSC

```
procedure CONTROL
  var CHECK: boolean; I: integer;
  begin CHECK := false;
    while true do begin
      AWAITEVENT (NEXTEVENT);
      case NEXTEVENT of:

      TSENDER-OUT:
        begin HI.SENDER := LOCID; HI.ST-VERSION := CURRENT-STATE.VERSION;
          HI.MSG-TIME := NEWTIMESTAMP;
          for I := 1 to N do
            if BROADCAST-AREA [I] then SEND-MESSAGE (HI, I);
          START (TSENDER)
        end;

      TCONTROL-OUT: begin RESET-TIMER (TCONTROL); FORGETEVENT; CHANGE end;

      TCHECK-OUT: begin CHECK := false; FORGETEVENT; CHANGE end;

      HI-IN:
        begin SYNCCLOCK (HI.MSG-TIME);
          if AREA-OF-CONTROL [HI.SENDER]
            then if STATETABLE [HI.SENDER] = UP
              then begin if HI.ST-VERSION > CURRENT-STATE-VERSION
                then begin CHECK := true; CHECK-VERSION := HI.ST-VERSION;
                  START-TIMER (TCHECK) end;
                START-TIMER (TCONTROL) end
              else begin CHECK := false; FORGETEVENT; CHANGE end
            else ignore
        end;

       STATE-TABLE-IN:
         begin SYNCCLOCK (STATE-TABLE.MSG-TIME);
           if STATE-TABLE.STATE.VERSION > CURRENT-STATE.VERSION
             then begin CURRENT-STATE := STATE-TABLE.STATE; GENAREAS;
               if CHECK then if CURRENT-STATE.VERSION ≥ CHECK-VERSION
                   then begin CHECK := false; RESET (TCHECK) end
         end end;

       REQUEST-STATE-IN:
         begin SYNCCLOCK (REQUEST-STATE.MSG-TIME);
           STATE-REPORT.SENDER := LOCID;
           STATE-REPORT.REQ-TIME := REQUEST-STATE.MSG-TIME;
           STATE-REPORT.MSG-TIME := NEWTIMESTAMP;
           SEND-MESSAGE (STATE-REPORT, REQ-STATE.SENDER)
         end;

       STATE-REPORT-IN: begin ignore end

  end end end;
```

```
procedure CHANGE;
  var ACTIVE: boolean; COUNT: integer;
  begin ACTIVE := true; COUNT := 0;
    REQ-STATE.SENDER := LOCID;
    REQ-STATE.MSG-TIME := NEWTIMESTAMP;
    for I := 1 to N do
      if I ≠ LOCID then SENDMESSAGE (REQ-STATE, I)
    START-TIMER (TCHANGE);
    for I := 1 to N do NEW-STATE [I] := DOWN;
    while ACTIVE do
      begin
        AWAITEVENT (NEXTEVENT);
        case NEXTEVENT of:

        TSENDER-OUT: see CONTROL;

        REQUEST-STATE-IN: see CONTROL;

        HI-IN: ignore;

        STATE-TABLE-IN:
          begin SYNCCLOCK (STATE-TABLE.MSG-TIME);
            if STATE-TABLE.STATE.VERSION > CURRENT-STATE.VERSION
              then begin CURRENT-STATE := STATE-TABLE.STATE;
                GENAREAS end
          end;

        STATE-REPORT-IN:
          begin NEW-STATE [STATE-REPORT.SENDER] := UP;
            COUNT := COUNT + 1;
            if COUNT = N - 1
              then begin RESET-TIMER (TCHANGE);
                ACTIVE := false end
          end;

        TCONTROL-OUT:
          begin ACTIVE := false end

      end;

    STATE-TABLE.TABLE := NEW-STATE;
    STATE-TABLE.MSG-TIME := NEWTIMESTAMP;
    for I := 1 to N do
      if NEW-STATE [I] = UP then SEND-MESSAGE (STATE-TABLE, I);
    CURRENT-STATE := NEW-STATE; CURRENT-STATE [LOCID] := UP;
    GENAREAS
  end;


procedure RSC;
  begin CHANGE; CONTROL end;
```

A QUORUM-BASED COMMIT PROTOCOL

Dale Skeen

Computer Science Department
Cornell University
Ithaca, New York

Abstract

Herein, we propose a commit protocol and an associated recovery protocol that is resilient to site failures, lost messages, and network partitioning. The protocols do not require that a failure be correctly identified or even detected. The only potential effect of undetected failures is a degradation in performance. The protocols use a weighted voting scheme that supports an arbitrary degree of data replication (including none) and allows unilaterally aborts by any site. This last property facilitates the integration of these protocols with concurrency control protocols. Both protocols are centralized protocols with low message overhead.

1. Introduction

A transaction is, by definition, an atomic operation on a distributed database system. Either all changes by the transaction are permanently installed in the database, in which case the transaction is said to be committed, or no changes persist, in which case the transaction is said to be aborted. It is the task of a commit protocol to ensure that a transaction is atomically executed.

In this paper we propose a commit protocol that is resilient to multiple occurrences of the following classes of benevolent failures: arbitrary site failures, lost messages, and network partitioning. It does not require that the type of failure be correctly determined, in fact, resiliency is guaranteed even if failures go undetected.

The protocol uses a weighted voting scheme to resolve conflicts during failures. When failures occur, a transaction is committed only if a minimum number of votes, called a commit quorum and denoted V_C, are cast for committing. Similarly, in the presence of failures, a transaction will be aborted only if a minimum number of votes, called an abort quorum and denoted V_A, are cast for aborting. A commit quorum does not have to equal an abort quorum, but their sum must exceed the total number of votes.

Voting schemes have been proposed previously for transaction management. Thomas introduced a majority voting scheme to ensure consistency in a fully replicated database ([THOM79]). Gifford extended the scheme by assigning weights to sites and using quorums rather than a simple majority ([GIFF79]). The proposed protocol differs from the previous work in several important ways:

(1) It is a commit protocol, not a concurrency control scheme. It provides atomicity at a per transaction basis. Nonetheless, it is straightforward to integrate any type of concurrency control protocol into this

protocol.

(2) It allows unilateral aborts during the first phase of the transaction. A site may decide to abort because of several reasons, for example, a deadlock is detected locally.

(3) It is primarily intended for partially replicated distributed databases where a transaction can read from any copy but must update all copies.

In addition, the protocol exhibits the following properties:

(1) It is a centralized protocol and, thus, benefits from the economy of centralized protocols.

(2) In the absence of failures it is no more expensive than previously proposed protocols that are resilient only to coordinator failures (and not to a partitioning of the network).

(3) If all failures are eventually repaired, then the protocol will eventually terminate.

(4) It is a blocking protocol -- operational sites must occasionally wait until a failure is repaired. This is an undesirable but necessary property exhibited by any protocol that is resilient to network partitioning ([SKEE81a]). However, the protocol can be tuned so that the frequency of blocking is low.

This paper is divided into six sections. The second section states our assumptions and defines the terminology used in the remainder of the paper. The third section develops a resilient quorum-based commit protocol, and the fourth section develops a resilient quorum-based recovery protocol. The recovery protocol is invoked whenever a group of sites can no longer communicate with the original coordinator (either it has failed or the network has partitioned). Like the commit protocol, it is a centralized protocol. The fifth section discusses performance, and the sixth section concludes the paper.

Although the protocols proposed are resilient to many classes of failures, this paper will focus on the problem of network partitioning. This class of failures is generally agreed to the most difficult class to handle. The other two classes, site failures and lost messages, can be cast as special cases of a partitioned network. In a site failure, a single site is isolated (partitioned) from the remainder of the network. A lost message can be viewed as a very short lived partitioning. In all cases, the protocols work without modifications.

2. Background

We assume that an underlying communications network provides point-to-point communication between any pair of sites. We also assume that it generates no spontaneous messages, and that garbled messages are detected and deleted. We do not assume that messages arrive in order nor that it detects lost messages.

A *partitioned network* occurs when there are two or more disjoint groups of sites where no communication is possible between the groups. Each of the disjoint groups is called a *partition*.

A distributed transaction T is decomposed into subtransactions T_1, T_2, ..., T_N, where a subtransaction is executed at one of the N participating sites. Any subtransaction can be unilaterally aborted, which results in the abortion of the entire transaction. Hence, for transaction T to be committed, *all* sites must agree to commit their subtransaction. We assume that a subtransaction can be atomically executed by a local transaction management system ([GRAY79,LIND79]).

It is the responsibility of a *commit protocol* to ensure that all subtransactions are consistently committed or aborted. One of the simplest commit protocols is the two-phase protocol ([GRAY79, LAMP76]) depicted in Figure 1. The protocol uses a central site, the coordinator, to direct the execution of the transaction at the other sites. Each slave has a chance to abort the transaction by replying with a "no" in the first round.

A commit protocol can be conveniently described by a set of state diagrams, one for each participating site ([SKEE81a]). The diagram for Site i describes the processing of subtransaction T_i. A state in the diagram is called a *local transaction state*.

In the two-phase commit protocol, a single state diagram (illustrated in Figure 2.) suffices to describe processing at all sites. For both the coordinator and the slaves, there are four distinct and easily identified local transaction states: the *initial state* (state q in the diagram), the *wait state* (w), the *abort state* (a), and the *commit state* (c). A site occupies the *initial* state until it decides whether to unilateral abort the

COORDINATOR	**SLAVE**
(1) Transaction is received. *Subtransactions* are sent to each slave.	
	Subtransaction is received. A reply is sent: *yes* to commit, *no* to abort.
(2) If all sites respond *yes* then *commit* is sent; else, *abort* is sent.	
	Either *commit* or *abort* is received and processed.

Figure 1. The two-phase commit protocol.

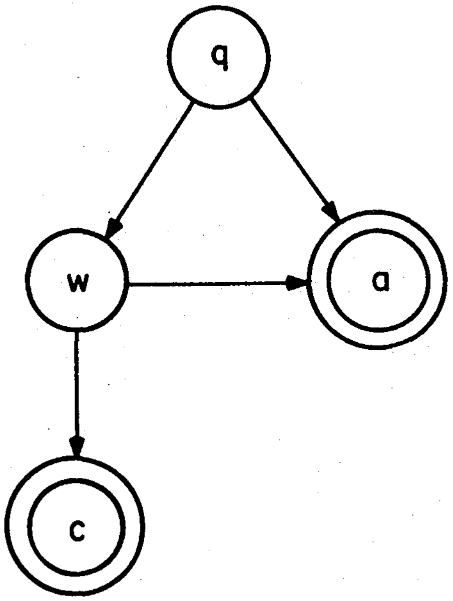


Figure 2. The state diagram for the two-phase commit protocol.

transaction. If the site decides against an abort, then the wait state is entered. This state represents a period of uncertainty for the site, where it has agreed to proceed with the transaction but does not yet know its outcome (i.e. committed or aborted). The commit and abort states are self-explanatory.

The local transaction states of any protocol form two disjoint subsets: the committable states and the noncommittable states. A site occupies a committable state only if all sites have agreed to proceed with the transaction. For example, the only committable state in the two-phase commit protocol is the commit state. A state that is not a committable state is a noncommittable state.

3. A Resilient Commit Protocol

The two-phase commit protocol is not a very robust protocol. Whenever the coordinator fails or becomes partitioned from the slaves, the slaves must block until the failure can be repaired.

In this section we develop a very resilient commit protocol that allows recovery from both of these types of failures. The section develops the commit protocol in detail; the next section discusses the associated recovery protocols for handling coordinator failures and partitioning.

Each site is assigned an integral nonnegative number of votes. (The number can be 0, in which case the site is a passive participant.) The basic idea is that whenever a group of communicating sites establishes a quorum, they are allowed to proceed. There are two distinct types of quorums - a commit quorum and an abort quorum.

Let V, V_C, and V_A represent the total number of votes, the number required for a commit quorum, and the number required for an abort quorum. A resilient quorum-based protocol must obey the following properties ([SKEE81c]):

(1) $V_C+V_A>V$ where $0<V_C,V_A<=V$

(2) When any site is in the commit state, then at least a commit quorum of sites are in committable states.

(3) When any site is in the abort state, then at least an abort quorum of sites are in noncommittable states.

These requirements are sufficient to ensure that a quorum-based protocol terminates in a consistent state -- if it does terminate ([SKEE81c]).

```
COORDINATOR                                              SLAVE'S RESPONSE

(1) Transaction is received.
    Subtransactions are
      sent to each slave.

                                                             Yes to commit
                                                              No to abort

(2) If all sites respond yes
      then
        prepare to commit is sent;
        continue to phase (3)
      else
        abort is sent;
        stop.

                                                                  Ack

(3) If the sum of the weights
    of the responding sites equals
    or exceeds V_C
      then
        send commit to all
      else
        block (wait until a "merge").

                                                                  --
```

Figure 3. The quorum based commit protocol.

The requirements are very similar to those for k-resiliency where a protocol can tolerate upto k arbitrary site failures (see [ALSB76] for a definition of k-resiliency and [SKEE81b] for a set of sufficient conditions ensuring k-resiliency in a commit protocol). In both cases a minimum number of sites must agree before an irreversible decision is made by any site.

The second requirement can be viewed as two subrequirements:

(2.1) Before the first site commits, a commit quorum of sites in committable states must be obtained, and

(2.2) After any site has committed, a commit quorum must be maintained.

As a consequence of (2.2), a site can safely move from a committable state to a noncommittable state if and only if it can be shown that no site has committed the transaction, or it can be shown that this will not destroy a commit quorum.

The third requirement, concerning abort quorums, is analogous to (2). Hence, there exists (3.1) and (3.2) which are the analogs of (2.1) and (2.2).

The two-phase commit protocol does not satisfy the second rule, nor can it be simply extended to satisfy it. Moreover, any protocol which has a single committable state (which must be the commit state) cannot satisfy the rule. Hence, in a quorum-based commit protocol, we need to introduce a new committable state, the prepared to commit (pc) state. This state will substantially increase the cost of the protocol, but unfortunately, it is necessary.

The new protocol is described in Figure 3 and its state diagram is given in Figure 4. It requires three phases to commit, two to abort. The new phase is the second phase, where all sites move into the prepared to commit state. The only explicit mention of quorums is in the third phase where the transaction is committed only if a commit quorum of sites advance to the prepared to commit state. Even though abort quorums are not explicitly mentioned, the third requirement is still satisfied. In fact, if any site unilaterally aborts (including the coordinator), then no site ever enters a committable state and the third rule is trivially satisfied.

The protocol is a pessimistic protocol -- if any site fails or a partition occurs during the first phase, then the coordinator immediately aborts the transaction.

4. Recovery

There are two aspects of recovery. When a group of sites is partitioned from the rest of the sites, they will execute a protocol that attempts to form a quorum and terminate the transaction. These protocols, called termination protocols are discussed in the first part of this section. If a quorum can not be achieved within the partition, then the sites must block until communication between partitions is restored. Once this is achieved, the sites within the new partition can execute a merge protocol and reattempt terminating the transaction.

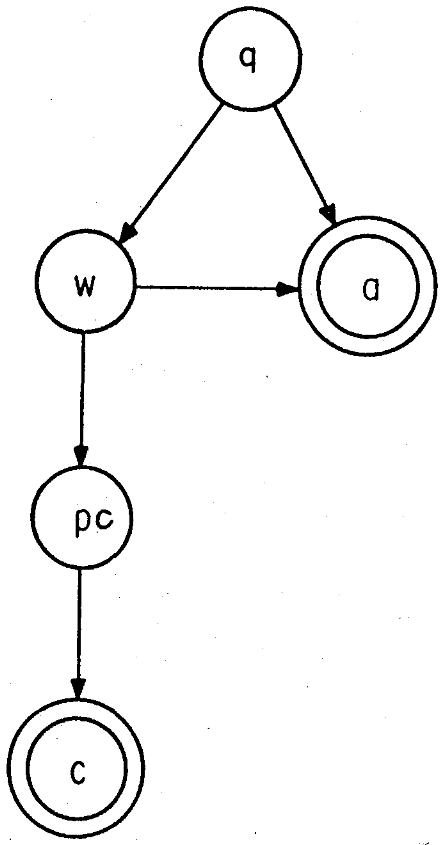


Figure 4. State diagram for the quorum based commit protocol.

Termination Protocol

As with the commit protocol, the major emphasis in the proposed protocol is on successful termination. Partially executed transaction will be aborted, when necessary, to achieve this goal.

When a group of sites detect that they are partitioned from the remainder of the network, they execute a two part termination protocol. The first part consists of electing a surrogate coordinator and the second part consists of an attempt to form a quorum.

There are several possible election protocols. We will not explicitly discuss election protocols except to note that it is possible to elect a unique coordinator at linear cost ([GARC81,HAMM79]). The resilency of a quorum-based protocol is not dependent on the uniqueness of the outcome of the election. Even if two surrogates are chosen, resiliency is guaranteed but performance suffers.

When the election completes the surrogate executes a protocol similar to the commit protocol in the previous section. The termination protocol is slightly more complex for two reasons. First, a surrogate works with less knowledge than the original coordinator, specifically, the surrogate may not know if a transaction is committable. Second, whereas there was a single coordinator originally, there many be many surrogates each operating in different partitions.[1]

For the first problem, a surrogate can attempt to form a commit protocol only if a site within the partition is in the committable state. For the second problem, a surrogate must explicitly form abort quorums. A site indicates its willingness to participate in an abort quorum by moving into a _prepared to abort_ state.

The termination protocol is given in Figure 5. Like the commit protocol, it consists of three phases. In the first phase the surrogate coordinator polls the sites about their local state, and these replies determine the action taken in the next two phases. If any site has committed

---

COORDINATOR

(1) Request local state.

(2)

slave responses	_coordinator's actions_
≥1 commit	send commit; terminate
≥1 abort	send abort; terminate
≥1 prepared to commit and weights of wait and prepared to commit $\geq V_C$	send prepare to commit continue with (3a)
weights of wait and prepared to abort $\geq V_A$	send prepare to abort continue with (3b)

```
(3a) if ≥$V_C$ ack's then send commit
     else block
(3b) if ≥$V_A$ ack's then send abort
     else block
```

(Slaves respond with their local state in Phase 1 and with an acknowledgement in Phase 2).

Figure 5. The quorum-based termination protocol.

---

[1]Or even in the same partition if the election protocol fails to uniquely elect a surrogate.

(aborted), then the transaction is immediately committed (aborted) at all sites. Otherwise, the surrogate will attempt to establish a quorum.

A commit quorum is possible if at least one site is in the prepared to commit state and the sum of the weights of the sites occupying the prepared to commit state and the wait states is at least V_C. If this is the case, the surrogates will attempt to move all sites in the wait state into the prepared to commit state. Barring additional failures, the surrogate will then commit the transaction.

However, additional failures may prevent sites either from making the transition or from acknowledging the transition. If an insufficient number of acknowledgements is received, then the protocol blocks.

An abort quorum is possible if the sum of the weights of sites occupying the wait state and the prepared to abort state is at least V_A. Unlike a commit quorum, an abort quorum does not require any sites to occupy the prepared to abort state. Again, the surrogate attempts to move an abort quorum of sites into the prepared to abort state -- aborting the transaction if it is successful, blocking otherwise.

The state transition diagram is given in Figure 6. A heavy line indicates the normal movement of the site into a "prepared" state and then into the corresponding final state. A dashed line indicates a path taken when the site is not a participant in the formation of the quorum.

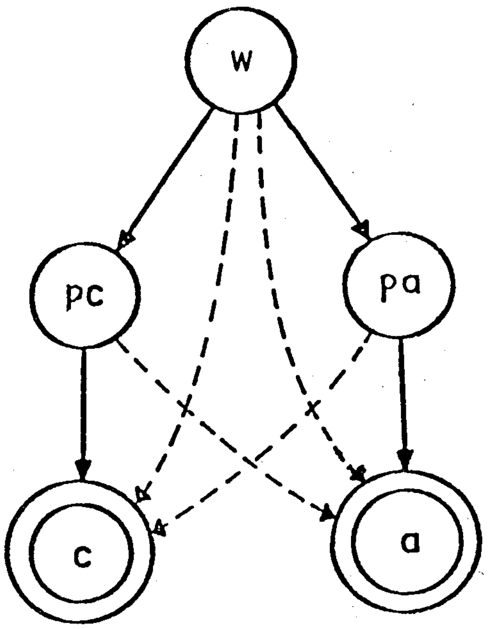


Figure 6. State diagram for the termination protocol.

Merging

Partition merging occurs whenever a failure is repaired and communication is established between two or more partitions. We assume that the re-establishment of communication paths is detectable[2].

The recovery strategy for merging is simple: execute the termination protocol described in the last section. In this case the election process can be streamlined -- the new coordinator can be chosen from among the old coordinators, e.g. let the coordinator with lowest site number become the new coordinator. The new coordinator then executes the three phases in the second part of the termination protocol.

Site recovery is equally simple -- it is a special case of merging where one partition contains a single site.

5. Performance

It is very difficult to analyze the expected performance of quorum-based protocols, even if very simple and independent probability distribution functions are used to describe site failures. For nonzero failure probabilities, it is clear that the worst case performance is unbounded, which is expected from the results of the Two Generals Problem (see [GRAY79] for an description of this problem and its ramifications).

However, we argue that if all partitions are eventually resolved, then the protocols will eventually terminate. They are acyclic, hence every state transition moves a site closer to termination, and they are deadlock free. This latter property is assured by the choice for the quorum sizes -- after the merging if all partitions, it must be the case that either an abort quorum or a commit quorum can be formed.

In environments where failures are rare, the most important cost measure is the cost of the commit protocol in the absence of failures. The quorum-based commit protocol requires 3 phases, 5 end-to-end message delays, and about 5N messages (where N is the number of participants). This cost is substantially higher than the cost of the two phase commit protocol -- higher by approximately 50%. However, the two-phase protocol is not very resilient. A more resilient protocol, specifically one that is resilient to a coordinator failure, requires at least three phases. While several three phase protocols are known ([GARC79, SKEE81b]), the quorum-based protocol is the only one resilient to network partitioning.

There are two sets of parameters that determine the performance of the protocol in the presence of failures: the weights assigned to individual sites, and the values for V_C and V_A.

The assignment of weights is often influenced by policy considerations external to implementation of the system. However, some factors that are relevant to performance are percentage downtime, failure rate, and percentage of data stored at the site. The most intuitive rule is to assign weights inversely proportional to the percentage downtime.

---

[2] A low level protocol can periodically attempt communication with other sites. Eventually it will detect the repair of the partition.

In choosing quorum sizes, it is not necessary for V_C to equal V_A. In fact, there are several strong arguments for choosing $V_C > V_A$. One argument concerns protocols allowing unilateral aborts: if a significant number of transactions are unilaterally aborted, then clearly V_A should be smaller. A stronger argument is that most site failures are expected to occur during Phase 1 of the commit protocol since most of the transaction execution time is spent in Phase 1. This phase is time consuming because the majority of the data processing takes place during it; whereas, Phase 2 and Phase 3 synchronize state information among the sites and require very little local processing. If sites fail during Phase 1, then the transaction must be aborted -- hence, it should be easy to abort.

An interesting heuristic for choosing V_A is based on a rough estimate of the failure distribution of the sites. This heuristic is useful in environments where site failures, rather than network partitions, predominate. Let $P(V_A)$ be the probability that at least an abort quorum is operational. $P(V_A)$ is a decreasing function in V_A. The point is to choose the maximum V_A such that $V_A <= V_C$ and $P(V_A)$ exceeds a minimum level of desired availability.

As mentioned before, the weight of a site can be zero, in which case the site contributes nothing toward forming a quorum. (However, such a site can still unilaterally abort the transaction.) When designing a protocol, a zero-weighted site can be eliminated from all phases requiring the formation of a quorum. In the extreme case, where only a single site has a non-zero weight, a quorum based commit protocol degenerates into the standard two-phase protocol with all of its disadvantages. Specifically, all sites must block on the failure of the only nonzero weighted site (which is normally the coordinator).

6. Conclusion

The use of quorums is a standard recovery technique for handling network partitioning (even primary site schemes, e.g. [STON79], are a degenerate case of using quorums). We have presented a very general quorum-based commit protocol that can be used with both replicated and nonreplicated data. Unlike previous schemes it allows a single site to unilaterally abort the transaction.

Quorum-based protocols are resilient because a site is allowed to participate in only one type of quorum. Quorum sizes are carefully chosen such that the formation of both a commit and an abort quorum requires the participation of a common site. In this way mutual exclusion is assured -- only one type of quorum can be formed during the execution of a transaction. (However, it is possible for multiple occurrences of a single type of quorum to be formed. For example, since abort quorums are usually small, more than one can be formed concurrently.) In such a scheme the concurrent execution of several coordinators, even if they are within the same partition, does not destroy consistency.

When a new coordinator is elected in the proposed recovery protocol, it polls all sites about their current local state. In making a commit decision, only the replies from the latest poll is used -- information obtained in earlier polls is ignored. Less conservative approaches which uses previous information can be found in [SKEE81c].

REFERENCES

[ALSB76] Alsberg, P. and Day, J., "A Principle for Resilient Sharing of Distributed Resources," Proc. 2nd International Conference on Software Engineering, San Francisco, Ca., October 1976.

[GARC79] Garcia-Molina, Hector, Ph.D. Thesis, Stanford University, 1979.

[GARC81] Garcia-Molina, Hector, "Elections in a Distributed Computing System," TR No. 280, Princeton University, December, 1980.

[GIFF79] Gifford, David, "Weighted Voting for Replicated Data" Operating Systems Review, 13, 5, Dec., 1979, pp. 150-9.

[GRAY79] Gray, J. N., "Notes on Database Operating Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[HAMM79] Hammer, M. and Shipman, D., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," Computer Corporation of America, Cambridge, Mass., July 1979.

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Storage System," Tech. Report, Computer Science Laboratory, Xerox Parc, Palo Alto, California, 1976.

[LIND79] Lindsay, B.G. et al., "Notes on Distributed Databases," IBM Research Report, no. RJ2571 (July 1979).

[SKEE81a] Skeen, D. and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System," IEEE Transactions on Software Engineering, (to appear).

[SKEE81b] Skeen, D., "Nonblocking Commit Protocols," SIGMOD International Conf. on Management of Data, Ann Arbor, Michigan, 1981.

[SKEE81c] Skeen, D., "Crash Recovery in a Distributed Database System," Ph.D. Thesis, University of California, Berkeley (in preparation).

[STON79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," IEEE Transactions on Software Engineering, May 1979.

[THOM79] Thomas, Robert, "A Majority Consensus Approach to Concurrency Control," Transactions on Database Systems, 4, 2, June 1979.

IMPLEMENTATION OF DISTRIBUTED TRANSACTIONS

Deborah J. DuBourdieu

Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Mass. 01701

ABSTRACT

This paper explores some of the issues encountered in the design and implementation of distributed transactions in an ongoing project at Prime Computer, Inc. An important feature of the concurrency control algorithm to be discussed is that retrieval-only transactions never wait. Implementation details relevant to this feature include the maintenance of a pool of previous data images, and more complex synchronization requirements in a distributed environment. Another important point discussed is an optimization of the Two-Phase Commit protocol in the event of coordinator failure. Finally, the design of an IPC suitable for support of distributed transactions is examined.

1 INTRODUCTION

A major strength of the Prime product line is Primenet(TM), which provides complete local and remote network communication services for all Prime systems [GORD79]. In geographically dispersed network configurations it allows Prime computers to communicate with other Prime computers, with computers from other vendors, and with terminals and computers attached to packet switching networks. In local network configurations, Primenet allows Prime computers to be attached via a high-bandwidth, multi-point ring arrangement to other Prime systems.

Another strength of the Prime product line is our CODASYL-compliant DBMS with full recovery, interactive database administration, and interactive query language/report writer. Part of our strategy in Data Management is to build upon these strengths by developing expertise in the issues of distributed DBMS. We have concentrated first upon remote data access and distributed transactions, and the system which incorporates this work is discussed below. Areas for future work include improved distributed schema management and optimization of distributed queries.

2 PRIME DATA MANAGEMENT ACCESS METHOD

This block-oriented access method is used by all Prime data management products which provide the service of transaction management, including concurrency control and data recovery. It is composed of several internal subsystems. The Data Manager provides I/O to local and remote data files. The Resource Request Manager is a general subsystem for the management of arbitrary types of locks on arbitrary objects, including control of deadlock. The Communication Manager provides a high-level interface to Primenet, our X.25-compatible network service. The Recovery Manager runs during system restart and makes use of log information left by the Transaction Manager to bring all databases into a consistent state following a system halt. Finally, the Recovery Manager can also bring databases into a consistent state following a disk crash by using log information written by the Transaction Manager.

The Transaction Manager uses the services of all the other internal subsystems to provide control of both local and global (or distributed) transactions. This paper will concentrate on some of the implementation problems of providing that service in a distributed environment. By "distributed environment" is meant one in which machines are connected by a local or long-haul network, so that users can access files which reside on their own machine or a remote machine, transparently to the application program. A "distributed transaction" is one which accesses data stored at more than one site. It is still an atomic unit of work, so that it sees a globally consistent database, and its updates are either installed in the permanent database at all sites or at none of them. In our current implementation data files may not be stored redundantly.

3 TRANSACTION MANAGEMENT IN LOCAL ENVIRONMENT

3.1 Synchronization Via Two-Phase Locking

The synchronization technique we use is two-phase locking [ESWA76] and the unit of locking is the disk page. Each read which is performed must be preceded by the acquisition of a read-lock on the page, which can be shared with other read-locks. No write may be performed on an item which is read-locked. Each write must be preceded by the acquisition of a write-lock, which cannot be shared with any other locks. Once a transaction has released a lock, it may not acquire any further locks, which is enforced in our implementation by the rule that locks must be held until the transaction completes.

Read-only transactions do not need to obey this protocol and never interfere with update transactions. All that is required for the read-only transaction is that reads be repeatable (computationally equivalent), seeing the output of the same write operation each time the same read is performed. The mechanism whereby this is accomplished is discussed in a later section.

3.2 Data Recovery

The recovery of data files after a soft or hard crash also depends on the transaction mechanism. The runtime side of this work is a variant of the DO-UNDO-REDO protocol [GRAY78], which we implement as two separate logs. Before entering an uncommitted update of a page into the permanent data file, the Transaction Manager must write an image of the current version of that data page into the UNDO log. If a soft crash occurs before the transaction is committed or is aborted at runtime, the Recovery Manager will use that image in the UNDO log to roll back the transaction, leaving the data file in the state that it was before the updater started, and therefore consistent. Before executing commit, which tells the user that the transaction definitely completed, the Transaction Manager must write an image of all of its updates to the Redo log. If a disk crash occurs later, the Recovery Manager uses those images to redo or roll forward the transaction, leaving the data file in the same (consistent) state that it was left by the original update transaction.

3.3 Multiple Version Variant

We employ an optimization of this technique which uses multiple versions of data items. An early developer of this technology was Christopher Earnest, currently of Prime Research. Multiple versions in the context of timestamp-ordering is discussed in [BERN81], and a design similar to ours is found in [BAYE80].

The "current database" at any moment of time is defined as the database which would result if all unterminated processes were to be aborted [ROSE78]. It is consistent because it contains only the output of committed transactions, which were controlled by the serializing 2PL scheduler. We distinguish between those transactions which only issue reads (Readers) and those transations which also issue writes (Updaters). We introduce the rule that for a Reader, the current database at the time the Reader starts is the database the Reader should continue to see throughout the life of the transaction. This rule is correct (yields serializable execution histories). The important implication of this rule is that if the current database is kept available to the Reader throughout its life, the Reader will never have any reads rejected by the concurrency control, regardless of update activity. Notice that Readers will not see the <u>most</u> current database at every point in time, which may be an important factor in some applications.

Updaters do not participate in this optimization. If Updater U reads the previous version of an item being updated by concurrent Updater V, then U must precede V in the ordering generated by the 2PL scheduler. This requires remembering all read-write conflicts which have occurred, in some cases even after the transactions have completed, adding extra complexity and overhead which we believe would more than cancel the benefit of the read optimization.

Each data item can be thought of as having its complete update history available (though in actuality this history is maintained only as far back as is necessary to satisfy possible requests). These previous versions are actually the same versions generated by Updaters for the UNDO log, so no extra work is done except to preserve the old images as long as there is a Reader who might make use of them. Transactions receive unique monotonically increasing transaction numbers which are interpreted as version or generation numbers. When an update is performed, the new value of the data item is stamped with the transaction number of the transaction which wrote it. When a Reader begins it is given a heavily encoded list of all transactions whose output is legal for it to read, namely, all those who had committed before the Reader started. This defines the Reader's current database.

When a Reader performs a read, the transaction number in the data block is checked to see whether it is on the Reader's list. If it is not, a chain of pointers to previous versions which begins at the data block is followed backwards in time until a version is found whose transaction number does appear on the Reader's list.

3.4 Deadlock Prevention Via PAD

If an Updater U requests a write-lock on a page which is currently read-locked by Updater V, rather than deny U and force it to abort, we permit U to wait for V to terminate and release the lock. Since V in turn may try to wait for another lock, possibly one held by U, the potential for deadlock is introduced.

Deadlocks are detected by looking for cycles in a directed graph which represents the relation "which transactions are currently waiting for which other transactions" (WAITS-FOR). We use the progressive acyclic digraph algorithm described by [HANS79]. Nodes represent transactions and an arc between T2 and T1 represents the state that T2 is waiting for T1 (because T1 has a resource, in this case a lock, that T2 wants). An arc may be added to the graph unless it would cause a cycle, in which case one or more nodes must be detached (those transactions have been pre-empted and their locks have been released). To optimize the detection of cycles, a topologically sorted list is kept of all nodes which have arcs. If transaction T2 wishes to wait for transaction T1 and T2 precedes T1 in this ordering, then there is no possibility of a cycle and no further work need be done. If T2 follows T1 in the topologic sort, then we must determine the hard way (by computing the transitive closure) whether T1 is really waiting for T2, since the ordering could simply be an artifact of the order in which the transactions arrived. This offers a significant performance improvement over computing the transitive closure for every case.

A timestamp which is retained across restarts is used to prevent cyclic restart (suggested by [ROSE78]). Suppose that T2 wants a lock which is currently owned by T1. If T2 can wait without causing deadlock then it will wait. If for T2 to wait would cause deadlock then either T2 must be refused (nonpreemptive) or T1 must be aborted (preemptive). We use the timestamp to help make the choice, selecting the transaction with the youngest timestamp for refusal or preemption. Since every transaction will eventually become the oldest transaction, cyclic restart and indefinite postponement are avoided.

4 TRANSACTION MANAGEMENT IN DISTRIBUTED ENVIRONMENT

4.1 Oriented Toward Site Autonomy

Our design is oriented towards decentralization and site autonomy. There is no central site for synchronization purposes, restart at each site is independent, and the effect of a failure at one point in the network is restricted as much as possible.

Each site continues to maintain its own series of transaction numbers and its own UNDO and REDO logs, so that management of a stricly local transaction operates in precisely the same way. A distributed transaction is implemented as a group of local transactions, one at each site the distributed transaction visits, including the originating site. The Transaction Manager at the originating site has extra responsibilities in that it must send requests against remote data files to the correct site using the Communications Manager, in addition to satisfying requests against local data files, and it must coordinate events at start and end of transaction. For this reason the orginating site and its local transaction are referred to as the Master, and all the other local transactions are referred to as "cohorts."

Since there is no central site dispensing transaction numbers, there is no bottleneck. Furthermore, system restart at each node is completely independent (except for unresolvable global transactions), since each site has its own UNDO log.

This design works very efficiently for global Updaters, who simply acquire a transaction number at any given local site when they first access a data file at that site. The mapping of a group of local transaction numbers into a single global update transaction is not needed at runtime, when the master and cohorts need only the information about one another which is necessary to send messages to each other. However, the mapping is written to the log for use by the Recovery Manager during system restart.

Global Readers, who still use the optimization of previous images, face a new problem. They must acquire not just one list of completed transactions, but a list at every site where they access data, and all of these must be consistent. The problem is that if a global Reader is in the process of acquiring these lists while a global update transaction is in the process of ending, the Reader may acquire a set of lists in which one cohort of the Updater may be marked as complete while another one may not yet have terminated and is therefore listed as incomplete. The global Reader would then see inconsistent data. Therefore, we synchronize this activity via a lock which must be obtained by global Readers when starting and by global Updaters when ending. It can be shared by any number of Readers or any number of Updaters but not between Readers and Updaters. We are investigating a more efficient alternative in which global Readers are synchronized by timestamp ordering.

4.2 Distributed locks

Each site has its own Resource Request Manager, and manages all locks on data residing at that site. This has the advantage of incurring the minimum overhead associated with acquiring locks, since it does not have the bottleneck problem and communication cost of a central lock manager site. The disadvantage is that the local algorithm for deadlock detection becomes insufficient because deadlock can now occur indirectly through a chain of waiting processes which crosses nodes.

4.3 Deadlock prevention via WOUND-WAIT

We handle deadlock via the hybrid WOUND-WAIT algorithm [ROSE78].

Deadlock handling consists of two questions: First, will letting T1 wait for T2 cause a cycle in the graph representing waiting processes (deadlock detection)? Second, if so, who is going to die (restart protocol)? In the local case we have all the information necessary to accurately answer the first question, and the second is answered using the timestamp in order to prevent cyclic restart. In the distributed case the first question cannot be answered accurately without sending all information about locks to some designated site. Instead a very crude heuristic is used, which is to presume that allowing global transaction T1 to wait for a younger global transaction T2 at one site might cause a global cycle. This determination is made by comparing the two transactions' timestamps, which are unique across the network. As in the local case, the restart protocol selects the younger transaction for restart, and does so by sending WOUND messages to all sites where that transaction has visited. In the stronger definition of WOUND, this message is interpreted as an abort when received at the active site, if termination has not yet started. However, such quick execution is not necessary to prevent deadlock. Under the alternate definition of WOUND, wounded processes may continue processing until they try to wait, at which point deadlock is again possible, and the wounded process must be aborted. Since forcing the older transaction to wait a while longer gives the wounded transaction one chance to complete, we use this definition.

We further reduce the total number of restarts by distinguishing between local transactions (those which have not left their original site) and global transactions (those which have). For local transactions deadlock continues to be handled in the same way as described in the previous section, using the locally computed WAITS-FOR relation, while global transactions are handled using WOUND-WAIT.

There are two costs in this method. First, some transactions will be aborted unnecessarily (younger ones who would not have caused a cycle in a centrally maintained WAITS-FOR graph). Second, when a transaction is wounded, extra messages (the WOUND message) are sent to all sites which the transaction has visited.

5 RECOVERY FROM SYSTEM HALT

The goal of this mechanism is to ensure that the system contiues to run, and to run correctly despite component failure and recoveries. A highly resilient network is thus more reliable in that it can be expected to:

1. Provide continued limited service at individual nodes in the presence of other failures, and

2. Maintain database consistency.

What "limited service" means is that new transactions may be run which are directed against data which is still accessible either locally or through the network. Of course, transactions which were interrupted by a failure elsewhere may still have resources (including locked data) allocated to them at the functioning node. This is a concern because when a failure occurs elsewhere in the network, there is a distinct possibility that the failing component will be out of service for a very long time, hours or days perhaps. We want those sites which are still functioning to be able to continue providing the best service possible under those conditions.

For the purposes of this analysis an active transaction has only two states: execution and termination. Termination is defined as the process of ending a transaction, which implies that it is either aborted or committed. "Commit" is defined as the act of installing all of a transaction's updates in the permanent database; "abort" is defined as undoing any updates performed by the transaction; both are followed by forgetting all information about the transaction except that it has terminated.

If a crash occurs during execution there is no doubt that the proper action is to abort the transaction at all sites. If a crash occurs during termination, the proper action is not immediately evident, since some cohorts may have successfully committed while the failed one may have lost some updates that were in volatile storage.

The Two-Phase Commit protocol, developed by Lampson and Sturgis in [LAMP76] and by Gray in [GRAY78], addresses this problem. Gray's "coordinator" is the Master in our implementation, and his "participants" are cohorts in our terminology. The coordinator sends a message to each participant, "Prepare to commit." Each participant then enters a state in which it is capable of either committing or rolling back the transaction, and then sends an acknowledgement back. When the coordinator has received all the acknowledgements it sends the message "Commit" to each participant, which then terminates. The participants terminate and send a final acknowledgement to the coordinator, who then terminates.

5.1 Optimization on Two-Phase Commit

A similar optimization was independently discovered by the SIRIUS DELTA Distributed Database System project of INRIA, and was mentioned briefly in their presentation at the Fifth Berkeley Workshop on Distributed Data Management.

In the original statement of the protocol above, an important implication is that if the coordinator fails, then the participants must wait for the Recovery Manager at the coordinator's site to get started and direct the participants to the conclusion of the transaction. Depending on what caused the crash at the coordinator site and how long it takes to fix it, the participants could wait for an appreciable period of time, during which all the locks held by those participants must remain held since it is not known whether the transaction has committed. This wait is not necessary in the event that all the other participants have survived and can communicate with each other. In this case they can each exchange all the information they have about the interrupted transaction. If any participants received a commit, they will all decide to commit. If none heard a commit, they will all decide to abort. If not all participants survived, then the survivors will have to wait as in the original protocol.

5.2 Pseudo-code for Participant Side of Two-Phase Commit

This follows the statement of the protocol in [GRAY78]. "IF COORDINATOR FAILS" is really a condition which is detected rather than a conditional statement.

```
PARTICIPANT_2_P_C:  PROCEDURE
   WAIT_FOR 'PREPARE TO COMMIT'
   IF COORDINATOR FAILS
      THEN ABORT AND RETURN
   ELSE
      FORCE UNDO REDO LOG TO NONVOLATILE STORE
      IF SUCCESS THEN
```

```
         REPLY 'AGREE'
      ELSE
         REPLY 'ABORT'
      END IF
   END IF
   WAIT_FOR VERDICT
   IF COORDINATOR FAILS
      WHILE TRANSACTION IS UNRESOLVED DO
         FOR EACH PARTICIPANT IN THIS GLOBAL TRANSACTION DO
            SEND (PARTICIPANT, 'DID YOU HEAR COMMIT?')
            RECEIVE (PARTICIPANT, 'YES/NO')
         END FOR
         IF ANY PARTICIPANT REPLIED 'YES' THEN
            COMMIT AND RETURN
         ELSE
            IF ALL PARTICIPANTS REPLIED 'NO' THEN
               ABORT AND RETURN
            ELSE
               WAIT FOR FAILED PARTICIPANTS OR
               COORDINATOR TO RECOVER
            END IF
         END IF
      END WHILE
   END IF
   IF VERDICT = 'COMMIT' THEN
      DO
      RELEASE RESOURCES AND LOCKS
      REPLY 'ACKNOWLEDGE'
      END
   ELSE
      DO
      UNDO PARTICIPANT
      REPLY 'ACKNOWLEDGE'
      END
   END PARTICIPANT
```

The system restart logic gets a bit more complicated at the coordinator site as well. Instead of merely examining the coordinator's log, the Recovery Manager must consider the possibility that in recovering from this particular crash, the participants made a decision on their own. The Recovery Manager can easily handle this possibility by querying any other site involved in the transaction as to what it believes the state of the transaction to be. The site will reply "Committed", "Undone", or "In doubt." At the coordinator site, then, the Recovery Manager will react to the first two responses by doing likewise, and to the third response by controlling resolution of the transaction as outlined in the original protocol.

This optimization introduces a subtle implementation problem, namely, that the participants and coordinator must share the same perception of whether the coordinator has failed. Otherwise the participants may be attempting to recover on their own while the coordinator is still attempting to end the transaction. In theory, failure of the coordinator should arise from only one cause, failure of the node where it resides. This is easy enough to determine since communication with that node becomes impossible. However, the coordinator may appear to have ceased responding when in fact the network is so slow that messages take much longer to arrive than expected, when the network loses messages, or when there is a system error at the coordinator site. All of these ugly cases must be accounted for in a commercial product.

6 PROBLEMS WE ENCOUNTERED, THINGS WE LEARNED

6.1 Cohorts and Master are Cooperating Processes

When moving from local to distributed transaction design, it is natural for the designer to continue to think of the system in terms of central control of execution. The physical end-user can only be at one site, so there is a central source of database commands. The role of the coordinator and participants in Two-Phase Commit looks very much like a master-slave relationship.

We believe now that a more accurate intuition about distributed transactions is that they are composed of cooperating processes. The fundamental reason for this is that exceptional events can occur out in the boondocks which require the cohorts to take action on their own. To the extent they can do so, the whole system is more robust. In this paper we have already seen an example of one such event, failure of the coordinator site. Under the optimization we have used, the cohorts each take charge of attempting to resolve their part of the transaction. Other examples are the various system errors that can occur asynchronously, including transaction timeout.

6.2 Desirable IPC

The factors discussed above carry implications for the underlying communications mechanism. In keeping with our original philosophy of "extending" the master process to remote sites where it acquires a slave process, our Communications Manager was based on the network primitive of remote procedure call. We would now prefer to use high-level network primitives composing an IPC that facilitates peer-peer communication. The ideal IPC for our purposes is sufficiently high-level that all details about the physical nature of the network, such as virtual circuit reset, are masked. It includes the following primitives:

GET_NODE_NAME (OBJECT, NODE_NAME)

For OBJECT which is known to the naming server (file system), return the NODE_NAME at which it is found. NODE_NAME is also known to the network and the IPC. This primitive is used by the Data Manager to determine the location of data files in the network.

Strictly speaking, this primitive is not part of an IPC at all. It is included as a reminder that the IPC must be designed to work in cooperation with the global naming server.

CREATE PROCESS (NODE, PROCESS_NETWORK_ADDRESS, ERR_STAT)

Create a complete process on machine NODE and return the process' "network address," which uniquely identifies the process throughout the network. Anybody who knows the process' network address can send messages to it without further ado in the same way that anybody who knows your street address can send you a letter. This address is an attribute of the process and is available to the network and the OS in the process control block. Some of the possible error returns are "No more processes available at NODE," "NODE is unavailable on the network." This primitive is used by the master to create cohorts.

TERMINATE_PROCESS (PROCESS_NETWORK_ADDRESS, ERR_STAT)

TERMINATE PROCESS tells the recipient to do all necessary clean-up and log itself out. There must be an error return whereby the recipient can indicate that it cannot terminate and why. Although from the IPC user's point of view TERMINATE could just as well be done via SEND like any other message it is probably easier for the network and OS to handle it as an explicit primitive, since its consequences are major. This primitive is used by the master to log out the cohorts.

SEND (PROCESS_NETWORK_ADDRESS, BUFFER, ERR_STAT)

SEND can handle any message of arbitrary length which is placed in its buffer. The semantics of the message must be determined by the recipient. Typical messages are "Execute this procedure", "Here's your data record," and "I heard a Commit." The options on SEND should reflect the underlying network in a useful way. For example, in our network it is possible to specify whether you wish to wait for your message to leave your machine before proceeding, thus giving an extra degree of confirmation. This should appear as an option on SEND. If the network provides a high priority message type for network users (as opposed to the network itself) this should be an option. Possible error is "Addressee does not exist."

RECEIVE (SYNCH/ASYNCH, BUFFER, PROCESS_NETWORK_ADDRESS, ERR_STAT)

In executing this primitive, the SYNCH option is the usual case of procedure execution where the procedure does not return until a message arrives in BUFFER. The ASYNCH option allows immediate return from the procedure. The network will put the message in BUFFER when it arrives, and it is up to the IPC user to poll the buffer to determine that event. A value for

PROCESS_NETWORK_ADDRESS as an input argument is optional. When given a value on input it means that the only message which should be placed in BUFFER is a message from that process. All other messages are to be treated as unexpected (see below). A null value on input means receive from any process. Its value on output is the network address of the process which sent the message currently in BUFFER. The network can determine the identity of the sender from its process control block and package it with the message transparently to the sender.

BROADCAST (LIST_OF_PROCESS_NETWORK_ADDRESSES, BUFFER, ERR_STAT)

There are several points in the Two-Phase Commit protocol where the same message is sent to all cohorts at the same time. While the functionality of BROADCAST can easily be built by the IPC user out of SEND, BROADCAST should be included as a primitive if the underlying network has such a facility (as does ours).

SEND_SIGNAL (PROCESS_NETWORK_ADDRESS, BUFFER, ERR_STAT)

RECEIVE_SIGNAL (PROCEDURE_NAME, BUFFER, SENDER_PNA)

The IPC must handle unsolicited messages, some of which are user interrupts. "Unsolicited messages" are those which are not expected by the recipient, who may not have been in previous communication with the sender. This is where the concept of a process' network address becomes so useful, since Process A can send a message to Process B even if B is unaware of A's existence, as long as A knows B's network address. An example of this situation is WOUND, where A wishes to wound B. Other examples are cohort recovery from failure of the master site, and the reporting of unexpected errors such as timeout. The other aspect of the unsolicited message is that it is necessary to interrupt or awaken the recipient in order to direct it to examine the messsage. The IPC at the receiver site can do this by signaling the recipient. This could probably be implemented as a system-defined condition and handled by the recipient via a PL/I-style on-unit.

It is also possible that the recipient does not have a receive outstanding, in which case there is no buffer available to the network in which to place the message. For this reason the IPC has to implement a simple buffering mechanism, although it is likely that if IPC users abuse this facility by failing to use RECEIVE_SIGNAL or disabling the signal for long periods, performance will suffer significantly.

SEND_SIGNAL directs the IPC to deliver the message in BUFFER to PROCESS_NETWORK_ADDRESS and to send a signal to that process. If the recipient does not have a RECEIVE or RECEIVE_SIGNAL outstanding, the signal will be trapped by the system default on-unit which will handle it as a user error and dump the message from the IPC buffer to the user error file. RECEIVE_SIGNAL tells the IPC that if a message arrives, put it in BUFFER and raise the predefined signal, which will cause the recipient to jump to procedure PROCEDURE_NAME. The receiver must also be able to turn off the signal while in critical code, during which time incoming messages are buffered by the IPC. When the receiver turns signaling back on, the signal should be raised once per message as the receiver drains them from the IPC buffer.

7 CONCLUSION

Many services familiar in the context of a single-machine environment change character in the context of a distributed processing environment. In this paper we have examined such a service, distributed transactions, and its implications for another basic service, inter-process communication. We believe this exploration helps to lay the groundwork for the development of a high-quality distributed DBMS.

8 ACKNOWLEDGEMENTS

I am grateful to Gordon D. McLean, Jr., for critically reviewing a draft of this paper, and for his many insights into concurrency control while a colleague at Prime. Marguerite McGuire supplied valuable comments on the PAD algorithm. This design was refined and implemented by Bob Gray, Kriss Kellermann, Marguerite McGuire, Jeannie Nakano, and Howard Spilke.

REFERENCES

[BAYE80] Bayer, R., Elhardt, K., Heller, H., and Reiser, A., "Distributed Concurrency Control in Database Systems," in Proc. 6th Int. Conf. Very Large Data Bases, Oct. 1980.

[BERN81] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, June 1981.

[ESWA76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The notions of consistency and predicate locks in a database system," Communications ACM, Nov. 1976.

[GORD79] Gordon, R. L., Farr, W. W., and Levine, P., "Ringnet: A Packet Switched Local Network with Decentralized Control," Computer Networks, Vol. 3, No. 6, Dec. 1979.

[GRAY78] Gray, J. N., "Notes on database operating systems," in Operating Systems: An Advanced Course, vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978.

[HANS79] Hansen, W. J., "Progressive Acyclic Digraphs--A Tool for Database Integrity," Communications ACM, Sept.

1979.

[LAMP76] Lampson, B., and Sturgis, H., "Crash recovery in a distributed data storage system," Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif. 1976.

[ROSE78] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M., "System level concurrency control for distributed database systems," ACM Transactions on Database Systems, June 1978.

Mutual Consistency of Copies of Files based on Request Characteristics

Gautam Barua

Department of Computer Science
University of California
Santa Barbara, CA 93106

ABSTRACT

An algorithm to maintain mutual consistency of copies of a file in a distributed file system is presented. The number of up-to-date copies of a file varies according to the characteristics of the requests. When an update takes place, there is one such copy. As other nodes read, the number of copies grow till it again shrinks to one when another update takes place. An outline of a proof that the algorithm ensures mutual consistency and that there is no starvation of requests is presented. An informal discussion of the performance of the scheme is also presented.

Keywords and Phrases: distributed file systems, distributed data bases, concurrency control, mutual consistency.

1. Introduction

In recent years many algorithms have been proposed that enable concurrent reading and updating of a distributed data base to take place correctly. Typically, in a distributed data base(DDB) the data is composed of a number of entities that are placed in sites that are distributed geographically. In addition, to improve the read response times, some (or all) entities are replicated and distributed over the sites. An entity is the smallest data unit that can be "locked" for exclusive use. The read/update control algorithm (the *Concurrency Control* algorithm) has to ensure that two types of *Consistencies* are met during operation:

i) all the copies of an entity contain the same information (this is the problem of ensuring *Mutual Consistency*).

ii) the data base as a whole has "consistent" data (variously referred to as the problem of ensuring *consistency* or *external consistency*).

If the data base is *fully replicated* at each site, then only mutual

This research was supported in part under NSF Grant MCS80-04257.

consistency is of concern (consistency of data within a site still has to be ensured, but it is now a problem of a centralized data base and ways of achieving this can be found in [PAPA]).

A survey of most of the algorithms proposed for concurrency control can be found in [KOHL] and [WILM]. [WILM] attempts a classification and a comparison of a few representative algorithms. The terms introduced above are explained in more detail in [KOHL]. The series of articles on the concurrency control method used in SDD-1, a DDB, is another useful source ([BERN]). General notions on consistency and concurrency control in a centralized data base can be found in [ESWA] and [PAPA].

In this paper we propose a concurrency control algorithm to ensure mutual consistency in a distributed file system (DFS). A DDB can be viewed as a DFS where the entities are files. However, the algorithm described in this paper incurs a space overhead that may be deemed unacceptable in a DDB if the size of entities is small. In any case, in all DDBs proposed, the number of copies of an entity is assumed to be constant and are in fixed places. When an update to an entity occurs, all the copies of the entity are updated. We propose a scheme where the number of up-to-date copies of a file varies according to the characteristics of the requests to the file. In the general case (see [BAR2]), the number of copies (up-to-date or otherwise) can vary, but for the purpose of this paper we assume that a fixed number of copies of every file exists at predetermined sites.

In the sections to follow, the above idea is presented in greater detail, an algorithm to ensure mutual consistency of files is presented, a proof of correctness of the algorithm is outlined and an informal discussion of the performance of the algorithm is presented.

2. The Environment.

There are M computer systems (to be referred to as nodes from here on) interconnected in some fashion by a communication network. Each node has secondary storage units attached to it along with other devices. Operating systems running on each node are independent of one another except that they share a common file system. Files private to a node are accessed in the usual way. Access to files that are shared by more than one node have to be made via the algorithm to be described below. At any instant of time, only a subset of the M nodes may be sharing a particular file. Thus, the directory structure maintained at a node will point to private files and to files that the node is currently sharing with others. We are not going into the details of the directory structure in this paper. It is possible that a request for a file originates at a node where no copy of the file is present. The node may not even know where a copy of the file exists. In such a case, the node obtains a copy of the file by searching the network. Details of this procedure is discussed elsewhere ([BAR2]). For simplicity of exposition we assume here that requests for a file originate only in those nodes that have a copy of the file.

A transaction ([PAPA]) is a read or an update of a single file. All transactions are processed at the node where the transaction originates. Thus a "valid" copy (what makes a copy valid will become clear later) of a file has to be present in the node before a transaction on that file can be processed (note that there are no "writes" to files: only reads and updates (which is a combination of reads and writes)).

No assumptions are made about the interconnection structure other than that there exists a physical path from every node to every other node. Thus every node is (logically) connected to every other node.

We make the following assumptions regarding the operation of the system:

Assumptions

1) There are no failures either of nodes or links (physical interconnections).

2) All messages from one node to another reach their destination within a finite amount of time.

3) No messages are lost, either in transit (ensured by 1) and 2)), or inside a node (due to overflow of queues etc.).

4) Messages sent from node i to node j reach j in the order they were sent from i.

5) All reads and updates of files take a finite amount of time.

3. The Algorithm

It is a distributed algorithm in that every node in the system follows the same algorithm. Since the algorithm controls access to one file , there has to be a separate invocation of the algorithm for each shared file in the system.

The number of up-to-date copies of a file present in the system varies with demand. At the end of an update exactly one up-to-date copy exists: at the node where the update took place. As other nodes read the file each gets an up-to-date copy of the file and the number of such copies grows until the next update when it again shrinks to one. At any moment exactly one node is the "Master". A file can be updated by a node only if it is the Master. This idea of a floating Master is the same as the "migrating primary sites" used in [MINO] to describe a concurrency control algorithm based on "two-phased locking" which efficiently takes care of multiple copies of a file.

For simplicity of exposition, we assume that when a node needs an up-to-date copy of a file the whole file is transported across to it.

In an implementation, only portions of a file will be transported. This can be achieved by using "version numbers" on files.

The controlling program (<u>the controller</u>) is driven by events. The following events can occur:

1) Internal Read Request (IRR).

2) Read request from node i (RR(i)).

3) Message "GRANT READ"from node i (GR(i)). A copy of the file is sent along with this message.

4) Internal Update Request (IUR).

5) Update Request from node i (UR(i)).

6) Message "GO TO EXCLUDED" arrives from node j with the originator of the message being node i (EX(i,j)).

7) Message "BECOME MASTER" arrives along with the queue(Q) of the previous master (BM(Q)). The file is sent along with this message.

8) Update Complete (UC).

The controller in each node has the following variables to control access to the file:

SUBSTATE: SUBSTATE $\in$ {CURRENT,PENDING,EXCLUDED}

CURRENT: A valid copy of the file exists in the node.

PENDING: The node is waiting for a reply to an update/read it sent out or it is the Master servicing an internal update request.

EXCLUDED: The node does not have a valid copy and there are no pending requests.

FATHER: Contains the name of another node in the network. All requests that cannot be locally processed are sent to the node pointed to by FATHER. It is not used if the node is in substate PENDING or is the 'MASTER'.

MASTER: MASTER $\in$ {TRUE,FALSE}
At most one node can be the MASTER at any point in time. A node can process an update only if it is the Master.

SONS: A list of "sons" in the present configuration. It is meaningful only when the node is in substate CURRENT.

QUEUE: A queue of pending requests.

THIS: Number of this node. Each node has a unique number.

The structure of the processes in a node is shown in Figure 1. We present below the algorithm followed by the controller. It waits for an event to occur. When one does, the action taken depends on the event and the substate of the node. After the response to the event is complete, the queue in the controller is examined if the current substate is not PENDING. If it is not empty, the top element in the queue is made the next event. If it is empty (or the current substate is PENDING), a "processing complete" signal is sent to the event handler and the controller waits for the next event to occur. The event handler, on receiving the above signal sends the next event present in its buffer. Thus events arrive from the event handler and also from within the controller.

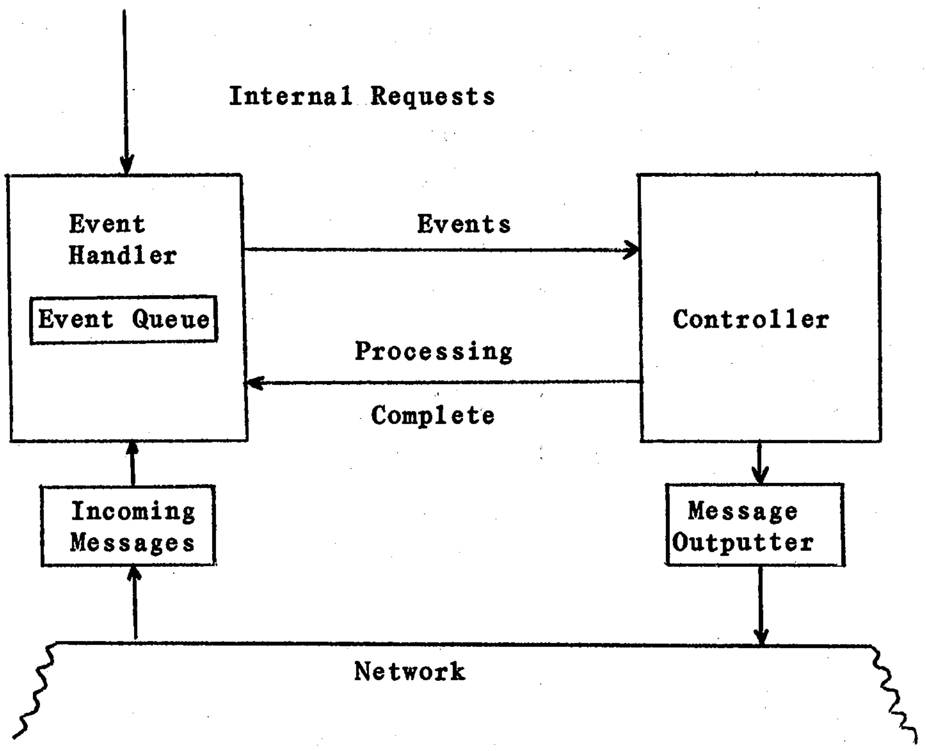


Figure 1 Structure of processes controlling access to a file in a node.

Algorithm I

```
Begin Program
 Loop Forever
  Wait(Event);
  Case SUBSTATE:
    SUBSTATE = CURRENT:
      Case Event:
        IRR     :
                  Service the request (Read);  {see note 1 below}
        RR(i)   :
                  Send message "Grant read" (GR(THIS)) to node i;
                  Add i to SONS;
        IUR     :
                  Send message "Go to Excluded" (EX(THIS,THIS))
                   to all SONS;
                  SUBSTATE := PENDING;
                  If MASTER = true then
                    Service the request (update);
                  Else
                    Send request to FATHER (UR(THIS));
                    Insert request into QUEUE;
                  Endif
        UR(i)   :
                  If MASTER = true then
                    Copy all "internal" requests in QUEUE
                     into TEMPQ; {a temporary queue}
                    Delete all these requests except the
                     first from QUEUE;
                    Send message "Become Master" (BM(QUEUE)) to i;
                    QUEUE := TEMPQ;
                    MASTER := false;
                    If QUEUE = empty then
                      SUBSTATE := EXCLUDED;
                      FATHER := i;
                    Else
                      SUBSTATE := PENDING;
                    Endif
                    If there is an update request in QUEUE then
                      {it must be an internal request}
                      Send EX(THIS,THIS) to all SONS;
                    Else
                      Send EX(i,THIS) to all SONS;
                    Endif {  for efficiency; see note 2 below}
                  Else {MASTER = false}
                    SUBSTATE := EXCLUDED;
                    Transmit request to FATHER;
                    FATHER := i;
                    Send EX(i,THIS) to all SONS;
```

```
                 Endif
        EX(i,j):
                 If FATHER ≠ j or MASTER = true then
                    Ignore;
                 Else
                    SUBSTATE := EXCLUDED;
                    FATHER := i;
                    Send EX(i,THIS) to all SONS;
                 Endif
     Endcase
```

<u>SUBSTATE</u> = <u>PENDING</u>

```
      Case Event:
        IRR    :
        RR(i)  :
        IUR    :
        UR(i)  :
                 Insert request into QUEUE;
                  {see note 4 below}
        GR(i)  :
                 FATHER := i;
                 SUBSTATE := CURRENT;
                 SONS := empty;
        EX(i,j):
                 Ignore;
        BM(Q)  :
                 MASTER := true;
                 Merge Q into QUEUE by "timestamp" order;
                  {see notes 3 and 5 below}
                 SUBSTATE := CURRENT;
                 SONS := empty;
        UC     :
                 SUBSTATE := CURRENT;
                 SONS := empty;
     Endcase
```

<u>SUBSTATE</u> = <u>EXCLUDED</u>

```
     Case Event:
        IRR    :
                 Transmit request  (RR(THIS)) to FATHER;
                 Insert request into QUEUE;
                 SUBSTATE := PENDING;
        RR(i)  :
                 Transmit request to FATHER;
        IUR    :
                 SUBSTATE := PENDING;
                 Insert request into QUEUE;
                 Send UR(THIS) to FATHER;
        UR(i)  :
                 Transmit request to FATHER;
```

```
                FATHER := i;
      EX(i,j):
                Ignore;
    Endcase
  Endcase
        {termination}
  If QUEUE not empty and SUBSTATE $\neq$ PENDING then
       Make "top" of QUEUE the next event;
       Remove "top" of QUEUE;
  Else
       Send signal "processing complete' to Event Handler;
  Endif
 Endloop

Endprogram
```

Notes

1) A request is serviced by waking up the relevant process waiting for the request to be granted. For a read request it is assumed that a copy of the file is made available to the process since the file itself may be updated by other requests while this request is still being serviced (if this is not feasible, then the controller should wait for the request to be complete before continuing processing as is done for updates).

2) If there is an update request in QUEUE then this node will become Master again "soon" so it is more efficient for nodes going into substate EXCLUDED to send their forthcoming requests to this node rather than to the new Master (and hence EX(THIS,THIS) rather than EX(i,THIS)).

3) To ensure fairness, each request has associated with it a "timestamp". A timestamp is generated by appending the number of the node where the request originates as the lower order bits to the current value of the local clock. To ensure that all timestamps are unique, we only require that the local clock be incremented at least once between two uses of it. See [LAMP] for more details on timestamps and ways of keeping the local clocks synchronised. The timestamp is generated at the time the request is submitted.

4) The insertion is done at a place such that the requests in QUEUE are in increasing timestamp order. However a request cannot be inserted at the top of QUEUE since the node will already have responded to the request at the top of QUEUE by sending out a message and the reply to this message must see the same request at the top of QUEUE for correct operation. So at any time the elements in QUEUE are in timestamp order except possibly for the request at the top of QUEUE.

5) When merging two queues by timestamp order, the top request in QUEUE must not be disturbed since the event in question (Become Master) is in response to this request. The situation is similar to that discussed in note 4 above.

6) In the algorithm as presented above, if the actions for an event are not specified for some substate then the occurrence of that event when the node is in that substate is an error condition. Thus, for example, the occurrence of the event GR(i) when the substate is CURRENT is an error condition.

7) When a node sends a particular message to "all sons" in response to a message from one of its sons it is assumed that it does not send the message to that particular son.

4. An Example

The way the above algorithm works can best be illustrated by an example. Consider the sequence of Figures 2(a) to 2(g). They depict the state of a typical system as time progresses starting with the state shown in Figure 2(a).

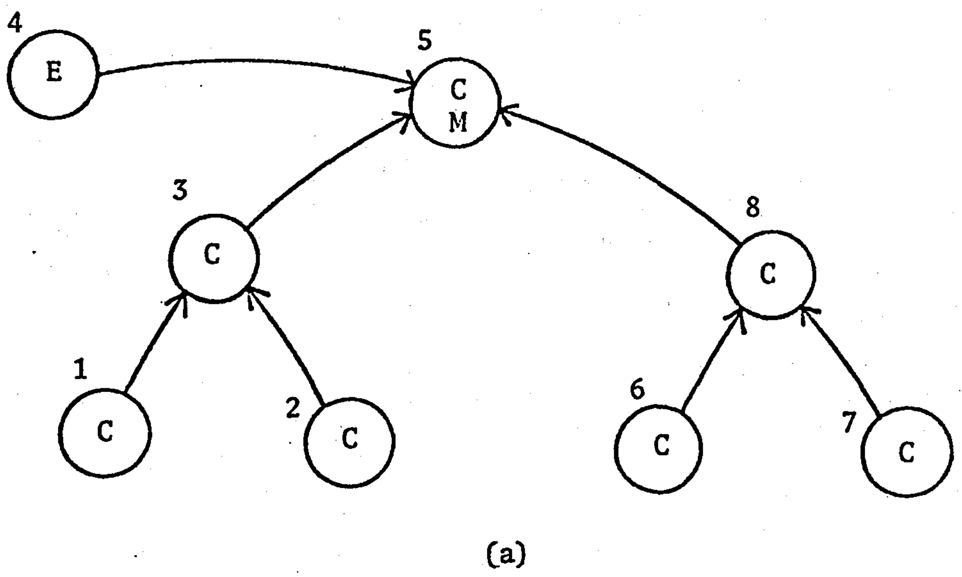
In the Figures, each node is labelled on the outside by a number (1,2,3,.. etc) which is the identifier of the node. The label inside a node, C(current), E (excluded), P (pending) specifies the substate of the particular node. A solid, directed edge points to the "Father" of a particular node. A dashed edge from i to j labelled B indicates that message B, sent from i, is in the communication network on its way to j. If a node has a second label "M" on the inside, then that node is the Master.

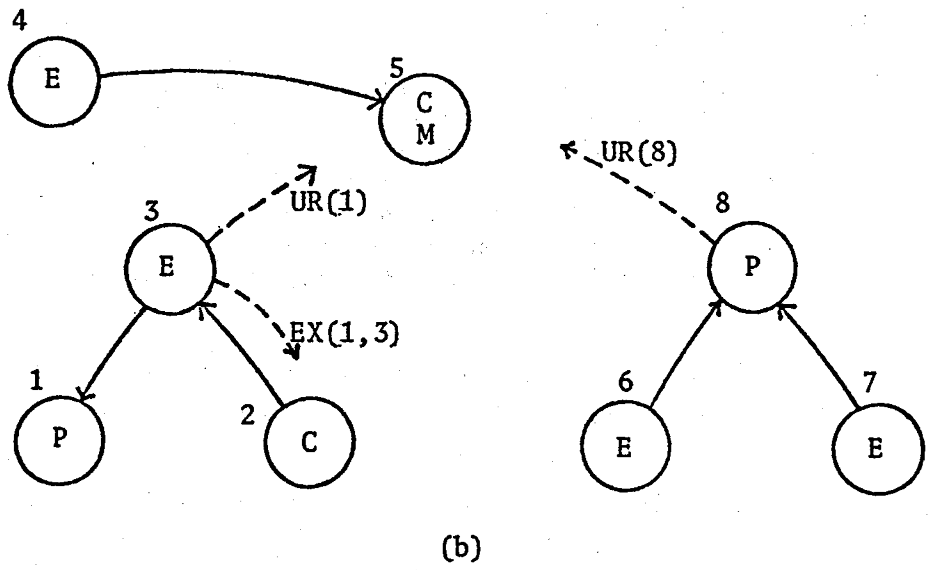
In Figure 2(a) node 5 is the Master and nodes 1,2,3,5,6,7 and 8 have valid copies of the file and are hence in substate CURRENT. Node 4 is in EXCLUDED substate.

Update requests originate simultaneously at nodes 1 and 8. Both the nodes go into substate PENDING and send the messages UR(1) and UR(8) respectively to their Fathers. Node 8 has Sons, so it "asks" them to go to substate EXCLUDED (message EX(8,8) is sent to both 6 and 7).

After some time the state depicted in Figure 2(b) is reached. By this time 3 has received UR(1) and in response to it has transmitted it to 5, has gone into substate EXCLUDED and has sent EX(1,3) to node 2. UR(8) has not yet reached 5.

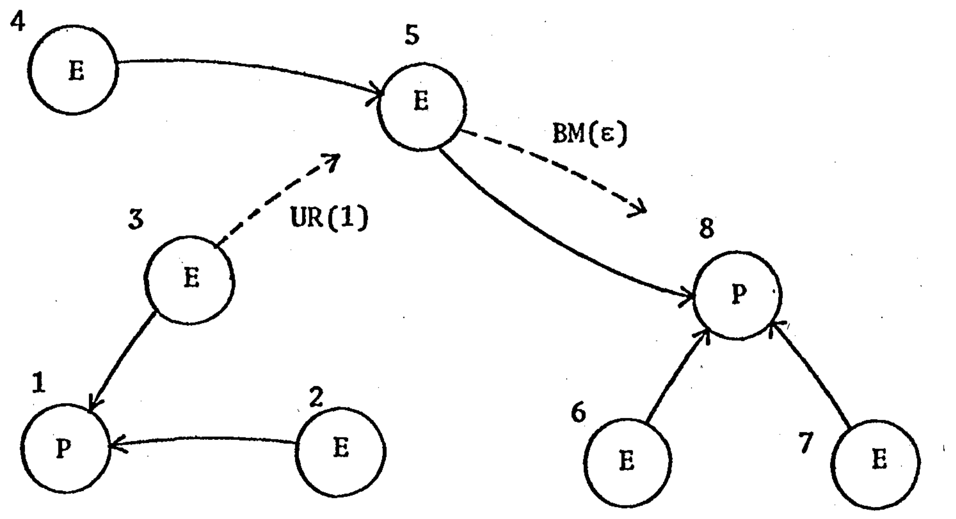
UR(8) then reaches 5. Node 5, on receiving it, goes into substate EXCLUDED, relinquishes the Master token and "asks" node 8 to become the Master (via message BM($QUEUE_5$); note that $QUEUE_5$ is empty). The situation at this point is depicted in Figure 2(c). The Figure also indicates that EX(1,3) has by now reached 2 which goes into substate EXCLUDED and makes 1 its Father in response.



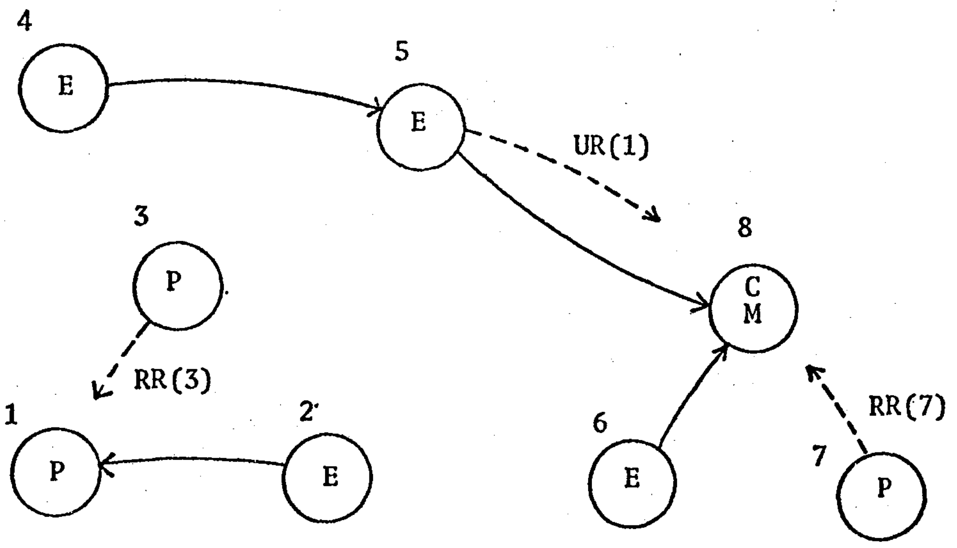


C - Current
P - Pending
E - Excluded
M - Master

Figure 2 An Example

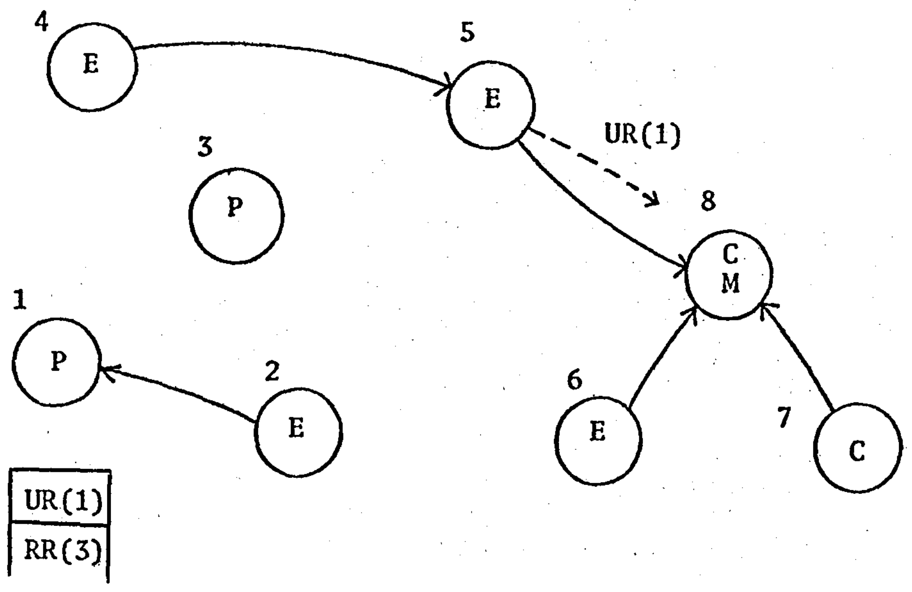


(c)

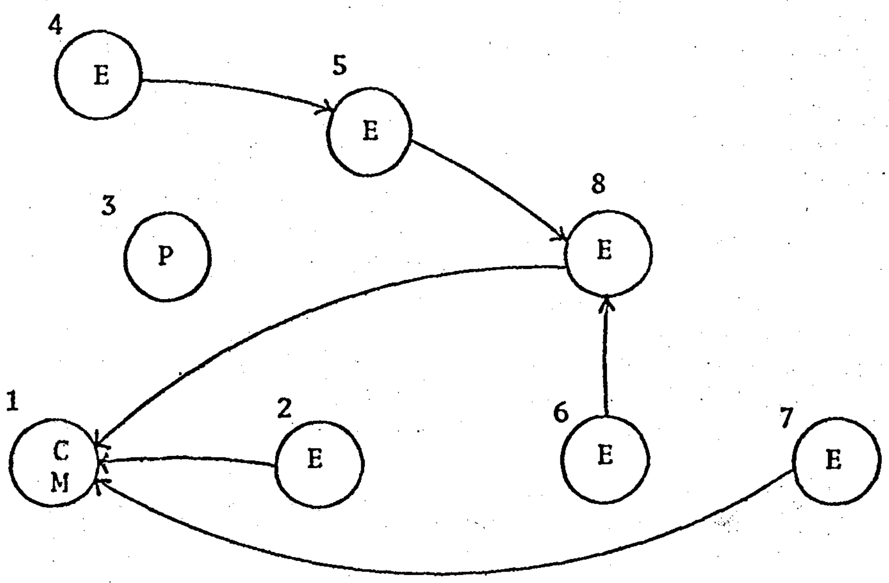


(d)

Figure 2 (contd)

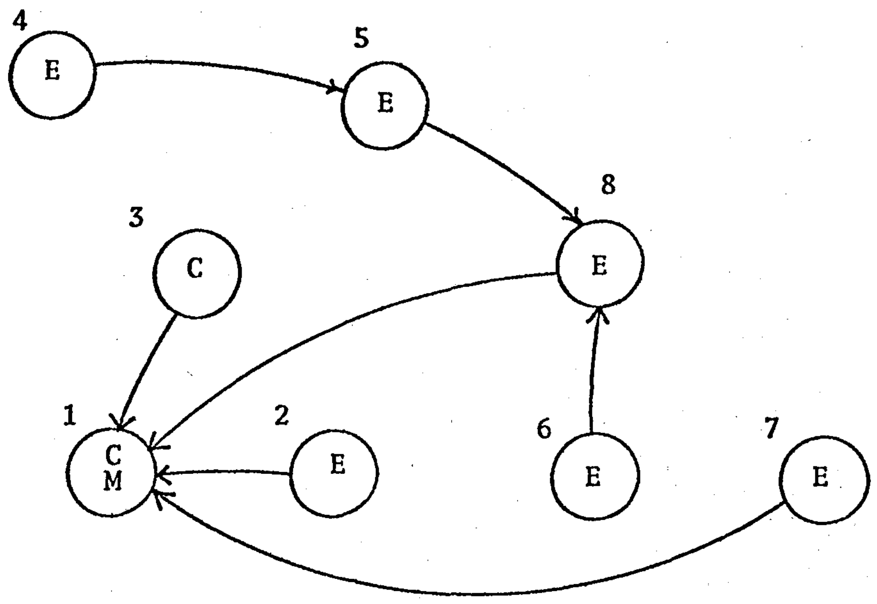


(e)



(f)

Figure 2 (contd)



(g)

Figure 2 (contd)

UR(1) then reaches 5 which merely transmits it to 8. Meanwhile, node 8 receives BM($QUEUE_5$), becomes the Master, does the update it was waiting on and goes into substate CURRENT. As this is going on, read requests originate in nodes 3 and 7. These nodes go into substate PENDING and send the messages RR(3) and RR(7) respectively to their Fathers. The situation at this point is shown in Figure 2(d).

RR(3) and RR(7) reach their destinations. On receiving RR(7), node 8 sends a copy of the file and a grant read (GR(8)) message to 7 which on receiving them goes into substate CURRENT and services the pending read request. When node RR(3) reaches node 1, the request is put into the queue in 1 (into $QUEUE_1$). Figure 2(e) illustrates this state. The queue at node 1 is drawn next to the node (strictly speaking, the queues at each pending node should be shown. However to avoid cluttering up the Figures, only queues with more than one entry are shown).

UR(1) finally reaches 8. Node 8 relinquishes the Master token in favour of 1, goes into substate EXCLUDED and asks its only son, node 7 to do likewise. Node 1, on receiving the "Become Master" message, services the pending update request and goes into substate CURRENT. It then finds RR(3) in its queue. So a valid copy of the file and a grant read message is sent to 3. The state at this point is shown in Figure 2(f).

When node 1 receives the grant read message, it services the pending read request, goes into substate CURRENT and we reach the state shown in Figure 2(g).

The above example has not been able to capture all the aspects of the algorithm. It only illustrates the salient features.

5. Proof of Correctness

We need to prove that the algorithm is correct with respect to the following properties:

1) Mutual Exclusion of updates is achieved.

2) There is no starvation of any requests.

The formal proofs will not be presented here. The interested reader should refer to [BAR1]. We present only the outlines here.

Mutual Exclusion

Update requests can be serviced at a node only if the node is the Master. Within a node, updates are serviced sequentially since a node goes into substate PENDING when an update begins and does not service any other request while this update is going on. Thus to prove 1), we need to prove that more than one node cannot be the Master at any instant. This is done by a simple case analysis.

While a node is updating a file another node could be reading the same file. A read need not therefore be of the most up-to-date copy of a file. Every node with a valid copy will however be informed ultimately of any update to the file. Thus the algorithm guarantees <u>weak</u> mutual consistency of the copies of a file.

<u>Starvation</u>

Consider an update request emanating from node i (UR(i)). If node i is the Master and in substate CURRENT, UR(i) gets service immediately according to the algorithm.

Now suppose i is not the Master and is in substate CURRENT or EXCLUDED. UR(i) is sent to the node pointed to by the variable FATHER, node i goes into substate PENDING and UR(i) is also placed at the top of QUEUE in i. We prove that in such a case UR(i) reaches a node other than i which, at that time, is in substate PENDING or is the Master, after a finite amount of time.

From the example presented above it should be clear that the "state" of the system at any instant corresponds to a forest with the roots of the constituent trees being nodes in substate PENDING or the node that is the Master. We prove by a case analysis that this is indeed the case. This result is then used to prove that UR(i) ultimately reaches a node in substate PENDING or with MASTER=true. If UR(i) reaches a node which is the Master and in substate CURRENT at that moment it gets service via a "Become Master" message that is sent to node i. Otherwise UR(i) is placed in QUEUE of some node j. At this point UR(i) will be in two queues: the one at the node of origin (i, in this case) and the one it has just been placed into (j).

If UR(i) had emanated from a node which was at that time in substate PENDING it would have been placed in QUEUE at i.

In any case UR(i) either gets service or is placed in QUEUE of some node j that is in substate PENDING.

So we prove that once an update request is placed in QUEUE of some node, it gets service within a finite amount of time. A distance measure is defined. The following example shows what this is: UR(i) is distance two if, UR(i) is in QUEUE of node j, node j is in substate PENDING, the request at the top of QUEUE in j is an update request from j itself which is also in QUEUE in node k and node k is the Master. A node gets service when its distance is zero. An induction on the distance of a request is used to prove that it gets service within a finite amount of time. The key to the proof is that requests are inserted into a queue by timestamp order and so at most a finite number of requests can "overtake" a particular request in a queue.

That read requests are not starved out can be shown similarly.

6. Performance Evaluation

N nodes use a file over a certain period of time ($N \leq M$, the total number of nodes). Now, because of the nature of the algorithm, at any point in time only n nodes have a valid copy of the file ($n \leq N$). If an update request originates in one of these nodes, every other node with a valid copy has to be informed and the Master has to send a grant message to the node (if the node is not already the Master). Thus n or n-1 messages have to be sent to complete an update. If on the other hand, a node without a valid copy wishes to update, more than n messages may be required. But in either case, at most N messages will be required. However, the price being paid is that read requests will also need to send messages if a valid copy is not present at the node where the request originates (contrast this with a scheme in which all nodes have a copy: reads are for "free" but updates are more "expensive"). Secondly, the read and update response times depends on the configuration. The read response time depends on the distance a node is from a node with a valid copy while the update response time depends on the distance of a node from the current Master.

The appeal of the algorithm lies in the fact that the configuration is dynamic. This implies that it is not totally dependent on some static characteristics of requests (one typically made in DDBs is that reads predominate writes), but can react to transients fairly adequately. Thus, for example, if within a certain period of time say 3 particular nodes do a series of updates while the other nodes are dormant, only these 3 nodes will participate in the operation, greatly improving performance over a strategy where every update has to be posted at all nodes. Thus, while the mean characteristic of the requests (where the requests originate, percentage of reads over updates) will affect the mean performance, the algorithm does not degrade from this behaviour when variations from this mean take place. So in environments where there is considerable fluctuation in the request characteristics over time, the algorithm becomes particularly attractive.

In order to get a quantitative measure of the performance, the system has been modelled as a Markov Chain and various performance measures have been obtained. The details of the analysis can be found in [BAR2]. Only the main results are presented here.

Let there be N nodes with a copy of a given file. We assume that requests are uniformly distributed over these N nodes and that the request arrival process is Poisson. Every request is a read request with probability p and an update request with probability q ($q=1-p$), independent of other requests.

If all N copies are kept up-to-date (we shall refer to any scheme that does this as an N-copy scheme), the number of messages required to service an update request is a standard performance measure. In Algorithm I, messages may have to be sent to service a read request also and so we define the <u>read cycle cost</u> as a performance measure. This is the number of messages that are sent by all nodes between two successive updates to the file. Note that the read cycle cost is the same as the cost to service an update in any N-copy scheme.

The second performance measure is the response time of a request in the absence of any other conflicting requests to the same file. Let $\overline{R}$ be the mean read response time and $\overline{U}$ the mean update response time. Then the mean response time $\overline{T}$ is obtained as

$$\overline{T} = p\overline{R} + q\overline{U}$$

It is assumed that it takes one time unit for a message to travel from one node to another and that all computations take time zero.

The read cycle cost is plotted against p for different values of N in Figure 3. The response time is plotted against p for different N in Figure 4. Also plotted in both these Figures are the corresponding quantities for Ellis' ring algorithm ([ELLI]), an algorithm that implements the N-copy scheme. The read cycle cost of this algorithm is 2N and the read response time is zero, while the update response time is taken to be N. All these quantities are independent of p for this algorithm. As the Figures indicate, only for high values of p is the performance of Algorithm I inferior to Ellis' algorithm. This checks with the intuition that it is better to update all N copies at the same time if it is likely that all N nodes will read the file before the next update (a high value of p makes this probable). It can be proved that the worst case mean read cycle cost for Algorithm I is 3N-2. This shows that even if the assumptions made about the input are not valid, the degradation in performance of the algorithm will not be great.

The above analysis does not include the cost incurred due to reliability considerations. If they are, the update response time of Ellis' algorithm will be 2N. If one were to use a centralized controller to maintain the mutual consistency of the N copies of a file, the read cycle cost will be 3N-2 and the update response time 4. The high read cycle cost is due to the need for using a two phase commit procedure (see [GRAY]) while posting an update at all the N nodes. Algorithm I has to be modified since it is possible that only one node has the up-to-date copy at a particular time. But reliability considerations do not require that N copies be maintained up-to-date at all times. If three copies are to be up-to-date at all times then the read cycle cost for the modified form of Algorithm I will increase by about 4. This is because the two phase commit procedure has to be invoked with respect to only 2 nodes instead of N-1 nodes in the centralized case. The read response time of Algorithm I will improve because of this change but the update response time will degrade because of the need to wait for an update to be posted at sites other than the site where the update originated. It was not possible to model the scheme with the reliability features added in and so the above remarks are qualitative in nature. Comparing the above figures with those indicated by Figures 3 and 4, we can conclude that the comparison of Algorithm I with N-copy schemes

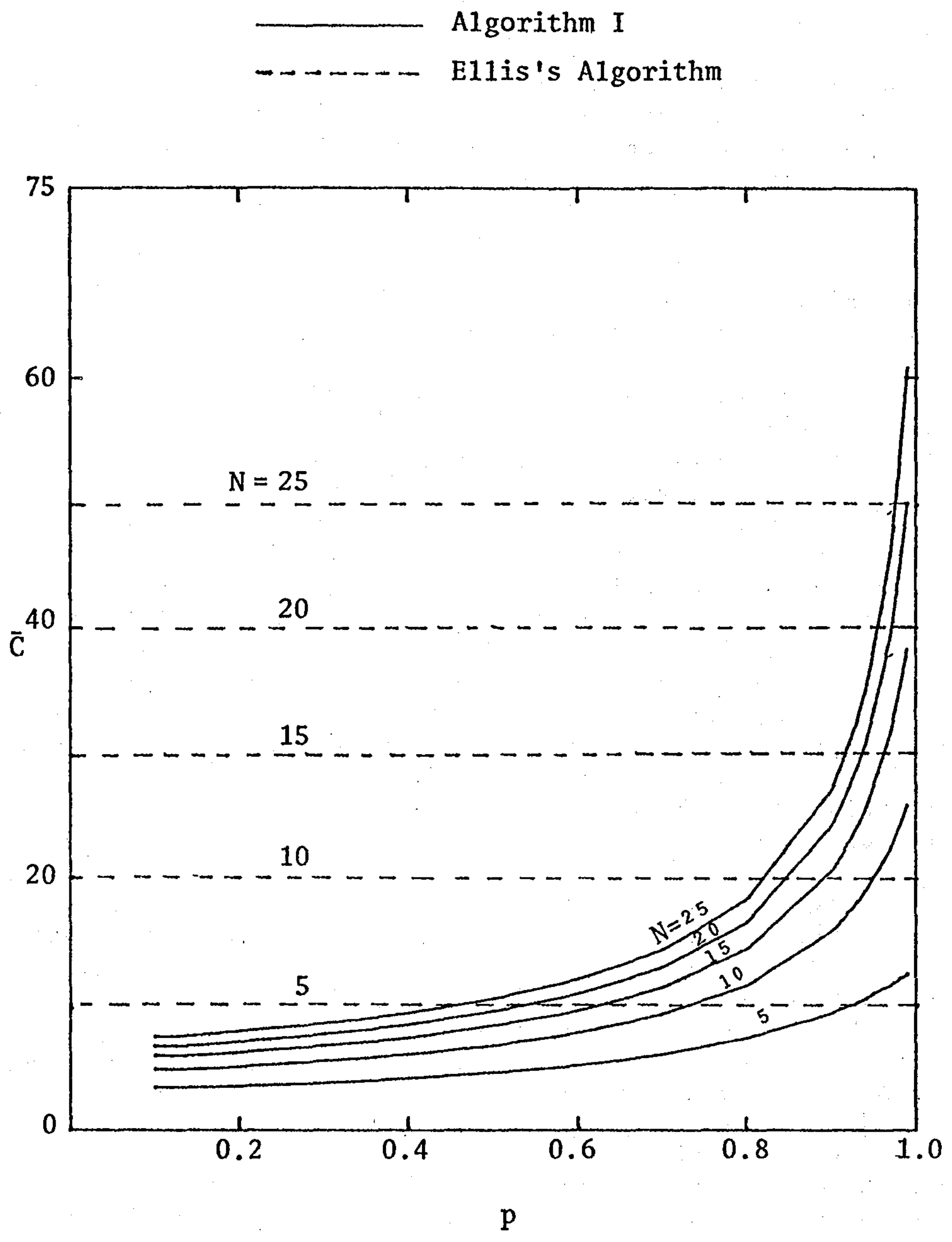


Figure 3 Mean cost per cycle ($\bar{C}$) versus the probability of a read (p) for different N, the number of nodes

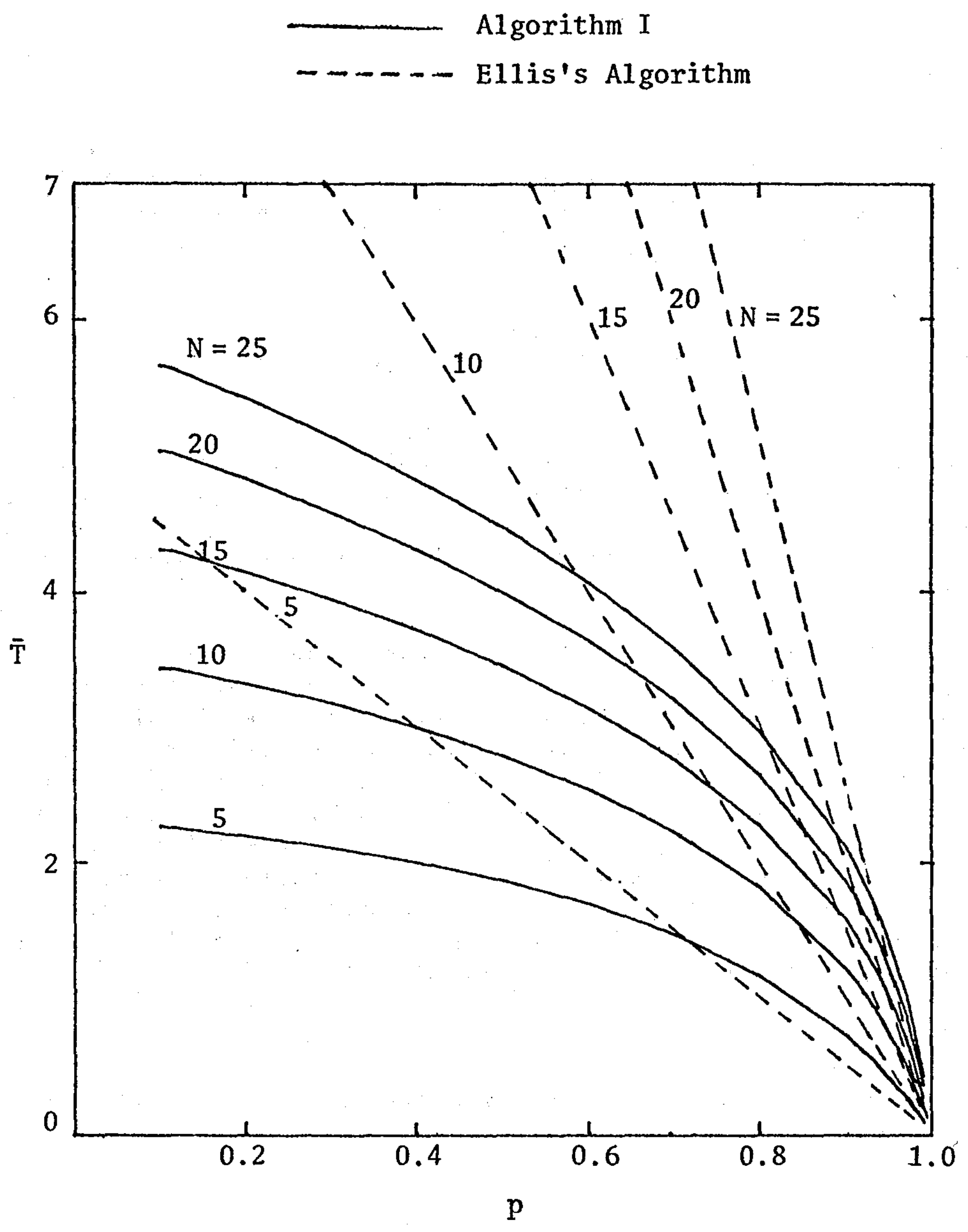


Figure 4 Mean response time ($\bar{T}$) versus p, the probability of a read, for different N, the number of nodes

becomes even more favourable when the schemes have to incorporate features to aid in failure recovery.

7. Conclusion

We have presented an algorithm for maintaining mutual consistency of multiple copies of a file in a distributed system. The algorithm keeps a variable number of up-to-date copies of a file with the number of such copies varying with the variation in the type and origin of requests in the system. We have proved that the algorithm guarantees mutual consistency and avoids starvation.

The scheme has been modelled and a fairly extensive analysis has shown that it compares favourably with other algorithms over a large range of request characteristics.

The algorithm has been extended to enable a node that wishes to access a file for the first time to do so and for a node that no longer wishes to maintain a copy to drop out. These aspects become important when a node has to destroy a copy of a file because of secondary storage space limitations.

The algorithm presented in this paper only guarantees mutual consistency of the copies of a file. It has been extended to service requests that have "tickets" assigned to them. Using a ticket allocation scheme (for example see [LELA]) along with this extended algorithm will then enable multi-file transactions to be serviced correctly. One of the prime criterion in this design is to allow single file transactions to access a file using just Algorithm I even in the presence of multi-file transactions. In an environment where most transactions are of the single file type, the efficiency of the concurrency control scheme will then be good. Finally, reliability aspects of these algorithms have been studied. The reader is referred to [BAR2] for details.

Acknowledgement

The author would like to thank his advisor, Professor John Bruno, for his valuable guidance.

References

[BAR1] Barua, G., A Demand Based Algorithm to maintain Mutual Consistency in a Distributed File System, Technical Report, Department of Computer Science, UCSB, May 1981.

[BAR2] Barua, G., Demand Based Concurrency Control in Distributed Systems, Ph.D Thesis, Dept. of Computer Science, UCSB, September 1981.

[BERN] Bernstein, P. A., D. W. Shipman and J. B. Rothnie, Concurrency Control in a System for Distributed Databases (SDD-1), ACM Transactions on Database Systems, Vol. 5, No. 1, Mar. 1980.

[ESWA] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger, The Notions of Consistency and Predicate Locks in a Database System, CACM, Vol. 19, No. 11, Nov. 1976.

[GRAY] Gray, J. N., Notes on Data Base Operating Systems, Operating Systems: An Advanced Course, Springer-Verlag, Berlin, 1978.

[KOHL] Kohler, W. H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, ACM Computing Surveys, Vol. 13, No. 2, June 1981.

[LAMP] Lamport,L., Time, Clocks and the Ordering of Events in a Distributed System, CACM, Vol. 21, No. 7, July 1978.

[LELA] Lelann, G., Algorithms for Distributed Data-sharing Systems which use Tickets, 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978.

[MINO] Minoura,T., A New Concurrency Control Algorithm for Distributed Database Systems, 4th Berkeley Conference on Distributed Data Management and Computer Networks, Aug. 1979.

[PAPA] Papadimitriou, C. H., The Serializability of Concurrent Database Updates, JACM, Vol. 26, No. 4, Oct. 1979.

[WILM] Wilms, P., Qualitative and Quantitative Comparison of Update Algorithms in Distributed Databases, Distributed Databases, ed. Delobel and Litwin, North Holland, 1980.

ON THE USE OF OPTIMISTIC METHODS FOR CONCURRENCY CONTROL IN DISTRIBUTED DATABASES

Stefano Ceri
Istituto di Elettrotecnica ed Elettronica
Politecnico di Milano
Piazza L. da Vinci, 32
I-20133 Milano - Italy

Susan Owicki
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford Ca 94305

Abstract

Optimistic concurrency control methods are based on the assumption that, in most real-life applications, conflicts between transactions are unlikely. To exploit this, transactions are allowed to execute freely, without the overhead of complex consistency-preserving mechanisms. However, transactions are validated before making their actions visible to other processes, and they are backed up whenever their actions would lead to inconsistency.

In this paper, one of the optimistic concurrency control methods presented by Kung and Robinson for single-site systems provides the framework for the development of a concurrency control method for distributed databases. Distributed transactions have to satisfy the requirements imposed by the local concurrency control systems, and also to respect global consistency requirements. In particular, the notion of global serializability is introduced, and an algorithm which guarantees the global serializability of distributed transactions is presented.

The work of Susan Owicki was partially supported by the Defense Advanced Research Project Agency under contract MDA903-79-C-0680.

1. Introduction

In this paper, we extend to a distributed database the Optimistic Concurrency Control methods for a single-site system presented by Kung and Robinson [9]. The approach is to assume that each node of the distributed database can use a local optimistic concurrency control algorithm and check for local consistency of transactions; local mechanisms are modified in order to ensure the global consistency of distributed transactions.

The main principle underlying optimistic concurrency control is the following: instead of ensuring the consistency of transactions a priori, using algorithms based on locking or time-stamp mechanisms, transactions are allowed to execute freely; however, a consistency check of the transaction execution (also called *validation*) is performed before making the effects of the transaction "visible" to other transactions; if a non-consistent execution is detected, then the transaction is backed-up and restarted.

Kung and Robinson [9] and Badal [2] give arguments that support the optimistic approach. In particular, they note that in many real-life applications the probability of conflicts between transactions is very low, and therefore the overhead due to backups of conflicting transactions is largely offset by avoiding more complex consistency-preserving mechanisms. In this paper, another assumption is made which justifies the use of optimistic methods in distributed databases, namely that most of the transactions in the distributed system are local to one site, and also that most of distributed transactions involve a limited number of sites. Thererefore the one-site optimistic control algorithm will be adequate for most transactions, and only in a limited number of cases will it be necessary to use the (rather heavy) global validation mechanism.

The paper is organized as follows. In Section 2, the optimistic approach described in [9] is reviewed; in Section 3, distributed transactions are described, and consistency criteria for distributed transactions are discussed. In particular, the notion of global serializability as an additional consistency requirement for distributed transactions is introduced. Section 4 describes an algorithm and a validation technique which provide the global serializability of distributed transactions; the method is derived from the *parallel validation* method of [9]. Finally, in Section 5 the correctness of the algorithm is proved and the approach is compared with other techniques presented in the literature.

2. The optimistic approach

2.1 Fundamentals

A distributed data base is a collection of named data objects, distributed over N different sites of a computer network. Any transaction consists of three phases: a *read phase*, a *validation phase* and a possible *write phase*. During the read phase, all writes take place on copies of the objects; during the validation, it is verified that the changes made by the transaction will not cause a loss of integrity. If the validation succeeds, then the copies are made effective in the write phase; otherwise the transaction is backed up.

The typical primitives provided to application programs allow one to create, delete, read, and write data objects; these primitives are implemented through calls to the concurrency control mechanism, operating as follows:

create(o)	adds a new object to the create set CS
write(o,v)	if $o \in CS$ or $o' \in WS$, writes the new value v in it; otherwise creates a copy o' of o, writes v in it, and inserts o' in the write set WS
read(o)	adds o to the read set RS; if a copy o' of o exists in WS, the value in that copy is read, otherwise the value of o is read
delete(o)	adds o to the delete set DS.

The sets are initialized to be empty by a *begin* call, and the validation phase is requested via an *end* call; the write phase simply consists of making the writes effective, by writing the values of copies $o' \in WS$ into the original objects, deleting the objects in DS, and deleting the temporary copies in WS. (It should be remarked that producing an output from a transaction is considered a write action).

2.2 Validation phase

The validation criteria used in [9] is serial reproducibility [5] or serializability [3,13]. Transactions are considered to consist of two atomic parts: the retrieval of the values of a set of database objects (the read set), followed by the update of the values of another set of objects (the write set) [13]. The criteria accepts those concurrent executions of transactions which are equivalent to some serial execution of the same transactions.

In [9], the criteria is enforced by verifying that the order in which transactions are actually executed is equivalent to their serial execution in the order in which they complete the read phase. If one of the following validation conditions holds, then the actual execution order is equivalent to one in which Ti completes before Tj starts [9,13]:

(1) Ti completes its write phase before Tj starts its read phase.
(2) The write set of Ti does not intersect the read set of Tj, and Ti completes its write phase before Tj starts its write phase.
(3) The write set of Ti does not intersect the read set or the write set of Tj and Ti completes its read phase before Tj completes its read phase.

Note that rules (1-3) are sufficient, but not necessary, conditions for serializability. To show that they are not necessary, consider rule (2). Let $\{o\} = WSi \cap RSj$ in a schedule having only transactions i and j, and assume that Ti writes o after the beginning of read actions of Tj but before the actual read of o by Tj. Then the validation fails, even if a perfectly legal sequence: "Ti reads o, Ti writes o, Tj reads o, Tj writes o" has occurred.

In the following, the control mechanism called *parallel validation* in [9] is outlined. The other mechanism introduced in [9], called *serial validation*, is unattractive for distributed databases becauses it forces the validations to be strictly serial. This is too restrictive for a distributed transaction, in which all the components of the transactions on the various sites must "agree" on either commit or abort.

In parallel validation, a global counter *tnc* is used for the purpose of assigning transaction numbers. Each transaction Tj reads tnc into Start-Tj at the beginning of the read phase (*begin* call). (Note that the value of tnc is not modified at this point: assignment of a transaction number for Tj takes place after validation.) At the end of the read phase (*end* call) the transaction is validated. Within a first critical section tnc is read into Finish-Tj and Tj is added to the set AS of transactions which are *active*, i. e. have completed the read phase but have not yet completed the write phase; Tj also takes a copy ASj of the active set at that time. Then, outside the critical section, validation of Tj takes place. Potential conflicts between Tj and transactions Ti with transaction number tn(Ti) such that Start-Tj < tn(Ti) <= Finish-Tj are checked using rule (2); potential conflicts between Tj and transactions Ti $\in$ ASj are checked according to rule (3). If the validation succeeds, then the write phase takes place; finally, within a critical section, Tj is assigned a transaction number and eliminated from the active set AS. Note that, because of this transaction number assignment schema, transactions with tn(Ti) <= Start-Tj necessarily satisfy rule (1).

The mechanism strictly enforces the serialization of transactions in the order in which they enter the active set. In the following, we use the fact that the order in which transactions enter the active set is the same as the serial schedule which is enforced by each local concurrency control mechanism.

3. A Model of distributed transactions

Transactions are classified as *local* (single-site) transactions, or *global* transactions, which operate on several nodes. A global transaction originates at a *master* site, which is in charge of coordinating the actions of the transaction on the other sites. In particular, the transaction master initiates several *sub-transactions* which are logically part of the transaction itself but run on different nodes. Subtransactions run concurrently, but at the end they commit to the transaction master using a two-phase commitment schema [8].

With respect to the concurrency model outlined in the previous section, sub-transactions can still be considered to consist of a read and a write part; however, each global transaction now consists of multiple pairs of read and write actions, one at each site where a sub-transaction is activated, including the master site.

Global transactions have to satisfy the following consistency requirements:

(1) Sub-transactions have to be properly synchronized with local transactions on their execution site, producing local serializable schedules; this is ensured by using the local concurrency control mechanisms. Let the relation "<" refer to the serialization

order which is enforced by the local concurrency control mechanisms.

(2) There is a global consistency requirement: global transactions should execute in such a way as to produce a *globally serializable* schedule. In the following, the notion of global serializability for this particular transaction environment is discussed.

First, as an example, consider a distributed database consisting of the nodes {1,2}, and two global transactions T1 and T2 each consisting of two sub-transactions. Let STij indicate the sub-transaction of Ti executed at node j. Finally, let RS1=RS2=WS1=WS2={x,y}, with x stored at site 1, y stored at site 2.

The notion of precedence between global transactions has to be formulated. A possible definition of global precedence between T1 and T2 requires that every action of T1 precede every action of T2; according to this definition, the following is a legal schedule:

"T1 reads RS1, T1 writes WS1, T2 reads RS2, T2 writes WS2".

However, this notion of serializability is rather unattractive in practice; it requires a strict serialization of actions which refer to objects that are not stored at the same site, and clearly this is a heavy requirement.

A better definition of global precedence requires that the concurrent execution of T1 and T2 on several sites produces the same effects as the serial execution of T1 and T2, without actually requiring strict serialization. The formal definition of global precedence is based on the order of transactions at sites where they conflict.

The definition of conflict between two transactions Ti and Tj executing at the same site is given as in [3,13]; it is:

$$\text{conflict}(Ti,Tj) \leftrightarrow (RSi \cap WSj \neq \Phi) \vee (WSi \cap RSj \neq \Phi) \vee (WSi \cap WSj \neq \Phi)$$

The relation conflict' takes into account the ordering of T1 and T2 in the local serialization; it is:

$$\text{conflict}'(Ti,Tj) \leftrightarrow \text{conflict}(Ti,Tj) \wedge Ti < Tj$$

Finally, the relation "<<" is the transitive closure of conflict'. Intuitively, T1<<T2 means that execution of T1 preceded and may have affected execution of T2.

Having defined the "<<" relation between transactions at a single site, it is possible to define a *globally serializable* execution of the distributed transactions. Let G=<N,E> represent a directed graph called the *global serialization graph*, where:

$$N = \{ Ti \mid Ti \text{ is a global transaction} \}$$
$$E = \{ \langle Ti,Tj \rangle \mid \exists\ k \in N : STik \ll STjk \}$$

G reflects the serializations that are forced at each site by using the local concurrency control mechanisms; then, the global transactions are globally serializable if the graph is acyclic.

In fact, with an acyclic global serialization graph, the effect is equivalent to the serial execution of the transactions in any order consistent with the "<<" relation. Note that the transitive closure of conflict' is needed in defining G, because the graph includes only global transactions, but the propagation of the effects of one global transaction on another can take place through local transactions.

One of the possible globally serializable executions of T1 and T2 in the above example corresponds to the following local schedules at nodes 1 and 2:

"ST11 reads x, ST11 writes x, ST21 reads x, ST21 writes x" at site 1;
"ST12 reads y, ST12 writes y, ST22 reads y, ST22 writes y" at site 2.

In this case, we have ST11 << ST21 and ST12 << ST22; the serialization graph has only an edge from T1 to T2, and is clearly acyclic.

Consider now a case in which the definition is violated. Let two local schedules be :

"ST11 reads x, ST11 writes x, ST21 reads x, ST21 writes x" at site 1;
"ST22 reads y, ST22 writes y, ST12 reads y, ST12 writes y" at site 2.

Here ST11 << ST21 and ST22 << ST12; the global serialization graph has a cycle, as there are two opposite edges between T1 and T2. This concurrent execution leads to evident inconsistency when the constraint $x=y$ holds, and T1 adds 10 to the (numerical) objects to which is applied, while T2 multiplies by 10 the (numerical) objects to which is applied.

4. An algorithm for global transaction validation

The algorithm for the validation of global transactions has been designed with the goal of keeping the activities related to the concurrency control as distributed as possible. Therefore, algorithms involving the presence of a unique *consistency monitor* that receives all information about the local schedules and checks for acyclicity of the global serialization graph have not been considered. This general criterion is motivated by considerations of reliability, efficiency, and site autonomy.

In the proposed algorithm, local transactions are executed under the control of local concurrency control systems, with minor modifications. Global transactions are subjected to a more complex concurrency control mechanism, described in the following. The term sub-transactions will refer to local portions of global transactions, as opposed to local transactions. Some new features are introduced with respect to [9]:

(1) Transactions are assigned unique *transaction identifiers* (TID); this requires the use of counters at each node, whose value is incremented for each new transaction; all the sub-transactions of a particular global transaction receive the same TID from

the transaction master, augmented with the master site index in order to preserve uniquenes of identifiers in the network.

(2) At each local site, a list is maintained of the local transaction or sub-transaction identifiers in the order in which they enter the active set: it is called the *active set list* (ASL). Note that transaction identifiers are recorded on the list in the same order in which their serialization is enforced; we have STi<STj whenever STi precedes STj in the list. After the validation of the transactions, the identifiers are either removed from the list, when the validation fails, or marked as *committed*, when the validation succeeds. Active set lists are periodically examined in order to delete from them those items which are not of interest for validation purposes; as it will become clear in the following, all identifiers preceding the earliest global transaction which is not marked are not of interest for the validation.

(3) Each sub-transaction collects into a *happened before set* HB the TIDs of the global transactions that have effectively executed before at that site, have influenced its execution, and have their identifier still recorded in the active set list. The happened before set is based on the relation "<<" already introduced. In the evaluation of "<<", only those local transactions and subtransactions whose TID is recorded in the ASL are considered; the HB set is then defined as follows:

$$HB(STih) = \{ TIDj \mid Tj \text{ is global} \wedge Tj \in ASL \wedge STjh \ll STih \}.$$

Local transactions are validated according to the parallel validation algorithm of [9], with the only difference that they have to insert their TID in the ASL when the validation starts and either delete it if the validation fails or mark it if the validation succeeds. The active set of [9] consists in this case of those transactions whose TID is stored in the ASL and is not marked. Moreover, the read and write set of the local transactions have to remain available until their marked identifiers are actually deleted from the ASL, as they are used for computing the "<<" relation.

The validation of sub-transactions consists of a local and a global validation phase. For the local validation, the parallel validation algorithm of [9] is used. If the validation fails, the sub-transaction is backed up and restarted locally. Finally, the sub-transaction enters the global validation phase, in which global serializability is checked. In the following, the global validation and commitment of a sub-transaction STih is described; the validation procedure for sub-transactions is also shown in Fig. 1, as an extension to the *end* consistency control call of [9].

```
end = (     /* validation of STih on node h */

    /* local validation, as in [9] */
    <Finish-Tn:=tnc;
     COPYih= (make a copy of ASL);
     {append TID to ASL}>;
    FA:=(make a copy of active transactions not marked in COPYih);
    valid:=true;
    for t from Start-Tn to Finish-Tn do
      if (write set of transaction number t intersects read set)
      then valid:=false;
    for i ∈ FA do
      if (write set of transaction STi intersects read set or write set)
      then valid:=false;
    if valid then  /* global validation phase */

        (   /* build the HB */
           HB(STih)={TIDi};
           while (no more TID can be added to HB(STih) ) do
             if ∃ j,k | TIDj∈COPYih ∧ TIDk∈HB(STih) ∧ conflict'(STjh,STkh)
             then HB(STih) = HB(STih) ∪ { TIDj };
          (remove TIDi and local transaction identifiers from HB(STih));

        /* testing for commitment or abortion of conflicting transactions */
           while  ( ∃ STjh ∈ HB(STih) ) and valid do
              if ( STjh ∈ ASL and STjh is marked ) or ( STjh ∉ ASL )
              then  HB(STih)= HB(STih) - {STjh}
              else ( /* time-out  and  second  attempt  */
                  go-to-sleep(timeout);
                  while  ( ∃ STjh  ∈ HB(STih) ) and valid do
                     if ( STjh ∈ ASL and STjh is marked ) or ( STjh ∉ ASL )
                     then  HB(STih)=HB(STih) - {STjh}
                     else valid := false;

        /* two-phase commitment */
          if valid then (
                  sendreply("ready to commit");
                  if message=commit
                  then (  (write phase);
                          <tnc:=tnc+1; tn:=tnc;
                          (mark TID in the ASL) > ))
                  else (  <ASL:=ASL - TID >;
                          (backup))

        /* failure of global validation */
          else ( <ASL:=ASL - TID >;
                  send("abort");
                  (backup) )) /* end global validation phase */

    /* failure of local validation */
    else ( <ASL:=ASL  - TID >;
           (local backup)  )).
```

Fig. 1: Extensions to end consistency control call for sub-transactions with respect to the *parallel validation* method of [9]. Critical sections are enclosed by symbols "<",'>'. Communications occur between the master site and site j; when the primitive sendreply is used, the sender waits until it receives the reply.

1) The HB set is computed; initially, it is HB(STih)={TIDi}, and then the TID of transactions in conflict' relation with some transaction of the HB set are included in the set itself; this construction is repeated until no more transaction can be added to the HB set. At the end, the TID's of local transactions and TIDi are eliminated from the set, which therefore contains only those global transactions which are in "<<" relation with STih.

(2) The global validation phase consists of verifying that all the transactions of the HB set have either committed or aborted before the point at which STih is validated; as it will be proved in the next section, this condition ensures global serializability. The commitment or aborting of conflicting transactions can be read from the ASL, without requiring exclusive access to it. In the case where the validation fails because there is some global transation STjh in HB(STih) which is still active, the transaction is kept waiting for a given timeout. Then the validation is repeated, and if it fails again Ti is aborted. In this case, a message is sent to the transaction master, which both broadcasts the abort command to all the other sub-transactions, and issues a new global transaction (assigning a new transaction number to it).

Let us consider how the validation attempt might fail. In a simple case, say Tih << Tjh and Tjk << Tik, transaction Tj cannot complete validation at site h until after Ti has committed or aborted globally, and Ti cannot complete validation at site k until after Tj has committed or aborted globally. Thus at least one of them will time-out and abort in its validation phase. In general, whenever committing a set of conflicting transactions would cause a cycle in the global serialization graph, there is a corresponding cycle in the validation phase, in which each transaction must wait for one of the others to commit or abort (this is proved in section 5). The wait cycle can only be broken by one (or more) aborting, and this prevents cycles in the serialization graph. Of course, this method can abort some transactions unnecessarily: a transaction may time-out and abort just before the transaction it was waiting for commits or aborts. However, given our assumption that conflicts are uncommon, a proper choice of time-out interval should make such un-needed backups rare.

(3) At the end of the global validation phase, the two-phase commitment takes place; a *ready to commit* message is sent to the transaction master, which collects the messages from all the sub-transactions; if none of them aborts, then finally a commit message is sent back. Then, just as in the parallel validation algorithm, subtransactions can perform the write phase, are assigned a local transaction number, and are marked in the local ASL. If instead one of the sub-transactions aborts, then all other subtransactions have to eliminate their TID from the local ASL's, and repeat their execution (with a new TID).

The global concurrency control mechanism is completed by a *clean-up transaction* which is issued periodically; it analyses the ASL and deletes from it all the TID's which precede the earliest non marked global TID. Moreover, read sets of transactions whose TID is deleted are also deleted, together with the write sets which are not of interest for the local concurrency control mechanism.

5. Proof of correctness of the algorithm, some practical considerations, and comparison with other approaches

Proof of Correctness

The correctness of the algorithm is proved informally here, though a formal proof based on the techniques of [12] is possible. We will show that the algorithm is safe in the sense that it does not allow construction of local schedules which are not globally serializable. There is no guarantee that a transaction will eventually be able to commit; one can only say that the assumption of low conflict between transactions makes eventual commitment very probable.

We first show that an edge between two transactions in the global serialization graph implies that there is a site at which the first one committed before the second completed its validation phase. (Note that the definition of the global serialization graph implies immediately that the first transaction committed before the second committed.)

Lemma: If there is an edge from Ti to Tj in the global serialization graph, then there is some site h such that STih committed before STjh completed validation.

Proof: Let h be a site such that STih << STjh; the definition of the global serialization graph implies that such a site exists. Consider the value computed for HB(STjh), assuming, for the moment, that no clean-up transaction have been run and thus the ASL contains all transactions that have ever committed or are still active at site h. In this case, computation of HB(STjh) captures all sub-transactions that are in the "<<" relation to STjh; in particular, TIDi is in HB(STjh). Now, the algorithm delays validation of STjh until after STih has committed, giving the required result.

Next, consider the effect of clean-up transactions. As long as TIDi is put in HB(STjh), the lemma is satisfied. The only way that TIDi can fail to appear in HB(STjh) is if it has been removed from the ASL by a clean-up transaction before HB(STjh) is computed. Because a transaction can only be removed after it has committed, this implies that STih committed before STjh began its validation phase, which is well before it ended validation. So the lemma is satisfied in this case too.

Theorem: The algorithm described above can never give rise to a cycle in the global serialization graph of committed transactions.

Proof: The algorithm prevents a cycle in the serialization graph by requiring conflicting global transactions to complete validation in the same order as they would appear in the graph. In the case of a cycle, this prevents any of the transactions from completing validation until one or more of the transactions in the cycle has aborted. We proceed to establish this by contradiction. Assume that a cycle of committed transactions exists, consisting of T1, T2, ... , Tn, Tn+1=T1. The lemma above implies that, for each pair of transactions Ti and Ti+1, there is some site h where Ti committed before Ti+1 completed validation. Since Ti could not commit before all of its sub-transactions completed validation, this implies that all sub-transactions of Ti completed validation before all sub-transactions of Ti+1 did so. Since the transactions form a cycle, this is impossible, and we have the required contradiction.

Practical considerations

The following modifications to the validation mechanism can be used to improve its efficiency or functionality in certain cases.

(1) There is an inefficiency in the local validation algorithm which was pointed out in [9], namely that "a transaction in ASL can invalidate another transaction, even though the former transaction is itself invalidated". A possible escape from this case was also pointed out in [9], and consists of "waiting for the invalidating transaction to either be invalidated, and hence ignored, or validated, causing the backup". This escape, however, holds for local transactions only, because keeping a global transaction waiting introduces potential distributed deadlocks (but see note 2 below).

(2) The algorithm currently prevents deadlocks among cyclically conflicting transactions by a time-out mechanism. Alternatively, one could detect such cycles, using an algorithm like Obermarck's [11], and abort transactions only when they are involved in a cycle. Given the assumption that conflicts are infrequent, this situation should seldom arise, so the extra mechanism required for cycle detection might not be used often enough to justify its inclusion.

(3) The computation of transitive closure of conflict can be rather lengthy, expecially with transactions with large read and write sets. In fact, more efficient algorithms than the one presented in Fig. 1 can be used for determining transitively the conflicts between transactions. For instance, it is clear that the local conflict history for a given transaction is all contained in its local HB set, and therefore this information can be used to build the transitive closure of conflicts. Note, however, that the method presented here relies on the optimistic assumption of having a very limited number of conflicts between transactions, and this should greatly simplify in practice the computation of HB sets.

(4) At each local site, the complexity of the validation process and the amount of information that has to be stored for the validation depends on the number of transaction which are stored in the ASL, and they ultimately depend on the earliest global transaction which is not validated. The size of the ASL could be controlled by introducing a mechanism to force the abortion of the latest global transaction, based either on time-out mechanisms or on the amount of local storage availlable for concurrency control.

(5) The possibility that a transaction never succeeds in its validation (starvation) has to be considered. In [9], lock-based techniques are recommended for those transactions which are repeatedly invalidated. This also applies to a distributed environment, with the additional problems which arise because of global deadlocks [11]. Badal [1] suggests the use of random time intervals for restarting transactions that have conflicted, in order not to repeat the same conflicting sequence.

(6) Although we have not explicitly considered replicated data, it fits into the algorithm quite easily. It is only necessary to implement a write on a replicated item

by writes to all copies of the item, using sub-transactions at each site where the item is stored; a read may be implemented by reading any copy. The global consistency mechanism will ensure serializability of these transactions, and this implies that multiple copies remain consistent. In fact, the more general weighting schemes described by Gifford [7] may be used to determine how many copies must be accessed by read and write operations.

Comparison with other approaches

An algorithm for concurrency control of distributed transactions which uses an optimistic approach is presented in [6]. However, the model of distributed transaction which is assumed in that paper is different from the proposed model; a distributed transaction in [6] corresponds to a sequential execution of sub-transactions at the various sites. In this way, each sub-transaction has a global knowledge of its conflict history, and therefore decisions on the conflicts that lead to non-serial executions can be progressively taken during transaction evolution. The concurrency control information is stored together with objects; each object has a stack-based log which stores the transaction identifiers of transactions which access the object, together with their past conflict history. The advantage of the method is in the possibility of anticipating decisions about conflicts; the disadvantage is in the "sequential" model of sub-transaction execution, that doesn't allow parallelism.

The notion of serializability used in [3, 4, 10, 13] was, in the authors' opinion, heavily influenced by the operational features of some distributed database system (see for instance SDD-1 [4]); in particular, the need in those systems of collecting the non-local information on the initiating site and executing the transaction there leads quite naturally to mantaining one read phase and one write phase in a distributed environment as well. The model which is proposed here, which allows several subtransactions to execute their read and write phase without requiring a strict sequentiality between all reads and all writes, is probably more general, and moves in the direction of considering more distributed models of transaction execution.

6. References

[1] Badal, D. Z. Correctness of Concurrency Control and implications in Distributed Databases, *Proc. COMPSAC 79*, Chicago, November 1979, pp. 588-593.

[2] Badal, D. Z. Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms, *Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1981, pp. 85-103.

[3] Bernstein, P. A., Shipman, D. W., and Wong, W. S. Formal Aspects of Serializability in Database Concurrency Control, *IEEE Transactions on Software Engineering*, Vol. 5 No. 3, May 1979, pp. 203-214.

[4] Bernstein, P. A., Shipman, D. W., and Rothnie, J. B. Concurrency Control in a System for Distributed Databases (SDD-1), *ACM TODS*, Vol. 5 No. 1, March 1980, pp 18-51.

[5] Eswaran, K. P., Gray, J. N., Lorie, R. A.,and Traiger, I. L. The Notion of Consistency and Predicate Locks in a Database System, *Comm ACM*, Vol. 19 No. 11, November 1976, pp. 624-633.

[6] Garcia-Molina, H. and Wiederhold, G. Read-only Transactions in a Distributed Database, Stanford Department of Computer Science, Report No. STAN-CS-80-797, April 1980.

[7] Gifford, D. K. Weighted Voting for Replicated Data, *Operating System Review,* Vol 13, No. 5, December 1979, pp 150-162.

[8] Gray, J. Notes on Database Operating Systems, in *Operating Systems: an Advanced Course,* R. Bayer, R. M. Graham, G. Seegmuller eds., Springer-Verlag, 1978, pp. 393-481.

[9] Kung, H. T. and Robinson, J. T. On Optimistic Methods for Concurrency Control, *ACM TODS*, Vol. 6 No. 2, June 1981, pp 213-226.

[10] Minoura, T. Resilient Extended True-copy token Algorithm for Distributed Database Systems, Dept. of Electrical Engineering, Ph. D. Dissertation, Stanford University, May 1980.

[11] Obermarck, R. Global Deadlock Detection Algorithm, IBM Rep. N. RJ2845(36131) 6/13/80, June 1980.

[12] Owicki, S. and Gries, D. An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica,* Vol. 6 No. 4, pp 319-340.

[13] Papadimitriou, C. H. The serializability of Concurrent Database Updates, *Journal of the ACM*, Vol. 26 No 4, October 1979, pp 631 - 653.

Performance of Two Phase Locking

Wen-Te K. Lin
Jerry Nolte

Computer Corporation of America

Abstract

Simulation and analytical modeling of the two phase locking in a DBMS is the subject of this study. It is only part of a larger project that is studying the performances of various concurrency control and reliability algorithms in a distributed DBMS. In the simulation model, the application environment is characterized by the transaction size -- the number of lockable units requested by each transaction -- and the system environment by the number of transactions running concurrently (multiprogramming level), total number of lockable units in the database, and the distribution of accesses to these lockable units. These environments are varied for different simulation runs. Output from these simulation runs includes the probabilities of a lock request involved in a conflict and deadlock respectively (PC and PD), and the average waiting delay (WT) and its standard deviation (DV) of a blocked lock request. The results show that the system behaves quite similarly for different access distributions -- PC, PD, WT, and DV all increase more than linearly with the multiprogramming level and the transaction size; the increase of PC is faster with multiprogramming level than with the transaction size, and the reverse is true for PD, WT, and DV. Regression analysis on the simulation results reveals interesting relationships between the granularity of the lockable units and PC, PD, and WT. Because of the assumption of fixed delay (excluding blocking due to lock conflict) between two consecutive lock requests by a transaction, the results apply to a centralized DBMS with little IO delay variation, and a distributed DBMS with little communication delay variation.

This research is supported by the Rome Air Development Center (TSR) of the Department of the Air Force under Contract Number F30602-81-C-0028. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Rome Air Development Center of the Department of the Air Force or the U.S. Government.

1. Introduction

In the two phase locking protocol as described in Gray [1], during the first phase transactions accummulate locks incrementally, acquiring each lock as its need arises, and during the second phase, release each lock as soon as its need ends. But to spare the end users the responsibility of requesting and releasing locks, most DBMSs implement implicit locking. The DBMSs request and release the locks automatically when the transactions request the data items and when the transactions end, respectively. Because a DBMS, not knowing enough of the syntax and semantics of the transactions, is ignorant of the time when each data item is no longer needed, it can only release the locks held by a transaction when the transaction ends. Besides, if locks held by a transaction are released before the transaction ends, then the abortion of the transaction causes roll-backs of all other transactions that have read data released by the aborted transaction. To avoid the problems discussed above, most DBMS release locks held by a transaction when the transaction ends. The performance of this modified two-phase locking is the subject of this study.

In this study we use several measures of system performance. We emphasize the blocking and restart behavior of transactions. We concentrate on the basic underlying factors of conflict, deadlock, and wait duration. The performance variables are listed as follows:

1. the average probability of a lock request conflicting with another one;
2. the average probability of a lock request causing a deadlock;
3. the average waiting delay of a conflicting lock request;
4. and the standard deviation of this delay.

Besides locking protocol, the performance of a DBMS depends on several system and application parameters:

1. the average number of locks requested by a transaction (transaction size);
2. the maximum number of transactions running concurrently (the multiprogramming level);
3. the size of the group that is the unit of locking (lockable unit size);
4. the size of the database (total number of lockable units);
5. and the distribution of lock requests to the lockable units of the database.

Two distributions of lock requests to the lockable units are simulated. The random access model assumes that all lockable units have the same probability of being accessed by a lock request. The 20/80 model assumes that 20% of the database is accessed 80% of the time.

Using simulation and statistical data analysis techniques, this paper studies the relationships between the performance of a DBMS and those system and application parameters affecting it.

A few researchers have attempted similar studies. In Lin [2], the same approach taken in this study was used to evaluate two timestamping protocols, but its results could not be extended to the two-phase locking protocol. In Naka [3], the result confirmed that concurrent updating of the database by transactions degrades the performance of a DBMS. In Spit [4], the two phase locking and the modified version (described above) were found to perform equally well in system-2000. In Mun [5], deadlock resolution methods were studied, and three were found to be superior: restarting the smallest, the one holding the least locks, and the one having consumed the least cpu time. In addition, it was found that simultaneous reduction of the sizes of the lockable unit and the transaction improves the performance. But the oversimplified definition of performance as the cpu utilization made the results less useful. In Ries [6], the scope and the objective of its simulation were much more ambitious than the previous three. Nevertheless, it emphasized the effects of the size of the lockable unit on the performance of the DBMS, which was defined as the cpu and IO utilizations, plus in some cases the response time and the system through-put. The main model required transactions to obtain all the required locks before they started, and the request-as-needed model was only briefly studied. It had many interesting results showing how the size of the lockable unit interacts with the system and application parameters to effect the performance. But its assumption that the multiprogramming level has no affect on performance is contradicted by this study. Also, performance was not related to system and application parameters as precisely and quantitatively as in the present study.

This study expands on Lin [2] and Ries [4], and presents the results in the same precise form as that of Lin [2]. The second section discusses the simulation model; the third section presents and analyzes the results of the random access model; the fourth section presents and summarizes the results of the 20/80 model; and the fifth section summarizes the results of this study.

2. Simulation Model

A complete description of a simulation model for a DBMS must include the database, the transactions, the computer system, and the output parameters.

The database consists of DZ (Database siZe) lockable units of equal size. The size of each lockable unit is irrelevant to our model. The database size DZ varies among different simulation runs.

We simulate two different access distributions to the database: the random access model in which all lockable units are equally likely to be accessed, and the 20/80 access model in which 20% of the database is accessed 80% of the time.

All transactions request only exclusive locks. Within each simulation run, all transactions request the same number TZ (Transaction siZe) of lockable units, but TZ varies among different simulation runs. Each transaction requests its lockable units sequentially, but different transactions request lockable units asynchronously. When a transaction requests for a lockable unit, a random number is drawn to select one among all the lockable units in the database except those held by the requesting transaction; thus a transaction never requests the same lockable unit more than once. If the drawn lockable unit is locked by another transaction, the requesting transaction is queued at the end of a FIFO queue. Otherwise, it sets a lock on the drawn lockable unit and waits one time unit before requesting another lockable unit. Since processing a lock request is assumed to be instantaneous, the simulation timer is advanced one unit only after all outstanding lock requests have been processed. The assumption that a transaction waits a unit of time (after obtaining a lockable unit) before requesting another one, implies that it takes one time unit to retrieve a lockable unit from the database, to wait for the cpu, and to process it. Each transaction releases all its lockable units after its completion or abortion.

We model the computer system at a high functional level. The cpu, IO devices, and other hardware components are invisible in the simulation model; their existence is implied by the processing time required for each lockable unit discussed previously. The system is a closed multiprogramming system, i.e., the number of transactions running concurrently remains at a constant level MP (MultiProgramming level); a new transaction starts as soon as one completes or aborts. Nonetheless MP varies among different simulation runs. A lock request conflicts if it requests a lockable unit already held by another transaction. The system maintains a lock with a FIFO queue for each lockable unit and places conflicting lock requests into the queue. It checks for deadlocks as soon as a lock request conflicts. If it detects a deadlock, the transaction of the conflicting lock request aborts and restarts immediately; it restarts with a new randomly drawn sequence of lock requests. Checkings of conflicts and deadlocks are instantaneous.

For each simulation run, the output includes the fraction of conflicting lock requests (which is the same as the probability of a lock request conflicting with another lock request PC), the fraction of conflicting lock requests causing deadlocks (which is the same as the probability of a lock request causing a deadlock PD), and the average waiting of a blocked lock request (WT) and its standard deviation (DV).

3. Simulation Results of the Random Access Model

Sixty four simulations were run for 4 values of multiprogramming level (MP), transaction size (TZ), and database size (DZ) each. The results are presented and analyzed in this section in the following order: PC, PD, WT, and DV. The analysis consists of three steps: visual inspection, regression analysis, and examination of the regression equations.

The results of PC are presented in Figure 3.1. The figure shows that for a fixed DZ, PC increases with both MP and TZ, and the increase is larger with MP than with TZ. This behavior is explained by the following observation during the simulation runs: the number of transactions deadlocked increases faster with the transaction size than with the multiprogramming level. Since a deadlocked transaction aborts and releases all held locks as soon as the deadlock occurs, the total number of locks outstanding (not released) increases slower with the transaction size than with the multiprogramming level.

If a diagonal line is drawn from the top left to the bottom right of each table in the figure, each number below the line is always larger than the opposite number across the line. Assuming DZ is fixed, two elements across the diagonal line represent the same load (L) defined as the product of MP and TZ divided by DZ. For example, a system with 16 transactions, each requesting 7 locks, imposes the same load (112 lockable units) on the database as a system with 7 transactions, each requesting 16 locks. This line shows that with the same load, the system with higher multiprogramming level has higher probability of conflict than the system with higher transaction size. This behavior is explained by the following observation during the simulation runs. Assuming the load L and the database size DZ are fixed, then on the average, a larger MP with smaller TZ implies less deadlocks and more locks outstanding. Since each lockable unit has the same probability of being accessed, more outstanding locks means higher probability of conflict. But higher probability of conflict does not necessarily means longer response time, because smaller transaction size causes conflicting requests to wait less and to deadlock less, as will be shown.

The differences across the diagonal line diminish as the database size DZ increases -- that is, the probability of conflict (PC) is approximately proportional to the load L when the load on the database is light, because increasing the database size without increasing the multiprogramming level or the transaction size is equivalent to decreasing the load on the database.

We applied regression analysis to the data in Figure 3.1, and found equation (3.1) a good fit. The residuals -- the differences between the actual values and the values predicted by the equation -- are within 2.5% of the actual values. We did a few simulation runs with larger values of DZ, MP, and TZ, and found that the equation is still a good fit for DZ of up to 12384, MP of up to 128, and TZ of up to 32; but we found that when the transaction size TZ gets much larger than 32, the equation under-estimates the probability of conflict (PC) substantially.

$$PC = \frac{0.72\ (MP-1)^{1.05+0.35L}\ TZ^{1.08-0.13L}}{DZ^{1.08+0.28L}} \qquad (3.1)$$

$$L = \frac{MP \times TZ}{DZ}$$

Next, we use the regression equation to examine the relationship between the size of the lockable unit and the probability of conflict.

If we split each lockable unit into k smaller units, then the database size increases to k times its original size. Because of the smaller lockable units, a transaction must request more lockable units; thus the transaction size increases to w ($1 \leq w \leq k$) times its original size. The value of w depends on how well the database is placed before the split. If the database is originally well placed, then all the data items contained in the original TZ lockable units are wanted by the transaction -- no frivolous data items are retrieved. In this case, when a lockable unit is split into k smaller ones, the transaction size increases to k times its original size (w=k). Otherwise, if the database is badly placed before the split, then the lockable units retrieved by a transaction contain a lot of unwanted data items. Thus, after the split, a transaction may request the same number of lockable units and still obtain all the data items it needs (w=1). In most cases, however, w will be larger than one and smaller than k.

Replacing DZ and TZ by kDZ and wTZ, equation (3.1) becomes equation (3.2),

$$PC' = t \times PC \qquad (3.2)$$

where

$$t = \frac{DZ^{0.28Lr} \; TZ^{0.13Lr} \; w^{1-(0.13Lw)/k}}{(MP-1)^{0.35Lr} \; k^{1-(0.28Lw)/k}} \qquad (3.2a)$$

and

$r = (k-w)/k.$

Setting w to k, equation (3.2a) becomes (3.2b).

$$t = \frac{1}{k^{0.41L}} \qquad (3.2b)$$

Since k is larger than one, t is smaller than one. Thus smaller lockable units imply a smaller probability of conflict whenever the database is well placed. But as we will show later, smaller probability of conflict with larger transaction size may result in a higher probability of deadlock and longer transaction response time. As L approaches zero, i.e., the load is light, t approximates one, and the difference between PC and PC' becomes insignificant.

Setting w to one in equation (3.2a) results in equation (3.2c).

$$t = \frac{DZ^{0.28Lr} \; TZ^{0.13Lr}}{(MP-1)^{0.35Lr} \; k^{1-(0.28L)/k}} \qquad (3.2c)$$

where

$r = (k-1)/k$

and

$t = 1/k$ as L approaches zero.

Equation (3.2c) shows that when the load L is smaller than 100%, which is within our simulation range and is realistic, t is less than one. Therefore, if the database is badly placed, smaller lockable units imply a smaller probability of conflict. In this case, since the transaction size remains the same, a smaller probability of conflict does imply a smaller probability of deadlock and shorter response time.

To sum up, smaller lockable units always imply smaller probability of conflict.

The probabilities of deadlock (PD) are presented in Figure 3.2. Notice that PD is the conditional probability of a lock request causing a deadlock, given that the request conflicts. The unconditional probability of deadlock is the product of PC and PD, which is presented in Figure 3.3. These data are also analyzed in three steps: visual inspection, regression analysis, and analysis of the regression equation.

Figure 3.3 shows that for a fixed DZ, PD increases with both the multiprogramming level MP and the transaction size TZ. But in contrast to PC, the increase is larger with TZ than with MP.

If the diagonal line discussed previously is drawn for each table in Figure 3.3, the number below the line is always smaller than the corresponding number across the line, in sharp contrast to PC of Figure 3.1. Thus assuming equal loads L, a system with larger transactions and lower multiprogramming level has a higher probability of deadlock than a system with shorter transactions and higher multiprogramming level.

Similarly, regression analysis shows equation (3.3) a good fit for the data of Figure 3.3.

$$PD' = PDxPC = \frac{0.012(MP-1)^{1.07-0.24L}\ TZ^{3.61-3.48L}}{DZ^{1.99-1.79L}} \qquad (3.3)$$

$$L = \frac{MP\ x\ TZ}{DZ}$$

We must emphasize that PD is the probability of deadlock for a lock request, not a transaction. Equation (3.3) shows that when the load L is larger than 80%, the coefficient c is smaller than the coefficient b. Therefore, for a fixed load of 80% or greater, a system with shorter transactions and higher multiprogramming level has a higher probability of deadlock than a system with longer transactions and lower multiprogramming level. This rather surprising behavior is not immediately apparent from inspection of Figure 3.3. This behavior occurs because when the load is high and transactions are long, transactions deadlock and abort frequently; and abortions of long transactions means that more locks are freed. Thus there is less probability of a lock request causing a deadlock.

To analyze the relationship between PD and the lockable unit size, we replace DZ by kDZ and TZ by wTZ, and equation (3.3) becomes equation (3.4).

$$PD'' = t \times PD' \tag{3.4}$$

where

$$t = \frac{(MP-1)^{0.54Lr}\ TZ^{3.7Lr}\ w^{3.5-(3.7Lw)/k}}{DZ^{2.1Lr}\ k^{1.9-(2.1Lw)/k}} \tag{3.4a}$$

and

$r = (1-w/k)$

Setting w to k, equation (3.4a) becomes (3.4b), which shows that if and only if the load L is less than one, which is within the range of our simulation and is realistic, t is greater than one.

$$t = k^{1.6(1-L)} \tag{3.4b}$$

Thus, when the database is well placed, smaller lockable units imply a larger probability of deadlock.

Setting w to 1 for the originally badly placed system, equation (3.4a) becomes (3.4c), which shows that, within the range of our simulation, t is less than one. Therefore smaller lockable units reduce the probability of deadlock.

In summary, larger lockable units in a well placed system and smaller lockable units in a badly placed system reduce the probability of deadlock for lock requests and transactions.

$$t = \frac{(MP-1)^{0.54Lr}\ TZ^{3.7Lr}}{DZ^{2.1Lr}\ k^{1.9-(2.1L)/k}} \tag{3.4c}$$

where

$r = (1-1/k)$.

The average waiting times of a conflicting lock request are shown in Figure 3.4, which shows that the average waiting of a conflicting lock request increases with the multiprogramming level and the transaction size, and the increase is larger with the transaction size than with the multiprogramming level. The result is consistent with our intuition, because a lock request blocked by a long transaction must wait until the long transaction completes or aborts; and it takes longer for a long transaction to complete or abort. Also, if a similar diagonal line is drawn for each table, the number above the line is always larger than the corresponding number across the diagonal line.

Regression analysis shows equation (3.5) a good fit for the data of Figure 3.4.

$$WT = \frac{0.19(MP-1)^{3.4(L+0.2)^2-0.3}\ TZ^{2.7(L+0.15)^2+0.8}}{DZ^{4.1(L-0.04)^2-0.16}} \qquad (3.5)$$

Assuming the database is well placed, to reduce the granularity of the lockable units to 1/k of its original size, we increase the database size DZ and transaction size TZ to kDZ and kTZ respectively in equation (3.5), resulting in equation (3.6a). Equation (3.6a) shows that when the load L is less than 1.4, which is realistic and within the range of our simulations, smaller lockable units imply longer waiting for a conflicting lock request. The result is consistent with the earlier observation -- longer transactions induce longer waiting.

$$WT = k^{1.25-1.37(L-0.41)^2} \qquad (3.6a)$$

Assuming the database is badly placed, to reduce the granularity of the lockable units to 1/k of its original size we increase the database size DZ to kDZ, but leave the transaction size TZ unchanged in equation (3.5), resulting in equation (3.6b). Equation (3.6b) shows that when the load is light and k is small, t is greater than one -- longer waiting for a conflicting lock request. As shown earlier this is because when a database is badly placed and the load is light, reducing the size of the lockable units reduces the probability of deadlock. With less deadlocks, more transactions complete and less transactions abort. Since a transaction takes longer to complete than to abort, a blocked lock request waits longer.

$$WT = \frac{DZ^{4.1rL(qL-0.08)}}{(MP-1)^{3.4rL(qL+0.4)}\ TZ^{2.7rL(qL+0.3)}\ k^{4.1(L/k-0.04)^2-0.16}} \qquad (3.6b)$$

where

$r = (1 - 1/k)$
$q = (1 + 1/k)$

In summary, whether the database is well placed or badly placed, smaller lockable units increase waiting delay for a blocked lock request, except when load is extremely heavy, the database is badly placed, and the reduction in lockable unit size is large.

We next examined the standard deviation of waiting delays. These results can be summarized very simply.

Regression on the data of Figure 3.5 results in equation 3.7, which shows that the waiting delay may be approximated by an Erlangian distribution.

$$DV = 0.86 \times WT \quad (3.7)$$

DZ = 256

MP/TZ	7	10	12	16
7	.077	.104	.118	.135
10	.113	.145	.159	.176
12	.135	.169	.182	.198
16	.174	.210	.224	.236

DZ = 1025

MP/TZ	7	10	12	16
7	.020	.029	.034	.045
10	.030	.043	.050	.064
12	.037	.052	.061	.076
16	.050	.069	.081	.098

DZ = 512

MP/TZ	7	10	12	16
7	.040	.056	.066	.081
10	.059	.081	.094	.112
12	.072	.097	.111	.130
16	.096	.127	.142	.160

DZ = 2048

MP/TZ	7	10	12	16
7	.010	.015	.017	.023
10	.015	.022	.026	.034
12	.019	.026	.031	.041
16	.025	.036	.043	.055

PC : Probability of a Lock Request conflicting With Another Lock Request

Figure 3.1

DZ = 256

MP/TZ	7	10	12	16
7	.031	.078	.112	.183
10	.039	.102	.143	.207
12	.044	.115	.156	.218
16	.061	.141	.179	.232

DZ = 1024

MP/TZ	7	10	12	16
7	.006	.014	.026	.050
10	.008	.019	.028	.061
12	.007	.019	.033	.068
16	.007	.025	.040	.090

DZ = 512

MP/TZ	7	10	12	16
7	.014	.037	.052	.102
10	.014	.041	.068	.130
12	.017	.049	.079	.144
16	.021	.067	.102	.168

DZ = 2048

MP/TZ	7	10	12	16
7	.003	.006	.011	.024
10	.003	.006	.011	.024
12	.003	.009	.014	.029
16	.003	.009	.015	.034

PD : Conditional Probability of a Lock Request Causing a Deadlock after Conflict

Figure 3.2

DZ = 256

MP/TZ	7	10	12	16
7	.0024	.0081	.0132	.0247
10	.0044	.0148	.0227	.0364
12	.0059	.0194	.0284	.0432
16	.0106	.0296	.0401	.0548

DZ = 1024

MP/TZ	7	10	12	16
7	.00012	.00040	.00088	.00225
10	.00024	.00081	.00140	.00390
12	.00026	.00099	.00201	.00517
16	.00035	.00173	.00324	.00882

DZ = 512

MP/TZ	7	10	12	16
7	.0006	.0021	.0034	.0083
10	.0008	.0033	.0064	.0146
12	.0012	.0048	.0088	.0187
16	.0020	.0085	.0145	.0269

DZ = 2048

MP/TZ	7	10	12	16
7	.000030	.00009	.00019	.0006
10	.000045	.00013	.00029	.0008
12	.000057	.00023	.00043	.0012
16	.000075	.00032	.00063	.0019

PCxPD : Absolute Probability of a Lock Request Causing a Deadlock after Conflict

Figure 3.3

DZ = 256

MP/TZ	7	10	12	16
7	3.76	6.18	7.85	11.01
10	4.64	8.25	10.55	14.40
12	5.36	9.52	12.09	15.70
16	7.27	12.52	15.24	18.65

DZ = 1024

MP/TZ	7	10	12	16
7	3.09	4.49	5.60	8.26
10	3.19	4.93	6.42	10.57
12	3.35	5.34	7.25	11.80
16	3.54	6.65	9.30	16.05

DZ = 512

MP/TZ	7	10	12	16
7	3.33	5.12	6.60	9.72
10	3.66	6.18	8.50	13.37
12	3.88	7.19	9.91	15.28
16	4.71	9.77	13.53	19.43

DZ = 2048

MP/TZ	7	10	12	16
7	2.94	4.11	5.00	7.01
10	3.01	4.39	5.42	8.13
12	3.07	4.49	5.64	8.88
16	3.14	4.88	6.35	10.91

WT : Average Waiting Time of a Conflicting Lock Request after the Conflict

Figure 3.4

DZ = 256

MP/TZ	7	10	12	16
7	2.86	5.28	6.90	10.09
10	4.02	7.59	9.88	13.56
12	4.93	9.03	11.26	14.78
16	7.05	11.77	14.19	17.66

DZ = 1024

MP/TZ	7	10	12	16
7	1.95	3.35	4.36	6.92
10	2.16	3.94	5.50	9.62
12	2.38	4.52	6.49	10.98
16	2.68	6.15	8.97	15.51

DZ = 512

MP/TZ	7	10	12	16
7	2.29	4.08	5.44	8.61
10	2.78	5.45	7.76	12.51
12	7.19	6.71	9.22	14.32
16	4.32	9.45	12.90	18.07

DZ = 2048

MP/TZ	7	10	12	16
7	1.80	2.80	3.49	5.46
10	1.89	3.09	4.06	6.93
12	1.94	3.21	4.53	8.04
16	2.09	3.92	5.52	10.58

DV : Standard Deviation of the Waiting Times of Conflicting Lock Requests

Figure 3.5

4. Results of 20/80 Access Model

The results of simulating the 20/80 access model are shown in Figures 4.1 through 4.5. They are similar to the results of the random access model with heavier load. The reason is that when 20% of the database is used 80% of the time, the same load of the random access model becomes a heavier load. The probability of conflict, the probability of deadlock, and the average waiting of a conflicting lock request still increases with both the transaction size and the multiprogramming level. The probability of conflict increases faster with the multiprogramming level than with the transaction size, while the reverse is true for the probability of deadlock and the average waiting of a conflicting lock request. If diagonal lines are drawn for the tables (as previously explained), the number below the line is always larger than the corresponding number above the line for the probability of conflict, and the opposite is true for the probability of deadlock and the average waiting of a conflicting lock request. But the differences diminish as the load becomes lighter.

Applying regression analysis to data in Figure 4.1 results in equation (4.1). Similar to equation (3.1), it shows that the coefficient b is always larger than the coefficient c. The major difference between this equation and equation (3.1) is that the coefficient a of equation (4.1) is equal 3.7, much larger than the 0.72 of equation (3.1).

$$PC = \frac{3.7(MP-1)^{1.08+1.51L}\ TZ^{1.08+0.58L}}{DZ^{1.13+1.39L}} \qquad (4.1)$$

where

$$L = \frac{MP \times TZ}{DZ}$$

To examine the relationship between the probability of conflict and the lockable unit size, we replace TZ by wTZ and DZ by kDZ in equation (4.1), and obtain equation (4.2).

$$PC' = t \times PC \qquad (4.2)$$

where

$$t = \frac{DZ^{1.39rL}\ w^{1+(0.58Lw)/k}}{TZ^{0.58rL}\ MP^{1.51rL}\ k^{1+(1.39Lw)/k}} \qquad (4.2a)$$

and

$$r = 1 - w/k.$$

If the database is well placed, then w is equal to k, and equation (4.2a) becomes equation (4.2b), which shows that smaller lockable units reduce the probability of conflict, consistent with the result of the random access case.

$$t = k^{-0.81} \qquad (4.2b)$$

If the database is badly placed, then w is equal to one, and equation (4.2a) becomes equation (4.2c). Equation (4.2c) shows that if the load L is less 50%, which is within the range of our simulations and is realistic, smaller lockable units reduce probability of conflict. In summary, whether the database is originally well or badly placed, reducing lockable units reduces the probability of conflict. This result is the same as in the random access model.

$$t = \frac{DZ^{1.39rL}}{TZ^{0.58rL}\ MP^{1.51rL}\ k^{1+(1.39L)/k}} \qquad (4.2c)$$

where

$r = 1 - 1/k.$

Regression of the data in Figure 4.3 results in equation (4.3), which shows that when the load L is greater than 33%, the coefficient c is smaller than the coefficient b. Therefore, for a fixed load of 33% or higher, a system with higher multiprogramming level and smaller transactions has higher probability of deadlock than a system with lower multiprogramming level and longer transactions. This result is similar to the random access model.

$$PD' = PDxPC = \frac{0.8(MP-1)^{1.41+2.66L}\ TZ^{3.88-4.74L}}{DZ^{2.33-0.13L}} \qquad (4.3)$$

where

$$L = \frac{MP\ x\ TZ}{DZ}$$

To examine the relationship between the probability of deadlock and the lockable unit size, we replace TZ by wTZ and DZ by kDZ in equation (4.3), and obtain equation (4.4).

$$PD'' = t \times PD' \qquad (4.4)$$

where

$$t = \frac{TZ^{4.74Lr}\, w^{3.88-(4.74Lw)/k}}{(MP-1)^{2.66Lr}\, DZ^{0.13Lr}\, k^{2.33-(0.13Lw)/k}} \qquad (4.4a)$$

If the database is well placed, then w is equal to k, and equation (4.4a) becomes equation (4.4b). Similar to equation (3.4b), it shows that when the load L is less than 34%, which is realistic and within the range of our simulations, t is greater than one. That means larger lockable units reduce the probability of deadlock. This result is similar to the one found in the random access model.

$$t = k^{1.55-4.61L} \qquad (4.4b)$$

For the badly placed database, setting w to one in equation (4.4a) results in equation (4.4c), which shows that, within the range of our simulations, smaller lockable units reduce the probability of deadlock. This result is also similar to the one found in the random access model.

$$t = \frac{TZ^{4.74Lr}}{(MP-1)^{2.66Lr}\, DZ^{0.13Lr}\, k^{2.33-(0.13L)/k}} \qquad (4.4c)$$

Regression on the data in Figure 4.4 results in equation (4.5).

$$WT = \frac{0.037(MP-1)^{11.7(L-0.1)^2-0.24}\ TZ^{14.8(L-0.22)^2+0.25}}{DZ^{13.4(L-0.2)^2-0.27}} \tag{4.5}$$

Replacing DZ by kDZ and TZ by kTZ, equation (4.5) becomes equation (4.6a), which shows, as does equation (3.6a), that t is greater than one -- longer waiting delay for a conflicting lock request.

$$WT = k^{0.1+1.4(L-0.4)^2} \tag{4.6a}$$

Replacing DZ by kDZ, but leaving TZ unchanged, equation (4.5) becomes equation (4.6b), which shows, as does equation (3.6b), that when the load is light and k is small, t is greater than one. Therefore, in general, reducing the size of lockable units increases the waiting delay of a conflicting lock request, except when the load is heavy, the database is badly placed, and the reduction of lockable unit size is large.

$$WT = \frac{DZ^{13.4rL(qL-0.4)}}{(MP-1)^{11.7rL(qL-0.2)}\ TZ^{14.8rL(qL-0.44)}\ k^{13.4(L/k-0.2)^2-0.27}} \tag{4.6c}$$

where

$r = (1 - 1/k)$
$q = (1 + 1/k)$

Regression on the data in Figure 4.5 results in equation (4.7).

$$DV = -0.88 + WT \tag{4.7}$$

DZ = 512

MP/TZ	7	10	12	16
7	.119	.150	.163	.176
10	.166	.198	.210	.221
12	.192	.225	.237	.245
16	.236	.268	.277	.284

DZ = 2048

MP/TZ	7	10	12	16
7	.033	.045	.054	.067
10	.048	.067	.078	.095
12	.059	.080	.092	.110
16	.078	.105	.119	.137

DZ = 1024

MP/TZ	7	10	12	16
7	.063	.085	.098	.116
10	.093	.122	.135	.151
12	.110	.142	.155	.173
16	.143	.177	.191	.207

DZ = 4096

MP/TZ	7	10	12	16
7	.016	.024	.028	.036
10	.025	.035	.041	.053
12	.030	.042	.050	.064
16	.040	.057	.067	.083

PC : Probability of a Lock Request conflicting With Another Lock Request

Figure 4.1

DZ = 512

MP/TZ	7	10	12	16
7	.057	.124	.168	.230
10	.072	.149	.189	.242
12	.081	.161	.201	.246
16	.102	.181	.214	.247

DZ = 2048

MP/TZ	7	10	12	16
7	.011	.028	.044	.086
10	.012	.032	.050	.104
12	.013	.035	.060	.113
16	.015	.046	.078	.141

DZ = 1024

MP/TZ	7	10	12	16
7	.025	.057	.089	.156
10	.028	.075	.117	.181
12	.032	.086	.126	.195
16	.042	.109	.149	.211

DZ = 4096

MP/TZ	7	10	12	16
7	.005	.012	.019	.037
10	.006	.013	.021	.046
12	.005	.014	.023	.051
16	.005	.016	.029	.067

PD : Conditional Probability of a Lock Request Causing a Deadlock after Conflict

Figure 4.2

DZ = 512

MP/TZ	7	10	12	16
7	.0068	.0186	.0274	.0405
10	.0120	.0295	.0397	.0535
12	.0156	.0362	.0476	.0603
16	.0241	.0485	.0593	.0701

DZ = 2048

MP/TZ	7	10	12	16
7	.00036	.00126	.00238	.00576
10	.00058	.00214	.00390	.00988
12	.00077	.00280	.00552	.01243
16	.00117	.00483	.00928	.01931

DZ = 1024

MP/TZ	7	10	12	16
7	.0016	.0048	.0087	.0181
10	.0026	.0092	.0158	.0273
12	.0035	.0122	.0195	.0337
16	.0060	.0193	.0285	.0437

DZ = 4096

MP/TZ	7	10	12	16
7	.00008	.00028	.00053	.00133
10	.00015	.00045	.00086	.00243
12	.00015	.00058	.00115	.00326
16	.00020	.00091	.00194	.00556

PCxPD : Absolute Probability of a Lock Request Causing a Deadlock after Conflict

Figure 4.3

DZ = 512

MP/TZ	7	10	12	16
7	4.21	6.81	9.49	11.11
10	5.44	8.94	10.83	13.08
12	6.50	10.19	11.93	14.39
16	8.44	12.64	14.26	16.32

DZ = 2048

MP/TZ	7	10	12	16
7	3.22	4.81	6.11	9.36
10	3.50	5.70	7.69	12.36
12	3.65	6.32	8.99	14.55
16	4.19	8.50	11.92	18.43

DZ = 1024

MP/TZ	7	10	12	16
7	3.51	5.71	7.37	10.83
10	4.25	7.54	9.75	13.89
12	4.75	8.71	11.43	15.79
16	6.04	11.49	14.53	18.92

DZ = 4096

MP/TZ	7	10	12	16
7	3.07	4.33	5.28	7.66
10	3.15	4.65	5.96	9.49
12	3.21	5.00	6.58	10.87
16	3.40	5.77	7.99	14.54

WT : Average Waiting Time of a Conflicting Lock Request after the Conflict

Figure 4.4

DZ = 512

MP/TZ	7	10	12	16
7	3.48	5.94	8.49	10.35
10	4.89	8.36	10.14	12.49
12	6.04	9.62	11.38	13.75
16	7.98	11.93	13.54	15.65

DZ = 2048

MP/TZ	7	10	12	16
7	2.17	3.75	5.00	8.01
10	2.57	4.83	6.95	11.63
12	2.79	5.69	8.45	13.70
16	3.56	8.22	11.47	17.18

DZ = 1024

MP/TZ	7	10	12	16
7	2.60	4.78	6.41	9.72
10	3.59	6.96	9.24	13.02
12	4.21	8.18	10.72	14.92
16	5.73	10.98	13.72	17.68

DZ = 4096

MP/TZ	7	10	12	16
7	1.91	3.05	3.97	6.26
10	2.07	3.53	4.83	8.55
12	2.17	4.02	5.72	10.25
16	2.48	5.08	7.61	14.02

STD-DEV : Standard Deviation of the Waiting Times of Conflicting Lock Requests

Figure 4.5

5. Summary

We simulated the two-phase locking in a DBMS with fairly constant communication and IO delays. We collected performance data, and regressed these data into equations relating the performance of the DBMS to the multiprogramming level, the transaction size, and the database size. Using these equations we examine the interaction between the performance of a DBMS and lockable units size.

We found the performance behavior of a DBMS with random database access distribution quite similar to that of the 20/80 access distribution -- the 20/80 system behaves as a random access system in heavy load. In fact, the same regression models (equations) with different coefficient values fit both access models well except for the standard deviation of the lock request waiting delay.

The probability of conflict of a lock request increases more than linearly with the multiprogramming level and the transaction size; the increase is larger with the multiprogramming level than with the transaction. The probability of deadlock, the average waiting, and its standard deviation of a conflicting lock request also increase more than linearly with the multiprogramming level and the transaction size. But the increase is smaller with the multiprogramming level than with the transaction size.

The waiting delay of a conflicting lock request can be approximated by an Erlangian distribution in the random access model. This result can be extremely useful for researchers who use queueing theory to model a DBMS.

The results of this study have been validated, and can be extrapolated for database size of up to 12384, multiprogramming level of up to 128, and transaction size of up to 32.

So far we have concentrated on the basic factors of PC, PD, WT, and DV. We will next briefly discuss the combination of these blocking and restart variables into system throughput, a measure of performance which is more directly useful to a system designer.

In the highly functional model used here, all system resources are represented by the time to process lock requests. Since each request consumes the same time, we measure throughput by number of lock requests processed by transactions which finish.

In every case, throughput decreases with increasing TZ, if MP and DZ are held constant. As noted above, for longer transactions there are more conflicts, more deadlocks, and longer delays. The message for applications program design is clear. Transactions should be made as small as possible.

Also, throughput increases with increasing DZ if MP and TZ are held constant. This is the "badly placed locks" case, and it also can be anticipated from the analysis above. For random access of data, small granules will provide better throughput when both blocking and restart behavior are considered. However, because of the increasing communications and processing costs of lock management, the response time will increase. The optimal granularity can be calculated from the regression equations.

Finally, throughput first increases, and then decreases with increased MP if TZ and DZ are constant. Given a particular granularity and transaction size, for light loads, significant gains in throughput can be attained by increasing the multiprogramming level. However, as the system load becomes heavier, the losses to deadlock and restart more than outweigh the gains from increased concurrency.

6. References

[1] Gray, J.N., et al., "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Proc. IFIP Working Conference on Modelling of Data Base Management Systems, Freudenstadt, Germany, January 1976.

[2] Lin, W.T.K., "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed DBMS", ACM-SIGMOD 1981 International Conference on Management of Data, Ann Arbor, Michigan, April 1981.

[3] Nakamura, et al., "A Simulation Model for a Database System Performance Evaluation", AFIPS Proc. 1975 NCC Conference, Volume 44, May 1975.

[4] Spitzer, J.F., "Performance Prototyping of Data Management Applications", Proc. CAM'76 Annual Conference, Houston, Texas, October 1976.

[5] Munz, R., et al., "Concurrency in Database System - A Simulation Study", Proc. ACM SIGMOD International Conference, Toronto, August 1977.

[6] Ries, D., "The Effect of Concurrency Control on Database Management System Performance", Ph.D. thesis, Electronics Research Lab, University of California, Berkeley, 1979.

A DISTRIBUTED FILE SYSTEM ARCHITECTURE SUPPORTING HIGH AVAILABILITY

D. Stott Parker
Raimundo A. Ramos

Computer Science Department
University of California
Los Angeles, CA 90024

ABSTRACT

It is difficult to maintain redundant copies of resources such as files (in the interest of availability) while simultaneously keeping these copies mutually consistent when the network over which they are accessed is subject to partitioning. One approach is to sacrifice consistency during network partitions and reconcile inconsistent data later, when the network is repaired; this attitude has been taken in the *LOCUS* system at UCLA. Here we extend the "version vector" mechanism originally used to implement this approach so as to detect inconsistency when more than one file is used by a transaction. We then show how the resulting general scheme may be implemented in a way which gives the user a consistent view of system operation, even during network changes.

This work is supported by ONR grant N00014-C-79-0866.
Mr. Ramos is also supported by BNDE and CAPES, Brazil.

1. *Introduction*

LOCUS is a UNIX-based homogeneous distributed operating system under development at UCLA in which the design goals of network transparency and availability have been given particular emphasis. The user is to be able to access files throughout the network without being aware of either his or the files' locations. In the design of *LOCUS* it was felt that software should not require knowledge of the location of resources, since a change in network configuration could require tedious re-writing of code.

Redundancy is the traditional approach to increase availability of a distributed file system. A file can be replicated at many sites making unlikely that one user will not be able to access one of the multiple copies of the file. However, the existence of multiple copies brings the problem of keeping the *mutual consistency* of the many copies of the file. All updates made to one copy of a file should be propagated to all copies. Updates originated at the various nodes must be performed in each copy. Transmission delays and the order in which updates are applied must be taken into account to maintain the mutual consistency of all the copies, as well as the internal consistency of each copy.

One can say that availability increases with the number of copies of a file. This statement is true in read-only situations. However, if updates are allowed, multiple copies may provide no improvement when mutual consistency is emphasized. When a network with redundant data becomes partitioned, independent updates may cause inconsistencies to arise. If we adopt the obvious solution to this problem, of avoiding updates completely during network partitions, then having redundant copies will not increase availability at all.

Unfortunately, a file can be made inaccessible by network failures or site crashes, so availability cannot always be attained. This is true even when files are replicated to a high degree. In fact, a number of proposed systems respond to network partitions by making all but one copy of each file inaccessible for the duration of the network outage, in order to guarantee consistency of the multiple copies [AD 76,Elli 77,Thom 78,etc.]. Since one has no a priori knowledge of the kind of updates to be made to a file, one prevents updates to it - except possibly in one copy - so that independent updates of individual copies are impossible. No update scheme is effective against partitioning in guaranteeing consistency of a file, unless the file is always kept accessible only in one partition.

A *network partition* occurs when two or more sites become temporarily isolated from each other, and unable to communicate through the network, even though some or all of them are operational. It is important to realize that network partitions are a common occurrence in many applications. Networks can be interrupted for environmental reasons (as when a submarine submerges), or simply economic ones (a corporation connects its network only at night, since at that time telephone rates are lower). Even in the Ethernet [MB 76], gateways can be inoperative for significant lengths of time, while the Ether segments they normally connect operate correctly. Whatever the cause of network partitions, the resulting availability vs. mutual consistency tradeoffs presents serious questions to system designers.

In *LOCUS*, no assumptions are made about the frequency of network partitions other than that they are a reality and will occur at intermittent intervals. However, an environment is assumed in which file update rates are moderate and "conflicts" occur only rarely. In this setting we feel that mutual inconsistency of files will not be a serious problem, and that emphasizing the availability of copies of files over their mutual consistency will be a workable scheme. This is not the kind of environment, for example, characterizing a database system with high transaction rates and volatility, and the results here will probably not be useful for that area.

The paper is organized as follows. Section 2 briefly reviews previous work on version vectors and detection of mutual inconsistency. Section 3 provides an extension of this approach which we call "log filters" permitting the detection of mutual inconsistency of sets of files, not just single files. Section 4 then describes various considerations in integrating the above mechanisms into a consistent user interface. These considerations lead us to an actual design in Section 5, which is consistent even during periods of network change.

2. Detection of Single-file Mutual Inconsistency

In [PPRe 81] a technique is described for detecting (and possibly resolving) mutual inconsistency of multiple copies of a single file, involving "origin points" and "version vectors". The technique works basically by encoding the partial order describing the set of updates made at various sites into "vectors". Independent updates, leading to incomparable versions in the partial order, have incomparable vectors as a result. Version vectors thus may be used to detect independently made updates accurately; they provide a better check for mutual inconsistency than various naive methods.

Put briefly, whereas previous mechanisms such as timestamps detected sufficient conditions for a conflict to exist, version vectors seek to provide a mechanism detecting necessary and sufficient conditions for conflict. The case is made that version vectors detect as much about inconsistency as possible, given that one does not know anything about the semantics of updates applied to the file. For example, the equality of two copies of a file modified independently during a network partition does *not* imply their mutual consistency. Note that when our bank account is debited equally in two mutually inaccessible locations, we cannot assume that our current balance is the new value stored at both these locations!

Below we review the results from [PPRe 81]. The exposition is brief, and the reader is referred to the paper for a more thorough treatment.

Define a *partition* to be a subset of sites in the network in which all copies of a given file are maintained with mutual consistency. Note that this definition is not strictly tied to the physical details of network failure. Instead, partitions are defined relative to files and to the higher concept of consistency. Although two sites with different versions of a file f may be communicating for some time, we do not consider the sites to be in a common partition relative to f unless this difference in the two versions is resolved.

Definition
An *origin point* OP(f) of a file f is a system-wide unique identifier which is generated when f is created. It is an immutable attribute of f, although f's name is not. No matter how many times f is renamed or modified, OP(f) will not change. (Here, "f" refers to a specific file and is not a filename.)

An origin point for a file might be something like a (creation time, creation site number)-pair. Origin points do not uniquely specify files, but indicate when two files are based on a common file. They permit us to distinguish between two kinds of "conflicts" that partitioning may cause in file systems:

Definition
After a partition, a *name conflict* is detected if two or more files from the various partitions are found, which have the same name but have different origin points. A *version conflict* is detected if two or more files are found which have the same origin point, but have different contents and/or different names, i.e., the files are different modifications of a common original file. Generally, we say that a *file conflict* is detected after a partition whenever either a name- or a version conflict is detected. File conflicts are *reconciled* when file names again uniquely specify a file. (Equivalently, file conflicts are resolved when mutual consistency is restored.) Reconciliation may be achieved by individually modifying, deleting, or renaming the various copies.

Definition
A *Partition Graph* G(f) for any file f is a directed acyclic graph (dag) which is labelled as follows: The source node (and the sink node if it exists) is labelled with the names of all sites in the network having copies of file f, and all other nodes are labelled with a subset of this set of names. Each node can only be labelled with site names appearing on its ancestor nodes in the graph; conversely every site name on a node must appear on exactly one of its descendants. In addition, a node is marked with a "+" if f is modified one or more times within the corresponding partition.

Taking into account the definition of a partition, we see each node in G(f) corresponds to a point in time at which the labelled sites "synchronize" their information about f. All sites appearing in the node label reconcile any differences that might exist among their copies of f. Of course, all connections in G(f) between nodes indicate transitions of the network under partitions or merges.

An example of a partition graph is shown in Figure 1. Here there are three sites, A, B, C, which support f. Multiple partitions of these initially connected sites occur, so that at first sites A and B can communicate, but are isolated from site C. Later A and B become isolated, as does C, but B and C resume communication. Ultimately all three sites are reconnected at the final node of the graph. The file f is modified first in the {A,B} partition, and again in the {B,C} partition. Note that this sequence of modifications should *not* result in a version conflict notice since site B at all times has the latest version of f; intelligent implementation of conflict detection should realize this fact and avoid notifying site A that their f versions conflict with the current one.

Definition
A *version vector* for a file f is a sequence of n pairs, where n is the number of sites at which f is stored. The i-th pair $(S_i : v_i)$ gives the index of the latest version of f made at site S_i. In other words, the i-th vector entry counts the number v_i of updates to f

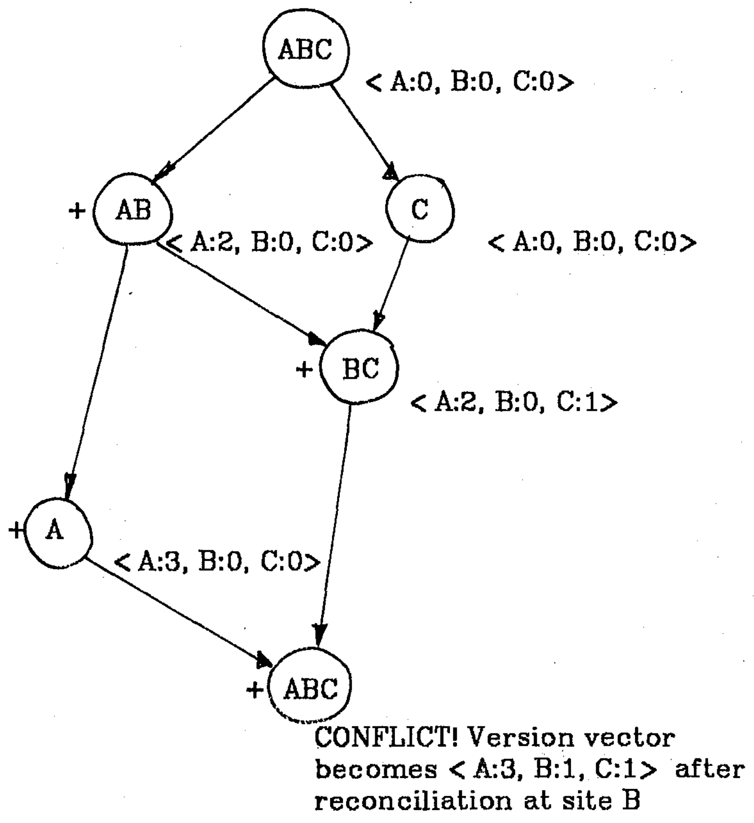


Figure 1. Partition graph G(f) for f with version vectors effective at the end of each partition

made at site S_i. (Actually, any strictly monotone value will suffice for v_i, such as a timestamp from site S_i.) We will use letters A,B,C,... to designate site names, and will write vectors in notation like < A:1, B:2, C:3>.

Definition
A set of version vectors are *compatible* when one vector is at least as large as any other vector in every site component for which they each have entries. A set of vectors *conflict* when they are not compatible.

For example, the version vector < A:3, B:4, C:2> dominates < A:2, B:1, C:2> so the two are compatible; and < A:3, B:1, C:2> and < A:2, B:4, C:2> conflict, but < A:3, B:1, C:2>, < A:2, B:4, C:2>, and < A:3, B:4, C:2> do not conflict, since the third vector dominates the other two. In Figure 1 version vectors are given for f in every partition. The vector < A:2, B:0, C:1> associated with the node labelled BC, indicates that f was modified twice at site A, once at site C, and nowhere else. Note in particular that during the {A,B} partition, the file is modified twice at site A. The final merge results in a conflict.

We adopt the following *usage of version vectors:*

[1] Each time an update to f originates at site S_i, we increment the S_i-th component of f's version vector by one. The vector is committed with the updated file.

[2] File deletion and renaming are treated as file updates. Deletion results in a version of the file of length zero, for example; when all versions of a file are of length zero, information on the file may be removed from the system.

[3] When version conflicts are reconciled within a partition, the S_i-th entry of the version vector for the reconciled file is set to be the maximum of the S_i-th entries of all of its predecessors, and in addition the site initiating the reconciliation increments its entry. This ensures future compatibility with any old versions of the file still remaining on the network.

[4] When copies of a file are subsequently stored at new sites, the version vector is augmented to include the new site information. The definition of compatibility above still holds. Thus vectors are not fixed in length, and may grow. Removal of copies from sites may be handled in an analogous manner.

Name conflicts and version conflicts are completely different in nature, and are detected differently. Keeping the origin point of a file is sufficient to detect name conflicts, but not version conflicts. The following result from [PPRe 81] indicates that version vectors detect necessary and sufficient conditions for there to be a version conflict.

Theorem 1
A version conflict must be reconciled at a node in G(f) if and only if f's version vectors conflict at that point.

As mentioned at the beginning of this section, then, version vectors serve to encode the partial order defined by the partition graph: If two updates in the graph are "incomparable", then the corresponding vectors are incomparable as well, and conflict.

3. Log Filters and Multi-file Inconsistency Detection

In the previous section a mechanism has been exhibited which guarantees that independent sets of updates to a single file f will be detected. While this will cover most of the usage patterns likely in an operating systems environment, this mechanism is not sufficient for detecting multi-file independent updates. Consider Figure 2, with the following scenario:

We have two files, f and g, replicated at sites A and B. A partition separates these sites. Initially, the copies of the files are mutually consistent, and each has version vector $<A:0,B:0>$. Transaction T1 at site A reads the contents of both f and g, then modifies file f, incrementing f's version vector to $<A:1,B:0>$. Similarly, transaction T2 at site B reads both f and g, but decides to modify file g, incrementing g's vector to $<A:0,B:1>$. At this point, the partition ends. The system discovers that f and g have each been modified, but only once. Using the results in §2, the system simply propagates the indicated updates. Unfortunately, this action is not correct: if we view f and g *together* as a data object (as we should in this case), we see that a conflict *should* occur.

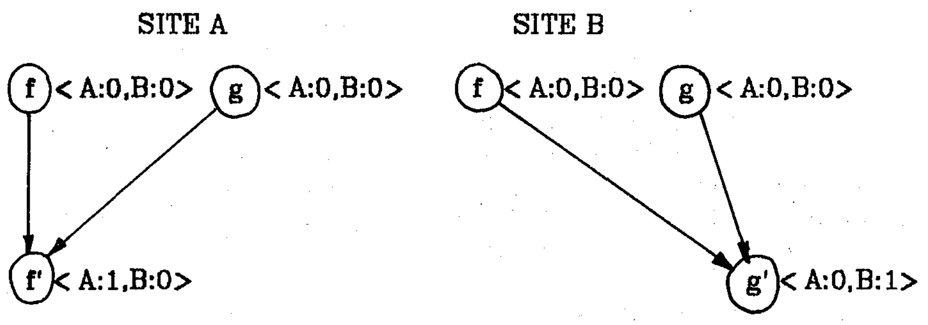


Figure 2. Multiple-file conflicts are not detected by version vectors alone

3.1. Conflicts and serializability

The problem of single-file and multi-file conflicts can perhaps best be described in terms of the traditional notion of serializability. If we consider the transactions T1 and T2 above as transactions, then by permitting them to execute "in parallel" in their individual partitions we achieve a result which is *not serializable;* cf. [BS 78].

Let us assume for the rest of this paper that all work done is broken into *transactions* which have the following structure:

[1] Initially a set $S = \{f_1{:}site_1, f_2{:}site_2, \ldots, f_m{:}site_m\}$ of files is specified, giving the sites at which the files are stored. Each of these copies are *locked* for the duration of the transaction. This set of files is the *readset* of the transaction.

[2] Some subset S' of these files is modified; all updates are committed simultaneously. The set S' is called the *writeset* of the transaction. Note that the writeset is always a subset of the readset.

[3] Messages are sent to all other sites holding copies of files in S', notifying them to change their copies to reflect the new modifications.

[4] The transaction completes, and all its locks are released.

A given transaction T_i may thus be viewed as having essentially two actions: first, a "lock" instruction L_i which locks its readset, then a "commit" instruction C_i which writes its writeset.

Clearly more sophisticated models of transactions are possible, and knowing more about the transaction's structure (both syntactic and semantic [PK 79]) will be useful in determining when conflict situations have arisen. Many extensions follow naturally, and we omit their description here. (For example, permitting transactions to create files is a straightforward extension, but requires handling of the "name conflicts" described above.) The assumption that the writeset is contained in the readset is not unquestionable, but we make it here mainly to avoid certain NP-completeness results [Papa 79].

Note in Figure 2 readset(T1) = readset(T2) = {f,g}, writeset(T1) = {f}, and writeset(T2) = {g}. However, more precisely, we must distinguish between the various copies of f and g. If we use subscripts to denote site locations, then we should say readset(T1) = $\{f_A, g_A\}$, writeset(T1) = $\{f_A, f_B\}$; and readset(T2) = $\{f_B, g_B\}$, writeset(T2) = $\{g_A, g_B\}$. It is immediate that in this case non-serializable execution is the cause of file conflicts.

This understanding is particularly important if we consider more intricate scenarios. For example, suppose there are three transactions T1, T2, T3 executed independently in different partitions, with:

readset(T1) = {f,g}   writeset(T1) = {f}
readset(T2) = {g,h}   writeset(T2) = {g}
readset(T3) = {f,h}   writeset(T3) = {h}

Although these transactions are pairwise-serializable, as a group they are not, and their independent execution will lead to file conflicts. Most treatments of serializability are centered around the notions of *histories* or *logs* [Papa 79]. However in this context such an approach is unsatisfying, not only because a global concept of time is intangible in distributed systems [Lamp 78], but also because we are concerned with operation of the system during network partitions. We therefore concentrate immediately upon graphs.

Definition
An *execution graph* $G = G(T_1, \ldots, T_n)$ is a directed acyclic graph with nodes $\{C_0, L_1, C_1, L_2, \ldots, L_n, C_n, L_{n+1}\}$ where $\{L_i, C_i \mid i=1,\ldots,n\}$ are the lock and commit operations from the transactions, C_0 initializes all files, and L_{n+1} reads all files. The edges of G are pairs (x,y) where either $x = L_i$ and $y = C_i$ or operation y reads what x writes.

Definition
For any pair of vertices x, y in a directed graph G, the relation $x <_G y$ is true if there is a path from x to y in G.

Definition
The *precedence graph* G' of an execution graph G contains G and all edges (x,y) such that $x <_G y$ is false and at least one of the following conditions holds:
(1) readset(x) $\cap$ writeset(y) $\neq \phi$
(2) writeset(x) $\cap$ readset(y) $\neq \phi$
(3) writeset(x) $\cap$ writeset(y) $\neq \phi$

Definition
An execution graph is *serializable* if its precedence graph is acyclic.

Definition
A set of files S is *put into conflict* if there exist transactions $T_1, \ldots, T_n$ whose execu-

tion graph is not serializable, and

$$S \cap \bigcup_{i=1}^{n} readset(T_j) \neq \phi .$$

Now intuitively, the problem posed by Figure 2 can be eliminated by saving sequences of version vectors for each *set* of files accessed by a process. With the example above, the version vectors for {f,g} at site A are < A:1,B:0> < A:0,B:0> , while at site B they are < A:0,B:0> < A:0,B:1> . These sequences of two vectors, viewed as single entities, are clearly incompatible if we make obvious modifications to the definition of compatibility.

If S is put into conflict then, after completion of $T_1, \ldots, T_n$, the version vector sequences for the sets $S_1, \ldots, S_n$ will be (intuitively) incompatible. This is the main insight we need to implement conflict detection in the next section. In the special case where $S_1 = \cdots = S_n = \{f\}$, we arrive at the previously-discussed result for single files, which is that version vectors are sufficient for detecting conflicts.

The big difference between what is being proposed here and previous work on serializability is that here we are concerned with accurate *detection* of serialization errors after they happen (as they would during network partitions), rather than with their *prevention*. Naturally, it is possible to extend the approach here as with timestamps to prevent these errors, and version vectors could be used to implement concurrency control. However, doing so would require restricted operation during partitions, in which at most one partition could modify a file, and we are more interested here in avoiding such restrictions. Our objective is to achieve high availability.

3.2. An implementation for multi-file conflict detection

In this section we overview the version vector/log filter mechanism for detection of file conflicts mentioned above. Due to space limitations we keep our presentation brief; it is hoped that by analogy with previous work for the single-file case various details of implementation will be clear.

To detect file conflicts for f, we must now monitor all transaction sets of files S containing f for serializability errors. This may be done as follows.

Let us imagine a *log* listing all process's filesets S as they are recorded. Define, for each file f, *extent*(f) to be the set of files involved with f by some filesets from the log. Formally,

$$extent(f) = \{ g \mid (f, g) \in R^+ \}$$

where R^+ is the transitive closure of the binary relation R defined by

$$R = \{(f_1, f_2) \mid \text{there is an S in the log such that } \{f_1, f_2\} \subseteq S\}.$$

This definition expresses the "extent" of all files involved in transactions with f.* Our idea here is to provide a simple means of finding *extent*(f) for all files f. Note that for example we can easily show

Prop
{f} is put into conflict *iff* *extent*(f) is put into conflict.

Another consequence of the above definition is that

Prop
$g \varepsilon \, extent(f) \longleftrightarrow extent(g) = extent(f)$

Thus *extent* divides the set of all files into equivalence classes. We store these equivalence classes, and update them as transactions occur. In fact the stored values of the equivalence classes and their version vector sequences are all we need to detect conflicts -- we do not, in particular, need the log. We call this stored set of classes a *log filter* because this equivalence class maintenance can be thought of as a process which filters the log information. Also, it turns out that the term "filter" is used in combinatorics to describe complete sub-partial orders. The log filter provides a simple refinement of the means of saving sequences of vectors suggested in the previous section.

Definition

A *log filter* LF is a family of sets, each set $S = \{f_1, \ldots, f_m\}$ drawn from a set of files. LF has the following two properties:

(1) If S, T are in LF, and $S \neq T$, then $S \cap T = \phi$.

(2) Each S in LF has associated with it a number of sequences of version vectors, each sequence consisting of m concatenated version vectors or null vectors, where m = cardinality of S. These sequences of version vectors (or null vectors) give the state of files following some transaction.

This may seem somewhat abstract, but it is really very straightforward. We are proposing a mechanism which is additional to version vectors for detecting multiple-file joint inconsistency. The filter LF for our file system is used in the following way:

[1] Initially, $LF = \phi$.

[2] Upon commit to a set of files $S = \{f_{i_1}{:}site_1, \ldots, f_{i_m}{:}site_m\}$ do the following things:

(a) if S is contained in some set $S' = \{f_1{:}site_1, \ldots, f_m{:}site_m\}$ in the LF already, attach the version vector sequence

$$\ldots < v_{i_1} > \ldots < v_{i_2} > \ldots < v_{i_m} > \ldots$$

* Viewed differently,

$$extent(f) = FIXPT \left[\bigcup_{\substack{S \text{ in log} \\ S \text{ contains } f}} S \;\cup \bigcup_{g \,\varepsilon\, extent(f)} extent(g) \right]$$

to S', where $<v_{i_j}>$ is the version vector for f_{i_j}, and the "..." indicate that null vectors should be used as placeholders.

(b) if S is not already contained in LF, incorporate S into LF using the fast UNION-FIND algorithm [AHU 74, Tarj 75] **:

```
$S_T := \phi$;
for each f in S do
    begin
    $S_f$ := FIND(f);  /* get current extent of f */
    if $S_f = \phi$ then $S_f$ = {f};
              /* f was newly created by transaction T
                - add it to the log filter */
    UNION($S_f$,$S_T$,$S_T$);
              /* add $S_f$ to extent of T */
    end
```

It is assumed here that the UNION operation also incorporates version vectors sequences in the obvious way.

[3] To check if a file is in conflict:

```
$S$ := FIND(f);  /* get extent in LF */
    if $S$ has incompatible version vector sequences
    then return (CONFLICT)
    else return (OK)
```

Here "incompatible" is a straightforward extension of the term in §2. We say one version vector sequence *dominates* another sequence if it is greater on all file entries where the two sequences are defined. A set of version vector sequences are *compatible* if there is one which dominates all the rest; otherwise, they are *incompatible*, or conflict.

[4] Entries in the log filter must be maintained as long as conflict might exist. This implies, for example, that during network partitions the filter may have to be fully maintained. When there are no network difficulties, however, the filter can drain as the system becomes quiescent.

** Recall that, in the UNION-FIND algorithm, we have a collection of mutually disjoint sets. The operation UNION(S_1,$S_{2,S}$) takes two of these sets, merges them, and names the resulting union S (S_1 and S_2 are destroyed). FIND(f) returns the name of the set containing f.

As observed in [Tarj 75] and [AHU 74], the UNION-FIND operations can be implemented very efficiently, so that for example a sequence of N-1 UNIONs and $M \geq N$ FINDs can be executed in time essentially linear in M. The overhead of log filter maintenance is thus small.

Upon reconnection of two (or more) partitions, entries from one log filter are added to the other(s) by using repeated application of step [2] above, where the sets S are now equivalence classes in the different log filters. The resulting log filter may be thought of as the union of its precursors, and is "less fine" in its partitioning of the set of all files.

We thus have a simple mechanism for detecting conflicts in a multiple-file environment. In the next sections we discuss problems of implementing such a mechanism in a real system environment.

4. Actual Resolution of File Conflicts

Suppose now that we are presented with a bona-fide file conflict. That is, following some partition we discover either two files which conflict in name, or a file which has been modified independently at two or more sites. What can be done? As has been noted elsewhere, there is no universal way to coalesce updates made in different partitions without knowledge of the update semantics. Since in many cases the semantics are sufficiently complicated that automated recovery is not possible, some kind of user intervention will be necessary.

The occurrence of some file conflicts is inevitable if one accepts the philosophy adopted here that file availability is important. Consistency and availability just appear to be fundamentally incompatible goals. However, the file conflict situation does not have to be made unpleasant for the user. As long as conflicts are handled consistently throughout the system, and the user's options are well-understood in any conflict situation, occasional mutual inconsistency of files should not offset the advantages obtained from increased file availability. The philosophy behind this statement is that inconsistency may be tolerable when one is aware of its possible extent. (Remember: we are assuming a network operating system environment, in which update rates are moderate and conflicts are rare.) This section therefore discusses what a coherent system policy on file conflicts involves, and then indicates how certain features might be implemented.

System policy must be defined for each of the following questions:

(1) When and how are file conflicts detected?

(2) Is permission to access a file altered by the fact that the file is in conflict?

(3) How are users informed of conflicts?

(4) What support does the system provide the user for resolving conflicts?

4.1. Detection of Conflicts

There are at least three times at which one might wish to detect conflicts following a partition merge:

(a) immediately after the merge, and

(b) upon access to files,

(c) sometime after (a) but before (b).

Which is best may ultimately depend upon other policy decisions made below (for example the means by which users are informed of conflicts). However it is easy to show that a general system policy cannot get by with conflict detection at only one of these three points. First, the system *must* cope with the case where file conflicts are detected upon access, because it is always possible that the user reference a file immediately following a partition merge (at the earliest point where conflict detection would be feasible). Moreover, system policy must be defined on what to do with active processes in different partitions which are caught independently modifying copies of a file (generating a conflict) when their partitions merge. Second, conversely, handling conflicts upon access is not adequate in itself, since for example the system must check for file deletions made in one partition which are to be propagated after the merge. Clearly conflicts must be detected using some kind of mixed strategy which operates at times (a), (b), and (c).

Detecting conflicts upon file access may be undesirable for at least two reasons: (1) File accesses will become slow following partition merges since all copies of an accessed file will have to be consulted in checking for a conflict. (2) Detection upon access may not provide a user with sufficient warning to recover from the conflict: he may be halfway through a long sequence of deletions and/or extensive file updates when he finally accesses the file and the conflict is noticed. This problem will be touched upon below in the discussion of how users are informed of conflicts.

Detecting conflicts immediately following partition merge may be undesirable because of its high temporary expense. Merges will be followed by heavy system traffic attempting to find name and version conflicts. Thus the system might grind to a halt every time faulty hardware is repaired! However, as with the consistency schemes in Section 1, it is easy to make arguments for the speed of information propagation. Having more information faster can never be detrimental.

Detecting conflicts gradually after merges might be implemented in several ways. One is to check all of a user's files when he logs onto the system, and notify him of any conflicts. Also he might be notified of repairs in network links, so that he can check for conflicts detectable after partition merges for himself. Another scheme is to set up detection processes which run constantly in background and, through repeated polling of directories, ultimately find all conflicts when the system is stable. Exactly what "gradually" might mean would be part of the system policy; one alternative is to assign these background processes a priority which could be varied as a reliability parameter; also one can parametrize the reliability levels of files, and have the background processes concern themselves with the highest reliability files first.

4.2. Accessibility of files in conflict

There are basically two alternatives for system policy on restricting access to files which are in conflict. First, one can permit all access regardless of the conflict. Second, then, one can restrict certain kinds of access (read, write, etc.) to certain users as soon as conflict is detected.

Our feeling is that the first alternative makes more sense. Restricting access is obviously inconsistent with the goal of availability, but a more compelling observation is that tasks which were not permitted to run after a partition merge because of access restrictions would have executed perfectly normally if the merge had occurred a few seconds later. This "discontinuous" behavior suggests real weakness in the second approach. In those situations where consistency is more important than availability, and version conflicts are likely, one would probably be better off prohibiting access after *network partitioning* instead of after file conflict detection.

4.3. Informing users of file conflicts

The first issue that must be addressed is: *Who* should be informed concerning a file conflict? Normally it might be most effective to notify the creator of a file, but more generally one should notify *all* users who have created conflicting versions, or have created different files which conflict in name. (One might even want to notify all users who have *referenced* the file in conflict, if this is possible, since they may wish to reference it again.) Whereas name conflicts are easily resolved with multiple users - everyone simply modifies their file name - version conflicts may require a fair amount of arbitration and possibly even intricate merging of the different file versions. This leads to the secondary question of what must be said to users when version conflicts arise.

The other main issue is now: How can the system react to, and inform the user of, conflicts when they are detected? Any policy decision here will strongly impact the system's structure. There are several possibilities, depending on when the conflict is detected. If the conflict is discovered upon file access, the system can:

(a) roll back the user's process (assuming that they can be "undone", in the sense of undoing a transaction), and notify the user somehow of the problem,

(b) suspend the user's process temporarily and enter an interactive routine permitting the user to inspect the alternative files and choose one,

(c) suspend the user's process temporarily and enter interactive conflict resolution software,

(d) just make an automated selection from the available choices using a fixed, well-publicized heuristic, and quietly continue the user's process.

Approach (a) is fine in transaction-oriented environments, where transaction processes can be re-spawned automatically to reattempt the same rolled-back task, but is less desirable in an interactive, network operating system environment since work can not always be recovered so easily. Approach (b) may not work for version conflicts in some applications, since no single version of the file may be adequate (consider an inventory file, for example). Further, approach (c) will also not generally work for version conflicts if different users have generated the differing versions, since then some arbitration may be necessary in finding a mutually agreeable resolution of the conflict. And clearly approach (d) will not always have the desired result, though in many situations it will make a reasonable choice.

In all cases the system must notify the affected users, via some kind of message or mail, of the conflict and of its resolution if one was decided upon. (The exact nature of the mail or message is yet another issue.) When the conflict is detected by the system "in background" with the users not immediately available for consultation, the only solution seems to be to notify the users by mail of their conflict.

Note that if we are prohibiting access from files in conflict, as discussed in Section 4.2, the situation here is somewhat clearer. Since this prohibition seems unwise, however, we will not pursue the subject further in this paper. An analysis appears in [Rudi 80,§4.5].

4.4. System support for conflict resolution

The system must provide at least three kinds of support for handling conflicts: (a) an interactive program for inspecting and choosing among conflicting files (Section 4.3), (b) automated detection and resolution of conflicts for files which will not be accessed (e.g., a file whose copies in different partitions have been independently deleted), and (c) a program for interactively editing and merging the various versions of a file into a consistent version. These facilities are vital in that a network-transparent system file structure provides the user with no nice way to distinguish between the conflicting versions (by definition, they all have the same name). Finally, the system may wish to perform (d) automated resolution for certain *types* of files whose structure makes reestablishment of mutual consistency routine. Directories and mailboxes are important examples [Fais 81].

In this section we will not attempt to describe the actual recovery mechanism used by the system in recovering from site or network failures. The actual dynamics of recovery (whether to use a centralized or distributed algorithm, what protocols should be employed, etc.) is a delicate problem with complex performance tradeoffs which will have to be deferred for later study. We remark, though, that the recovery strategy will be most effective if it handles failures of short duration differently than those of long duration. Transient faults should not bring about the exhaustive comparison of directories and system information at nodes in the network that one expects from recovery after long network partitions.

The system either solves a file version conflict automatically by calling a conflict-reconciliation software or informs the user that real conflict was found where automated resolution is not possible. In the latter case the user has to reestablish file consistency. This system policy puts the burden of reestablishing file consistency squarely on the shoulders of the user, but only when there is a real conflict which cannot be dealt with otherwise.

User file conflict reconciliation might raise a lot of problems. First, there is the problem of deciding who is the user responsible for reconciling the conflicting files. We could say that only the owner of the file can perform file reconciliation. However, for some applications it would be desirable that either a superuser or a set of users could reconcile the file. Second, there is the problem of inconsistencies kept for a long time if the user responsible for doing reconciliation is not present or if he does not want to decide right away. Finally, there is the problem of rolling back transactions of users not present. They would not be aware of the reconciliation operation and they would not know that their transactions must be redone. It would take a long time until the user logs in and discovers a message from the system asking him to redo some transactions.

We might have even political battles between users. Since files are shared, each user could decide that his updates are the correct ones. Users could have the tendency to reconcile files by deleting others' updates and keeping their own updates. Users would prefer to maintain their transactions and would ask other transactions to be rolled back. It seems that this procedure does not work. Users will either have heavy discussions every time a multiple copy file is inconsistent and a conflict resolution decision has to be made or they would create new files for own use destroying the advantages of having multiple file copies and reducing availability.

The advantage of automatic reconciliation is that inconsistencies would be corrected as soon as they are detected, thus, files would be as consistent as possible. Automatic reconciliation can be performed only by furnishing the system enough semantic information for each application. The system must provide to the users tools for reconciling conflicts. Automated reconciliation procedures can be made available using default parameters. Users can set these parameters according to the semantics of each application. Users could get together for deciding in one user group a reconciliation agreement for each file. This agreement would create a standard reconciliation procedure avoiding group meetings over reconciliation and avoiding long-term complex inconsistency. Another advantage is that all users will be aware in advance of system behavior for file conflict resolution. Moreover, inconsistencies will not be kept for a long time since the system can call the conflict-resolution procedure just after finding a conflict or upon partition merge. As a result, automatic reconciliation is advantageous over user reconciliation achieving a system with high availability and at the same time providing data consistency.

5. A basic system policy

In section 3 we have presented a simple multi-file conflict detection mechanism. The proposed approach uses sequences of versions vectors and log filters. Instead of keeping a list of sequences of version vectors for every update made in the system, log filters are used to reduce the number of sequences of version vectors the system needs to store as log information. In order to detect conflicts we need keep only the "latest" sequence, i.e., those sequences which are not dominated by any other sequence.

In section 4 we have discussed problems of implementing conflict detection and problems of defining a consistent user interface. Section 4 discusses what a coherent system policy on file conflicts involves, and then indicates how certain features might be implemented.

In this section we outline a simple policy for system management of a (potentially inconsistent) distributed file system having the transparency properties described above.

Files in our system must be implemented in such a way that they permit conflict detection as in section 3.

Our conflict resolution policy is based on the notion of a *transaction*. Any file update operation must be within the boundaries of a transaction, limited by begin and end statements. The transaction-like format would be as follows:

```
begin;
  get $f_1, f_2, \ldots, f_n$
    .
    .
    .
end;
```

The idea is that the get statement tries to establish an "environment" (somewhat like the pseudo-temporal environments presented in [Reed 78]). The user is guaranteed that this environment will exist for the duration of his transaction *unless* there is a non-recoverable failure. Such a failure will lead to the transaction being aborted.

The *get* statement has the following functions:

[1] The get statement informs the system the set $S = \{f_1, f_2, \ldots, f_m\}$ of files which the user plans to use. These files are assumed to be read subsequently, and some of them are eventually updated. This set S of files is the readset of the transaction and forms an environment where the transaction is permitted to work on.

[2] The get statement checks that each of the files f_i is consistent (i.e., it does not have a file conflict). If a file conflict is found the transaction is not executed. The system either solves the conflict automatically by calling a conflict-reconciliation software or informs the user that real conflict was found where automated resolution is not possible. In the latter case the user has to reestablish file consistency.

[3] Each file specified in the get statement is locally locked for the duration of the transaction.

At the transaction's end, that is, when the *end* statement is executed the following situation occurs:

[1] The system updates the log filter.

[2] All updates are committed simultaneously. Update propagation is performed by sending messages to all other sites holding copies of files specified in the writeset of the transaction.

[3] We might wish to check for conflicts at this point. If a file conflict is found the transaction should be rolled back. Again, this is not really necessary, since

conflicts will be detected later. However, it may be desirable to enforce consistency to this degree. This mechanism is appropriate for applications where file inconsistencies must be detected as soon as possible.

[4] The transaction completes, and all its locks are released.

The transaction solution actually solves the problems of implementing conflict detection and defining a consistent user interface as presented in Section 4. For example:

[1] Using log filters is a nice way to handle the 'How do we solve the after-partition merge?" problem. File accesses would become slow following partition merges since all copies of a file will have to be consulted in checking for a conflict. However, using log filters, conflict detection can be done easily either on demand or by a constantly-running background process.

[2] Using the *get* statement eliminates surprises in the middle of execution ('I am sorry, your file is in conflict. Please log off"). The user may be halfway through a long sequence of deletions and/or extensive file updates when he finally accesses one file which is in conflict. Our policy avoids this by requiring the user to inform the system in advance of the files he plans to use. File conflicts can be checked at this time, giving the user an environment which is conflict-free when the transaction starts to execute.

The system policy we are proposing here can be seen as a method for implementing concurrency control. The difference between traditional concurrency control mechanisms and our proposed method is that here we are performing *detection* rather than *prevention* of synchronization errors. In some cases prevention cannot be attained forcing us to use detection. For example, currently we recommend against global locking (for the needs of a distributed file system at least). Global locking is hard to be implemented and even impossible under network partition. This still leaves us with a consistent approach if local locking is used (i.e., only one process may use a particular copy of a file) since non-serializable updates can simply be detected as conflicts later on. Such an approach is not desirable where conflict reconciliation is expected to occur a lot or where it is especially painful.

By checking for conflicts both in the beginning and at the end of a transaction, we avoid all update conflicts within partitions. If a conflict is detected at the end of a transaction, then another user has updated one of the transaction's files. The transaction is not completed at all, but it is rolled back and the user informed of the problem (note that this process can be easily automated).

When users are processing files independently in different partitions, one user is notified of file conflicts whenever he starts a transaction and the partitions were already merged. The transaction approach assures that once a file conflict is detected no updates are performed at all on that file until the file is reconciled. Note that here we are avoiding the so called "domino" effect.

The described transaction approach might present the two following drawbacks:

[1] First, there is the possibility of a *livelock*, i.e., starvation of one or more transactions. Suppose the following situation occurs. Transaction T1 starts executing using file f, version <1>. Upon execution of the get statement, in the beginning of the transaction, the system verifies that there is no conflict for file f allowing the transaction to proceed. Assume now that transaction X starts executing using the same file f. Since transaction T1 has not finished we still have version <1>. While transaction X is executing, transaction T1 ends and writes file f version <2>. After that, when transaction X ends it will find out that a conflict exists since file f has been updated by someone else. Therefore, transaction X has to be rolled back and started again. This situation could repeat for other transactions. Consequently, transaction X would never complete successfully and we say that X starves to death. This situation is known as a livelock and it is illustrated by the figure below. Note that transactions T1, T2, and T3 complete successfully, whereas transaction X never executes up to completion.

If the degree of concurrency is high, conflicts would occur frequently causing waste of processing power, since transactions are executed up to the end before a conflict is detected. Moreover, processing time and effort is necessary to roll back transactions.

The transaction approach presented here is suitable for systems with low degree of concurrency and where conflicts occur only rarely. However, it has been pointed out by [BSR 80] that in a large class of applications, most transactions require little or no synchronization at all because they never interfere with each other. In most applications the operations of transactions are known a priori and most of them do not conflict.

[2] Second, there is the possibility of occurring *unfairness*. Suppose there is one long transaction T1 updating file f version <1>. After a substantial amount of time, a short transaction T2 starts, updates just one record of file f, and terminates successfully writing file f version <2>. Later on, when transaction T1 ends, a conflict is detected since file f has been modified by transaction T2 having a different version. Consequently, the system has not been fair to transaction T1.

The longer the transaction, the greater the probability of occurring a file version conflict. The system would not be fair for long transactions due to the presence of small transactions accessing the same files. In order to solve this problem we do not need to limit the transaction size. The system will work well for either small or long transactions. What we need is to have transactions with approximately the same size. However, long transactions are not desirable, because if it is necessary to roll them back more work needs to be redone.

One user might complain saying that the above transaction approach is not suitable for him because he does not always know in advance the set of files he would use in one transaction. He might have subroutines which require files not present in the set of files listed in the get statement at the beginning of the transaction. Therefore, the system must provide more flexibility by allowing the user to specify multiple get statements. One transaction with multiple get statements looks like as the one

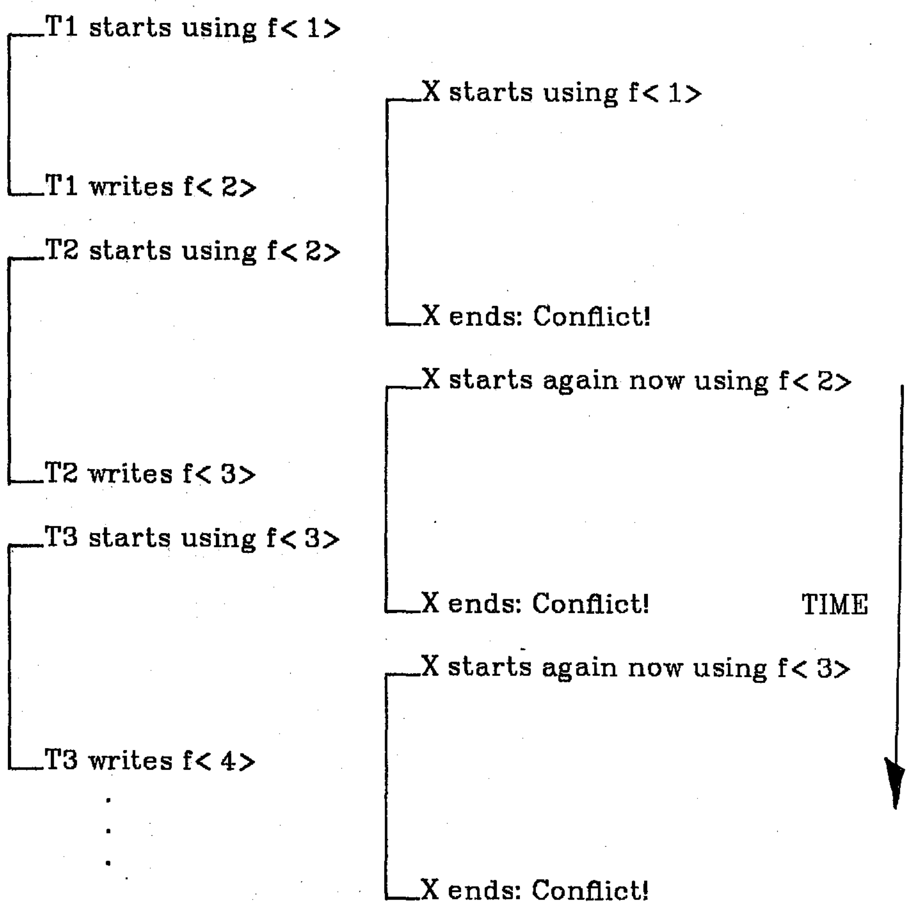


Figure 3. Transaction X in a livelock situation.

shown below:

```
begin;
  get $f_1, f_2, \ldots, f_n$
  .
  .
  .
  get $g_1, \ldots, g_m$
  .
  .
  .
end;
```

Note that transactions still have boundaries established by the begin and end statements. Multiple get statements would request *multiple* checks for version conflicts. For each issued get statement, the system might check the version vectors looking for conflicts. The difference between the single-get and the multiple-get is that, in the former case, the user has no surprises in the middle of the transaction (the transaction is not started unless there is no conflict), while in the latter case, the user can have surprises in the middle of his transaction ("Sorry, your transaction cannot proceed. A file version conflict was found."). If a file version conflict is found after the execution of a inner get statement the transaction has to be rolled back and started again. This mechanism works similarly to the conflict checking performed by the end statement, which may also cause transactions to be rolled back and started again. Eventually, to roll back transactions is really not necessary if we can live with some inconsistency. The system might just declare a conflict and inform the user who can decide either to roll back the transaction or to proceed even though the files are not consistent. Without knowing the semantics of each application the system cannot take this decision. One must provide the user flexible tools to help him to take the best solution.

Note that the single-get statement given at the beginning of the transaction assures a conflict-free environment when the transaction starts to execute but requires the user to inform in advance of the set of files to be used. Multiple-get statements give more flexibility to the user by allowing the specification of other files to be used in the transaction. However, with multiple get statements the environment for the transaction must be adjusted multiple times, and this adjustment is not guaranteed to be successful.

6. Conclusions

We have shown that it is possible to implement a distributed file system which supports redundant copies of files effectively, even in the face of network partitioning. The kind of environment in which the results presented here will be useful are those akin to a network operating system in which file availability is a greater problem than mutual consistency among file copies. The proposed system is very simple in structure, and involves the addition of really only a few new constructs to the file system design: file origin points, version vectors, and the log filter. These constructs were shown to be adequate for detecting "file conflicts" (mutual inconsistencies) in an extremely straightforward way, requiring little system overhead. Strategies for resolution of detected file conflicts were then discussed. A simple conflict resolution policy was described, based on the notion of a transaction. The resulting user interface, although more restrictive than most interactive system interfaces, provides a consistent view on the state of the network and various other benefits.

References

[AD 76] Alsberg, P.A. & J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," Proc 2nd Intnl. Conf. on Software Engineering, 13-15 October 1976.

[AHU 74] Aho, A.V., J.Hopcroft, J.D.Ullman, 'The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974, sections 4.6-4.7.

[BS 78] Bernstein, P.A., Shipman, D.W., "A Formal Model of Concurrency Control Mechanisms for Database Systems", Proc 3rd Berkeley Workshop on Distributed Data Management & Computer Networks", August 29-31, 1978.

[BSR 80] Bernstein, P.A., Shipman D.W., Rothnie J.B., "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, March 1980.

[Elli 77] Ellis, C.A., "A Robust Algorithm for Updating Duplicate Databases," Proc 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, 1977, pp. 1146-158.

[Fais 81] Faissol, S., "Operation of Distributed Database Systems Under Network Partitions", Ph.D. dissertation, UCLA Dept. of Computer Science, July 1981.

[FH 72] Farber, D.J. & Heinrich, F.R., 'The Structure of a Distributed Computer System -- The Distributed File System," Proc ICCC, 1972.

[Gray 78] Gray, J., "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Ed. by R. Bayer et al., NY: Springer, 1978.

[HS 78] Hammer, M. & D. Shipman, "An Overview of Reliability Mechanisms for A Distributed Data Base System," Spring Compcon 78, Feb 28-Mar 3, San Francisco, pp. 63-65.

[KR 79] Kung, H.T. & J.R. Robinson, "On Optimistic Methods for Concurrency Control," Proc. 5th VLDB Conf., October 1979, Rio de Janeiro. To appear in ACM TODS.

[Lamp 78] Lamport, L., 'Time, Clocks, & the Ordering of Events in a Distributed System," CACM vol. 21, 7, 558-565 (July 1978).

[LS 76] Lampson, B. & H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Technical Report, Xerox PARC, 1976.

[MPM 77] Menasce, D.A., G.J. Popek & R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Systems," Tech. Rept. UCLA-ENG-7808, Dept. of Computer Science, UCLA, October 1977.

[MB 76] Metcalfe, R.M. & D.R. Boggs, 'Ethernet: Distributed Packet Switching for Local Computer Networks," CACM Vol 19, 7, 395-404 (July 1976).

[Papa 79] Papadimitriou, C.H., "The Serializability of Concurrent Database Updates", Journal of the ACM, v.26, no.4, 631-653 (October 1979).

[PK 79] Papadimitriou, C.H., Kung, H.T., "An Optimality Theory of Concurrency Control for Databases", Proceedings 1979 ACM SIGMOD, Boston, May 30-June 1

[PPRe 81] Parker, D.S., Popek, G.P, Rudisin, G., et al., "Detection of Mutual Inconsistency in Distributed Systems," Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks, February 1981.

[Reed 78] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," MIT LCS Technical Report number MIT/LCS/TR-205, 1978.

[RG 77] Rothnie, J.B. & N. Goodman, "A Survey of Research and Development in Distributed Database Management", Proc. 3rd VLDB, Tokyo, October 1977, pp. 48-61.

[Rudi 80] Rudisin, G.J., "Architectural Issues in a Reliable Distributed File System," M.S. Thesis, Dept. of Computer Science, UCLA, Report UCLA-ENG-8014 SDPS-80-001, April 1980.

[SM 78] Shapiro, R.M., & R.E. Millstein, "Failure Recovery in a Distributed Data Base System," Proc. Spring COMPCON, Feb 28-Mar 3, 1978, pp. 66-70.

[Ston 79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Trans. on Software Engg., vol. SE-5, 3, 188-194 (May 1979).

[Tarj 75] Tarjan, R.E., "Efficiency of a Good But Not Linear Set Union Algorithm", JACM 22:2, April 1975, 215-225

[Thom 78] Thomas, R.F., "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases," Proc. Spring COMPCON, Feb 28-Mar 3, 1978, pp. 56-62.

[TSF 78] Thomas, R.F., R.H. Schantz, H.C. Forsdick, "Network Operating Systems," Technical Report RADC-TR-78-117, Rome Air Development Center, May 1978.

SITE INITIALIZATION, RECOVERY, AND BACK-UP IN A DISTRIBUTED DATABASE SYSTEM*

Rony Attar
Philip A. Bernstein
Nathan Goodman

Aiken Computation Laboratory
Harvard University
Cambridge, Massachusetts 02138

August 1, 1981

*This work was supported by the National Science Foundation, grant number MCS79-07762, by the Office for Naval Research, contract number N00014-80-C-647, by Rome Air Development Center, contract number F30602-81-C-0028, and by the Dr. Chaim Weizmann Post-Doctoral Fellowships for Scientific Research.

ABSTRACT

Site initialization is the problem of integrating a new site into a running distributed database system (DDBS). Site recovery is the problem of integrating an old site into a DDBS when the site recovers from failure. Site backup is the problem of creating a static backup copy of a database for archival or query purposes. We present an algorithm that solves the site initialization problem. By modifying the algorithm slightly, we get solutions to the other two problems as well.

Our algorithm exploits the fact that a correct DDBS must run a serializable concurrency control algorithm. Our algorithm relies on the concurrency control algorithm to handle all inter-site synchronization.

1. THE SITE INITIALIZATION PROBLEM

Site initialization is the problem of integrating a new site into a distributed database system (DDBS). The goal is to make the new site look like all other sites. In particular, transactions must be able to access data at the new site in the same way as they access data at all other sites. The main problem is to bring the database at the new site up-to-date relative to the rest of the system. The problem is caused by replicated data: if the new site stores datum X and there are copies of X elsewhere in the system, then the value of X at the new site must agree with its value in the rest of the system. There is a simple brute force solution to the problem: just turn off the DDBS, wait for all activity to subside, and then load the new site's database in bulk. Our solution is almost as simple as this, but far more practical.

Our algorithm exploits the fact that a correct DDBS typically runs a serializable concurrence control algorithm (cf. [BG]). Concurrency control is the activity of coordinating transactions that access a database concurrently. The goal is to prevent concurrent transactions from interfering with each other. This goal is usually formalized by the concept of serializability (e.g. [BSW, EGLT, Pa, SLR, Th]): an execution of transactions is *serializable* if it is equivalent to an execution in which transactions execute serially, one after the other with no concurrency. Many algorithms are known for attaining this goal, e.g. *two phase locking* and *timestamp ordering*.

As we will see, the site initialization problem can be neatly framed in terms of serializable executions. Once stated in these terms, a simple solution will stare us in the face. All we have to do is:

(1) turn on the concurrency control algorithm at the new site;

(2) tell all other sites to begin updating the replicated data at the new site; and

(3) not let any transaction read a datum X at the new site until X has been updated at least once.

These three steps are a sketch of our algorithm. The rest of the paper fills in the details, and explains why the algorithm works. We also show how to use the algorithm to solve site recovery and backup problems.

2. BASIC CONCEPTS

A *distributed database system* (DDBS) is a set of sites interconnected by a network. Each site runs two software modules: a transaction manager (TM), which supervises the execution of transactions; and a data manager (DM), which processes read and write operations on data stored at the site.

A *logical database* is a set of *logical data items*, denoted X,Y,Z. A copy of a logical data item stored at a site is called a *physical data item*. We use $x_1,\ldots,x_m$ to denote the physical copies of X.

A *transaction* is a program that accesses the database by issuing READ and WRITE operations on logical data items. For notational convenience we assume that a transaction issues all of its READ's before any of its WRITE's.

Each transaction's execution is supervised by one TM. When a transaction issues an operation READ(X), its TM selects a copy of X, say x_i, and issues an operation read(x_i) to the DM that manages x_i. (We use upper case for logical operations and lower case for physical ones.) When a transaction issues an operation WRITE(X), its TM issues an operation write(x_i) for *every* copy x_i of X.

The logical data items that a transaction reads (respectively writes) are called the transaction's *readset* (respectively *writeset*).

We mathematically model executions of transactions in a DDBS by a log. A log describes the order in which read and write operations are processed by

DM's. Formally, a *log* is a partial order* of read and write operations. For example,

$$\begin{tikzcd}
 & & r_2[x_1] \arrow[r] & w_2[x_1,x_2] \arrow[r] \arrow[dd] & r_3[x_1] \arrow[r] & w_3[z_1] \\
L_1 = & w_o[x_1,x_2,y_1,z_1] \arrow[ur] \arrow[dr] & & & & \\
 & & r_1[y_1] \arrow[r] & r_1[x_2] \arrow[r] & w_1[x_1,x_2,y_1] &
\end{tikzcd}$$

is a log. Notationally, $r_i[x_j]$ (resp. $w_i[x_j]$) denotes the execution of a read (resp. write) operation by transaction i on data item x_j. The arrows indicate the partial order, which represents the order in which operations were executed. So, in L_1, $w_o[x_1,x_2,y_1,z_1]$ executed before any other operation; $r_2[x_1]$ executed before $w_2[x_1,x_2]$ and $r_1[x_2]$, but it executed concurrently with $r_1[y_1]$; and so forth.

We place one more constraint on the allowed form of logs: for each physical data item x_i, all *conflicting* operations on x_i must be totally ordered,** where two operations on x_i *conflict* if (at least) one of them is a write. That is, for each x_i, we know the exact order in which conflicting operations occurred.

---

*A *partial order* is a binary relation, $\leq$, that is *reflexive* ($a \leq a$), *anti-symmetrical* ($a \leq b$ and $a \leq b$ implies $a=b$), and *transitive* ($a \leq b$ and $b \leq c$ implies $a \leq c$). Traditionally, a distributed execution is modelled as a set of sequential logs, one per DM [BG]. We prefer using partial orders because they allow operations from different DM's to be ordered and they do not require ordering unrelated operations that can be executed concurrently at the same DM.

**A *total order* is a partial order in which every pair of elements are related (i.e., $a \leq b$ or $b \leq a$). A total order is the same as a *sequence*.

Two logs are *equivalent* if they represent executions that produce the same final database state, and if each transaction performs the same computation in both executions. The following proposition states a well-known, and very useful, characterization of log equivalence. We need one more definition first. Two operations *conflict* if they operate on the same physical data item and one is a write.

Proposition 1 Two logs are equivalent if they contain the same operations, and every pair of conflicting operations appear in the same order in both logs.

3. CORRECTNESS CONCEPTS

The correctness of any system must be defined relative to users' expectations. Intuitively, a system is correct if it does what users want it to do. We assume that users expect a DDBS to behave like a *serial transaction processor*; that is, users expect the DDBS to behave as if it were processing transactions one at a time, against the logical, non-replicated database. (This assumption is adopted almost uniformly in the literature.) A DDBS is *correct* if it behaves like a serial transaction processor in this sense.

In this section we analyze DDBS correctness using the basic concepts of Section 2.

A *serial log* is a total order of operations such that for every pair of transactions, all operations of one transaction precede each operation of the other. For example,

$$L_2 = w_0[x_1,x_2,y_1,z_1] \rightarrow r_2[x_1] \rightarrow w_2[x_1,x_2] \rightarrow r_3[x_1] \rightarrow w_3[z_1] \rightarrow r_1[y_1] \rightarrow r_1[x_2] \rightarrow w_1[x_1,x_2,y_1]$$

is a serial log.

Consider any read operation in a serial log, e.g. $r_2[x_1]$ above. This operation is said to be *read-from* the nearest write operation before it that writes into its argument. E.g. in L_2, $r_2[x_1]$ reads-from $w_o[x_1,x_2,y_1,z_1]$, while $r_3[x_1]$ and $r_1[x_2]$ read-from $w_2[x_1,x_2]$. Similarly, transaction T_i *reads-x_k-from* T_j if T_i indeed reads x_k, T_j writes x_k, and $r_i[x_k]$ reads-from $w_j[x_k]$. E.g. in L_2, T_2 reads-x_1-from T_o.

A serial log is *one-copy equivalent* (or simply *1-serial*) if for each transaction T_i, and for each x_k that T_i reads, T_i reads-x_k-from the last transaction before T_i that writes into *any copy* of X.

The reader can verify that L_2 is 1-serial. However, if we change $w_2[x_1,x_2]$ to $w_2[x_2]$, the resulting log is not 1-serial.

$$L_3 = w_o[x_1,x_2,y_1,z_1] \rightarrow r_2[x_1] \rightarrow w_2[x_2] \rightarrow r_3[x_1] \rightarrow w_3[z_1] \rightarrow r_1[y_1] \rightarrow r_1[x_2] \rightarrow w_1[x_1,x_2,y_1]$$

L_3 is not 1-serial, because T_3 reads-x_1-from T_o, which is not the last transaction before T_3 that wrote into any copy of X.

A 1-serial log represents a serial execution of transactions in which the replicated copies of each data item behave like a single logical data item. Therefore, *every 1-serial log is correct* in the sense defined at the beginning of this section.

A log is *serializable (SR)* if it is equivalent to a serial log. A log is *1-serializable (1-SR)* if it is equivalent to a 1-serial log. Since every 1-serial log is correct, and since every 1-SR log is equivalent to a 1-serial log, *every 1-SR log is also correct.*

We adopt 1-SR as our basic notion of correctness for the rest of this paper.

If sites are never added to a DDBS and sites never fail, attaining 1-SR is little more than a concurrency control problem. All we have to do is:

(1) make sure that each transaction writes into all physical copies of its writeset, as described in Section 2; and

(2) synchronize read and write operations using any serializable concurrency control algorithm, such as two-phase locking.

The following proposition states the correctness of these steps in terms of logs.

Proposition 2 A log is 1-SR if every transaction in the log writes into all copies of its writeset, and the log is SR.

4. SITE INITIALIZATION ALGORITHM

Suppose we have a DDBS that is running correctly -- i.e. its execution is 1-SR -- and suppose we add a new site to the system. We need to integrate the site into the DDBS in such a way that (1) all data at the site can eventually be read, and (2) the resulting execution remains 1-SR.

In this section we describe an algorithm that accomplishes this task. First, we use the concepts of Sections 2 and 3 to specify the kinds of executions permitted by our algorithm, and to argue that these executions are *correct* (i.e. satisfy requirements (1) and (2)). Then, we demonstrate an algorithm that meets the specification.

Specification and Correctness

The logs that our algorithm will allow satisfy the following properties.

A1. Each transaction writes into all copies of its writeset, except possibly those copies at the new site.

A2. By some time t, every data item at the new site has been written into at least once.

A3. No transaction reads a data item at the new site until that data item has been written at least once.

A4. The log is SR.

A5. Let x_{new} be a copy of X at the new site, and let T_x be the first transaction that writes into x_{new}. By A1, T_x also writes into the other copies of X. Let T'_x be any transaction that writes into any copy of X *after* T_x wrote into that copy. Then T'_x must also write into x_{new}.

Stated a bit loosely, A5 simply means that once any transaction writes into x_{new}, all later transactions also write into x_{new}.

We now argue that if a log satifies A1-A5 then it is correct.

(1) A2 and A3 ensure that all data items at the new site are eventually readable, thereby attaining the first correctness requirement.

(2) It remains to prove that if log L satisfies A1-A5, then L is 1-SR. By A4, L is SR; let L_s be any serial log equivalent to L. Consider any reads-from relationship in L_s, e.g. T_i reads-x_k-from T_j. L_s looks like:

$$L_s = \quad \ldots \quad \longrightarrow w_j[x_k] \longrightarrow \ldots \longrightarrow r_i[x_k] \longrightarrow \ldots$$

and we must show that no transactions between T_j and T_i in L_s writes into any copy of X. We will show this by proving that every transaction that follows T_j and updates any copy of X, also writes into x_k.

Let T_ℓ be any transaction that follows T_j and updates X. If x_k is not a "new" data item, then T_ℓ writes into x_k by A1. Now suppose x_k is "new". By A1 and Proposition 1, T_ℓ follows T_j in L, and so T_ℓ writes into x_k by A5. In either case, since T_ℓ writes into x_k, and since T_i reads-x_k-from T_j (and *not* from T_ℓ), T_ℓ cannot come between T_j and T_i. Q.E.D.

Algorithm

Rules A1-A5 form a blueprint for a simple site initialization algorithm. Let us see how these rules can be attained algorithmically.

A1 and A3 are trivial to implement. A4 is merely concurrency control. Any serializable concurrency control algorithm can be used. The remaining rules can be implemented as follows:

A2. For each logical data item X stored at the new site, run a *copier* transaction that reads a copy of X at an old site and writes that value into the new copy. (I.e. there is one copier transaction per X.) Copiers must be synchronized by the concurrency control algorithms exactly like all other transactions.

A5. For each logical data item X stored at the new site, designate a *guardian copy* x_g of X at some old site. Beginning at some (arbitrary) time t after the new site is added, the site holding x_g alerts all transactions that update x_g to write into the new copy of X also. No transaction updates a data item at the new site unless told to do so by its guardian. (In practice, if the site holding x_g fails, a mechanism is needed to appoint a new guardian copy for X. We do not consider this problem here.)

These five rules constitute our proposed site initialization algorithm.

This description of our algorithm may seem too abstract, mainly because we have not pinned down the concurrency control algorithm. For definiteness, let us see how the algorithm works in conjunction with two-phase locking. We begin by reviewing the basic two-phase locking algorithm.

Associated with each physical data item is a set of *locks*. At any time, the set of locks on a physical data item may contain no locks, one *write lock*, or a set of *read locks*.

Suppose x_i is stored at DM_i. Before processing read(x_i) on behalf of transaction T_j, DM_i must set a read lock on x_i for T_j. Before processing write(x_i) on behalf of T_j, DM_i must set a write lock on x_i for T_j. If DM_i cannot set a lock for an operation, it delays the operation until the lock can be set.* When a transaction terminates, all of its locks are released.

Now let us see how to add a new site to the system. Suppose sites 1,2,...,n-1 are running properly and we wish to add site n. Site n begins the process by sending an "I'm up" message to the DM's at sites 1,2,...,n-1.

Suppose the DM at site i, DM_i, receives an "I'm up" message from site n. From then on, for each guardian copy x_g at DM_i, when DM_i processes a write(x_g), it tells the TM that issued the write to also issue a write(x_n), where x_n is the copy of X at the new site. (An optimization is for DM_i to issue the write(x_n) directly.)

The DM also instructs its local TM to execute copier transactions for each of its guardian copies. The copier for x_g must obtain a read lock on x_g and a write lock on x_n; i.e. it must be synchronized like any other transaction.

DM_n uses the same two-phase locking algorithm as every other DM. However, it refuses to process a read(x_n) until x_n has been updated at least once.

For each logical data item X, a TM issues a write(x_n) on behalf of a transaction that updates X, if *and only if* one of its writes on x_g has been acknowledged by a message telling it to do so. The TM must not update x_n until this point in time.

*Since operations can be delayed while waiting for locks, deadlock is possible. Deadlocks can be resolved by any of the standard techniques in [BG].

5. SITE RECOVERY

Site recovery is the problem of integrating a site into a DDBS when the site recovers from failure. As for the site initialization problem, the goal is to make the recovered site look like all other sites. Once again, the main problem is to bring the database at the recovered site up-to-date relative to the rest of the DDBS.

Site recovery is obviously an important problem, but it has received little attention in the literature. One early paper on DDBS reliability [AD], which mainly studies reliable *concurrency control algorithms*, disposes of site recovery with these few words:

> How the new host is brought up to date depends on the application. It may be done by transferring to that host the journal of all updates since the host went down. It may require transferring the database. [AD, p. 568].

Other related work includes [HS, LS, LSP, MPM, Th, Sk, SS]. Some of these papers [MPM, Th] are like [AD] in that they mainly study reliable concurrency control. (A piece of the algorithm in [MPM] is called Single Node Recovery. But the algorithm only recovers the concurrency control algorithm at the site, not the database.) Other papers study atomic commitment [HS, LS, Sk, SS], site monitoring to keep track of which sites are up [HS], and other distributed decision problems [LSP]. Again, site recovery in our sense is not studied.

One paper that does treat site recovery is [HS]. The solution is based on the concept of Reliable Network (Relnet), a virtual machine that guarantees reliable message delivery despite site failures. The Relnet is intended to be a very general facility suitable for many kinds of distributed systems. Because of this generality, the mechanism is rather complex.

Our approach to site recovery is narrower (and, we hope, simpler) than the Relnet approach. We are *not* trying to attain reliability for arbitrary distributed systems; nor are we trying to solve all DDBS reliability problems. Our goal is simply to integrate the database at a recovered site into a running DDBS.

Evidently, site recovery and site initialization are almost identical problems, and the algorithm of Section 4 can be directly applied here. There is one major caveat: our algorithm says nothing about multiple failures. We believe the algorithm can be generalized to handle multiple failures, but offer no hard evidence in this respect. Despite this caveat, the algorithm of Section 4 solves a big piece of the site recovery problem.

An Optimization

When using the initialization algorithm for site recovery, an important optimization is possible. It is not necessary to fire up copier transactions for all X in the logical database. Suppose we are recovering site f. Only those X that were updated after site f failed need to be recovered. Any X that was not updated while f was down still has the correct value at f when the site recovers. If a spool (or journal) of all writes to site f is maintained while f is down, as in SDD-1 [HS], then when f recovers the following processing can be done. Scan the spool to produce a list of data items that were written while f was down. All data items not on the list can be immediately marked as readable at DM_f. Copiers are executed only for data items on the list.

Notice that we are not proposing that spooled write operations be processed in FIFO order, as in SDD-1. If X was written several times while f was down, only the last value should be sent to f. If earlier values are sent, the algorithm will not work correctly.

6. SITE BACKUP

A *backup database* is a static copy of the database that is consistent but potentially out-of-date. One use of backup databases is to speed up the processing of queries. By reading the static backup, a query does not interfere with updates, and so will not be delayed or restarted for concurrency control reasons. The cost is that it may read out-of-date data. Backup databases are also useful for archiving data.

Creating a backup database is similar to initializing a new site or recovering a failed site -- similar enough that we can use our initialization algorithm to do most of the job.

We begin by pretending that the backup database is a new site being added to the DDBS. We run the initialization algorithm to bring the backup database up-to-date, until all data items in the backup have been written at least once. Now we must freeze the backup, by shutting off writes to it. However, we must shut off writes carefully, so that the backup is frozen in a consistent state. We can do this simply by running a query that (conceptually) reads the entire backup database, and by ensuring that no data item is written once the query has read it. This freezes the backup copy in the state read by the query. Since the query is synchronized by a serializable concurrency control algorithm, the frozen state is consistent.

For example, suppose we use the two-phase locking initialization algorithm of Section 4. When all backup data items have been written at least once, we run a query that sets a read lock on every backup data item. (The query may deadlock while trying to obtain its locks, and so may need to be aborted and restarted.) When all backup data items are locked, we shut off updating by refusing to process any more writes against the backup. The resulting backup database is consistent and can be correctly queried without synchronization.

One problem with this algorithm is that the "shut-off" query may deadlock repeatedly and never finish. This problem can be fixed as follows. Once the query begins, the backup should refuse to grant any write lock requests from transactions that have not already set a lock on some backup copy. These requests are simply blocked, and the transactions delayed, until the query manages to get all of its locks. Then a very counterintuitive event happens -- the lock requests are unblocked, but since the backup is now frozen, the transactions no longer need the locks! (It does not work to unblock the transactions earlier.)

7. CONCLUSION

We have presented an algorithm that can be used to initialize a database at a new site in a DDBS, to recover a database at a formerly failed site, or to create a consistent, static backup database. The algorithm is simple, yet introduces little overhead beyond what is normally needed for concurrency control. We therefore believe it is a practical solution to all three problems.

The methodology that we used to describe our algorithm is also interesting, we believe. First, we defined *correctness*, i.e. what it means for an algorithm to correctly solve the problem; this definition was stated in terms of executions (i.e. logs). Second, we *specified* the kinds of logs that our algorithms would allow, and proved that every allowable log is correct. Third, we described an *abstract algorithm* that meets the specification. Finally, we gave a *concrete implementation* of the abstract algorithm. These four steps,

(i) defining correct logs,

(ii) specifying an allowable subset of the correct logs,

(iii) designing an abstract algorithm that produces allowable logs,

(iv) engineering a concrete implementation of the abstract algorithm,

help structure the problem and the search for solution.

One benefit is that we can engineer new concrete algorithms for specific systems or problems just by redoing step (iv). For example, the concrete implementation of the backup algorithm in Section 6, may have bad performance characteristics. By locking the entire backup database, the "shut-off" query interferes with many updates. This performance problem is not inherent in the abstract algorithm; it is just an artifact of the concrete implementation we gave. A better implementation would use a concurrency control algorithm for the backup in which queries and updates interfere less. Multiversion concurrency control algorithms [BHR, Re, SR] are likely candidates for this role. Engineering a backup algorithm that uses multiversion concurrency control is by no means a trivial task. But structuring the problem as we have done, the designer does not have to start from scratch.

REFERENCES

[AD] Alsberg, P.A., J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," *Proc. 2nd Intl. Conf. Software Eng.*, Oct. 1976.

[BG] Bernstein, P.A., and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, (June 1981).

[BHR] Bayer, R., H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Syst.*, Vol. 5, No. 2 (June 1980), pp. 139-156.

[BSW] Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Softw. Eng.*, Vol. SE-5, No. 3 (May 1979).

[EGLT] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System." *Commun. ACM* Vol. 19, No. 11, (Nov. 1976), pp. 624-633.

[HS] Hammer, M.M., and D.W. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Trans. Database Syst.* Vol. 5, No. 4 (Dec. 1980), 431-466.

[LS] Lampson, B., and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, CA, 1976.

[LSP] Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," Tech. Rep., SRI International, March 1980.

[MPM] Menasce, D.A., G.J. Popek, and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," *ACM Trans. Database Syst.* Vol. 5, No. 2, (June 1980), pp. 103-138.

[Pa] Papadimitriou, C.H., "Serializability of Concurrent Updates," *J. ACM* Vol. 26, No. 4 (Oct. 1979), pp. 631-653.

[Re] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System", Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, MA, Sept. 1978.

[Sk] Skeen, Dale, "Nonblocking Commit Protocols," *Proc. 1981 ACM-SIGMOD Conf. on Management of Data*, ACM, N.Y., pp. 133-147.

[SLR] Stearns, R.E., P.M. Lewis, II, and D.J. Rosenkrantz, "Concurrency Controls for Database Systems," in *Proc. 17th Symp. Foundations Computer Science (IEEE)*, 1976, pp. 19-32.

[SR] Stearns, R.E., and D.J. Rosenkrantz, "Distributed Database Concurrency Controls Using Before-Values," in *Proc. 1981 ACM-SIGMOD Conf. on Management of Data*, ACM, N.Y., pp. 74-83.

[SS] Skeen, D., and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System", *Proc. 5th Berkeley Conference on Distributed Data Management and Computer Networks*, 1981, pp. 129-142.

[Th] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", *ACM Trans. on Database Syst.*, Vol. 4, No. 2 (June 1979), pp. 180-209.

VIEW DEFINITION AND GENERALIZATION FOR DATABASE INTEGRATION IN MULTIBASE: A SYSTEM FOR HETEROGENEOUS DISTRIBUTED DATABASES*

Umeshwar Dayal
Computer Corporation of America
Cambridge, Massachusetts

and

Hai-Yann Hwang
The University of Texas at Austin
Department of Computer Sciences

ABSTRACT

Access to a heterogeneous, distributed database system can be simplified by providing users with a logically integrated interface or global view. There are two aspects to database integration. Firstly, the local schemas may model objects and relationships differently and, secondly, the databases may contain mutually inconsistent data. This paper identifies several kinds of structural and data inconsistencies that might exist. It describes a versatile view definition facility for the Functional Data Model and illustrates the use of this facility for resolving inconsistencies. In particular, the concept of generalization is extended to this model, and its importance to database integration is emphasized. The query modification algorithm for the relational model is extended to the semantically richer functional data model with generalization.

---

*This research was partially supported by the Defense Advanced Research Projects Agency of the Department of Defense and the Naval Electronic Systems Command under Contract No. N00039-80-C-0402. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Naval Electronic Systems Command or the U.S. Government.

1. INTRODUCTION

Retrieving information from a collection of independently designed databases is a formidable task. The component databases typically have different schemas, expressed in different data models, and are implemented on different database management systems, each with its own retrieval language. Formulating and implementing queries that require data from more than one database poses many problems for the user. These problems include resolving discrepancies between the databases, such as differences in representation and naming conflicts; resolving inconsistencies between copies of the same information stored in different databases; and transforming a query from the user's language into a set of queries expressed in the different retrieval languages supported at the different sites.

Multibase is a system that relieves the user of many of these problems [SMITH et al.81]. It presents users with a logically integrated global view of the data stored in the local databases, without requiring that the databases be physically integrated. The schema architecture of Multibase is shown in Figure 1.1. Translating the local host schemas (LHSs) into local schemas (LSs) in a common data model shields the users from differences in the data models and languages. In addition, an integration database, described by the integration schema (IS), may be needed to record information required for integration. For example, if the databases record the same data values in different scales, the IS stores the mapping between the scales. The global schema (GS) is defined as a view of the LSs and IS. The definition of the GS incorporates directives for resolving discrepancies and inconsistencies between the local databases. Therefore, the GS provides users with the illusion of a homogeneous and integrated database.

Users express their queries against the GS in a common data language. However, since the global database is not physically materialized, these queries must be modified into equivalent queries against the local schemas. Since these queries are still in the common data language, they finally must be translated into programs that will execute on the local hosts.

In this paper, we are not concerned with the problem of mapping the LHSs into LSs in a common model, nor with the corresponding translation of queries. Instead, we focus on the issue of database integration: how to define the GS to resolve discrepancies between the local databases, and how to modify a global query into a query against the collection of LSs. Specifically, we show that the concept of generalization [SS77, MBW80, LG78], when coupled with extensive view definition facilities, (exceeding even those provided by typical relational systems

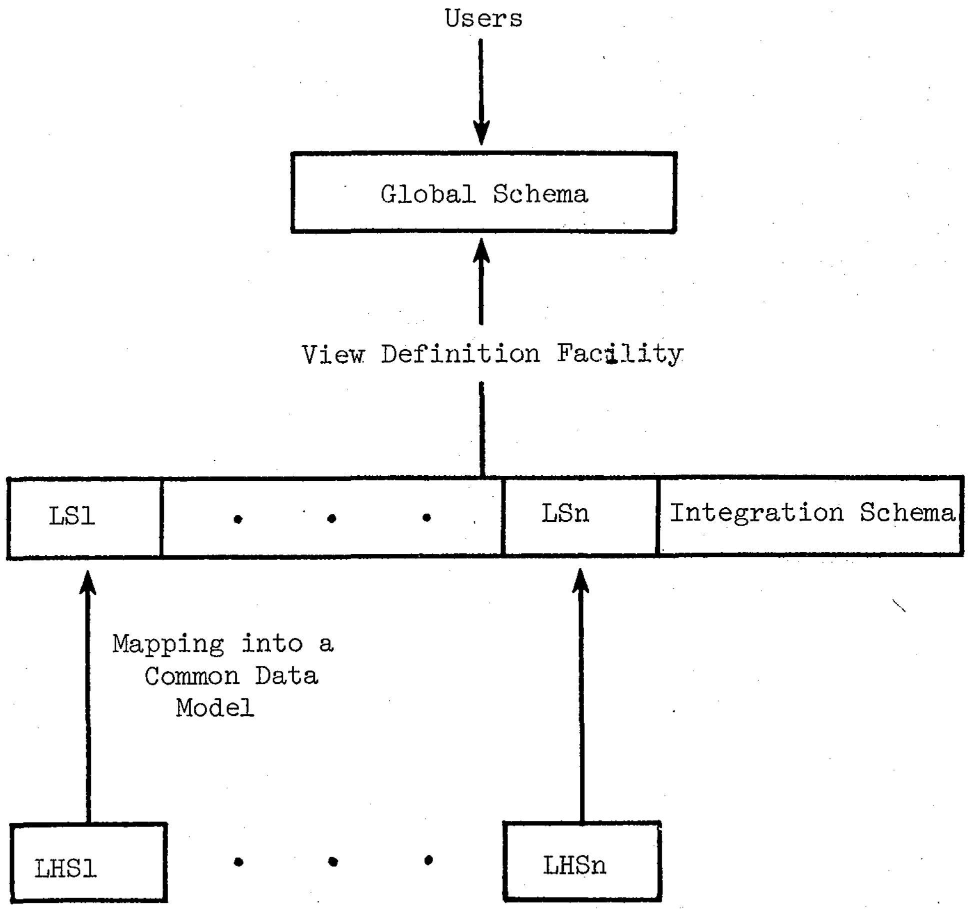


Figure 1.1 Schema Integration Architecture

[HSW75, CGT75]), is a powerful tool for database integration. This idea was explored in [KG81]. There it was assumed that the local databases were disjoint (i.e., contained no entities in common), and the use of generalization for schema integration was demonstrated. For instance, consider two databases, one of U.S. Ships and the other of Soviet Ships. The global schema can generalize these to the entity type Ship (which has all the attributes common to the two specialized entity types). However, the most interesting (and difficult) problems occur when the local databases do overlap. It is that class of problem which we attack here. For instance, suppose two databases, DB1 and DB2, contain data on employees and their salaries. For employees in DB1-DB2 (or DB2-DB1), it seems clear that the salary value in the global view must be derived from that in DB1 (or DB2, respectively). However, for an employee in both databases, what should be done if there is a conflict? The Database Administrator (DBA) could decide that DB1 is more credible and, hence, derive the salary in the view from DB1. Another possibility is to include both salaries in the view (since they might be salaries for two different jobs). Other solutions might be appropriate for different applications. The view definition mechanism must be flexible enough to accommodate these alternatives.

The problem of defining a global schema as a view (called a "superview" by the authors) is also addressed in [MB81]. Some interesting operators (including generalization) for view definition are introduced, but these are not powerful or versatile enough for resolving inconsistencies of the kind described here.

Multibase uses the Functional Data Model (FDM) and DAPLEX [SHIP81] as its common model and mapping language for defining the LSs, IS and GS, and for expressing global queries. In Section 2, we embellish the basic Functional Data Model with ISA relationships (generalization). For instance, we can say USShip ISA Ship, implying that every entity of type USShip is also of type Ship. In addition to the conventional ISA relationship for entities, we introduce an ISA relationship for functions. This enhances the semantic modelling capabilities of the FDM, and provides greater flexibility in defining global schemas. Our goal in this paper is to discuss the concepts and view definition capabilities required for database integration. Toward this goal, we introduce a simple nonprocedural query language that embodies these concepts and capabilities. This language is described in Section 3. How to "smoothly" incorporate these ideas into an existing full-blown user language such as DAPLEX is currently under investigation. In Section 4, we catalogue various kinds of discrepancies that might exist between databases and we illustrate the use of our view definition mechanism in resolving them. Section 5 describes our query modification algorithm. View definition and query modification in the relational model have been studied in [STON75]. Our contribution is in extending those ideas to the functional data model with generalization, which is syntactically and semantically richer than the relational model. For views defined by generalization, the global query is modified into a collection of subqueries against the local schemas. However, the straightforward approach of syntactically substituting the view definition into the query could lead to a proliferation of irrelevant subqueries. The algorithm we describe generates only the essential subqueries. In the Appendix, we enhance the power of the language by giving users control over duplicate elimination. We show that this enhancement has little impact on query modification.

2. THE FUNCTIONAL DATA MODEL WITH GENERALIZATION

In this Section we first review the basic model. Then we extend it to include generalization and describe a query language for use with it.

2.1 The Basic Model

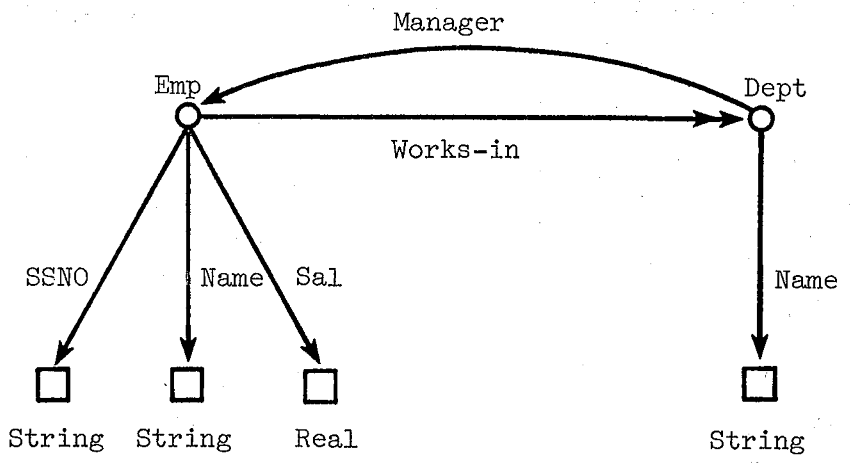
The Functional Data Model uses two constructs: the _entity_ and the _function_. Entities are intended to represent real-world objects, and functions to represent properties of these objects or relationships among objects.

A _database schema_ in this model is represented by a labelled directed multigraph, G, whose nodes are labelled with _entity types_ and edges with _function names_. Functions may be _singlevalued_ or _multivalued_.

In every state of G, there is a finite set E of entities. Each element of E is of a specified entity type. The _extension_ in E of entity type X is the set of all entities in E of type X. (We sometimes call this the _entity set_ X). For each singlevalued function name f: X -> Y, there is a function f from entity set X into entity set Y; for each multivalued function name f: X ->-> Y, there is a function f from entity set X into the power set of entity set Y. A state of G may also be represented by a directed multigraph in the obvious way. (We distinguish between user defined entity types and constants, i.e., system defined entity types such as integer, real, string, Boolean. The extensions of the constants are fixed and cannot be altered by users). An example FDM schema is given in Figure 2.1.

2.2 Augmenting the Basic Model With Generalization

Generalization is an abstraction that groups classes of objects with common properties into a generic class of objects [SS77]. For example, if a STUDENT entity is described by properties SSNO, NAME, ADDRESS, MAJOR, GPA, and an EMPLOYEE entity by properties SSNO, NAME, ADDRESS, SALARY, JOBHISTORY, then a generic type PERSON can be formed with their common attributes SSNO, NAME, ADDRESS. Applications that are not concerned with the special properties of students or employees need not distinguish between them, and can, instead, treat them as persons. Thus, incorporating the concept of generalization in the data model



Note: For convenience, we denote the constant types by □ nodes

Figure 2.1 A Schema in the FDM

provides greater flexibility in semantic modelling. This idea has been suggested several times in the literature (e.g., [SS77, HM78, ROUSS79, MBW80, MB81, CODD79, HK81]).

The concept of generalization assumes even greater importance for database integration in Multibase. The local schemas, having been developed independently, may contain entities at different levels of generalization. For example, assume that one schema, LS1, models people as PERSON entities, whereas another, LS2, models them as STUDENT and EMPLOYEE entities. The integrated global schema may be satisfactorily defined neither at the level of LS2 (if it has no information available to distinguish between persons in LS1 who are students and those who are employees), nor at the level of LS1 (for then the distinction between students and employees in LS2 would be lost to global users). A reasonable solution is to include all three entity types, viz., STUDENT, EMPLOYEE, and PERSON, in the GS with the relationships STUDENT ISA PERSON, EMPLOYEE ISA PERSON explicitly defined between them.

We propose another (less obvious) use of generalization for database integration. Suppose the "same" entity type occurs in two different schemas but with different properties. For example, suppose EMPLOYEE entities in LS1 have properties SSNO, NAME, SALARY, AGE, whereas EMPLOYEE entities in LS2 have properties SSNO, NAME, SALARY, ADDRESS. Conventional data modelling would suggest that the EMPLOYEE entity set in GS be defined as the "outer join" of the entity sets in LS1 and LS2. (The "outer join" operator sets SALARY values to NULL for employees not in the first database, and ADDRESS values to NULL for employees not in the second database [CODD79].) Instead, using generalization to define the GS (as in Figure 2.2), we achieve the same effect without introducing artificial NULL values. As we shall see in Section 5, imposing this additional structure on the view assists in query modification.

We extend the concept of generalization to generalization over functions. Reconsider the example of Figure 2.2. Suppose that two additional functions HOMEPHONE and WORKPHONE were defined for EMPLOYEE entities in LS1, and a multivalued function PHONES for EMPLOYEE entities in LS2. Then a convenient abstraction would be to define the generic multivalued function PHONES for the generic entity type EMPLOYEE in GS. Thus, if employee e has workphone p1 and homephone p2 in the first database, and the set of phones {p2, p3} in the second database, then p1, p2, p3 are all included in PHONES(e) in the global view.

Proceeding formally, we extend the basic model of Section 2.1 as follows. A schema now is a triple S= (G, ISAe, ISAf) where G is a labelled directed multigraph (as before), ISAe is a binary relation on the nodes of G, and ISAf a binary relation on the edges of G. A state of S is as before, except that now each entity can be of more than one type. ISAe and ISAf obey the axioms of Figure 2.3.

If X1 ISAe X2, then we call X1 a _subtype_ of X2, and X2 a _supertype_ of X1. The extensions of subtypes may overlap. The Extensionality Constraint implies the following Inheritance Rule: a subtype inherits all the functions of its supertype.

If F1 ISAf F2, then we call F1 a _subfunction_ of F2, and F2 a _superfunction_ of F1.

Figure 2.4 gives an example of integrating two local schemas using the ISA relationships. In Section 3, we show how to define this global schema as a view of the local schemas.

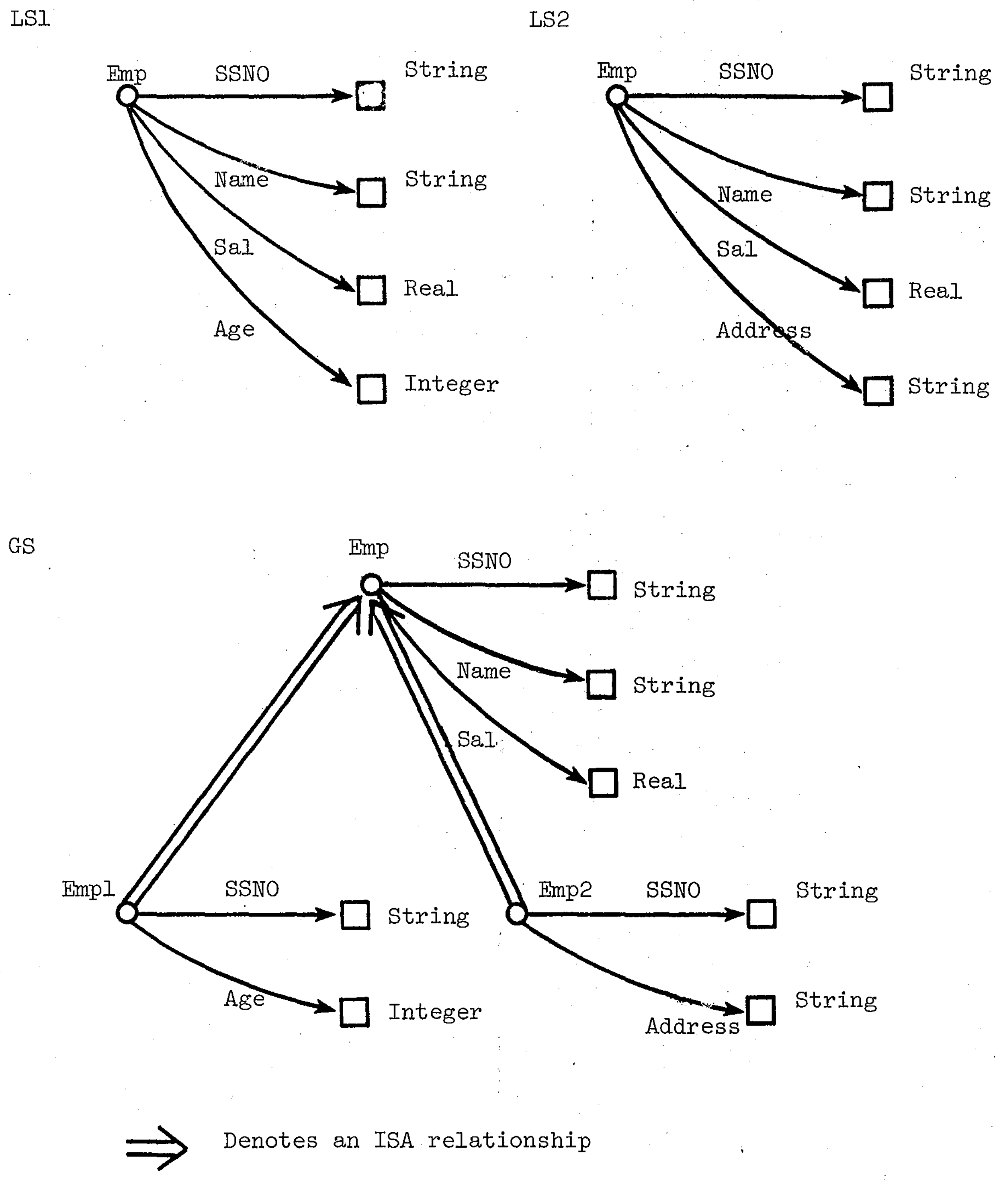


Figure 2.2 Integration by Generalization of Entity Types

a) Axioms for ISAe

1. Intrinsic Axiom
   ISAe is a partial order, i.e., it is reflexive, antisymmetric, and transitive.

2. Extensionality Constraint
   If x is an entity of type X1 and X1 ISAe X2, then x is also of type X2.

3. Range Restriction
   ```
   If X1 ISAe X2 and F: X1 -> Y1
                     F: X2 -> Y2,
   then Y1 ISAe Y2.
   ```

4. Cotype Constraint
   If x is an entity of types X1 and X2, then there is a type X3 such that X1 ISAe X3 and X2 ISAe X3.

b) Axioms for ISAf

1. Intrinsic Axiom
   ISAf is a partial order.

2. Structural Constraint
   ```
   If F1 ISAf F2, and F1: X1 -> Y1
                      F2: X2 -> Y2,
   then X1 ISAe X2 and Y1 ISAe Y2.
   ```

3. Extensionality Constraint
   If F2 ISAf F2, F1 is a function from X1 to Y1, and F2 is a function from X2 to Y2, then, for every entity x of type X1,
   $F1(x) = F2(x)$ if both F1 and F2 are singlevalued
   $F1(x) \subseteq F2(x)$ if both F1 and F2 are multivalued
   $F1(x) \in F2(x)$ if F1 is singlevalued and F2 is multivalued

Figure 2.3 Axioms for ISAe and ISAf

2.3 A Query Language for the Model

The query language we use to illustrate our ideas is a variant of NQUEL, a nonprocedural language for the general network model introduced in [DAYAL79, DB82]. (NQUEL is based on the relational query language QUEL [HSW75]. We use this language

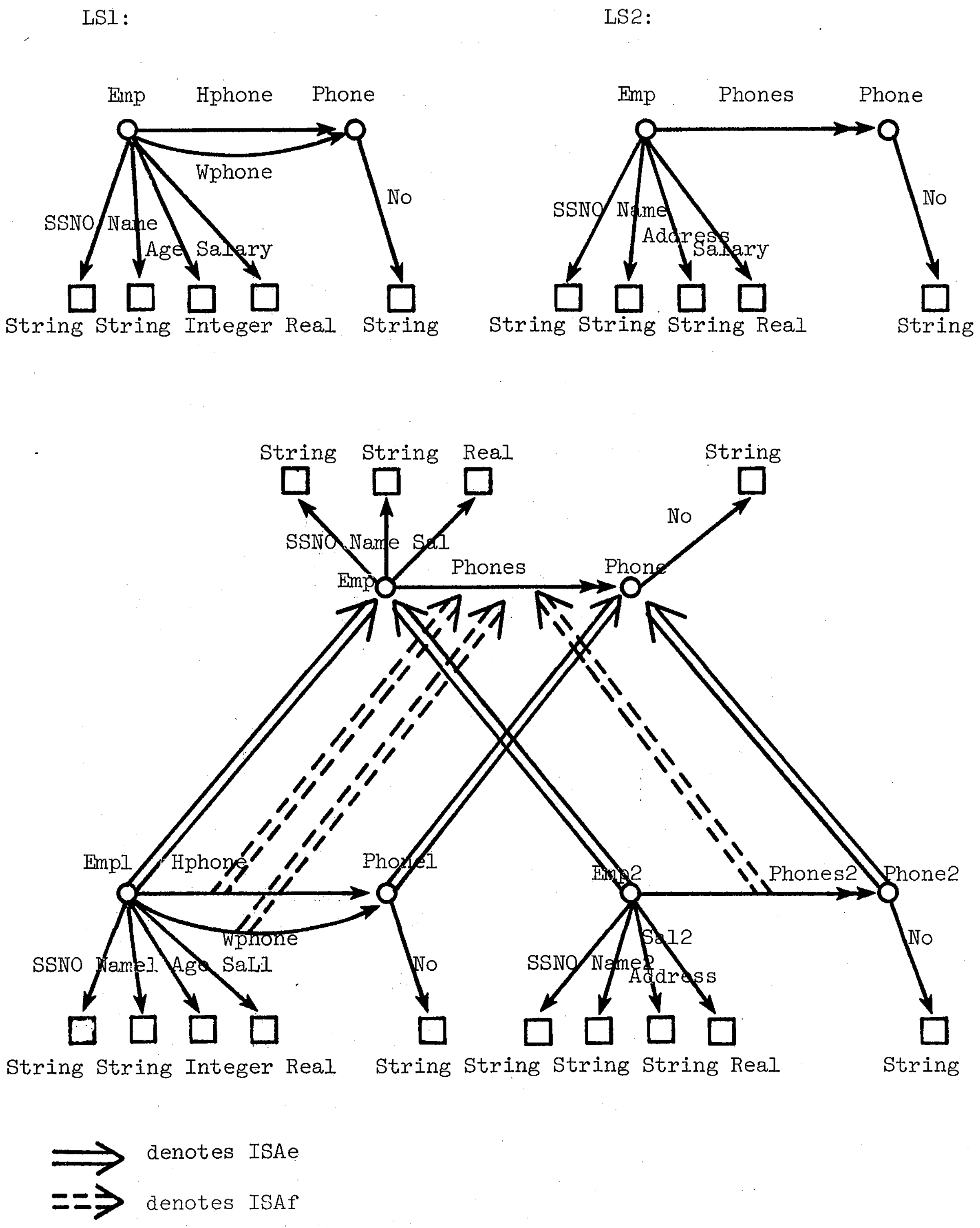


Figure 2.4 Integration by Generalization of Entity Types and Functions

merely as an example. The techniques can be applied to other algebraic or calculus-based languages such as DAPLEX [SHIP81].)

Queries are formulated using entity variables, which are declared in range statements: RANGE OF <entity-var> IS <entity-type>.

Queries are retrieval statements of the form
RETRIEVE INTO <result-entity-type> (<target_list>)
   WHERE <qualification>
<target-list> is a list of assignments

<singlevalued function name> := <term> or
<multivalued function name> := <set>;
<term> is an entity variable, a constant, a singlevalued function applied to an entity variable, or a composition of singlevalued functions applied to an entity variable; <set> is a multivalued function applied to an entity variable, or an expression of the form {<entity variable> IN <entity type> WHERE <qualification>}; <qualification> is a Boolean combination of atomic formulas of the form <term> <op> <term> or <entity variable> ISIN <multivalued function applied to an entity variable>; <op> is one of the comparison operators =, <, etc.

We do not preclude entity variables ranging over constant (i.e., system defined) entity types, such as integer, real, string, Boolean. However, we do require that any such variable x be bound to a finite range by an atomic formula x=f(y) or x ISIN f(y) in every disjunct when the qualification is cast into Disjunctive Normal Form.

The interpretation of a query RETRIEVE INTO E (f1:=c1,...,fk:=ck) WHERE qualification is as follows: construct the Cartesian product of the ranges of all free entity variables appearing in the query; eliminate those tuples of entities that do not satisfy the qualification; for each of the remaining tuples of entities, perform the target-list computations c1,..., ck; for each unique k-tuple <t1, ..., tk> computed, create a new entity of type E with functions fi := ti, $1 \leq i \leq k$. (How these entities are printed or displayed to a user is irrelevant to our discussion.) Examples of queries are given in Figure 2.5. Observe that the query in (i) retrieves an "unnormalized" relation.

Since Multibase is a retrieve-only system, we ignore updates, although they could be adapted from those in NQUEL [DAYAL79, DB82].

Schema of Figure 2.1

i. Retrieve the Soc. Security Nos. and Names of all Employees earning over 50K together with the Names of the Departments they work in.

```
RANGE OF e IS EMP
RANGE OF d IS DEPT
RETRIEVE INTO RESULT (SSNo := SSNo(e), Name := Name(e)
                      DeptNames := {n IN STRING WHERE n = Name(d)
                                    AND d ISIN Works-in(e)})
              WHERE Sal(e)>50000.
```

ii. Retrieve the Soc. Security No. and Name of every employee and the name of each of his/her managers.

```
RANGE OF e,m IS EMP
RANGE OF d IS DEPT
RETRIEVE INTO WORKSFOR
       (SSNo := SSNo(e), EmpName := Name(e), MgrName := Name(m))
             WHERE m = Manager(d) AND d ISIN Works-in(e).
```

Figure 2.5 Examples of Queries

3. VIEW DEFINITION IN THE MODEL

In this Section, we describe constructs for defining the global schema as a view of the local schemas. Facilities are provided for including entities from the local databases into the view, for defining new (virtual) entities, and for defining ISA relationships on entity types and functions in the view.

3.1 Inclusion

INCLUDE <entity variable> AS <entity type> (<function-list>) WHERE <qualification>. This statement causes those entities in the range of the entity variable that satisfy the qualification to be included in the view. The AS clause permits renaming the type of these entities. Only the functions in the function list are visible in the view. These functions may be renamed thus: <function-name> AS <new-function-name>.

3.2 Defining Virtual Entities and Functions

i. DEFINE ENTITY TYPE <entity type> (<target_list>) WHERE <qualification>. The semantics of this statement are similar to the semantics of a query.

ii. DEFINE FUNCTION <function name>
FOR <entity variable> IN <entity type>
<assignment>

This statement is useful for defining new functions for previously defined entity types in the view. Depending upon whether the right hand side of the assignment is a term or a set, the function being defined is singlevalued or multivalued.

3.3 Defining Supertypes and Superfunctions in a View

Defining a Supertype X in a view involves the following steps: i) Specify X's subtypes X1,...,Xk. This defines the extension of X to be X:= $\bigcup_{i=1}^{k}$ (extension of Xi). ii) Specify a list IDX of singlevalued functions that identify entities of type X. (IDX must include the identifiers of all Xi, $1 \leq i \leq k$.) This partitions $\bigcup_{i=1}^{k}$ (extension of Xi) based on IDX values, and merges entities in each block of the partition. Thus, after the merge, the following constraint holds: ($\forall$x, x'$\in$X) (IDX(x) = IDX(x') => x = x'). Consequently, entities of subtype Xi and Xj having the same IDX values are treated as being the same entity. Observe that this permits us to define overlapping subtypes. iii) Specify how the functions on supertype X are derived from those on X1,...,Xk. In addition to the constructs described in Section 3.2, two additional features are necessary.

(a) A function for X may be declared to be a superfunction of some functions f1,...,fk on X's subtypes. For the example in Figure 2.4, the relationships Hphone ISAf phones, Wphone ISAf phones, phones2 ISAf phones imply the following constraint: ($\forall$ a $\in$ PHONE, $\forall$e $\in$ EMP) [a $\in$ phones(e) iff [(e $\in$ EMP1 AND a $\in$ PHONE1 AND (a = Hphone(e) OR a = Wphone(e))) OR (e $\in$ EMP2 AND a $\in$ PHONE2 AND a $\in$ phones2(e))]].

(b) The value of a function on an entity of supertype X may depend on the subtype(s) to which the entity belongs. For example, in Figure 2.4, the Salary function on EMP may be defined by

$$\mathrm{Sal(e)} = \begin{cases} \mathrm{Sal1(e)}, \text{ if } e \in \mathrm{EMP1} - \mathrm{EMP2} \\ \mathrm{Sal2(e)}, \text{ if } e \in \mathrm{EMP2} - \mathrm{EMP1} \\ \mathrm{Sal1(e)} + \mathrm{Sal2(e)}, \text{ otherwise} \end{cases}$$

To define such <u>conditional functions</u>, a conditional assignment is required.

Supertype definitions are summarized below.

```
DEFINE SUPERTYPE <entity-type-0> BY
      <entity-type-1> ISAe <entity-type-0>,...,
      <entity-type-k> ISAe <entity-type-0
      ID: <list of singlevalued function names>
      FOR <entity variable> IN <entity-type_0>
      [<list of superfunction declarations>]
      [<conditional assignment list>]
```

A superfunction declaration is of the form

```
  DEFINE SUPERFUNCTION <function-name-0> BY
    <function-name 1> ISAf <function-name-0>,...,
    <function-name-p> ISAf <function-name0>
```

A conditional assignment is of the form

```
   <function name > : = CASE
        <condition1> => <term1> WHERE <qualification>|
                                      <set 1>
        <conditionN> => <termN> WHERE <qualification>|
                                      <set N>
```

A condition is: <entity variable> ISIN <range>, where <range> is a well formed set expression over entity types 1..., k. The N ranges in the CASE statement must partition the extension of <entity_type_0>.

Figure 3.1 illustrates the use of these statements for defining the view of Figure 2.4. (In our examples, we shall assume that the terms of the language are extended to include arithmetic computations. Permitting embedded calls to arbitrary procedures written by the DBA would provide even more flexibility for database integration. Furthermore, it might be convenient to extend the conditions in a conditional assignment statement to include arbitrary formulas.)

3.4 Defining Subtypes in a View

For some applications it might be required to define specializations of entity types and functions in the view.

```
DEFINE SUBTYPE <entity_type_0> BY
   <entity_type_0> ISA <entity_type_1>,...,
   <entity_type_0> ISA <entity_type_k>
```

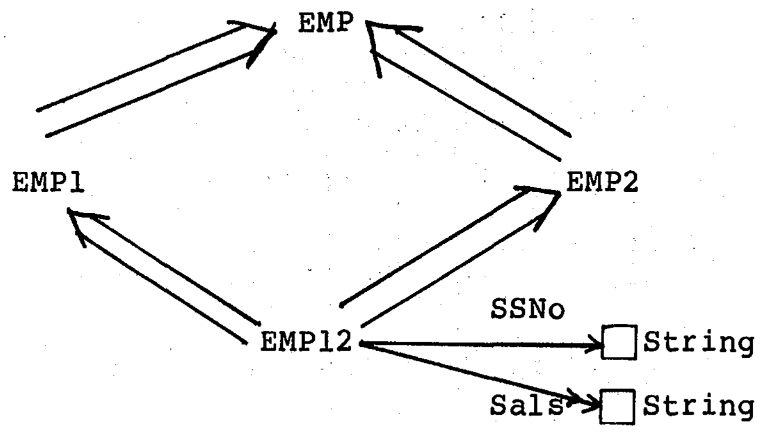
```
RANGE OF e1 IS LS1.EMP, RANGE OF e2 IS LS2.EMP
RANGE OF p1 IS LS1.PHONE, RANGE OF p2 IS LS2.PHONE

INCLUDE e1 AS EMP1 (SSNo, Name AS Name1, Age, Sal AS Sal1,
                          Hphone, Wphone)
INCLUDE p1 AS PHONE1 (No)
INCLUDE e2 AS EMP2 (SSNo, Name AS Name2, Address, Sal AS Sal2,
                          Phones AS Phones2
INCLUDE p2 AS PHONE2 (NO)

DEFINE SUPERTYPE PHONE BY
       PHONE1 ISAe PHONE, PHONE2 ISAe PHONE
       ID : No
DEFINE SUPERTYPE EMP BY
       EMP1 ISAe EMP, EMP2 ISAe EMP
       ID : SSNo
   FOR e IN EMP
      DEFINE SUPERFUNCTION Phones BY
             Hphone ISAf Phones, Wphones ISAf Phones,
             Phones2 ISAf Phones
      Name := CASE
             e ISIN EMP1 => Name1(e)
             e ISIN EMP2 - EMP1 => Name2(e)
      Sal := CASE
             e ISIN EMP1 - EMP2 => Sal1(e)
             e ISIN EMP2 - EMP1 => Sal2(e)
             e ISIN EMP1 ∩ EMP2 => Sal1(e) + Sal2(e)
```

Figure 3.1 Definition of the Global Schema of Figure 2.4

```
FOR <entity-variable> IN <entity_type_0>
    <target_list>
```

The extension of a subtype is the intersection of the extensions of all its supertypes. Subtype definition is illustrated in Figure 3.2.

Suppose that we have already defined the view of Figs. 2.4 and 3.1. We now want to define a subtype EMP12 of EMP1, EMP2.



```
DEFINE SUBTYPE EMP12 BY
       EMP12 ISAe EMP1, EMP12 ISAe EMP2
       FOR e IN EMP12
          Sals := {s IN STRING WHERE s = Sal1(e) OR
                                     s = Sal2(e)}.
```

Figure 3.2 Example of Subtype Definition

4. DATABASE INTEGRATION USING VIEWS AND GENERALIZATION

If the LSs were identical (i.e., contained the same entity types and functions) and there were no conflicts among data values, then database integration would be a trivial task. We could define the GS to be identical to each LS, and the extension of each global entity type or function to be the union of the corresponding extensions in the local databases. However, in general, there are two sources of difficulty: the LSs might not be identical (schema differences), and data values stored in different databases that represent the same information might conflict (data differences). It might seem that a simple solution to these problems is to define the GS (and its extension) to be the disjoint union of the LSs (and their extensions). This solution is unacceptable, however, for it places the onus of integration entirely on the user, who must then understand the semantics of all the local databases to formulate queries. In this

Section, we show via examples how the DBA can design global user views to resolve various kinds of schema and data differences.

The constructs described in Section 3 may be used to define a wide variety of views that meet a wide variety of application requirements. The approach we adopt in our examples is to preserve all available information from the local databases. For specific users, the DBA may wish to suppress some of the information. Such views are easily defined, once we have shown how to define the more exacting views that preserve all information. The approach we suggest consists of two steps. First, resolve schema differences so that the entity types of interest to the user's application in all the LSs "look similar". Then, we combine these entity types using generalization; data differences are resolved by appropriately defining the functions on the supertype.

4.1 Schema Integration

Schema integration includes the resolution of naming conflicts, scale differences, structural differences, and differences in abstraction.

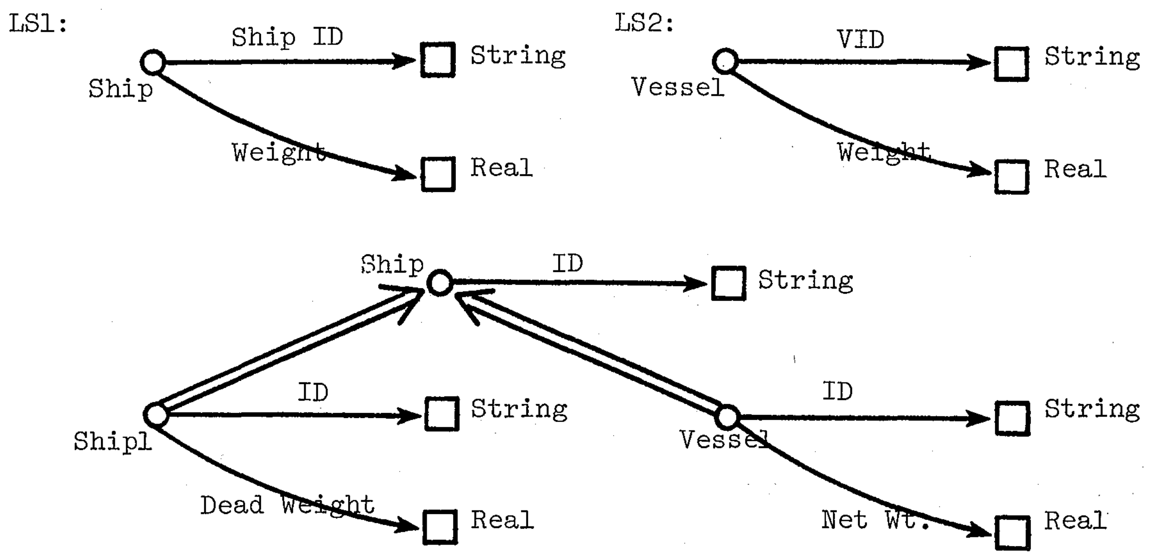
4.1.1 Naming Conflicts

Naming conflicts are easily handled by renaming. If entity types (or functions) representing the same real-world object (or relationship) have different names (synonyms) in different LSs, then give them the same name. This is illustrated by the Ship and Vessel entity types in Figure 4.1a in the GS. If the entity types (or functions) representing different objects (or relationships) in different LSs have the same name (homonyms), then give them different names in the GS. This is illustrated by the Dead-Weight and Net-Weight functions in Figure 4.1a. (Observe how we use generalization to merge the entities in the two databases.)

4.1.2 Scale Differences

The same function values might be stored using different scales in different databases. For example, in Figure 4.1b, Height is measured in inches in one database and in cms in the other; similarly, Weight is measured in lbs in one database and is encoded on a scale of Light/Medium/Heavy in the other. These differences are resolved by using unifying scales with the Height and Weight functions on the supertype. For Height, since there

a. Naming Conflicts



Definition: Include Ship and Vessel in the view, with appropriately renamed functions; then define Ship as the supertype.

b. Scale Differences

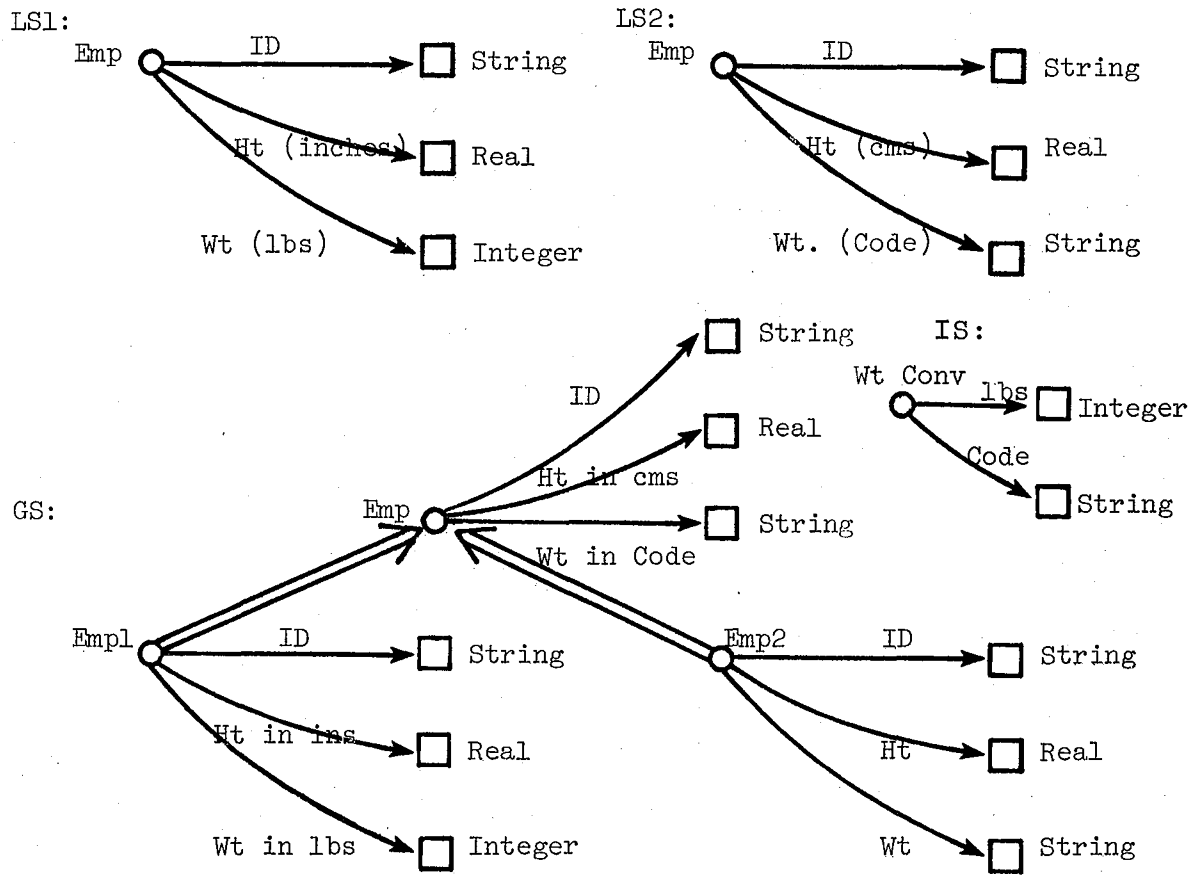
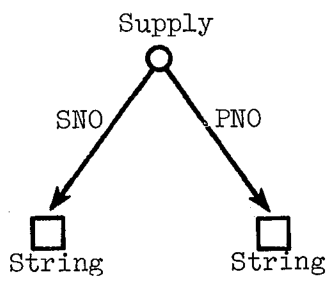


Figure 4.1 Examples of Integration

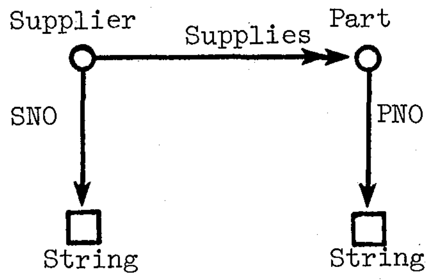
```
Definition: Include LS1.Emp as Emp1, LS2.Emp as Emp2 in GS.
RANGE OF c IS Wtconv
DEFINE SUPERTYPE Emp BY Emp1 ISAe Emp, Emp2 ISAe Emp
        ID : ID
    FOR e IN Emp
       Htincms := CASE
                   e ISIN Emp2 => Ht(e)
                   e ISIN Emp1-Emp2 => 2.54*Htinins(e)
       Wtincode := CASE
                    e ISIN Emp2 => Wt(e)
                    e ISIN Emp1-Emp2 => code(c) WHERE
                                         lbs(c) = Wtinlbs(e)
```

c. Structural differences

LS1:



LS2:



GS:

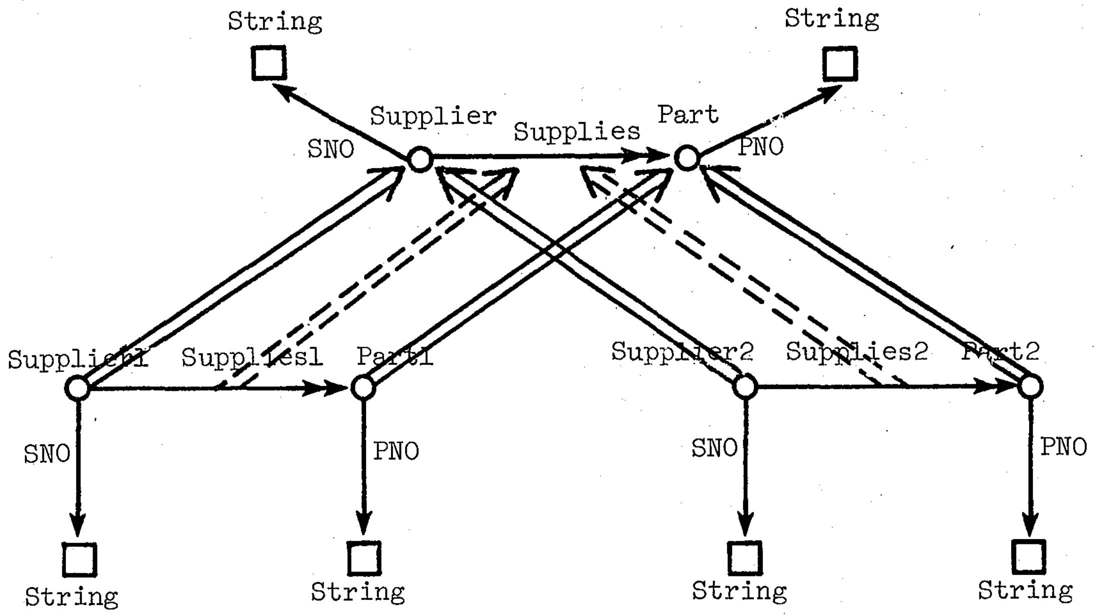
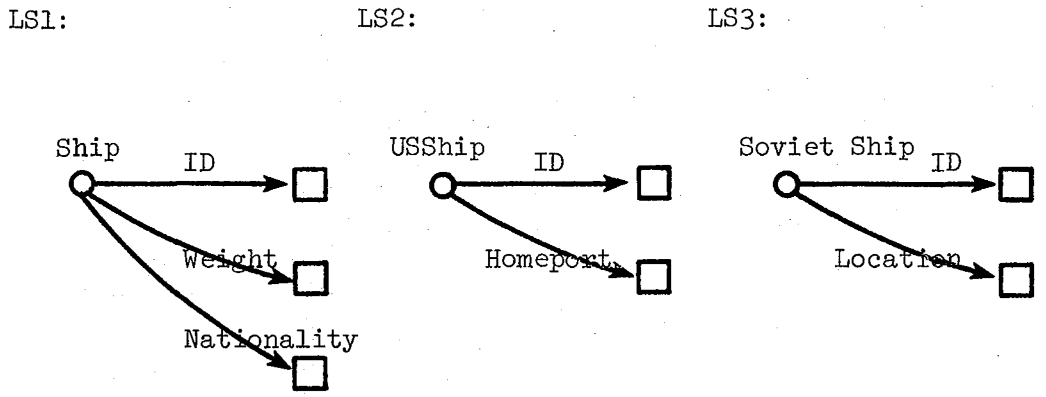


Figure 4.1 Examples of Integration (continued)

```
Definition: RANGE OF y, y' is Supply
            DEFINE ENTITY TYPE Part1 (PNo := PNo(y))
            DEFINE ENTITY TYPE Supplier1 (SNo := SNo(y)),
                  Supplier1 := {p in Part1 WHERE SNo(y') = SNo(y)
                                           AND   PNo(y') = PNo(p)})
Include     LS2.Supplier as Supplier2 and LS2.Part as Part2.
Define      Supplier as the Supertype of Supplier1, Supplier2
            Part as the Supertype of Part1, Part2
            Supplies as the Superfunction of Supplies1, Supplies2.
```

d. Entity types at different levels of generalization



GS1 (least integrated):

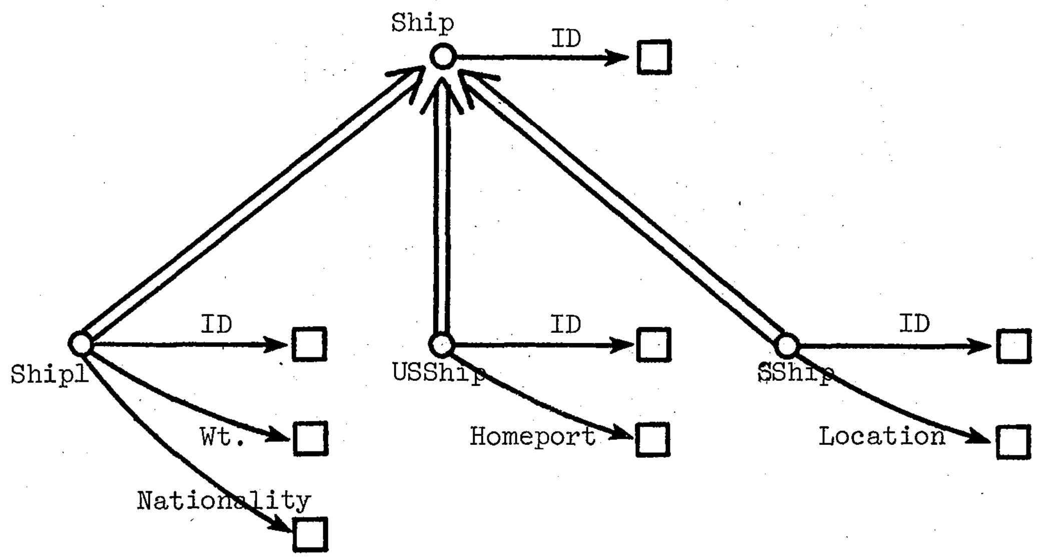
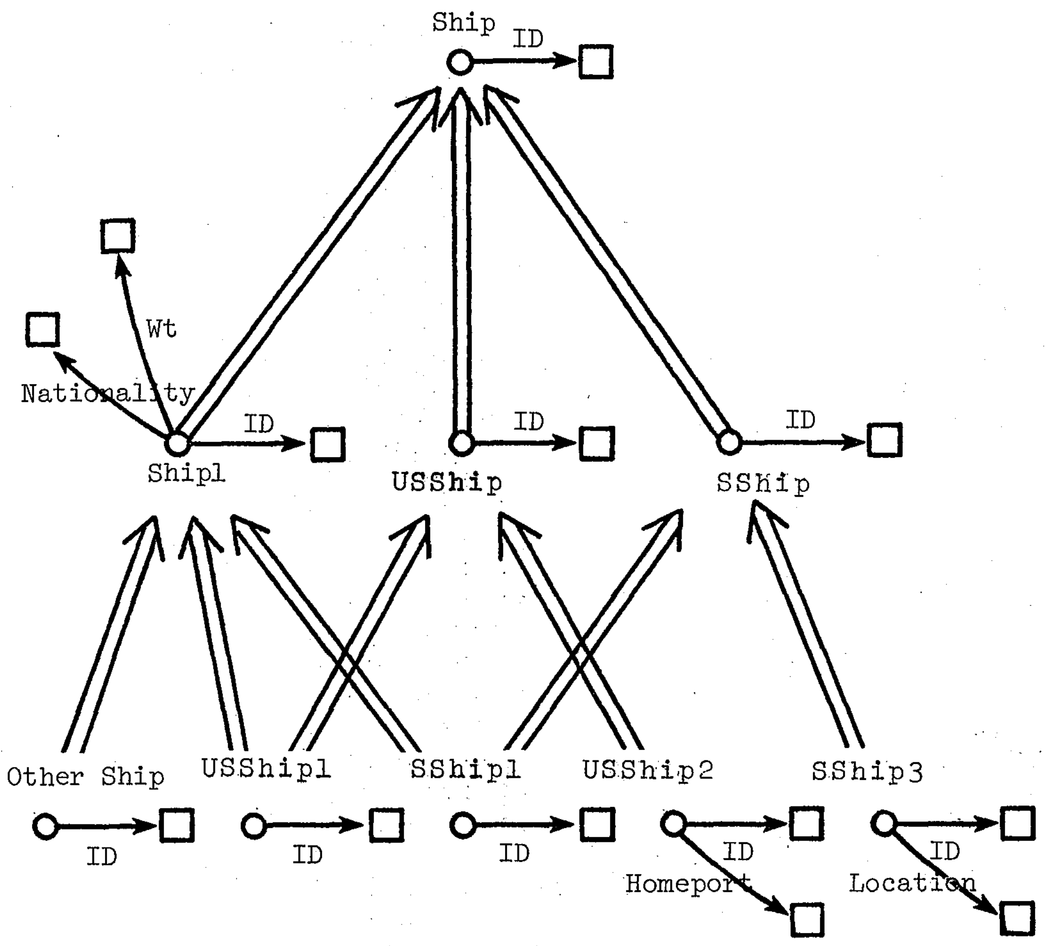


Figure 4.1 Examples of Integration (continued)

GS2 (most integrated):

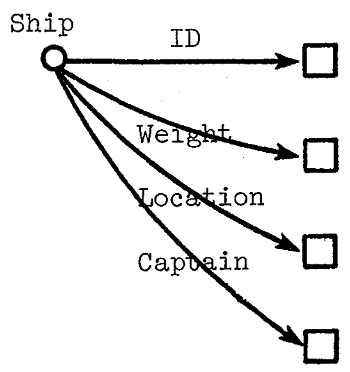


Definition:	Include LS1.Ship with appropriate conditions on Nationality as OtherShip, USShip1, SShip1.
Include	LS1.Ship as Ship1. (The ISAe links from OtherShip, USShip1, and SShip1 to Ship1 are not needed in the definition of these types, but are added merely as integrity constraints.)
Include	LS2.USShip as USShip2, and LS3.SovietShip as SShip3.
Define	USShip to be the supertype of USShip1 and USShip2; SShip to be the supertype of SShip1 and SShip3. Finally, define Ship to be the supertype of Ship1, USShip and SShip.

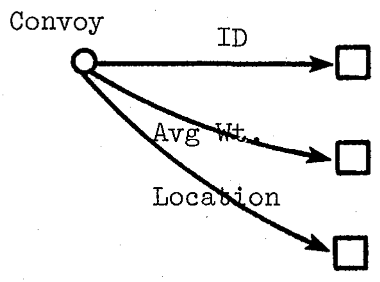
Figure 4.1 Examples of Integration (continued)

e. Set abstraction and summarization

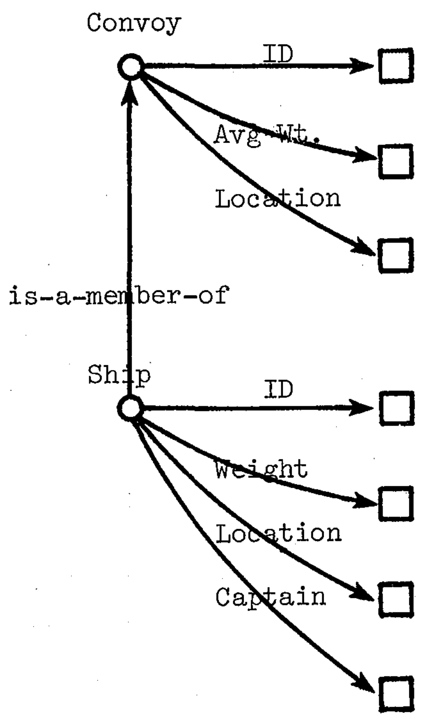
LS1:



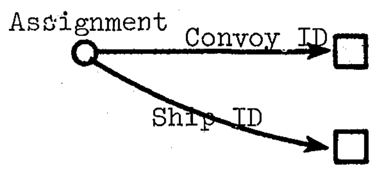
LS2:



GS:



IS:



```
Definition: Include LS1.Convoy and LS2.Ship.
            Range of c is Convoy, RANGE of a is ASSIGNMENT
            DEFINE FUNCTION isamemberof
               FOR s IN Ship
                   (isamemberof := c)
               WHERE ID(c) = ConvoyID(a) AND ID(s) = ShipID(a)
```

Figure 4.1 Examples of Integration (continued)

is a bijection between inches and cms, we can choose either function; in this example, we choose cms, and include the conversion formula from inches to cms in the view definition. For Weight, there is no conversion formula; instead, there is a table for converting between lbs and encoded weights. This table is stored in the Integration database and used in the view definition. Observe that we use the coarser scale for the supertype. (Of course, we could have used some other unifying scale; then we might have had to store two conversion tables in the Integration Database.) In more complex situations, conversion might require calling an arbitrary DBA-defined procedure (see Section 3.2).

4.1.3 Structural Differences

By structural differences we mean that the local schema graphs are not isomorphic. These differences include: missing functions and entity types (i.e., differences in aggregation [SS77]); and modelling a real-world object or relationship by an entity type in some LSs and by a function in other LSs. Missing functions are easily dealt with using generalization -- only the functions common to all the subtypes are defined on the supertype (see Figure 2.4). To integrate schemas with different entity types and functions, first define virtual entity types and functions, and then use generalization. There is considerable flexibility in designing the GS. We illustrate one possibility in Figure 4.1c.

4.1.4 Differences in Abstraction

The entity types and functions in the LSs may have been defined at different levels of generalization. The natural way of integrating these schemas is via generalization. Figures 2.4 and 3.1 give an example of integration when the functions on the entity types are at different levels. Figure 4.1d applies this technique to LSs with entity types at different levels. Again, several solutions are possible, and of these we show two.

Besides generalization, the LSs might differ in other forms of abstraction. In Figure 4.1e, the Convoy entity type in LS2 is a <u>set abstraction</u> of the Ship entity type in LS1 (a ship is a member of a convoy) [HM78]. The Average Weight function on Convoy is a <u>summarization</u> of the Weight function on Ship. Both of these abstractions are handled by defining a virtual function relating each ship to the convoy of which it is a member. This function is defined using additional information supplied by the DBA via the Integration database.

4.2 Data Integration

Once the structure of the global schema has been decided upon, its extension must be defined in terms of the extensions of the local schemas. But the extensions might disagree on the value of some functions. We discuss some causes of data discrepancy here.

1. The local databases are mutually inconsistent, but correct. One reason for this might be that entities that appear to be the same are actually different. Consider two identical LSs containing entity type EMP and functions EmpNo:EMP->INTEGER, Sal:EMP->INTEGER. EmpNo is the local identifier in each database. Suppose there is an entity e1 with EmpNo(e1)=1, Sal(e1)=25 in one database, and an entity e2 with EmpNo(e2)=1, Sal(e2)=30 in the other database. Suppose that although entities e1 and e2 have the same EmpNo value, they represent different employees in the real world. This implies that EmpNo is not an identifier for Emp entities in GS. This apparent discrepancy is easily resolved by concatenating a Database_ID with the local entity_ID for use as the global_ID.

   Another reason for the apparent discrepancy might be that although EmpNo is, in fact, an identifier for Emp in GS (so e1 and e2 do represent the same employee), the two functions are different; e.g., they represent salaries for two different jobs. This implies that the Sal functions are homonyms. We can resolve this discrepancy by including both functions (appropriately renamed) for the supertype, by treating Sal as a superfunction, or by deriving the Sal function on the supertype by some computation on the two Sal functions on the subtypes. The third solution was adopted in Figure 3.1.

   A third reason for the discrepancy might be obsolescence. Again, we can treat the two functions as homonyms, and use both (renamed appropriately, e.g., current_Sal and last_year's_Sal) for the supertype. Alternatively, we can use the more recent value (this is easily determined if data is timestamped). For the latter solution, Sal must be defined as a conditional function (with the conditions extended to include tests on timestamps).
2. The local databases are mutually inconsistent and incorrect. In this case, we again have several options. One is to use the more credible data. (If this can be determined a priori, a conditional function definition will suffice. This solution is adopted for the name function in Figure 3.1.) Alternatively, we can specify how to compute a value from the conflicting data, or trigger some appropriate action, e.g., notifying the user that a conflict has been detected. This last alternative requires the ability to embed procedure

calls in view definition statements. The procedures are invoked if necessary during execution of the modified query.

5. QUERY MODIFICATION

We have seen how to define the GS as a view of the LSs. We now describe algorithms for modifying queries against the GS into queries against the LSs. (We insist that global queries request the retrieval of only constant entities.)

For views defined using the constructs of Sections 3.1 and 3.2 (inclusion, virtual entity type and function definition), a straightforward extension of the algorithm described in [STON75] will work. The main steps are: given query q, for each entity variable x in q, do the following: replace the range statement RANGE of x IS X by a collection of range statements RANGE OF xi IS Xi, $1 \leq i \leq n$, where X1,..., Xn are the entity types in terms of which X was defined; replace each occurrence of f(x) in q's target list or qualification by its definition; finally, conjoin the qualification of X's definition to q's qualification.

But there are two features of queries and view definitions in the functional model that have no relational counterparts and must be specially dealt with. (i) Compositions of singlevalued functions must be unraveled before query modification. Replace g(f(x)) where f: X->Y, g: Y->Z, and x ranges over X, by g(y) where y is a new variable ranging over Y, and conjoin y=f(x) to the qualification. (ii) After modification, an atomic formula (x ISIN f(y)) may become (x ISIN {x' IN X WHERE qual}). This is simplified to (qual with x' replaced by x).

For views defined via generalization, query modification is more complicated. It might seem at first that the only complication is the presence of variables ranging over supertypes; for each such variable, the query must be replaced by the union of a set of subqueries, one per subtype. However there are two other issues to consider.

1. The query might refer to several conditional functions on X, whose definitions involve different partitions of X. We have to construct the coarsest common refinement π(i.e., the greatest lower bound with respect to the refinement partial order) of all these partitions. Then, the query must be replaced by the union of a set of subqueries, one for each block of π. This principle is illustrated in Figure 5.1a. (Observe that the range statements in the subqueries are not in the syntax described in Section 3. We shall show later how to rectify this.)

2. The query might refer to a superfunction f, e.g., in a formula x=f(y). The query must then be replaced by the union of a set of subqueries, one for each pair of subranges of x and y for which a subfunction of f is defined. This principle is illustrated in Figure 5.1b. (Note that if two subfunctions (e.g., homephone and workphone, have the same domains and the same ranges, then we can combine the two corresponding subqueries.)

Applying these two principles independently, however, would lead to a proliferation of subqueries, many of which might be irrelevant. For example, if we first apply Principle 1 to all the entity variables in the query of Figure 4.1, we get 6 subqueries (3 subranges for e, viz., EMP1-EMP2, EMP2-EMP1, EMP1 ∩ EMP2; times 2 subranges for PHONE, viz., PHONE1, PHONE2). But on subsequently applying Principle 2, we discover that only 4 of these are relevant -- the (EMP1-EMP2, PHONE2) and (EMP2-EMP1, PHONE1) pairs are ruled out by the qualification. The algorithm we describe below avoids generating unnecessary intermediate subqueries.

For describing the algorithm, we require the following definitions. Let q be a query, and let x be a supertype variable ranging over X. Let f1, ..., fn be the functions occurring in q either in a term fi(x) or in a formula x=fi(y) or x ISIN fi(y); For each i, let πfi be the partition of X induced by the definition of fi. (If fi is a conditional function, πfi is the partition induced by the conditions in the CASE statement defining fi; otherwise πfi is defined to be {Y|Y is a subtype of X}.) Let π(x) be the coarsest common refinement of all such partitions. The <u>subrange table</u> SRT(x) for variable x has n+1 columns. The first column is labelled x, and the others are labelled x.fi, 1 $\leq$ i $\leq$ n. Each row contains a block of π(x) and the corresponding definitions of the functions fi for that subrange of x. (If fi is a conditional function, this is the right hand side of the corresponding conditional assignment. If fi is a superfunction, this is the set of subfunctions defined over the subrange. (See Figure 5.2 for an example.) Each row of the table tells us which definition or subfunctions of fi to use for a given subrange of the range of x.

In the query modification algorithm below, we assume that the target list does not contain a set assignment of the form B:=<set> where <set> contains a reference to a supertype or a superfunction. This is not a restrictive assumption, for a query that violates it can be replaced by two queries. Thus, replace
q: RETRIEVE INTO RESULT (A:=f(x), B:=g(y)) WHERE qual;
where g:Y->->Z is a superfunction, and range of y is Y, by
q1: RETRIEVE INTO RESULT (A :=f(x), B:={z IN Z WHERE z ISIN g(y)}) WHERE qual;
q1 is now a special case of the following
q: RETRIEVE INTO RESULT (A:=f(x), B:={y in Y WHERE qual1}) WHERE

Assume the LSs and GS of Figures 2.4 and 3.1.

a. Conditional function

```
    Query: RANGE OF e IS EMP
           RETRIEVE INTO WEALTHY (Name := Name (e))
                         WHERE Sal(e) > 50K
```

The definition of Name induces the partition:

{EMP1, EMP1-EMP2}

The definition of Sal induces the partition:

{EMP1-EMP2, EMP2-EMP1, EMP1 ∩ EMP2}

Coarsest common partition = partition induced by the definition of Sal.

Subrange of e	Name	Sal
EMP1-EMP2	Name1(e)	Sal1(e)
EMP2-EMP1	Name2(e)	Sal2(e)
EMP1∩EMP2	Name1(e)	Sal1(e)+Sal2(e)

Subqueries:

```
1. RANGE OF e IS EMP1-EMP2
   RETRIEVE INTO TEMP1 (Name:= Name1(e)) WHERE Sal1(e)>50K
2. RANGE OF e IS EMP2-EMP1
   RETRIEVE INTO TEMP2 (Name := Name2(e)) WHERE Sal2(e)>50K
3. RANGE OF e IS EMP1 ∩ EMP2
   RETRIEVE INTO TEMP3 (Name := Name1(e))WHERE
                         (Sal1(e)+Sal2(e))>50K
Then, WEALTHY := TEMP1 ∪ TEMP2 ∪ TEMP3
```

Figure 5.1 Query Modification Principles

b. Superfunction:

```
Query: RANGE OF e IS EMP, RANGE OF p IS PHONE
       RETRIEVE INTO Z (SSNo := SSNo(e)) WHERE
                        1234 = No(p) AND p ISIN Phones(e)

       Phones has the subfunctions
         HPhone : EMP1 --> PHONE1
         WPhone : EMP1 --> PHONE1
         Phones2: EMP2 ->> PHONE2

Subqueries:

1. RANGE OF e IS EMP1
   RANGE OF p IS PHONE1
   RETRIEVE INTO Z1 (SSNo := SSNo (e)) WHERE
            1234 = No(p) AND (p=HPhone(e) OR p=WPhone(e)
2. RANGE OF e IS EMP2
   RANGE OF p IS PHONE2
   RETRIEVE INTO Z2 (SSNo := SSNo (e)) WHERE
            1234 = No(p) AND p ISIN Phones2(e)
   Then, Z=Z1 ∪ Z2
```

Figure 5.1 Query Modification Principles (continued)

qual; which must be replaced by the sequence

```
RANGE OF Y IS Y
RETRIEVE INTO TEMP (A:=f(x), B:=y) WHERE qual AND qual1
RANGE OF t,t1 IS TEMP
RETRIEVE INTO RESULT (A:=A(t), B:= {y IN Y WHERE
                     y=B(t1) AND A(t)=A(t1)}).
```

The first query retrieves a "flat" relation; then the second converts it into the unnormalized form requested by q.

<u>Algorithm</u>

Given query q

1. (i) Construct the natural join N of the following tables:

   SRT(x) [x.f=y.f] SRT(y), for every pair of supertype variables x, y, and superfunction f that occur together in a formula y=f(x) or y ISIN f(x) in q's qualification.

   (ii) Form the product of N with SRT (x), for every supertype variable x occurring in q and not used in (i).

2. Let T be the table constructed in 1. For each row r of T, construct a subquery by replacing the range of each supertype variable with the corresponding subrange given in the x entry

of r; each formula x = f(y), or x ISIN f(y), where f is a superfunction, with the formula $\bigvee_{i=1}^{k}$ x = fi(y), or $\bigvee_{i=1}^{k}$ x ISIN fi(y), where {f1,...,fp} is the y.f entry of r; and each term g(x), involving a conditional function g, with the corresponding definition given in the x.g entry of r (if the definition includes a qualification, this is conjoined with the qualification of the subquery).

3. The ranges in the subqueries constructed in 2 may involve set operations. These are eliminated as follows.
   i. Replace a subquery having range of e is E1 $\cup$ E2 by two subqueries having range of e is E1 and range of e is E2.
   ii. Replace range of e is E1 $\cap$ E2 by
      range of e1 is E1
      range of e2 is E2
      and conjoin ID(e1) = ID(e2) to the qualification. Replace e in every term f(e) in the subquery by f(e1) if f is defined on E1, and by f(e2) if f is defined on E2. Similarly, replace e in every clause e = f(x) or e ISIN f(x) by e1 if f : x -> E1, and by e2 if f:X->->E2, where X is the range of x.
   iii. Replace range of e is E1 - E2 by
      range of e is E1
      range of e2 is E2
      and conjoin ($\forall$e2) (ID(e2) $\neq$ ID(e) to the qualification.

   [Note: this requires augmenting the simple query language of Section 2 with quantifiers. This extension has no effect on query modification.]
4. Modify the subqueries constructed in 3.

Figure 5.2 illustrates the query modification algorithm. We omit the proof that this algorithm is correct, i.e., it modifies a query into an equivalent collection of subqueries; and non-redundant, i.e., none of the subqueries generated by it produces a result that is always empty or always subsumed by the result of some other subquery. (The proof is by induction and case analysis, and is given in [HWANG].)

For queries over a subtype, replace the subtype by the intersection of its supertypes, and replace its functions by their definitions.

6. CONCLUSION

Simplifying access to a heterogeneous distributed database system requires the definition of a logically integrated global view of the local databases. There are two aspects to the

Assume the LSs and GS of Figures 2.4 and 3.1.

Query

```
        q: RANGE OF e IS EMP, RANGE OF p IS PHONE
           RETRIEVE INTO RESULT (Name := Name(e))
             WHERE Sal(e) > 50K and 1234 = No(p) AND
             p ISIN Phones (e)
```

SRT(e):

e	e.Name	e.Sal	e.Phones
EMP1-EMP2	Name1(e)	Sal1(e)	{HPhone,WPhone}
EMP2-EMP1	Name2(e)	Sal2(e)	Phones2
EMP1 EMP2	Name1(e)	Sal1(e)+Sal2(e)	{HPhone,WPhone}
EMP1 EMP2	Name1(e)	Sal1(e)+Sal2(e)	Phones2

SRT(p):

p	p.Phones	p.No
PHONE1	{HPhone,WPhone}	No(p)
PHONE2	Phones2	No(p)

Subqueries

```
q1: RANGE OF e IS EMP1-EMP2, RANGE OF p IS PHONE1
    RETRIEVE INTO T1 (Name :=Name1(e)) WHERE
        Sal1(e) >50K AND 1234 = No(p) AND
        (p = HPhone(e) OR p = WPhone(e))

q2: RANGE OF e IS EMP2-EMP1, RANGE OF p IS PHONE2
    RETRIEVE INTO T2 (Name := Name2(e)) WHERE
        Sal2(e) >50K AND 1234 = No(p) AND p ISIN Phones2(e)

q3: RANGE OF e IS EMP1∩EMP2, RANGE OF p IS PHONE1
    RETRIEVE INTO T3 (Name := Name 1(e)) WHERE
       Sal1(e)+Sal2(e) >50K AND 1234 = No(p) AND
         (p=HPhone(e) OR p=WPhone(e))

q4: RANGE OF e IS EMP1 ∩ EMP2, RANGE OF p IS PHONE2
    RETRIEVE INTO T4 (Name := Name1(e)) WHERE
       Sal1(e)+Sal2(e) >50K AND 1234 = No(p) AND
        p ISIN Phones2(e).
```

Then, RESULT = T1 ∪ T2 ∪ T3 ∪ T4.

Each subquery must then be modified to eliminate the set expressions in the range statements, and to replace entity types and function names with those used in the local schemas.

Figure 5.2 Illustrating the Query Modification Algorithm

integration problem. First, the local schemas may model real world objects and relationships differently; second, the databases may be mutually inconsistent. This paper identified various kinds of structural and data inconsistencies that might exist. It described a versatile view definition facility and illustrated the use of this facility for resolving inconsistencies. In particular, the importance of the semantic modelling concept of generalization was emphasized. It has been postulated that the derivation of an integrated global schema from the local schemas can be automated [MB81]. However, as we show in this paper, there usually are many different global views that can be defined for a given collection of local schemas. Which one the DBA should choose depends strongly on the semantics of the local databases and on individual application requirements. Our approach, therefore, is more pragmatic. We suggest a two-step procedure as a guide to the DBA: first, resolve naming conflicts, differences in the representation of real-world objects and relationships, etc., by renaming or defining virtual entity types and functions; then, generalize to resolve differences in aggregation and data inconsistencies.

Once the global view is defined, users can pose queries against it. These queries have to be modified into equivalent queries against the local databases. We described an algorithm for query modification. The problem of optimally processing the subqueries generated by this algorithm is currently under investigation.

Acknowledgment

We would like to thank John Smith, Terry Landers, Nat Goodman, and the other members of the Multibase group at the Computer Corporation of America, Ron Pinter of MIT, and Mohamed Gouda of the University of Texas at Austin for their comments on the ideas presented in this paper. Some of these ideas are currently being implemented as part of the Multibase system at CCA.

7. REFERENCES

[CGT75] Chamberlin, D.D., J.N. Gray and I.L. Traiger. "Views, Authorization and Locking in a Relational Database System". Proc. AFIPS NCC 1975, pp. 425-430.

[CODD79] Codd, E.G. "Extending the Database Relational Model to Capture More Meaning". ACD TODS 4:4, Dec. 1979, pp. 397-434.

[DAYAL79] Dayal, U. "Schema_Mapping Problems in Database

Systems". Ph.D. Dissertation, Tech. Rep. TR-11-79, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, August 1979.

[DB82] Dayal, U. and P.A. Bernstein. "On the Updatability of Network Views -- Extending Relational Views Theory to the Network Model". Information Systems, 7:1, 1982 (to appear).

[HK81] Hecht, M.S. and L. Kerschberg. "Update Semantics for the Functional Data Model". Database Research Rep. 4, Bell Labs., Holmdel, NJ, January 1981.

[HM78] Hammer, M. and D. McLeod. "The Semantic Data Model -- A Modelling Mechanism for Database Applications". Proc. SIGMOD Conf., 1978, pp. 26-36.

[HSW75] Held, G.D., M.R. Stonebraker and E. Wong. "INGRES -- A Relational Database System". Proc. AFIPS NCC, 1975, pp. 409-416.

[HWANG] Hwang, H.-Y., Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, (in preparation).

[KG81] Katz, R. and N. Goodman. "View Processing in Multibase -- A Heterogeneous Database System". in Entity-Relationship Approach to Information Modelling and Analysis (P.P. Chen, ed.), ER Institute, Saugus, Calif., 1981, pp. 259-280.

[LG78] Lee, R.M. and R. Gerritsen. "Extended Semantics for Generalization Hierarchies". Proc. SIGMOD 1978, pp. 18-25.

[MB81] Motro, A. and P. Buneman. "Constructing Superviews". Proc. SIGMOD, 1981, pp. 56-64.

[MBW80] Mylopoulos, J., P.A. Bernstein and H.K.T. Wong. "A Language Facility for Designing Database-Intensive Applications". ACM TODS 5:2, June 1980, pp. 185-207.

[ROUSS79] Roussopoulos, N. "CSDL: A Conceptual Schema Definition Language for the Design of Database Applications". IEEE Trans. on Software Engineering, 5:5, September 1979, pp. 481-496.

[SHIP81] Shipman, D.W. "The Functional Data Model and the Data Language DAPLEX". ACM TODS 6:1, March 1981, pp. 140-173.

[SMITH et al.81] Smith, J.M., P.A. Bernstein, U. Dayal, N. Goodman, T. Landers, K.W.T. Lin and E. Wong. "Multibase -- Integrating Heterogeneous Distributed Database Systems". Proc. AFIPS NCC 1981, pp. 487 - 499.

[SS77] Smith, J.M. and D.C.P. Smith. "Database Abstractions: Aggregation and Generalization". ACM TODS 2:2, June 1977, pp. 105-133.

[STON75] Stonebraker, M.R. "Implementation of Integrity Constraints and Views by Query Modification". Proc. SIGMOD Conf., 1975, pp. 65-78.

A. CONTROL OVER DUPLICATE ELIMINATION

The calculus-based language that we described in Sections 2 and 3 always eliminates duplicates from the result of a query. The user has no control over duplicate elimination. Sometimes, however, duplicates may be desired in the output. Consider, for example, the schema of Figure A.1 Suppose we want to retrieve the names of all employees. The obvious RETRIEVE statement:

```
RANGE OF a IS ASSIGN
RETRIEVE INTO RESULT (Name(a))
```

returns each name only once, although there might be many employees with the same name. Adopting the "prime" option of QUEL [HSW75]

```
RANGE OF a IS ASSIGN
RETRIEVE' INTO RESULT (Name(a))
```

will not work either, because now the name of an employee assigned to two projects will appear twice. Augmenting the language with an iterative statement: FOR EACH <entity variable> solves this problem. The query in the above example then becomes:

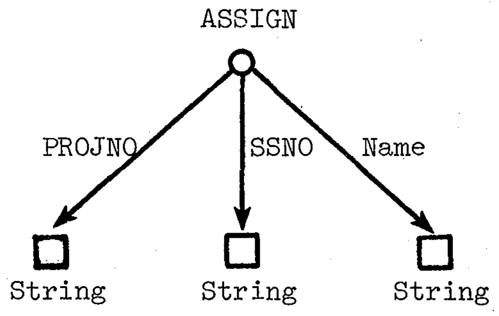
```
RANGE OF a IS ASSIGN
RANGE OF no, na IS STRING
FOR EACH no
FOR EACH na
   RETRIEVE INTO RESULT (na) WHERE ( ∃ a) (SSNO(a)=no
          AND Name(a)=na)
```

(We augment qualifications to include quantifiers, as in Section 5.)

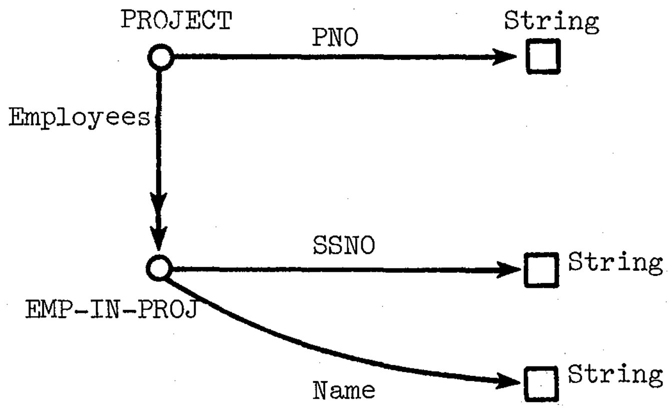
An analogous problem arises in view definition. Suppose we want to define the hierarchical view of Figure A.1(b) over the schema of Figure A.1(a). We have to eliminate duplicates to define the PROJECT entity type, but then retain duplicates in defining EMP_IN_PROJ. Figure A.1(c) shows how to define this view using the iterative construct (for details, see [DB82]).

Query modification is not much affected by the iterative statement if duplicates are to be eliminated in the result of the query. However, if duplicates are not to be eliminated (i.e., if the query is formulated using FOR EACH), then care must be taken in substituting for any variable whose range was defined by duplicate elimination. Figure A.2 illustrates query modification with two queries, one eliminating duplicates and the other not. Observe that for the second query, the iteration over PROJECT is replaced by an iteration over strings in the image of the PROJNO function, since there is one PROJECT entity per string in PROJNO(ASSIGN).

a. Schema



b. View



c. View definition

```
RANGE OF a IS ASSIGN
DEFINE ENTITY TYPE PROJECT (PNO := PROJNO(a))
RANGE OF p IS PROJECT
FOR EACH p
FOR EACH a
    DEFINE ENTITY TYPE EMP-IN-PROJ (SSNo := SSNo(a),
                                    Name := Name(a))
        AND MAP FROM p UNDER employees
        WHERE PNo(p) = PROJNO(a).
```

Figure A.1 Using the Iterative Statement

```
a. Query: RANGE OF p IS PROJECT, RANGE OF e IS EMPINPROJ
          RETRIEVE INTO RESULT (PNo := PNo(p), Name := Name(e))
                   WHERE e ISIN employees(p)

   Modified query: RANGE OF a, a′ IS ASSIGN
                   RETRIEVE INTO RESULT (PNo := PROJNO(a), Name := Name(a′)
                      WHERE PROJNO(a)=PROJNO(a′).

   After simplification: RANGE OF a′ IS ASSIGN
                         RETRIEVE INTO RESULT (PNo := PROJNO(a′), Name :=
                                                                 Name(a′)).

b. Query: RANGE OF p IS PROJECT, RANGE OF e IS EMPINPROJ
          FOR EACH p
          FOR EACH e
              RETRIEVE INTO RESULT (PNo := PNo(p), Name := Name(e))
                       WHERE e ISIN employees(p)

   Modified query: RANGE OF a, a′ IS ASSIGN, RANGE OF s IS STRING
                FOR EACH s
                FOR EACH a′
                    RETRIEVE INTO RESULT (PNo := s, Name := Name(a′))
                            WHERE s = PROJNO(a′) and (∃a) (s = PROJNO(a)).

   After simplification: RANGE OF a′ IS ASSIGN, RANGE OF s IS STRING
                      FOR EACH s
                      FOR EACH a′
                          RETRIEVE INTO RESULT (PNo := S, Name := Name(a′))
                                  WHERE s = PROJNO(a′).
```

Figure A.2 Query Modification with the Iterative Construct

Selective Broadcast in Packet-Switched Networks

David W. Wall
Computer Science Department
The Pennsylvania State University

Abstract

Many researchers in the area of distributed networks have found it convenient to assume the existence of a facility for routing broadcast messages to all the nodes in a network, or selective broadcast messages to some subset of those nodes. However, relatively little work has been done on the design of such a facility. This paper extends a mechanism developed by Dalal for total broadcast to the problem of selective broadcast, by converting an existing sequential algorithm on graphs into a distributed algorithm suitable for network use. Because of this approach, the results may provide some insight to the general problem of designing distributed algorithms.

1. Introduction

A loosely-coupled store-and-forward network like the ARPANET routes a message from one node to another along some series of links starting at the source node and ending at the destination. The problem of selecting the best route for a given message has been considered in detail, and a simple but effective mechanism is provided by the ARPANET [8,9].

This material is based upon work supported by the National Science Foundation under Grant MCS-8102278 and the Joint Services Electronics Project under contract N-00014-75-C-0601.

Much of the recent work on the effective use of such a network has assumed the existence of a mechanism for _message broadcast_, by which a node can send an identical message to every other node in the network; or more generally a mechanism for _selective broadcast_, by which the node sends an identical message to several nodes but not necessarily to the whole network. Broadcast and selective broadcast would be useful in updating a distributed data base [4,9], in maintaining a distributed file system [3] and other distributed resources [1,2], and in the use of parallel systems to speed up problem solving in artificial intelligence [6] and elsewhere [11]. Unfortunately, not enough work has been done on actually providing such a service; no explicit mechanism for broadcasting is available in the ARPANET.

This paper describes one approach to the design of a selective broadcast facility. Our aim will be to consider a total broadcast mechanism developed by Dalal [3] and then generalize it to the problem of selective broadcast. We will do this by examining a sequential algorithm on graphs presented by Kou, Markowsky, and Berman [7] and modifying it for use in a distributed environment, using Dalal's algorithm as a subroutine. This discussion may therefore be of some relevance to the general problem of applying sequential algorithms to distributed applications.

2. Selective broadcast

There are several easy techniques for selective broadcast (described by Wall [12], and previously by Dalal [3] in the context of total broadcast) that do not require any special structure to be imposed on the network beyond that structure necessary for the single-destination mechanism such networks already have. These techniques bear the fault of their virtues, however - because they are general enough to work at any time and for any destinations, without any special preparation, they necessarily involve a certain amount of overhead or redundancy, incurred each time a broadcast is sent. For example, if we simply send a separate copy of the broadcast to each destination, we may pass several redundant copies over the same link, because the routes from the source to several different destinations include that link.

On the other hand, if the network is fairly stable and broadcasting among a given group of nodes is fairly frequent, it may be worthwhile to impose some additional structure on the network so as to make broadcasting easier. In this way we accept some initialization time and occasional maintenance time so that we can avoid the redundancy or overhead of some simpler scheme.

A particularly useful structure to impose is a complete or partial spanning tree. A spanning tree of the network is a set of links that connects all the nodes without including any cyclic paths. If there are n nodes in the network, a spanning tree will consist of n-1 links. We can route a broadcast from node to node along the branches of a spanning tree: The node that initiates the broadcast sends a copy along each incident branch, and every other node forwards the broadcast by sending a copy along each incident branch except the one on which the broadcast arrived. A broadcast routed in this manner will be copied only n-1 times, which is the minimum since there are that many destinations. If we select the spanning tree carefully, we can arrange for it to have other useful properties as well.

For example, Dalal [3] considers the problem of minimum-cost total broadcast. If we assign to each link an estimate of the cost of sending a message across that link, we can build the minimum spanning tree (MST) of the network - that is, the tree for which the sum of all these costs is as small as possible. A broadcast routed along the minimum spanning tree will incur a cost to the network as a whole that is as small as possible.

Dalal presents a distributed algorithm for constructing such a tree in a network environment. The algorithm is based on Prim's principle [10] that any fragment of a minimum spanning tree, including any single vertex, is connected by a branch of the tree to the nearest vertex of the graph that is not in the fragment; that is, the cheapest edge leading away from the fragment is a branch of the MST. In Dalal's algorithm, each vertex starts by marking the cheapest incident edge as a branch; the larger fragments that this creates proceed to join into still larger fragments, until finally some vertex discovers that there is only one fragment, namely the MST. This description somewhat oversimplifies the algorithm, which is performed asynchronously and need not fall into such well-defined phases.

How can we generalize this to selective broadcast? Consider the small network in Figure 1. The heavy links are the branches of a minimum spanning tree such as Dalal's algorithm might find. Suppose that nodes A, B, C, and D will be working together for a while and want to be able to do broadcasting among themselves without bothering the rest of the network more than necessary. We will call such a set of nodes a <u>broadcast group</u>. If this group uses only the direct links AB, BC, and CD, each broadcast will have a cost of 50, but if the group can convince Z to join as a passive member, then the group can broadcast via the tree consisting of edges AZ, BZ, CZ, and DZ, at a cost of only 40. In either case there is little resemblance between the tree used for selective broadcast and the local portion of the minimum spanning tree used for total broadcast. Note also that a broadcast group might not form a connected subgraph, in which case the addition of extra vertices would be unavoidable.

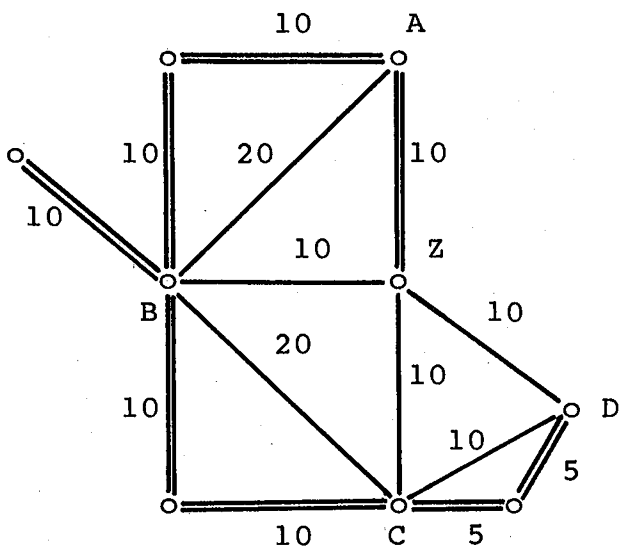


Figure 1. Network with minimum spanning tree

This leads to a generalization of the minimum spanning tree. Given a connected weighted network, and a subset S of the nodes in the network, a <u>Steiner tree</u> is a tree of network links that spans all the nodes in S but is not necessarily restricted to them. A <u>minimum Steiner tree</u> for S is one whose cost is smallest over all Steiner trees for S. Thus if we have a set of nodes trying to form a broadcast group, we could build the minimum Steiner tree for that set of nodes, and broadcasts via that tree would cost as little as possible. Unfortunately, the problem of finding a minimum Steiner tree is NP-complete [5].

The next best thing would be to use an approximation to the minimum tree. Kou, Markowsky, and Berman [7] present a sequential algorithm that builds a tree whose cost is less than twice the minimum cost. We turn to this algorithm next.

3. The KMB algorithm

If we are given a graph G and a subset S of the vertices, which we will call the set of Steiner points, the problem is to find a good (if not necessarily minimum) Steiner tree T_H for the set S on the graph G. The algorithm described by Kou, Markowsky, and Berman proceeds as follows.

Step 1. Build the complete undirected distance graph G_I for S over G as follows. The vertices of G_I are the vertices in S. Construct an edge for G_I between every pair of vertices. Assign a cost to each such edge by finding its endpoints in the original graph G and computing the cost of the cheapest path in G between these vertices. Thus G_I represents a summary of the costs of paths in G between the Steiner points.

For example, if we are using the network and broadcast group of Figure 1, we obtain the graph G_I shown in Figure 2.

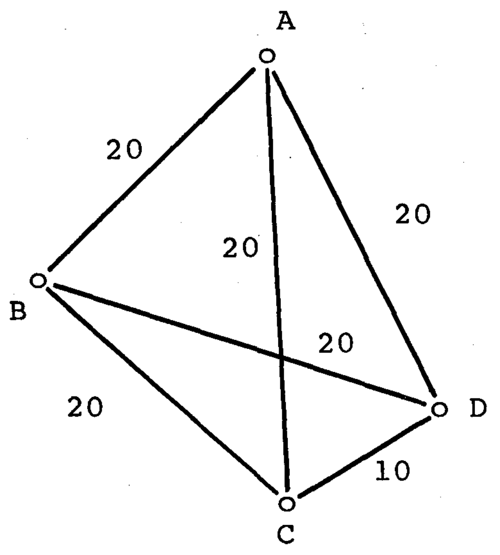


Figure 2. The graph G_I

Step 2. Find any minimum spanning tree T_I of G_I. For example, a minimum spanning tree of the G_I in Figure 2 appears in Figure 3. Note that there were several other choices.

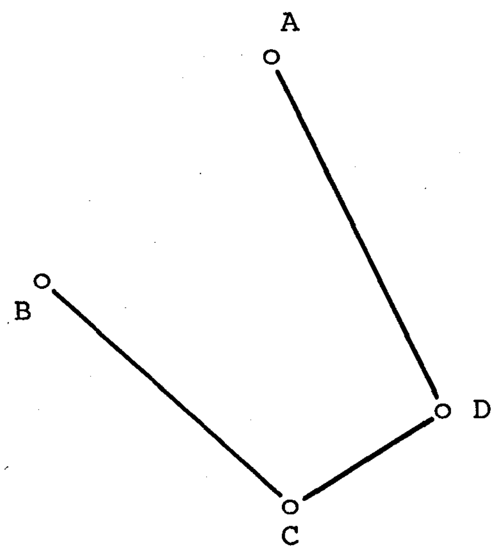


Figure 3. The minimum spanning tree T_I

Step 3. Build the subgraph G_S of the original graph G by replacing each branch in T_I by any corresponding cheapest path in G. If such a path includes vertices not already in G_S, add them to G_S as well. For example, Figure 4 contains such a subgraph obtained from the T_I of Figure 3. Again, note that other choices were possible for each of the branches.

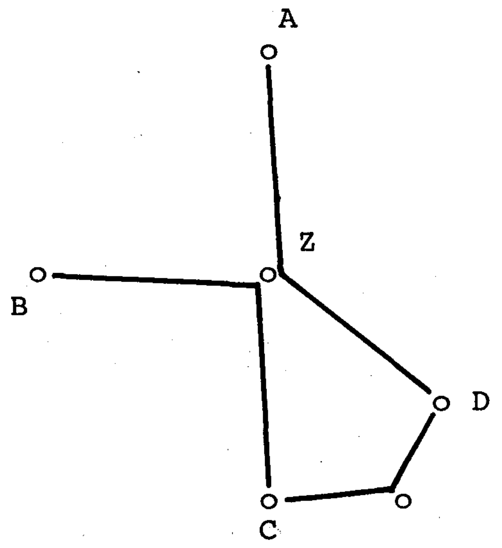


Figure 4. The subgraph G_S

Step 4. Find any minimum spanning tree T_S of the subgraph G_S. Figure 5 is an example.

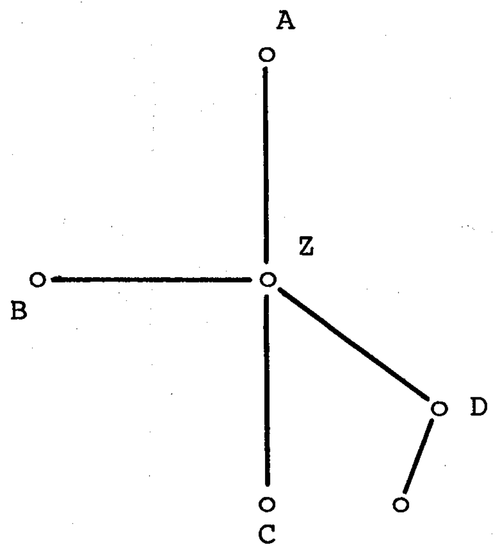


Figure 5. The second minimum spanning tree T_S

Step 5. Build a Steiner tree T_H from T_S by deleting branches from T_S, if necessary, so that all the leaves of T_H are Steiner points. For example, in the T_S of Figure 5, the lower right vertex is not a Steiner point and does not lie between two Steiner points, and so we can delete it, producing the tree in Figure 6.

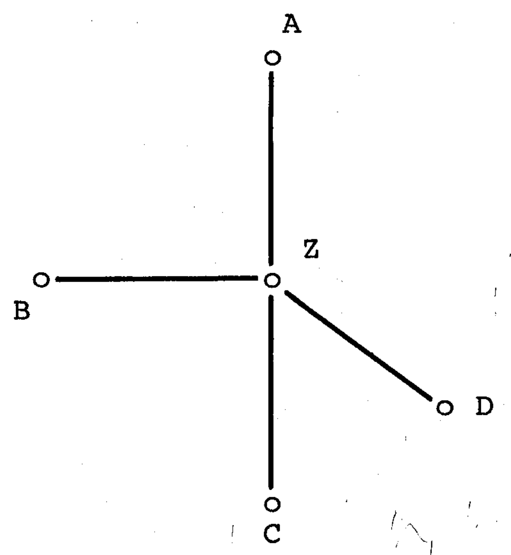


Figure 6. The good Steiner tree T_H

In this example we have constructed the minimum Steiner tree. Kou, Markowsky, and Berman show that in general the cost of T_H is less than twice the cost of the minimum Steiner tree. In practice it seems likely to do better than this, as is suggested by the following result, whose proof appears elsewhere [12].

Theorem. If a minimum Steiner tree exists that spans only the Steiner points, then the KMB Algorithm will find a minimum tree if it uses a suitable tie-breaking rule.

Briefly, if we break ties between edges in favor of those that correspond to direct edges in G as opposed to those that represent multiple-edge paths of the same cost, we can build a minimum-cost Steiner tree that is also restricted to the Steiner points, if only such a tree exists. In this case our broadcasting will incur minimum cost, and our broadcast group will have essentially no impact on the rest of the network. So perhaps the KMB Algorithm is a good algorithm to consider further.

4. Distributing the KMB Algorithm

We plainly have a head start on the problem of distributing this algorithm, because a good deal of the work involves building minimum spanning trees, a problem already explored by Dalal. Dalal's approach must be slightly generalized, since it was originally developed to find the MST of the graph whose topology corresponds to the network doing the work, which is not the case in Step 2 of the KMB Algorithm. But by depending on the underlying mechanism for single-destination message routing, we can make Dalal's algorithm find the MST of any graph as long as there is a correspondence between the vertices of the graph and the nodes of the network.

We will consider the five steps of the KMB Algorithm in turn.

Step 1 presents a bit of a problem. This step dominates the time-complexity of the sequential algorithm, simply because a cheapest path might meander through a lot of vertices before finally reaching its destination. We must find something like the transitive closure of the network if we want to know what these cheapest paths are, even if our broadcast group is very small. We can sidestep this issue by depending on the network as a whole to maintain this information, a decision we can justify in two ways. First, it is possible that the network may already maintain the information for the benefit of the underlying single-destination mechanism; the ARPANET, for example, keeps the transitive closure of the link delays rather than the link costs, but a network more interested in cheap routing than in fast routing might maintain the costs instead. Second, there will presumably be many other groups besides the one we are currently building, and they need the transitive closure of the costs as well; thus it seems reasonable not to charge the maintenance of this information solely to the group now forming.

Step 2 builds a minimum spanning tree of G_I. As discussed earlier, we can use Dalal's algorithm for this, if we remember that the graph under consideration is not isomorphic to the network.

Step 3 is a nuisance. It amounts to telling everyone, including the nodes being added to the group, what the paths are. This turns out to be a little messy, since a given added node might appear on more than one path. Fortunately, we can absorb this step into Step 2; whenever we declare an edge of G_I to be a branch of T_I, we can send a message along the corresponding network path, bringing in the necessary added nodes and informing all concerned about the connections of the path in what is to become G_S.

Step 4 is the easiest so far; since we are now working with a portion of the network, we can simply use Dalal's algorithm as it is, without any special considerations.

Finally, Step 5 prunes any unnecessary added nodes from the tree. We can do this using a variation of a broadcast that we might call a convergecast. Some node performing Dalal's algorithm in Step 4 will decide that the MST is complete; it can send a broadcast to that effect out on this newly-built tree. When this broadcast reaches the leaves it bounces back as follows: if a leaf is an added node, it prunes itself by sending back a message to that effect; if not, it sends back a simple acknowledgement. An interior node prunes itself if it is an added node and all but one of its neighbors prune themselves; it sends a message to that effect to the remaining neighbor. Eventually these responses converge somewhere in the middle of the tree, and everyone who should be pruned has been pruned.

Thus it is shown (at least informally) that we can build a low-cost Steiner tree in a distributed environment by modifying the KMB Algorithm and using a slightly extended form of Dalal's algorithm as a subroutine. A more detailed discussion of the problems involved appears elsewhere [12].

5. Going a step further

A moment's reflection may lead one to wonder just how much the KMB Algorithm accomplishes with its second MST construction. The graph G_S is likely to be pretty sparse; in fact it is not hard to show that if there are no ties then G_S is a tree already, in which case the second MST construction is not needed. This demand is too stringent, however, and we can state a more relaxed requirement that still lets us omit that step.

We will say that a consistent tie-breaking rule has two properties:

Edge rule. Ties between pairs of equal edges with a common endpoint are broken consistently with respect to the other endpoints. For example, in Figure 7 we have a pair XM and XN of equal edges that share an endpoint X, and another pair YM and YN that share an endpoint Y. If we are using a consistent edge rule, then we must break the ties in favor of XM and YM or else in favor of XN and YN. This is not a harsh requirement; Dalal assumes a similar tie-breaking rule to prevent his algorithm from creating a cycle in his minimum spanning tree.

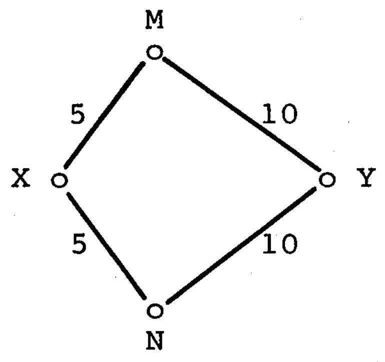


Figure 7. Applying the edge rule

<u>Path rule</u>. The tie between a pair of equal paths whose end-points are the same must be broken the same way if the two paths are extended by a common edge. For example, in Figure 8 we have a pair of equal paths between M and N. If we break that tie in favor of the upper path, we must also choose the upper path from M through N to Z in favor of the lower path through those points.

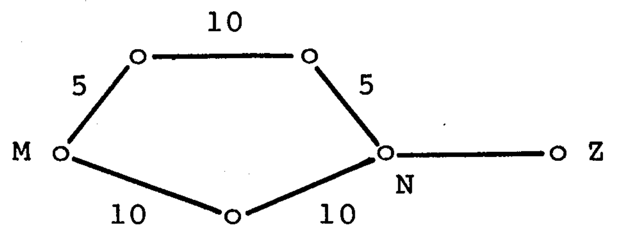


Figure 8. Applying the path rule

Given this definition of a consistent tie-breaking rule, we can prove the following result [12].

Theorem. If we perform the KMB Algorithm using a consistent tie-breaking rule, then the graph G_S produced in Step 3 is a tree.

This means that Step 4 is unnecessary.

A useful easier result is the following.

Theorem. If G_S is a tree, then none of its leaves are added nodes; that is, all its leaves are in S.

This is simply because a node gets added because it is on a path between two Steiner points. Each such added node must therefore have a degree in G_S of at least two, and hence cannot be a leaf. The important consequence is that if we can omit Step 4, we can also omit Step 5.

6. Summary

By distributing the KMB Algorithm in a naive fashion, we can build a low-cost Steiner tree suitable for use in selective broadcast. This involves using Dalal's algorithm twice to build certain minimum spanning trees, and using the interesting technique of the convergecast to do the pruning at the end.

By taking a deeper look at the sequential KMB Algorithm, however, we can do better. If we use a consistent tie-breaking rule, we can eliminate the second MST construction and hence also eliminate the final pruning step. We have already absorbed Step 3 into the main MST construction, and have argued that the network as a whole should maintain the transitive closure information computed in Step 1, for the benefit of all broadcast groups. Thus we are left with a single application of Dalal's algorithm, slightly complicated by the fact that the input graph is not isomorphic to the network and by the fact that the resulting tree must be incrementally translated into a tree in the original network. It still seems fair to say that low-cost selective broadcast need not be much harder to provide than Dalal's minimum-cost total broadcast.

A fairly obvious additional moral has been pointed out by others in other contexts, but may nevertheless be worth repeating. Namely, the better you understand an existing algorithm, the better you can fit it to your application.

Acknowledgements

I would like to thank Susan Owicki for vast amounts of advice and encouragement, and my colleagues John Gilbert, Brent Hailpern, and Sam Bent for letting me bend their ears.

References

[1] J. Eugene Ball, Jerome Feldman, James R. Low, Richard Rashid, and Paul Rovner. RIG, Rochester's intelligent gateway: System overview. IEEE Transactions on Software Engineering 2, 4 (1976), pages 321-328.

[2] David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. Pup: An internetwork architecture. Report CSL-79-10, Xerox Palo Alto Research Center, July 1979.

[3] Yogen Kantilal Dalal. Broadcast Protocols in Packet Switched Computer Networks. PhD thesis, Stanford University, April 1977. (Computer Systems Lab Technical Report 128.)

[4] Jim Gray. Notes on data base operating systems. IBM Research Report RJ2188 (30001), San Jose, California, 1978.

[5] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, Complexity of Computer Computations, pages 85-103. Plenum Press, New York, 1972.

[6] William A. Kornfeld. ETHER - a parallel problem solving system. Sixth International Joint Conference on Artificial Intelligence, August 1979, pages 490-492.

[7] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. Acta Informatica 15 (1981), pages 141-145.

[8] John M. McQuillan. Adaptive Routing Algorithms for Distributed Networks. PhD thesis, Harvard University, May 1974 (BBN Report 2831).

[9] John M. McQuillan, Ira Richer, and Eric C. Rosen. An overview of the new routing algorithm for the ARPANET. Sixth Data Communications Symposium, November 1979, pages 63-68.

[10] R. C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, November 1957, pages 1389-1401.

[11] Reid Garfield Smith. A Framework for Problem Solving in a Distributed Environment. PhD thesis, Stanford University, December 1978. (Computer Science Department Technical Report HPP-78-28.)

[12] David Wayne Wall. Mechanisms for Broadcast and Selective Broadcast. PhD thesis, Stanford University, June 1980 (Computer Systems Lab Technical Report 190).

PERFORMANCE ANALYSIS OF A SHORTEST-DELAY PROTOCOL*

Liang Li, Herman D. Hughes, Lewis H. Greenberg
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824
(517) 353-5152

ABSTRACT

A generalized shortest-delay access method (SDAM) protocol for local networks is defined and evaluated. This protocol differs from a previously reported SDAM [16] in that it accommodates a branching-bus topology instead of a single-bus network. It is shown that for small bus-delays, SDAM performs very close to that of M/D/1--with perfect scheduling. In this paper, the performance evaluation of SDAM is more pragmatic in that the effects of various protocol overheads (e.g., decoding, turnaround time, initializing packets, etc.) are taken into account. An analysis of the tradeoffs between exhaustive and nonexhaustive transmission disciplines is also presented.

Keywords: Local network, Protocol, Virtual token, Bus propagation delay, Branching-bus topology, Protocol overhead, Carrier-sensing, Exhaustive/non-exhaustive transmission, Queueing delays

* Research supported in part by NSF - Grant No. PRM-8115413

1. Introduction

In a recent paper [16], Li and Hughes proposed an access-level protocol for local computer networks (LCN). This protocol employs a scheme which is analogous to the one-directional shortest-seek-time-first (SCAN) algorithm advanced by Denning et al. [6,7,8] and is referred to as the "shortest-delay access method" (SDAM). Briefly, the SDAM protocol has the following properties [16]:

- works on a single-trunk bus-structured local network,
- has a decentralized control,
- maintains conflict-free transmissions,
- uses simple algorithms and little control overheads,
- performs closely to M/D/1 with perfect scheduling in ideal cases (taking into account the bus propagation delay). In particular, the performance of SDAM exceeds that of the popular CSMA/CD protocol in medium to high loads,
- provides adequate services to a large number of users (nodes) (e.g., 1000 nodes).

Recognizing the inconvenience of a single-bus topology (e.g., reconfiguring the network or adding new nodes at certain locations), this paper generalizes the SDAM protocol to a branching-bus network. A discussion of this generalization is presented in Section 2 of this paper.

In Section 3, two variants of the SDAM protocol, the closed SDAM (C-SDAM) and the open-ended SDAM (OE-SDAM), are closely examined for their relative merits. That is, the performance of these protocols is evaluated by both analytic and simulation models and compared to that of M/D/1 with perfect scheduling. Since the OE-SDAM provides equal access to all nodes, the performance evaluations throughout Section 3 will focus on this protocol.

In order to claim that the implementation of SDAM is feasible, the effects of the following three operating overheads of SDAM are evaluated: the decoding/turnaround time, the carrier-sensing time, and the token-initializing packet time.

The last part of Section 3 considers two transmission disciplines (exhaustive and non-exhaustive) which are possible for any local network protocols. The performance differences between these two disciplines are analyzed in terms of the distributions, means, and variances of their respective queueing delays.

Finally, in Section 4, a summary of this paper is presented.

2. The Shortest-Delay Access Method (SDAM) Protocol

2.1 Basic Concepts of SDAM

The underlying concept of SDAM is to reduce the delay (i.e., the "change-over" time) between two consecutive transmissions by different nodes. In order to do this, the nodes on the bus must be numbered sequentially from left to right or vice-versa. We can then draw an analogy between the virtual-token passing of

SDAM and the scanning action of the read/write head of the disk (see [6]). The token may be viewed as the disk head, scanning across the tracks (i.e., the nodes on the bus), and en route processes requests referencing those tracks (i.e., triggers transmission of packets from the nodes). To achieve distributed control and still avoid conflicts, SDAM uses a "token-direction" code on each packet to indicate the direction of current scan. The "virtual-token," as perceived by a node, is actually the absence of any more packets following a passing packet, thus allowing the node to start its packet transmission. The packets that missed the token will have to wait until the token passes the node again.

2.2 Network Configuration

Before we describe the rules of the SDAM protocol, let us first define a general configuration for which the algorithm will apply.

1. The transmission medium is a single common bus (e.g., coaxial cable), and is assumed error-free. The end-to-end propagation delay on the bus is $\underline{a}$. Later we generalize this network topology to a branching-bus network.
2. There are N nodes connected to the bus via communication interface units (CIUs), which function as decouplers and buffers. Therefore, we may consider the network as composed of functionally homogeneous nodes. These nodes are numbered 1 to N from one end of the bus to the other.
3. Each CIU has the carrier-sensing capability. In addition, it is assumed that the CIUs can identify the source and the destination addresses as well as the "token-direction" code on the passing packets.
4. There is a decoding/turnaround time $\underline{t}$ that is needed for the node to change from the receiving state to the transmitting state. This time is independent of the time required for carrier-sensing, which is assumed to have length $\underline{d}$.
5. There may be either one or two end-nodes attached to the bus. These nodes contend for the access of the bus as normal nodes do--only they generate a control packet (or token-initializing packet, TIP) that contains a special bit pattern to initialize the token passing. It is also possible to add this feature to the user-nodes located on either end of the bus, so that they serve as both user- and end-nodes.
6. The data packets are of fixed length, and each is assumed to require one time unit to transmit. The control packet requires $\underline{c}$ time units to transmit.

2.3 The SDAM Protocol for a Single Bus

There are two variants of the SDAM protocol: the first uses both end-nodes to pass the TIP back and forth and is called the closed SDAM, or C-SDAM; the second uses only one end-node and is referred to as the open-ended SDAM, or OE-SDAM.

Under both SDAM variants, each user-node can be represented as having four states (refer to Figure 1). Originally, all nodes are in the IDLE state. When a packet is generated at a node, say node n_i, the node becomes "busy" and enters the WAIT state. When an end-of-packet from node n_j is sensed on the channel and the token-direction on that packet is the same as the packet's traveling direction, the node enters the READY state. After a decoding/turnaround time $\underline{t}$ plus the cumulated carrier-sensing delays $(|n_i-n_j|-1)d$ along the path, the node is ready to send a message. But before this happens, it keeps monitoring the channel status; and if the channel becomes busy, the node goes back to WAIT state. Otherwise, the node enters the TRANSMIT state and stays there until either its buffer is emptied (for exhaustive transmission) or some transmission limit is reached (for non-exhaustive transmissions), and then it returns to the IDLE or WAIT state. Note that all packets transmitted carry the same token direction code as that of the most recently passed packet.

For the end-node of the C-SDAM protocol, the state diagram is the same as that of a user-node, except the end-node always has at least one packet (the TIP) to send when the token arrives, and the packet always carries an opposite token direction so as to send the token backwards.

For the end-node of OE-SDAM, a counter of $(2a + t + Nd)$ is used. After generating the first TIP, the end-node activates the counter and monitors the channel constantly. Whenever a packet is detected, the countdown is interrupted until the packet has passed, and a packet decoding/turnaround time $\underline{t}$ is added back to the counter. When this counter expires, or if the end-of-packet from the last node of the network is sensed from the channel, the end-node generates a new TIP and starts another round of token passing.

2.4 SDAM on a Branching-Bus Topology

Although it is always possible to use a single bus to connect any set of nodes scattered in a local area, a branching bus network is more desirable for its shorter propagation delay as well as its flexibility in regards to future expansion and reconfiguration (refer to Figure 2). SDAM can easily be generalized to support this topology. Since any complex branching topology can be decomposed into simple three-branch structures, as shown in Figure 3, it is sufficient to show that the algorithm of SDAM works on such a three-branch network. The reader can easily see that the same principle applies to networks with any finite number of branches.

Because there are three end-nodes E1, E2, and E3 on the three branches for the C-SDAM protocol, we need to change the token

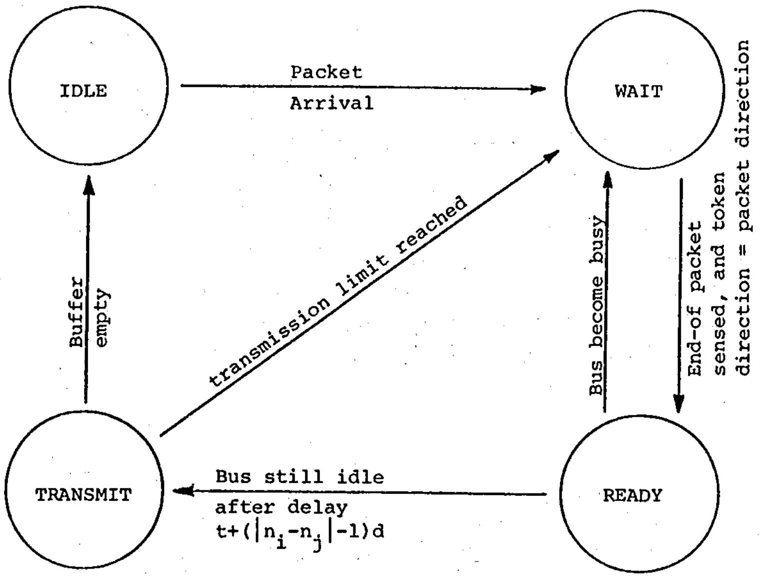

IDLE
Packet Arrival
WAIT
transmission limit reached
Buffer empty
Bus become busy
End-of packet sensed, and token direction = packet direction
TRANSMIT
Bus still idle after delay $t+(|n_i-n_j|-1)d$
READY

Figure 1. The State-diagram for the User-nodes of the SDAM Protocols.
t = decoding/turnaround time for the passing packet
d = carrier sensing time
n_i = the node wishing to transmit
n_j = the node that transmitted last

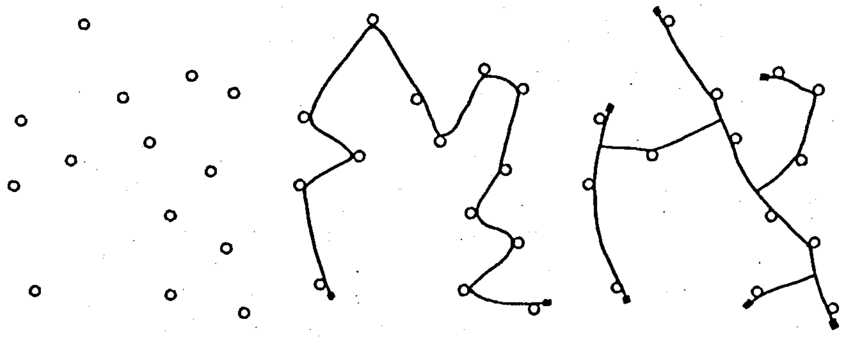


Figure 2.

(a) A set of nodes in in local area.
(b) Using a single bus to connect the nodes.
(c) Using a branching bus to connect the nodes.

direction code from left or right to: (E1-->E2), (E2-->E3), or (E3-->1). Each user-node will be on precisely two of such paths (refer to Figure 3a), and its access scheme remains unchanged, except for the new token directions. After receiving the network token, an end-node will now generate a TIP carrying a token direction that points toward the next end-node in sequence.

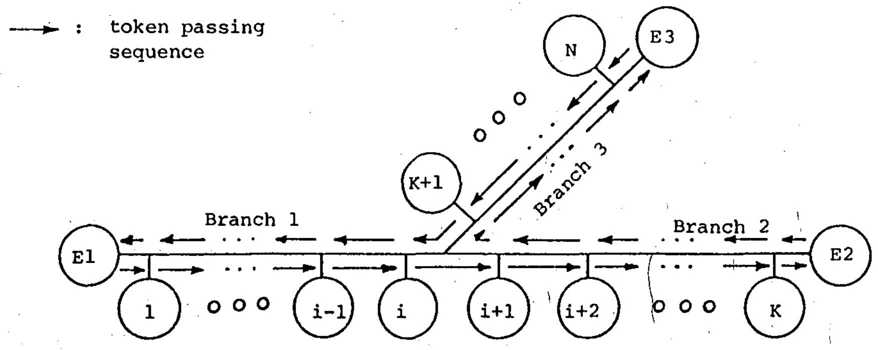
For the OE-SDAM protocol, two token directions are possible: one is on branches 1 and 2 and the other is on branch 3. Let us assume that the K nodes on branches 1 and 2 are sequentially numbered from 1 to K, and the remaining (N-K) nodes on branch 3 are sequentially numbered from K+1 to N. Then the first K nodes on branches 1 and 2 are treated as if they are on a single bus network. For each node n_i on the third branch, however, a counter of $((n_i-1)d + t + 2a_2)$, where a_2 is the propagation delay on branch 2, is used to determine when the token will arrive at node n_i. As soon as the node n_i detects the end of a TIP, it activates the counter and monitors the channel status. Whenever a packet from branch 1 or 2 is heard, the countdown is temporarily interrupted, and a decoding/turnaround time $\underline{t}$ is added back to the counter. If instead, a packet from a node on branch 3 is sensed, then n_i switches back to the single-bus scheme described earlier.

To illustrate how the requests from different nodes are coordinated, let us assume first that the network has been up and running for awhile. Now, suppose node i received the token at time x_o and had just finished a packet transmission at time x_i, with the token direction pointing toward branch 2 from branch 1 (refer to Figures 3a and 3b). After a propagation delay, the end-of-packet signal reaches each of the following nodes: i-1, i+1, and K+1. Nodes i-1 and K+1, noticing that the packet's traveling directions (toward branches 1 and 3, respectively) is different from the token direction (toward branch 2), will refrain from any transmission attempts. On the other hand, node i+1 will be able to go through the READY state to the TRANSMIT state and send its packet onto the channel. Nodes i+2, i+3,..., K, although sensing the same information as node i+1, will stop at the READY state and re-enter the WAIT state because node i+1 has already jumped ahead of them and occupied the channel. When node i+1 finally completes its transmission, the above procedure is repeated again.

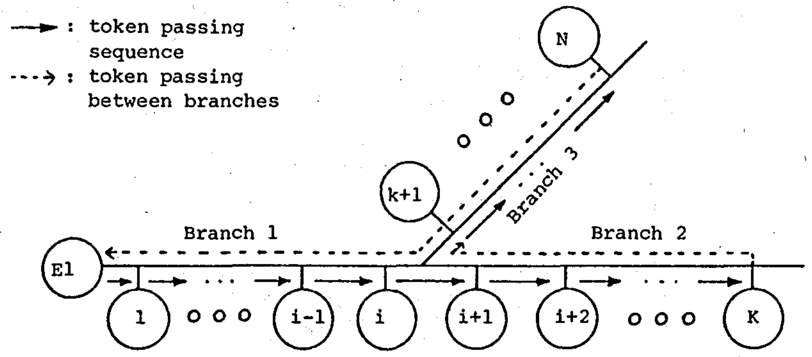
Continuing this process, the token will eventually reach the end of branch 2. For the C-SDAM protocol, the end-node E2 simply generates a TIP with a new token direction (E2-->E3) and sends the token toward E3 of branch 3 (refer to Figure 3a). The OE-SDAM protocol requires node K+1 on the third branch to wait for its counter to expire; then it claims the token and passes it to branch 3 (refer to Figure 3b).

2.5 Network Reliability under SDAM Protocol

As we have described earlier (in Section 2.3), each node executes an identical algorithm independently according to the information (e.g., channel status, token direction, etc.) provided



(a) Token passing in C-SDAM. The token is passed from E1 of Branch 1 to E2 of Branch 2. Then E2 changes the token direction toward E3 instead of to E1. Finally, E3 passes it back to E1 and completes a cycle.



(b) Token passing in OE-SDAM. The end node, E1, passes the token toward the other end of Branch 2. The first node on Branch 3 waits a time-out period then starts the token passing down to Branch 3. After another time-out period, the left end node E1 recovers the token.

Figure 3. Token passing of SDAM in a branching bus.

through the common channel. Therefore, any single node-failure will not affect the network's operation. However, network failure could still occur if (1) the end node(s) fails to generate a control packet; (2) there is an error in transmission (e.g., a noise on the channel, causing a later node to start transmission prematurally); and (3) cable failure occurs, disabling a part of the network. For contention based schemes, there does not exist a network token. All transmission errors may be viewed as just another data collision and can be handled as such. But for a token-passing scheme (e.g., SDAM), some procedure must be employed to restore the proper network operation from network failures. For the case of OE-SDAM, the error recovery procedure stipulates that:

1. whenever a node detects an unrecognizable address or token direction, it abandons any transmission attempts until the next token arrives;
2. each node is preassigned a time-out value whose size varies with the distance between the user-node and the end node (i.e., the shorter the distance, the smaller the time-out value);
3. if the bus has been sensed idle by a node for a period of time longer than its time-out value, then the user-node may infer that all the nodes with smaller time-out values have failed; therefore, it will generate a (data or control) packet to start the token passing again.

With this procedure, any error in transmission will be handled by using rule no. 1, followed by the end node generating a new token. If the end node should fail, then the user next to this node will resume its duty after a time-out period. Any subsequent user-node failure can be handled by the same procedure. In this fashion, even when the cable is physically cut into several pieces, each piece of the cable can be used to form a network, provided the proper cable terminators are added.

For the C-SDAM protocol, if one of the end nodes should fail, the other end node can detect this after a prolonged time-out period; hence it can automatically switch to the one-end-node OE-SDAM scheme as previously described.

3. Analysis of SDAM Protocol

The performance of both C-SDAM and OE-SDAM for an ideal single bus case have been analyzed in a previous paper by Li and Hughes [16] using the principle of a polling system [15]. Assuming nodes are uniformly located on the bus and the arrivals are Poisson, the network's throughput-delay formula for OE-SDAM (exhaustive transmission) is derived as

$$E(\text{Delay}) = \frac{\delta^2 a}{2Nr} + \frac{S}{2(1-S)} + \frac{a}{2N}\left(1 - \frac{S}{N}\right)\left(1 + \frac{Nr}{1-S}\right) \qquad (1)$$

in terms of packet transmission time, where

N = total number of nodes on the network
$\underline{a}$ = end-to-end bus propagation delay
$\underline{S}$ = network throughput in equilibrium (i.e., packet arrival rate X packet transmission time < 1)

and

$\frac{a}{N}$ = the change-over time between two adjacent nodes
$\underline{r}$ = average token passing time between two nodes (in units of $\frac{a}{N}$)
δ^2 = variance of token passing time between two nodes.

For a single-bus network, $\underline{r}$ and δ^2 can be approximated as follows. Excluding packet transmission times and their associated turnaround times, it takes the token (a+(N-1)d) time to reach the last node of the bus from the starting end-node, then another (a+d) time to travel back to the end-node. Therefore, the average token passing time is

$$r = \frac{1}{N}\ \{(c+t) + [a+(N-1)d] + (a+d)\}/(\tfrac{a}{N})$$

$$= 2 + (c+t+Nd)/a \qquad (2)$$

in terms of (a/N), where (c+t) is the network overhead associated with the initial TIP. The variance, δ^2, can then be determined as

$$\delta^2 = \frac{1}{N}\{(N-1)[1-r]^2 + 1\cdot[(c+t+a+d+\tfrac{a}{N})/(\tfrac{a}{N})-r]^2\} \qquad (3)$$

where the second squared term represents the token-passing delay between user-node N and user-node 1.

For the generalized branching bus as shown in Figure 3, we let

$$a = \sum_{i=1}^{3} a_i$$

where a_i is the bus propagation delay on branch i, i=1,2,3. Then equations (1) and (2) still hold true, but the variance of the token passing time must be modified to:

$$\delta^2 = \frac{1}{N}\{(N-2)[1-r]^2 + 1\cdot[(a_2+d+\tfrac{a}{N})/(\tfrac{a}{N})-r]^2$$

$$+ 1\cdot[(c+t+a_3+a_1+d+\tfrac{a}{N})/(\tfrac{a}{N})-r]^2\} \qquad (4)$$

where the second squared term is the token passing delay between the last node of branch 2 and the first node of branch 3; and the last squared term is again the token passing delay between user-nodes N and 1.

For simplicity and without loss of generality, we will still assume a single-bus network for our analyses in this paper. Due to difficulties in analytic modeling, the C-SDAM protocol will be analyzed by simulation methods.

In the remainder of this section, we will compare the performances of C-SDAM and OE-SDAM in detail. Also, the performance degradation due to various operating overheads of OE-SDAM is analyzed. The exhaustive transmission discipline will be assumed for a l our analyses unless otherwise specified.

3.1 Comparison of C-SDAM and OE-SDAM

The following parameters are used for comparisons of the delay performance of C-SDAM and OE-SDAM.

N = 50 nodes
packet size = 1000 bits (packet time = 1)
$\underline{a}$ = 0.01, 0.1, 0.5, 1.0 for propagation delay
$\underline{c}$ = 0.03 (30 bits) for TIP
$\underline{t}$ = 0.02 (20 bit-time) for turnaround time
$\underline{d}$ = 0.002 (2 bit-time) for carrier-sensing time.

When $\underline{a}$ is small (a = 0.01), the difference in the token passing time between these two protocols is also small. Therefore their performances are very close to each other (see Figure 4). As $\underline{a}$ increases to 0.1, the performance of C-SDAM begins to exceed that of OE-SDAM. When a=0.5, the average delay of OE-SDAM is 12% larger than that of CSDAM. This difference expands to 22% as $\underline{a}$ increases to 1.0.

The C-SDAM protocol, however, has one serious performance drawback. That is, it tends to discriminate against nodes located near either end of the bus (see Figure 5). The same phenomena has been observed and explained by Coffman et al. in their analysis of the disk access schemes [7]. Therefore, although the delay performance of CSDAM could be viewed as the "delay lower-bound" of all token-passing schemes, it is not suited for implementation unless such discriminations can be justified for some practical applications.

3.2 Comparison of OE-SDAM and Other Popular Schemes

The throughput-delay curve is compared to those of MSAP/BRAM [4, 12] and CSMA/CD with various $\underline{\nu}$ values ($\underline{\nu}$ is the probability of a busy node to attempt its transmission in a given time slot) [19] in Figure 6. It shows that, with a small bus propagation delay (a=0.01), in light loads SDAM performs very close to the CSMA/CD protocol with an optimal $\underline{\nu}$ value, and in higher loads SDAM exceeds CSMA/CD for all values of $\underline{\nu}$. For a larger propagation delay (e.g., a=0.1), the performance of CSMA/CD degrades drastically, while SDAM maintains a very good performance. In any circumstances, the performance of SDAM far exceeds other collision-free protocols reported in the literature [4,12,20].

The throughput $\underline{S}$ versus the offered traffic load $\underline{G}$ (defined as the total average number of packets available for transmission in the network) is plotted in Figure 7. Again, it shows that SDAM is highly efficient throughout the entire spectrum of G and remains stable (i.e., throughput does not degrade due to SDAM's collision-free property) for high values of G.

3.3 Effect of Decoding/Turnaround Time

Since a decoding/turnaround time t is associated with each transmission of a packet, we can envision the packet as being enlarged by a ratio of t (i.e., packet time t´=1+t), with the added portion being blank-filled. The throughput-delay curve can then be approximated using equation (1), with S modified to S´=S(1+t). The results are plotted in Figure 8 for t = 0.0 to 0.1 (0 to 100 bit-times). This figure shows that, for light loads the values of t do not significantly affect the network's performance. However, for high loads, larger t values have devastating effects on the average packet delay as well as the network's maximum achievable throughput. Generally, the maximum network throughput is bounded above by (1-t). So, if the turnaround time is t=0.1, then the network can only achieve an average of 90% throughput.

3.4 Effect of Carrier-Sensing Time

In an ideal case (as most people assume for their protocols), the time is negligible for each CIU to detect the absence (or presence) of a carrier and generate its own packet. Under such an assumption, SDAM's performance is independent of the number of nodes on the network [16]. However, this is an unrealistic assumption. Since all nodes on the network take turns to sense and access the channel, each node must allow its predecessors enough time to complete their actions before it can safely start its own. So, the time spent in carrier-sensing by each node will cumulate as the token is passed from node to node. Figure 9 shows the degradation of network performance under various values of d and N. It is clear that in a heavily populated network (N>100), even a subtle change in carrier-sensing time will have a profound effect on the network's performance.

3.5 Effect of the Token-Initializing Packet (TIP)

In SDAM protocol, a TIP is required to initialize each cycle of t ken-passing. This packet can be a special bit pattern that all nodes recognize as being generated by a particular end-node; or it can be a shortened data packet that contains nothing but a source address and a token direction code. In any case, this packet will occupy a fraction of the channel time, and therefore must be considered as a network overhead. Analysis shows, however, that when the number nodes on the network is large (n>50), the size of TIP has little effect on the network's performance. The effect of varying TIP sizes under an extreme light load ($s \simeq 0.0$) is plotted in Figure 10. In higher loads, this effect becomes negligible.

3.6 Effect of Exhaustive/Nonexhaustive Transmissions

In analyzing the exhaustive/nonexhaustive transmission disciplines, it is important to specify the type of workload imposed on the network. Here, we are mainly concerned about workloads with uniform Poisson arrivals; no attempt is made to study the situation where unbalanced loads are presented.

The exhaustive transmission discipline allows a node, upon receiving the network token, to transmit all the packets in its buffer, including the ones that arrived during the transmission process. The nonexhaustive discipline, on the other hand, limits the number of packets that a node may send at one time. In practice this limit may vary from node to node, but here our focus is directed on the case where only one packet is allowed to be transmitted per channel access.

Generally speaking, the exhaustive discipline provides a better average delay and a higher throughput of the network. In extreme cases, a busy node with a large file to transfer may monopolize the entire channel for a long period of time, causing networks throughput to temporarily reach 1, while making other nodes suffer long waiting times. The nonexhaustive scheme, on the other hand, guarantees fairness among the users and eliminates the above monopoly at the cost of increased token-passing time (hence the increase of average delay). However, in light loads ($S<0.5$) where the average number of waiting packets at each node is much less than 1, these two schemes are practically the same. Also, if the bus delay is small ($a<0.1$), then the performance of nonexhaustive discipline remains close to that of the exhaustive one (refer to Figure 11). When the bus delay is large, the difference in performance becomes significant (at $a=1.0$, $S=0.8$, the nonexhaustive scheme is 20% worse; at $S=0.9$, it is 66% worse).

In terms of the queueing delay distribution, the nonexhaustive discipline has a larger variance than its exhaustive counterpart. This is again due to the fact that the efficiency in consecutive transmissions is compromised by the requirement of fairness. Therefore, the distribution curve is "flatter" and the delay values are spread wider. Figure 12 shows the queueing delay distribution of both exhaustive and nonexhaustive transmission disciplines at $a=0.1$ and $a=1.0$. Table 1 summarizes the mean, standard deviation, median, and the 95 percentile of each of these distributions.

4. Summary and Conclusion

In this paper, we analyzed the generalized "shortest-delay access method" (SDAM) protocol previously proposed for a single-bus network [16]. The generalized network configuration now includes a branching-bus topology and three protocol overheads (the decoding/turnaround time, the carrier-sensing time, and the token-initializing packet (TIP) time), which makes SDAM more powerful and more practical.

It is shown that the throughput-delay formula for SDAM on a branching-bus is identical to that of a single bus, except that the total bus delay a is shorter and the variance of the token-passing time may be larger. Therefore, it was possible to reduce a complex branching-bus to an equivalent single bus configuration so as to simplify our analysis.

Two variants of the SDAM protocol--the C-SDAM and OE-SDAM--are analyzed. The C-SDAM, using back-and-forth token passing, is the most efficient of all the token passing schemes. The OE-SDAM, on the other hand, employs one-way token passing and provides uniform queueing delays to all nodes. When the network propagation delay a is small and the number of users N is not very large, these two schemes perform closely to the M/D/1 with perfect scheduling. But their performances are most impressive when a and/or N are large (e.g., a=1.0 and N=1000).

Of the three overheads evaluated in this paper, the TIP time is unique to the SDAM protocol. However, analysis shows that the size of the TIP has little effect on network's performance. The turnaround time t, which we think should be considered in every access protocol, shows a devastating effect on network performance. Given that the packet size is fixed and that the workload is balanced, the maximum throughput of the network is bounded above by (1-t), and the average queueing delay of the packets approaches infinity as the network's throughput approaches this limit. The carrier-sensing time d, although very small, becomes an important factor in network's performance when the number of users is large. For N>100, even a subtle change in d will have a profound effect on network's queueing delays.

The exhaustive transmission discipline provides the most efficient use of the channel time, while permitting a busy node to monopolize the channel access. The nonexhaustive discipline, on the other hand, eliminates such a monopoly and thus guarantees an upper bound to a node's waiting time. Analysis shows that the latter scheme causes larger mean and variance in queueing delay. But in low loads or with a small bus delay, these two schemes are practically the same. Therefore, the nonexhaustive discipline is recommended. However, in high loads with a large propagation delay, the difference in performance could become significant, and some form of tradeoff must be made.

In summary, the SDAM protocol provides a very efficient access method for a local network with a branching bus topology and is worth considering for implementation, particularly to large-scale networks.

REFERENCES

1. A. Abramson, "The ALOHA System--Another Alternative for Computer Communications," AFIPS 1970 Fall Joint Computer Conference, pp. 281-285.
2. I. Chlamtac, "Issues in Design and Measurement of Local Area Networks," Proc. CMG-XI International Conference on Computer Performance Evaluation, Dec. 1980, pp. 32-34.
3. I. Chlamtac, W. R. Franta, P. C. Patton, and W. Wells, "Performance Issues in Local Computer Networks," Technical Report 79-16, Computer Science Department, U. of Minnesota.
4. I. Chlamtac, W. R. Franta, and K. D. Levin, "BRAM: The Broadcast Recognizing Access Method," IEEE Trans. Comm., Vol. COM-27, No. 8, Aug. 1979, pp. 1183-1190.
5. D. D. Clark, K. T. Pogran, and D. P. Reed, "An Introduction to Local Area Networks," Proc. IEEE, Vol. 66, No. 11, Nov. 1978, pp. 1497-1517.
6. E. G. Coffman, Jr., and P. J. Denning, "Operating Systems Theory," Prentice-Hall, Englewood Cliffs, N.J., 1973, Chap. 5.
7. E. G. Coffman, Jr., L. A. Klimko, and B. Ryan, "Analysis of Scanning Policies for Reducing Disk Seek Times," SIAM J. Comput., Vol. 1, No. 3, Sept. 1972.
8. P. J. Denning, "Effects of Scheduling on File Memory Operations," AFIPS 1967 Spring Joint Computer Conf., pp. 9-21.
9. H. A. Freeman, "Tutorial Notes: Introduction to Local Computer Networks," 5th Conference on Local Computer Networks, Minneapolis, Oct. 6-7, 1980.
10. H. A. Freeman and K. J. Thurber, "Issues in Local Computer Networks," IEEE 1979 International Communications, pp. 20.3.1-20.3.5.
11. L. Kleinrock, "Queueing Systems," Vol. 2, Computer Applications, Wiley-Interscience, 1976.
12. L. Kleinrock and M. Scholl, "Packet Switching in Radio Channels: New Conflict-free Multiple Accesss Schemes for a Small Number of Data Users," Proc. ICC., June 1977, pp. 22.1-105-22.1-111.
13. L. Kleinrock and F. A. Tobagi, "Packet Switching in Radio Channels: Part I--Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics," IEEE Trans. Comm., Vol. COM-23, No. 12, Dec. 1975, pp. 1400-1416.
14. L. Kleinrock and F. A. Tobagi, "Packet Switching in Radio Channels: Part III--Polling and (Dynamic) Split-Channel Reservation Multiple Access," IEEE Trans. Comm., Vol. COM-24, No. 8, Aug. 1976, pp. 832-845.
15. A. G. Konheim and B. Meister, "Waiting Lines and Times in a System with Polling," JACM, Vol. 21, No. 3, July 1974, pp. 470-490.
16. L. Li and H. D. Hughes, "Definition and Analysis of a New Protocol," Proc. 6th Conference on Local Computer Networks, Minneapolis, October 12-14, 1981.
17. T. T. Liu, L. Li, and W. R. Franta, "The Analysis of a Conflict-Free Protocol Based on Node Clusters," Proc. 6th

Conference on Local Computer Networks, Minneapolis, Oct. 12-14, 1981.

18. R. M. Metcalf and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Comm. ACM, Vol. 19, No. 7, pp. 395-403, 1976.

19. F. A. Tobagi and V. B. Hunt, "Performance Analysis of Carrier Sense Multiple Access with Collision Detection," Proc. LACN Symp., May 1979, pp. 217-244.

20. C. Tropper, "Models of Local Computer Networks," Mitre Corp. Report ESD-TR-80-111.

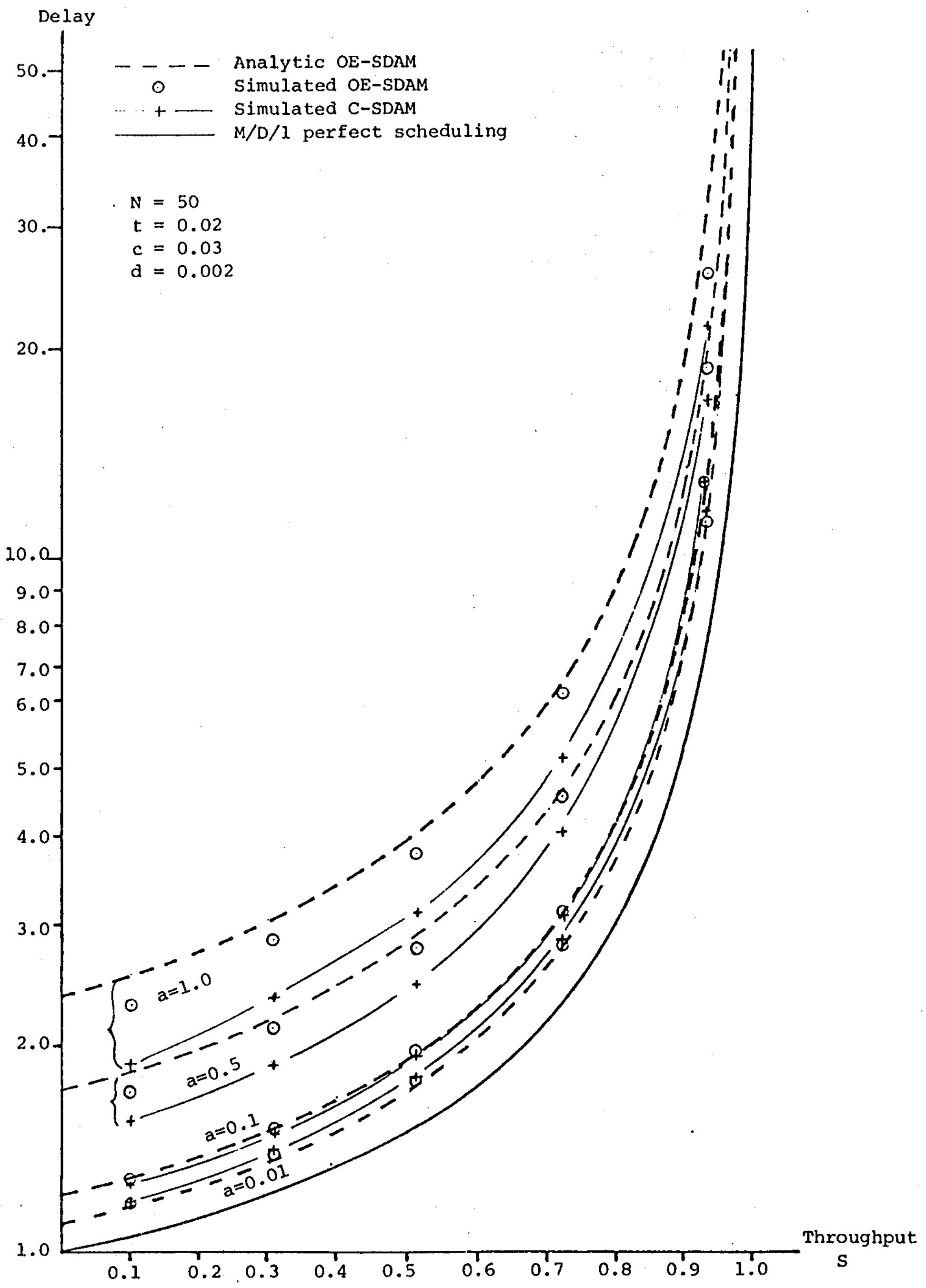


Figure 4. Delay performance of closed-SDAM and open-ended SDAM.

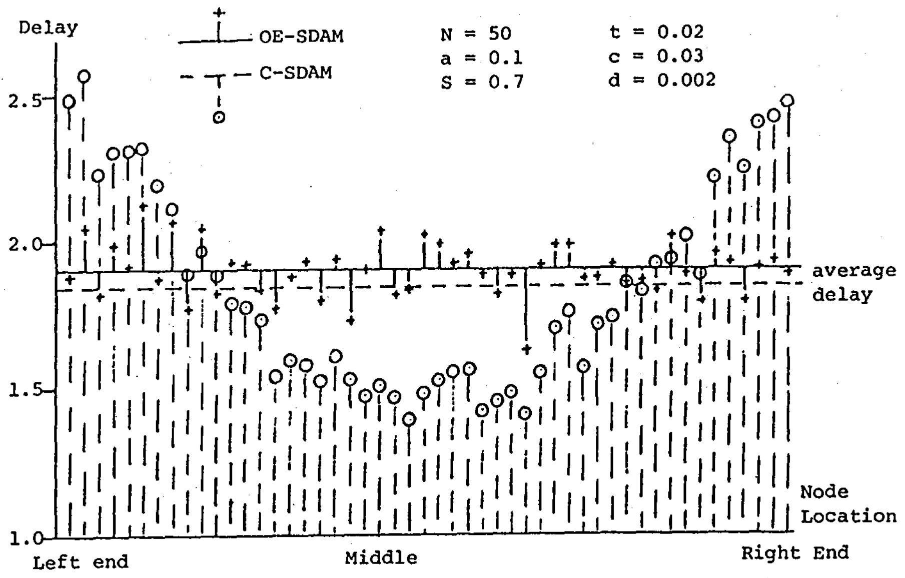


Figure 5. The packet delay versus node location on the bus.

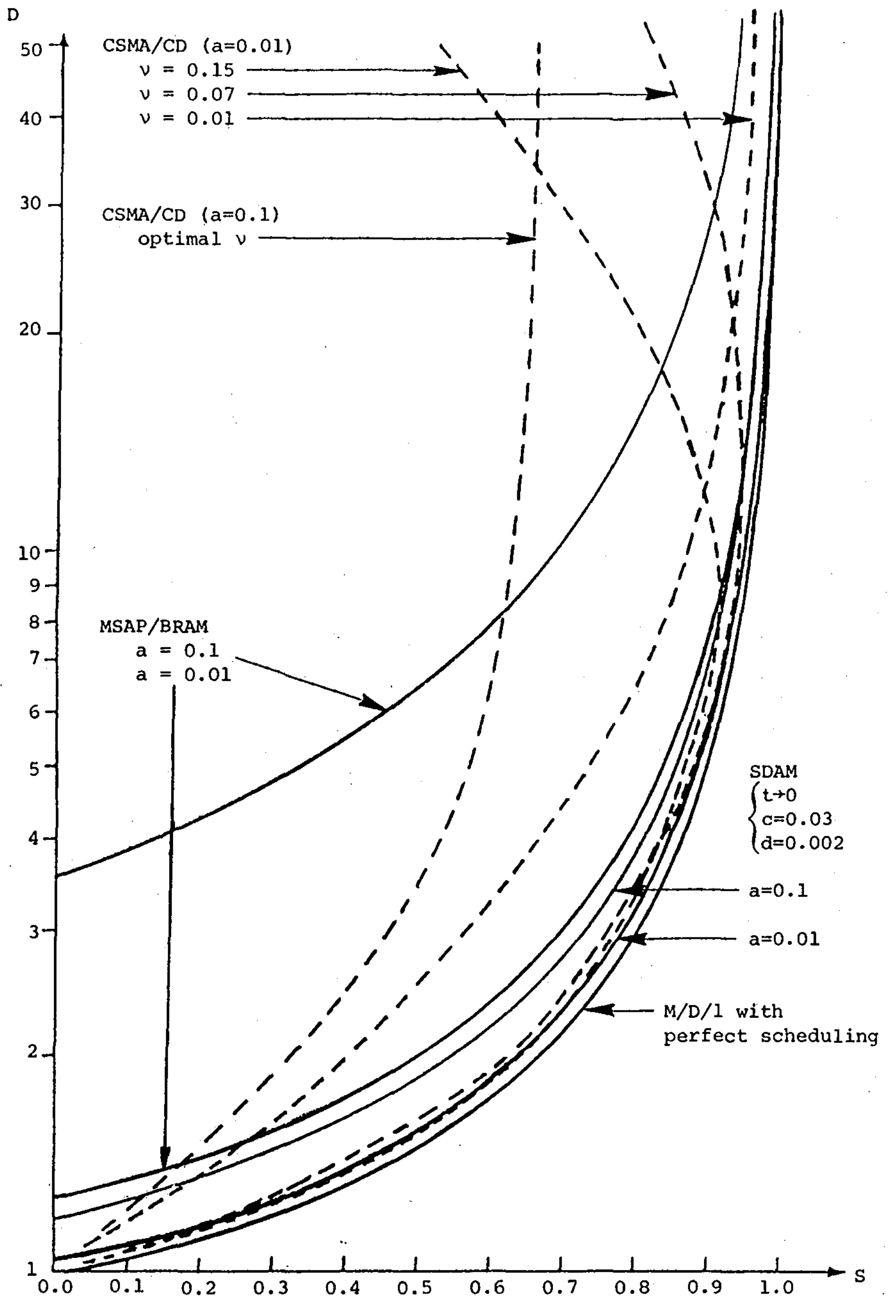


Figure 6. The throughput-delay curve of SDAM compared to MSAP/BRAM, CSMA/CO, and M/D/1. (N=50) (ν=transmission probability per time slot for CSMA/CD.)

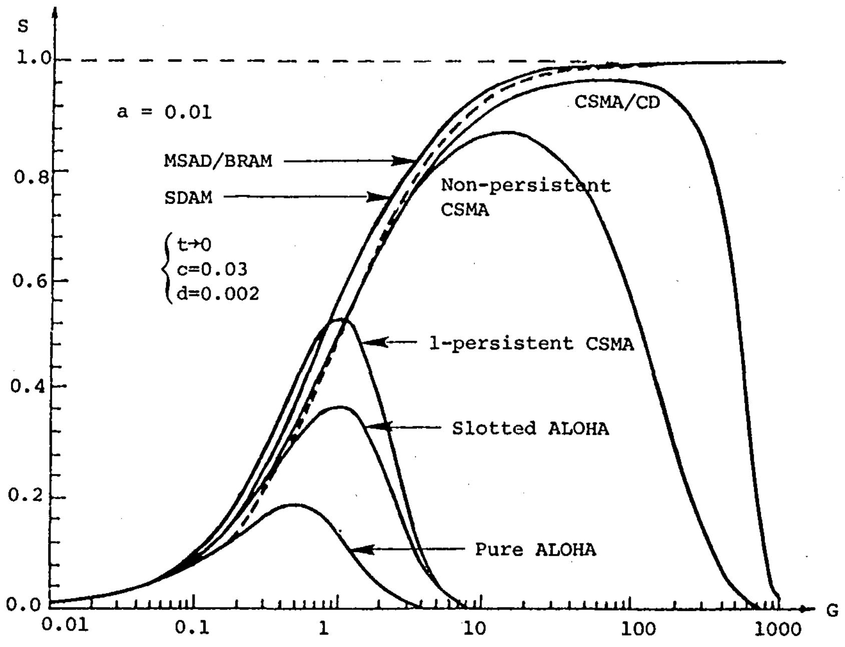


Figure 7. Offered load G vs. throughput S.
(Infinite population arrivals)

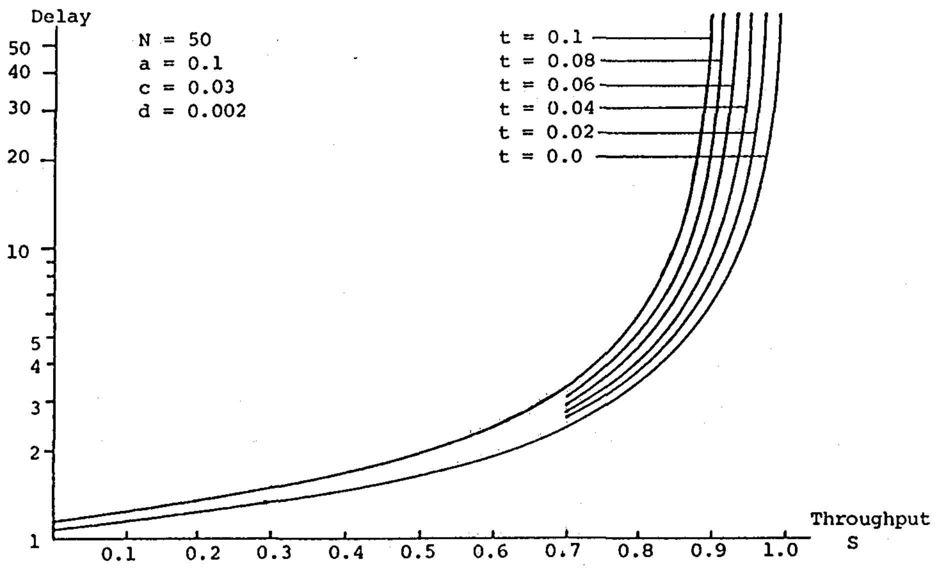


Figure 8. The effect of turnaround time on network's queueing delay.

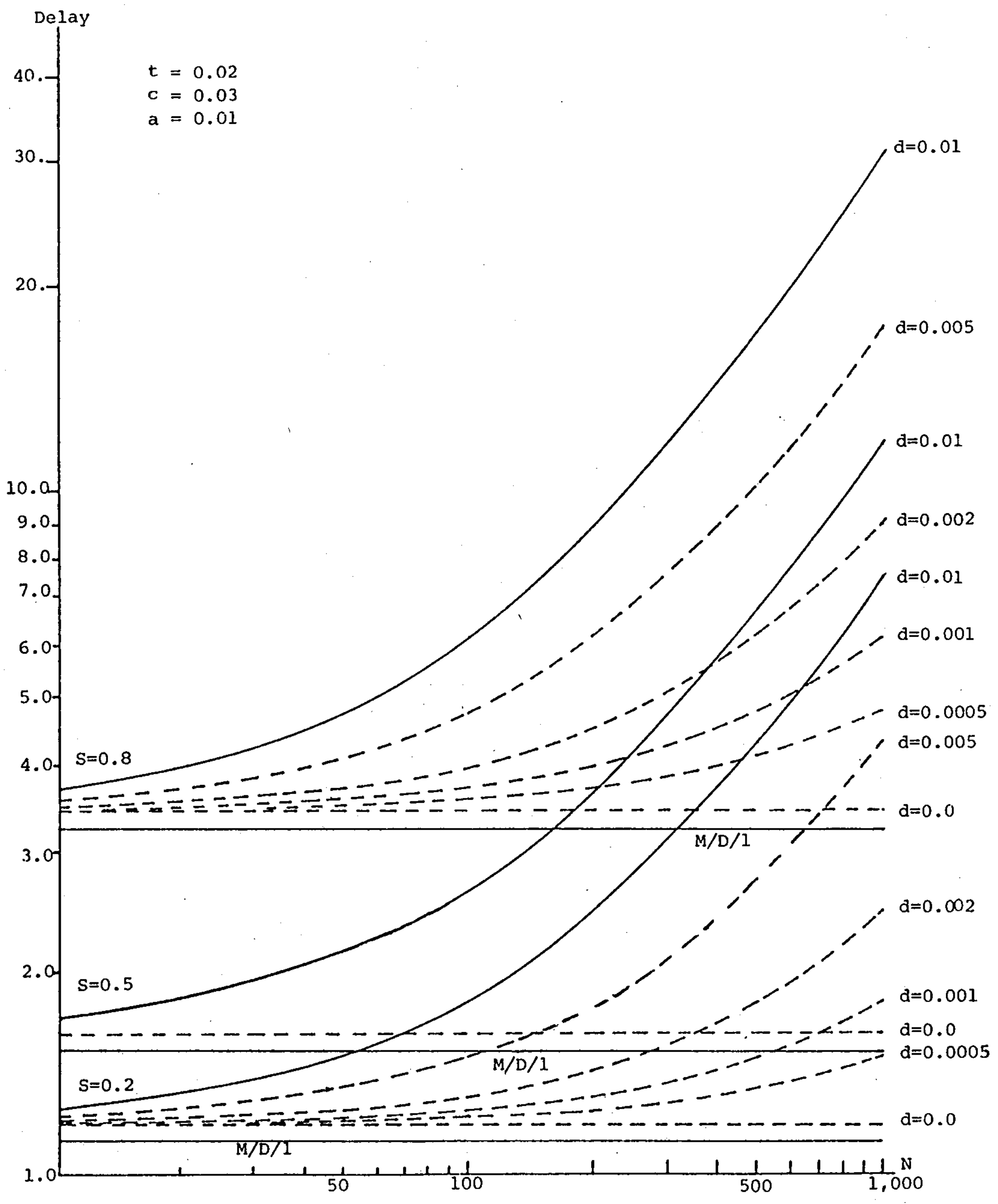


Figure 9. Effect of the carrier-sensing time d versus queueing delay.

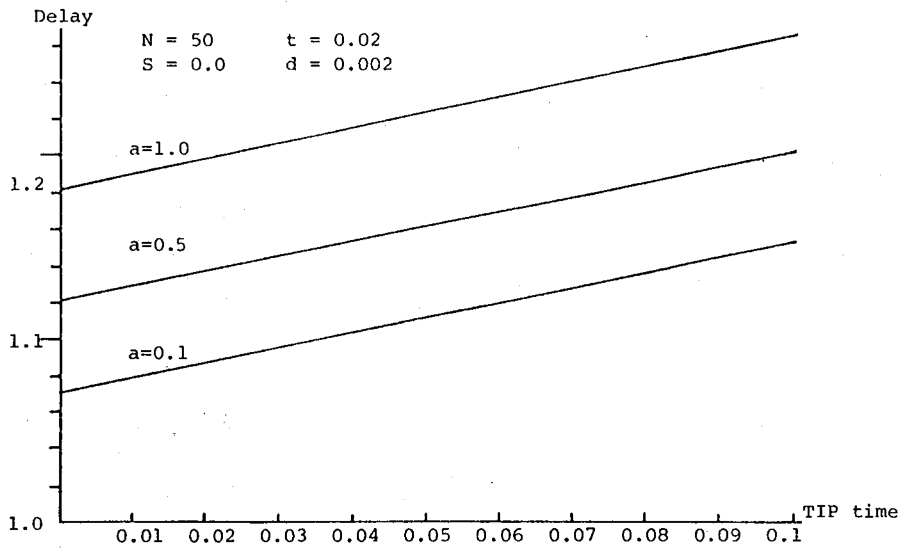


Figure 10. The token-initializing packet (TIP) time versus network's queueing delay.

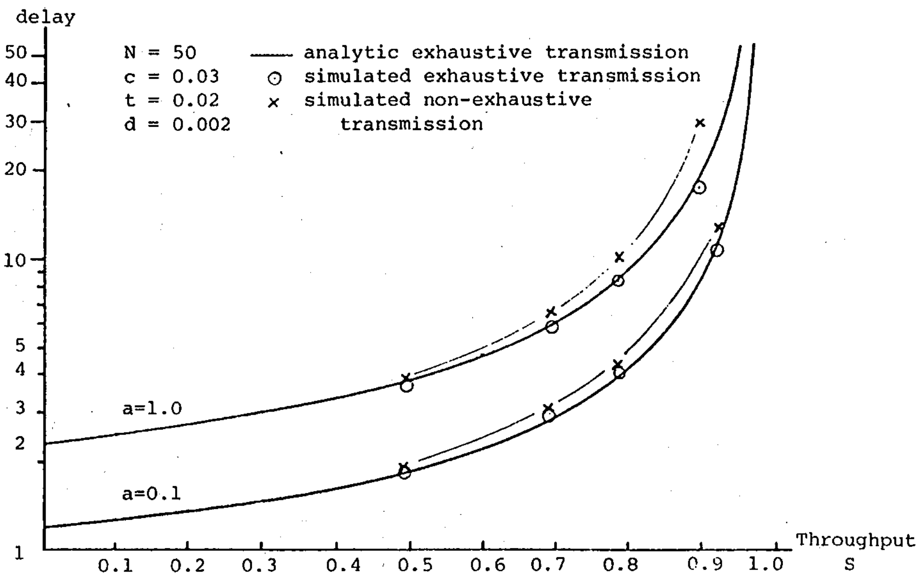


Figure 11. Differences in the queueing delays of the exhaustive and the non-exhaustive transmission disciplines.

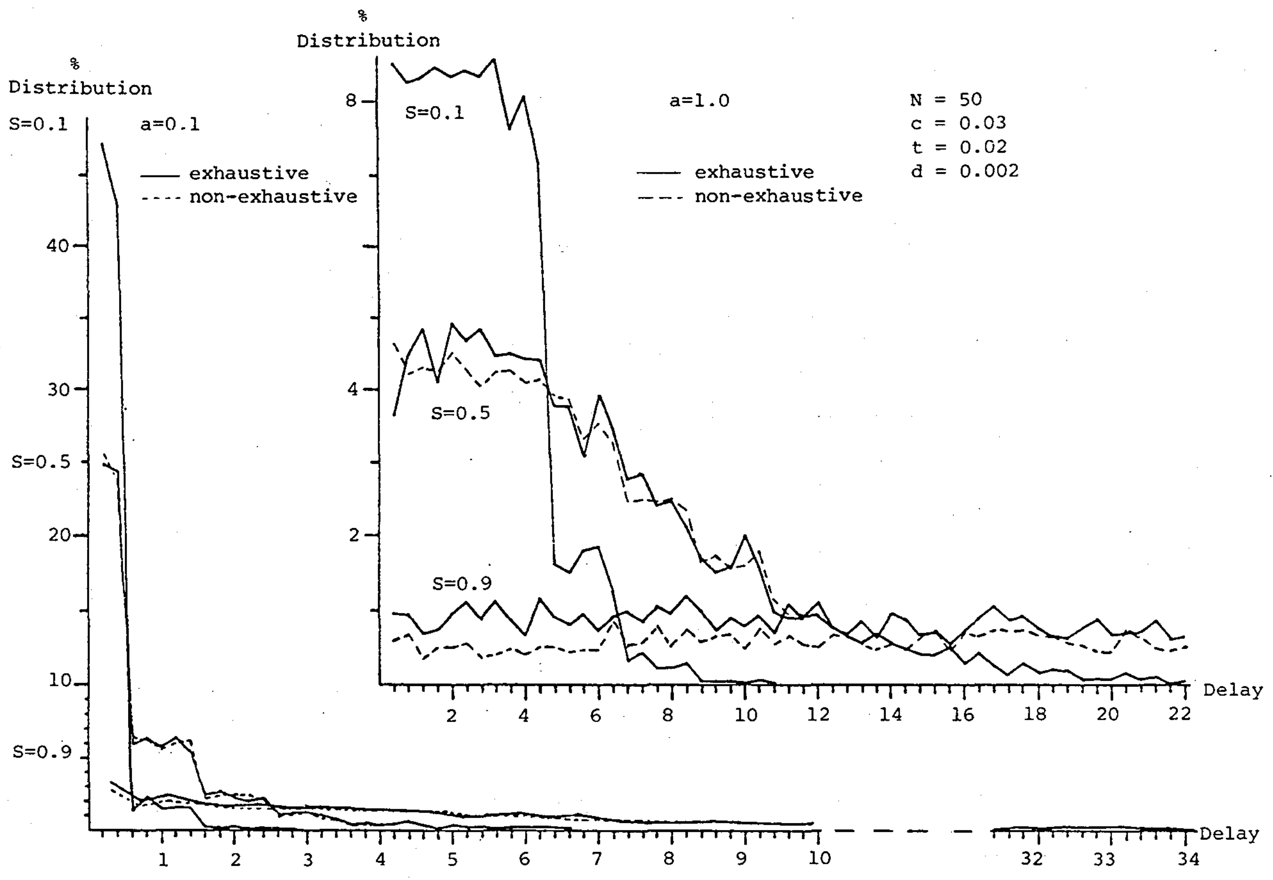


Figure 12. Distributions of queueing delays (excluding transmission time) for the exhaustive and the non-exhaustive transmission disciplines of the OE-SDAM protocol (simulated results).

		Propagation Delay a = 0.1					Propagation Delay a = 1.0				
Exhaustive	S	0.295	0.492	0.688	0.787	0.923	0.295	0.492	0.688	0.786	0.893
	Mean	1.496	1.916	2.932	4.208	10.766	2.769	3.738	5.873	8.397	17.681
	S.D.	0.627	1.123	2.344	3.996	10.987	1.300	2.183	4.287	6.534	13.122
	Median	0.3	0.4	1.2	1.8	6.0	1.5	2.3	3.4	5.6	12.2
	95%	1.8	3.2	6.8	11.2	33.6	4.2	7.0	13.4	20.4	**
Non-exhaustive	S	0.295	0.492	0.688	0.787	0.923	0.295	0.492	0.688	0.786	0.893
	Mean	---	1.927	2.977	4.354	12.850	---	3.878	6.503	10.070	29.407
	S.D.	---	1.160	2.522	4.587	15.618	---	2.391	5.292	9.229	31.736
	Median	---	0.4	1.2	1.9	6.8	---	2.3	4.2	6.6	18.4
	95%	---	3.4	6.8	11.2	41.6	---	7.6	15.0	25.8	**

--- : omit **: out of range

Table 1. Distributions of queueing delays (excluding transmission time) for the exhausted and the non-exhaustive transmission disciplines of the OE-SDAM protocol (simulated results).

THE BX.25 CERTIFICATION FACILITY

J. A. Melici
Bell Telephone Laboratories
Piscataway, N. J. 08854
(201) 981-2597

ABSTRACT

BX.25 is a data communications protocol which has been adopted as the standard protocol for host access to the Bell System's Operations System Network (OSN). BX.25 is based upon the Consultative Committee on International Telephone and Telegraph (CCITT) recommendation X.25. This paper describes a facility to automatically test that an implementation of BX.25 conforms to the BX.25 specification. The facility, called the BX.25 Certification Facility (BCF), consists of a data base of tests, software to execute the tests, and a microprocessor dedicated to handling BX.25 communication. The output of the facility is a report which provides information concerning the degree to which an implementation conforms to the BX.25 specification.

1. INTRODUCTION

BX.25 [1] is a data communications protocol which has been adopted as the standard protocol for host access to the Bell System's Operations System Network (OSN) [2]. BX.25 is based upon CCITT recommendation X.25 [3]. This paper describes a facility to automatically test that an implementation of BX.25 conforms to the BX.25 specification. The facility, called the BX.25 Certification Facility (BCF), consists of a data base of tests, software to execute the tests, and a microprocessor dedicated to handling BX.25 communication. The output of the facility is a report which provides information concerning the degree to which an implementation conforms to the BX.25 specification.

The remainder of this paper is divided into nine sections. Section 2 discusses certification in general, and defines some terms. Section 3 briefly describes BX.25. Section 4 describes the BX.25 Certification Facility. Section 5 presents a scenario for certifying BX.25 implementations. Section 6 discusses the generation of tests and reports. Section 7 suggests some enhancements to the facility. Section 8 discusses "proving in" the facility. Section 9 describes related work, and section 10 presents the conclusions of this paper.

2. GENERAL DISCUSSION OF PROTOCOL CERTIFICATION

Protocol certification involves executing a particular implementation of the protocol to test whether it complies with the protocol specification [4]. The tests are derived from the protocol specification. Only the external behavior of the protocol implementation is tested. How that behavior is achieved is unimportant for certification purposes. Certification of a protocol is not a formal proof of the correctness of the implementation [4]. Certification does provide a high degree of confidence that the protocol has been implemented correctly.

Certification of data communications protocols is different from verification and validation of protocols. Verification means demonstrating that the protocol definition is "logically" correct. Validation, a subset of the verification problem, means the protocol definition exhibits certain general properties, e.g., freedom from deadlock. Sunshine [5] defines validation and verification, as well as some techniques for protocol verification.

3. BRIEF DESCRIPTION OF BX.25

BX.25 is a layered data communications protocol [6] based upon CCITT recommendation X.25. X.25 defines the "interface between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) for terminals operating in the packet mode on public-data networks" [3]. Levels 1, 2, and 3 of BX.25 correspond to levels 1, 2, and 3 of X.25, level 1 being the physical layer, level 2 the link layer, and level 3 the packet layer. BX.25 is compatible with X.25 at these levels, but differences do exist. In addition, BX.25 defines an optional multi-link layer, and a session layer.

Level 1 of BX.25, the physical layer, specifies the physical and electrical interface between a DTE/DCE pair, or a DTE/DTE pair. Currently either the Electronics Industry Association (EIA) RS-232C interface, or CCITT Recommendation V.35 is utilized. This level provides a bit-serial, full-duplex, point-to-point, synchronous transmission path.

Level 2 of BX.25, the link layer, utilizes the Link Access Procedure B (LAPB) of X.25. This level defines procedures which provide an essentially error-free, transparent link between a DTE/DCE or a DTE/DTE. Some functions of this level are link initialization/disconnection, link level error control, and flow control.

Level 3 of BX.25, the packet layer, defines procedures for the interchange of packets. The major facilities of this level are Permanent Virtual Circuits (PVC) and Virtual Calls (VC). The major functions of this level are packet-level error control and flow control, multiplexing of packets over the single physical link, and packet level reset and restart procedures. For complete information about BX.25 and X.25 refer to [1], [3], and [7].

4. DESCRIPTION OF THE BX.25 CERTIFICATION FACILITY

4.1 An Abstract Model of a Protocol Certification Facility

This section presents an abstract model of a protocol certification facility. The model is applicable to any protocol, including BX.25, and higher layer protocols.

The following description refers to figure 1. The components of this model of a certification facility are:

- Parameterized test file - The tests necessary to certify that the protocol has been implemented correctly. The test file is parameterized, so the same tests apply to all protocol implementations, regardless of the values chosen for the protocol parameters. For example, the value of the T1 timer is a parameter of BX.25 level 2.
- Parameter file - A file containing the definitions of all the parameters of the protocol, e.g., for BX.25 level 2, the value of the T1 timer is specified.
- Test preprocessor - The parameterized test file, and the parameter file serve as input to the test preprocessor. The test preprocessor substitutes the *values* of the parameters in the parameter file, for the parameters in the parameterized test file. The output of the preprocessor is a test file tailored to the environment of the Implementation Under Test

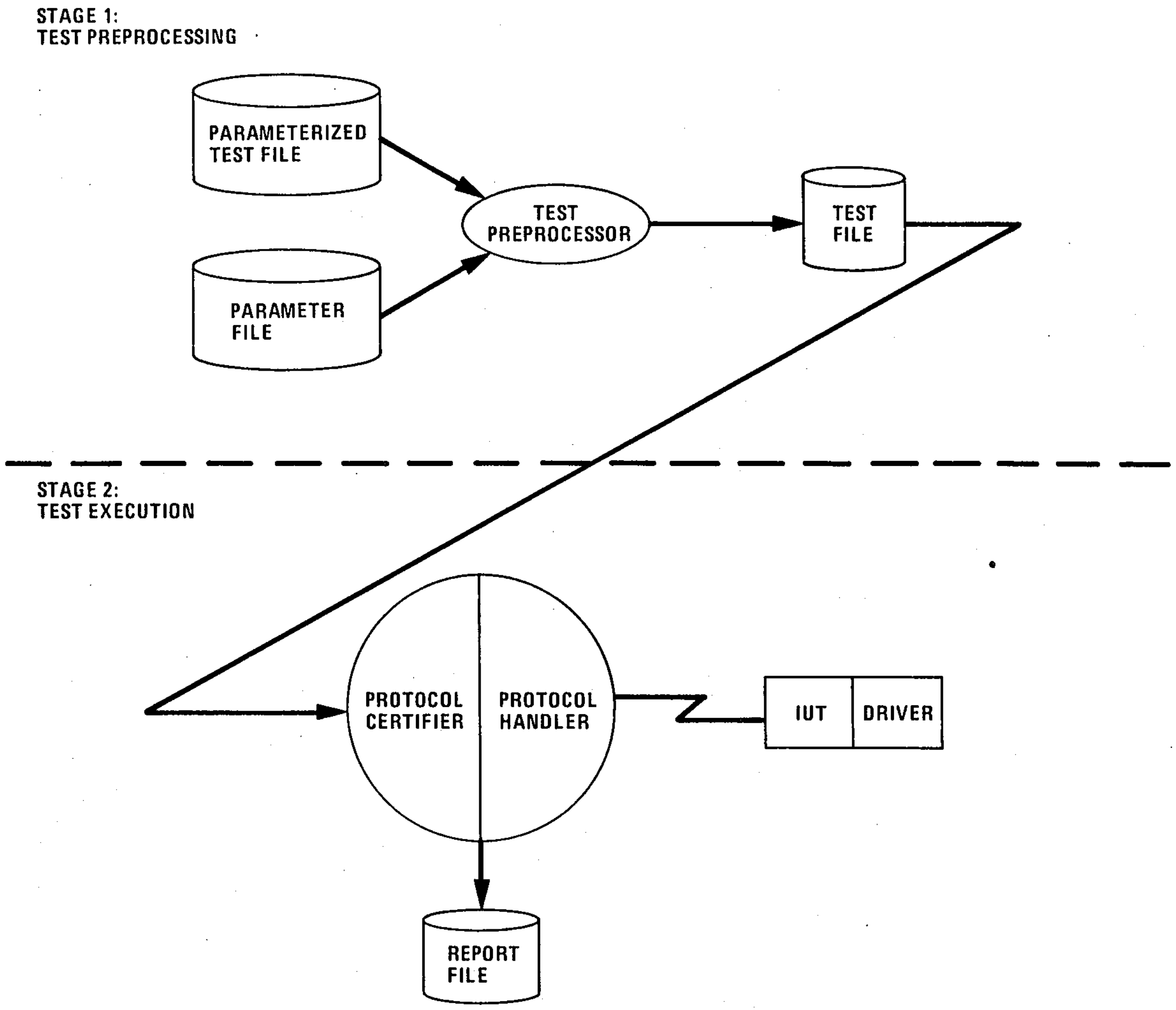


Figure 1

AN ABSTRACT MODEL OF A PROTOCOL CERTIFICATION FACILITY

(IUT).

- Protocol certifier - The test file serves as input to the protocol certifier. The commands in the test file direct the actions of the certifier. Based on these commands the certifier directs the protocol handler to send a particular protocol message to the IUT, e.g., for BX.25 level 2 a DISC (disconnect) frame. The response of the IUT is forwarded to the certifier, for analysis, by the protocol handler. The certifier automatically generates a report file.
- Report file - This file provides information concerning the degree to which the IUT conforms to the protocol specification.
- Protocol handler - Handles all the processing associated with communicating the protocol. The handler has two interfaces. The IUT is connected to one interface. It is over this connection that the protocol is communicated. The certifier is connected to the other interface. The certifier sends commands to the handler to force the transmission of a particular protocol message. The handler transmits only when directed to by the certifier. This allows for the purposeful violation of the protocol, to check that the IUT responds correctly. The handler sends to the certifier a trace of all transmitted and received protocol messages.
- Driver - To comprehensively test an implementation a driver is required. In most protocol implementations there are certain events which must be initiated from a level above the implementation. As an example, in BX.25 level 2, the transmission of an I frame is initiated from the level above the BX.25 level 2 implementation. The behavior of the IUT when these events occur should be tested as part of a certification procedure. In this model the driver acts as the next layer above the IUT. The driver sends control signals to the IUT, initiating events which otherwise would not occur.
- Communications equipment - The necessary equipment (e.g., modems, cables) to establish the physical transmission path to the IUT.

Protocol certification is a two stage process. Stage one, test preprocessing, transforms the parameterized test file into a test file tailored to the environment of the IUT. Stage 2, test execution, involves running the tests in the test file against the IUT. The output of this stage is the report file.

4.2 Implementation of the BX.25 Certification Facility

The components of the BX.25 Certification Facility correspond to the components of the model presented in the previous section. The facility is used to certify BX.25 level 2, or BX.25 level 3 implementations. The need to test BX.25 implementations developed as part of the Bell Administrative Network Communications System (BANCS) [8] [9], which is the internal switching network for the OSN, as well as BX.25 implementations which communicate on a point to point basis, provided the stimulus for the development of a BX.25 Certification Facility.

The following description refers to figure 2. The test preprocessor and the BX.25 certifier are two programs which execute in user space of a VAX-11/780 running the UNIX^{TM} [10] operating system.

The test file describes the expected behavior of the IUT when it is subjected to certain stimuli. The different tests in the test file attempt to fully exercise the protocol, checking both normal and error conditions. In particular the test file:

- Is divided into a number of separate and independent tests.
- Each test specifies when to transmit a particular frame, for BX.25 level 2 certification, or packet, for BX.25 level 3 certification. The type of frame/packet to transmit is specified mnemonically, e.g., disc means send a DISC frame. Moreover, any of the fields in a frame or packet may be specified.
- Each test specifies the expected response of the IUT upon receipt of a particular frame or packet.

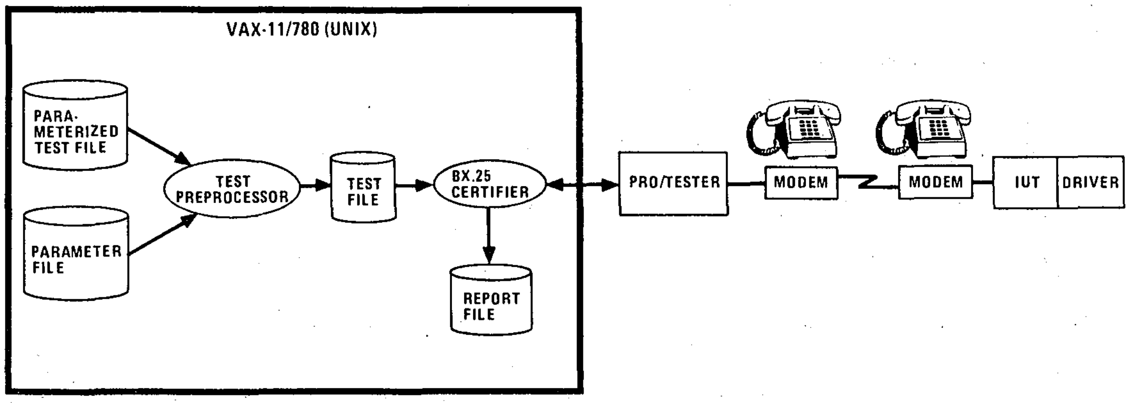


Figure 2

IMPLEMENTATION OF THE BX.25 CERTIFICATION FACILITY

The Test Input Language (TIL), used to write the tests, will be defined in section 6.

The PRO/TESTER[1] [11], commercially available from Applied Data Communications (ADC), is a microprocessor with software dedicated to X.25 communication. The facility utilizes the PRO/TESTER for handling the processing associated with BX.25 communication. The major features of the PRO/TESTER are:

- Simulation of a DTE or DCE.
- Tracing of all frames/packets received from the IUT.
- Communication at level 2 of X.25 (i.e., turn level 3 off), or level 3 of X.25.
- A mnemonic command language to specify the transmission of all X.25 frames/packets, with the capability to specify the fields in a frame/packet. Frames or packets can also be constructed from hexadecimal.
- Either level can operate in either the automatic or manual mode. In the automatic mode the PRO/TESTER adheres to the X.25 protocol, and accepts commands (over its asynchronous interface) directing its actions. In the manual mode the PRO/TESTER only transmits frames/packets when directed to do so via commands. The PRO/TESTER operates in the manual mode for the level being certified. This allows for the purposeful injection of errors, to test that the IUT responds correctly.
- The capability to generate Cyclic Redundancy Check (CRC) errors, and abort the transmission of a frame.

The PRO/TESTER has two interfaces, an asynchronous interface, and a synchronous interface. The asynchronous interface accepts commands directing the PRO/TESTER's actions. Moreover, the trace of received/transmitted frames/packets is output over this interface. Figures 3 and 4 show the trace information provided by the PRO/TESTER for level 2, and

1. PRO/TESTER is a trademark of Applied Data Communications.

LINK LAYER TRACE INFORMATION:

R/T	- Direction of transmission. "TRN" for transmitted, "REC" for received.
ADR	- The address (either A or B) of the frame.
FRAME	- The type of frame.
P/F	- The poll/final bit. "P" if the poll bit is set, "F" if the final bit is set.
NS	- The send sequence number of the frame.
NR	- The receive sequence number of the frame.
HEX	- The hexadecimal representation of the frame.

EXAMPLE:

R/T	ADR	FRAME	P/F	NS	NR	HEX
TRN	A	I-FRAME	P	05	03	037A4444

Figure 3
BX.25 Level 2 Trace Information Provided by the PRO/TESTER

PACKET LAYER TRACE INFORMATION:

RT	- Direction of transmission. "T" for transmitted, "R" for received.
TYPE	- The type of packet.
LCN	- The logical channel number of the packet.
Q	- The qualifier bit."Q" if this bit is set.
D	- The delivery confirmation bit. "D" if this bit is set.
M	- The more data bit."M" if this bit is set.
PS	- The send sequence number of the packet.
PR	- The receive sequence number of the packet.
HEX	- The hexadecimal representation of the packet.

EXAMPLE:

RT	TYPE	LCN	QDM	PS	PR	HEX
T	DATA	0001		4	3	10016844

Figure 4
BX.25 Level 3 Trace Information Provided by the PRO/TESTER

level 3 of X.25 respectively. In general, the trace information provided for both level 2 and level 3 is:

- The direction of transmission (either transmitted or received).
- The type of frame or packet.
- The contents of various fields of the frame or packet. Refer to [7] for a description of the meaning of each of the fields.
- The hexadecimal representation of the frame or packet.

The synchronous interface is used to communicate with the IUT.

The PRO/TESTER is only used to transmit and receive frames/packets. The certifier directs the PRO/TESTER as to which frame/packet to send, based on the test file. Even though the PRO/TESTER communicates **X.25**, the test file defines the expected behavior of a **BX.25** IUT.

A pair of dial-up Bell 208, or Bell 212 modems provides the means for establishing the physical connection between the PRO/TESTER and the IUT. Communication takes place at 4800 bits per second (bps), or 1200 bps. The facility provides only one of the modems. The other modem must be provided by the IUT.

The report file provides information concerning the degree to which the IUT conforms to the BX.25 specification. In particular the report file:

- Indicates which tests passed or failed. If a test fails, the reason the test failed is included in the report file.
- Provides a trace of all frames/packets exchanged between the facility and the IUT. The trace information provided is exactly the trace information of the PRO/TESTER.
- Optional time-stamping of frames/packets.
- Summarizes the total number of tests which passed or failed, and lists the number of failed tests, up to some limit.

The exact format of the report file is specified in section 6.

To comprehensively test an IUT a driver is required. The driver acts as the level above the BX.25 IUT. The level 2 and level 3 drivers interpret single-byte codes contained in I frames, or DATA packets, respectively. The code may specify "do nothing", cause the IUT to go into a particular state, or cause the IUT to send a particular frame/packet. The specifications of the driver, and a prototype driver, written in the C language [12], are provided as part of the BX.25 Certification Facility. Figures 5 and 6 contain the codes recognized by the level 2 and level 3 drivers, respectively. Without drivers, 52% of level 2 and 74% of level 3 can be tested. The states which cannot be tested without a driver represent "pathological cases," for example, the REJ Sent & Station Busy state, which do not occur very often. Thus, even though to *certify* an IUT requires a driver, a fairly comprehensive test can still be performed on an IUT without a driver.

5. SCENARIO OF OPERATIONS

This section presents a scenario for the certification of either level 2 or level 3 of BX.25. The same scenario applies to each level. The test file and driver required to certify level 2 is different, and independent, of the test file and driver for level 3. The scenario is:

1. The level (either 2 or 3) to be certified is determined.

2. The parameters of the BX.25 IUT are determined. The parameter file is modified appropriately.

3. The parameterized test file (either the level 2 or level 3 test file) is preprocessed to produce the test file to be input to the BX.25 certifier.

4. The appropriate driver (either the level 2 or level 3 driver), if available, is installed on top of the implementation to be tested.

5. Hardware communication is established via the dial-up Bell 208 modems or the dial-up Bell 212 modems.

Code	Action
0	Do nothing
1	Send a Local Start command to the IUT
2	Send a Local Stop command to the IUT
3	Create a Station Busy condition for N seconds*
4	Echo I frame back N times*
5	After waiting N seconds, create a Station Busy condition for M seconds*

* N and M are contained in the second and third bytes, respectively, of the I field of the I frame.

Figure 5
Codes Recognized by the BX.25 Level 2 Driver

Code	Action
0	Do nothing
1	Create a Station Busy condition on logical channel identifier (LCI) N, for M seconds*
2	Echo DATA packet over LCI N, M times*
3	Transmit an INTERRUPT packet over LCI N*
4	Transmit a CALL REQUEST packet

* N and M are contained in the second and third bytes, respectively, of the User Data field of the DATA packet.

Figure 6
Codes Recognized by the BX.25 Level 3 Driver

6. The BX.25 certifier is executed. The appropriate flags are set for frame/packet level testing and time-stamping of frames/packets.

7. The report file is delivered to the BX.25 implementor.

6. GENERATION OF TESTS AND REPORTS

6.1 Generation of Tests

The BX.25 state tables, provided as part of the BX.25 specification, provide the framework for systematically generating the tests. Tests are organized by the BX.25 states. All possible inputs, for a particular state, are applied to the IUT. The response of the IUT, as compared against an expected response, determines the success or failure of a particular test. The expected response is determined from the state table, augmented where necessary by the written description of the protocol.

Moreover, to assure the independence of tests, each test contains the proper sequence of commands to bring the IUT from any state into the state being tested. This ensures that the success or failure of a particular test does not affect any other test.

For level 2, approximately 1700 tests are required. For level 3, approximately 650 tests are required. Each test consists of approximately eight lines of Test Input Language commands.

6.2 The Test Input Language

The tests are written in the Test Input Language (TIL). There are five commands in the language. Only one command per line is allowed. The commands are:

- The TEST command marks the beginning of a test and the end of any previous test. The syntax is:

  TEST "description of test"

  The description is required.

- The expected-reply command specifies which frame or packet should be received next. The syntax is:

```
[frame/packet type,field1= value1,...,fieldn= valuen]
{OR [frame/packet type,field1= value1,...,fieldn= valuen]}* : nnnn
```

Each frame/packet type is enclosed in square brackets. The frame/packet type is expressed mnemonically, e.g., DISC represents a disconnect frame. The frame/packet type should be all upper-case letters. The value of a field is also expressed mnemonically, e.g., nr= 3 means the receive sequence number of the received frame should be three. This syntax supports alternation, specified by ORing individual frame/packets together. ({...}* is extended Backus-Naur Form (BNF) [13] for "zero or more occurrences of.") The numerical value after the colon specifies the number of seconds before which the frame/packet should be received. If this is omitted, a default, which can be specified, is in effect. Receiving a response other than one of the responses specified in the expected-reply command causes the failure of the current test. In this case the BX.25 certifier flushes to the next TEST command, or end-of-file, whichever comes first.

- The DEFTIMEOUT command modifies the default timeout value to use for an expected-reply command without a time value specified. Initially the default timeout is ten seconds. The syntax is:

```
DEFTIMEOUT nnnn
```

where nnnn is given in seconds.

- The comment command identifies everything that follows, up to the end of the line, as a comment. The syntax is:

```
! text of comment
```

A comment whose text begins with a "!" will appear in the report generated by the facility. Other comments are ignored by the BX.25 Certifier.

- PRO/TESTER commands represent a whole class of commands. These are sent to the PRO/TESTER, which executes them. PRO/TESTER commands are identified by being specified in lower case letters. Most of the PRO/TESTER commands appearing in a test file cause the transmission of a frame or packet. Any command is allowed though. Usually the only other PRO/TESTER command appearing in a test file is "dte", used to option the PRO/TESTER to act like a DTE. The default mode of the PRO/TESTER is DCE. Refer to Appendix 1 for a list of the PRO/TESTER commands.

Figure 7 is a portion of a test file used to certify BX.25 level 2 implementations. The numbering of the tests is done automatically by the test preprocessor. The numbers also appear in the report file, and serve to allow easy reference to failed tests.

6.3 Reports Issued by the BX.25 Certification Facility

The following description refers to figure 8, a report issued by the BX.25 Certification Facility, corresponding to the tests in figure 7. Reports for level 3 certification differ only in that packets, instead of frames are part of the trace.

Several points should be observed about this report:

- Tests are numbered for easy reference.
- The data before the trace information is the time stamp, in seconds, that the frame was received. Time is relative to the start of testing, beginning at time= zero.
- The trace information is exactly the trace information provided by the PRO/TESTER.
- A frame received during the transmission of a frame is marked "UNEXPECTED".
- A received frame different from an expected frame, in the expected-reply command, is marked "INCORRECT". The frame expected also appears in the report file. This fails the current test.
- At the end of the report, a summary of the success/failure of the tests is provided.

```
!!
!! The following are the level 2 parameter assignments:
!!
!! The address of the Implementation Under Test is B
!! The value of T1 is 10
!! The value of T2 is 25
!! The value of N1 is 256
!! The value of N2 is 5
!! The value of K is 2
!!

! Set the address of the facility
dte

! Set the default timeout to T1 = 10 seconds
DEFTIMEOUT 10

TEST     1 "Test receipt of a bad receive sequence number (n(r)) in the information transfer state (S5)"
disc,*
disc,*
[UA,addr=01,pf=1] OR [DM,addr=01,pf=1]
sabm,*
[UA,addr=01,pf=1]


rr,*,nr=05
[FRMR,addr=01,pf=1,ifield=B10008]

TEST     2 "Test receipt of a set asyncronous balanced mode (SABM) command in the disconnected state (S1)"
disc,*
disc,*
[UA,addr=01,pf=1] OR [DM,addr=01,pf=1]


sabm
[UA,addr=01,pf=1]
```

Figure 7

PORTION OF A TEST FILE USED TO CERTIFY BX.25 LEVEL 2

```
BX.25 CERTIFICATION FACILITY (RELEASE 1.0, 10/1/81)

FORMAT OF BX.25 LEVEL 2 TRACE INFORMATION:

TIME STAMP R/T ADR FRAME    P/F NS NR HEX

!!
!! The following are the level 2 parameter assignments:
!!
!! The address of the Implementation Under Test is B
!! The value of T1 is 10
!! The value of T2 is 25
!! The value of N1 is 256
!! The value of N2 is 5
!! The value of K is 2
!!


***TEST      1 "Test receipt of a bad receive sequence number (n(r)) in the information transfer state (S5)"
0000000017 TRN B   DISC     P          0153
0000000017 REC B   DM       F          011F**** UNEXPECTED ****
0000000018 TRN B   DISC     P          0153
0000000018 REC B   DM       F          011F
0000000019 TRN B   SABM     P          013F
0000000019 REC B   UA       F          0173
0000000019 REC A   RR       P       00 0311**** UNEXPECTED ****
0000000020 TRN B   RR       P       05 01B1
0000000020 REC B   FRMR     F          0197B10008
***TEST      1 PASSED


***TEST      2 "Test receipt of a set asyncronous balanced mode (SABM) command in the disconnected state (S1)"
0000000021 TRN B   DISC     P          0153
0000000021 REC B   DM       F          011F**** UNEXPECTED ****
0000000022 TRN B   DISC     P          0153
0000000022 REC B   DM       F          011F
0000000023 TRN B   SABM                012F
0000000023 REC A   RR       P       00 0311**** INCORRECT **** EXPECTED:    [UA,addr=01,pf=1]    ****
***TEST      2 FAILED


TOTAL NBR OF TESTS:              2
TOTAL NBR OF TESTS THAT PASSED: 1 ( 50%)
TOTAL NBR OF TESTS THAT FAILED: 1 ( 50%)

TESTS WHICH FAILED (UP TO 50):
TEST 2
```

Figure 8

A REPORT ISSUED BY THE BX.25 CERTIFICATION FACILITY

7. POSSIBLE ENHANCEMENTS TO THE BX.25 CERTIFICATION FACILITY

Several enhancements to the BX.25 Certification Facility have been identified and are being considered for inclusion in the facility. The enhancements are:

- Selectively suppressing portions of the report.
- Expanding the Test Input Language to include variables and elementary control structures (e.g., if-then-else).
- Including "loadtesting," or stress testing, as part of the certification of an implementation.
- Certifying levels other than 2 and 3 of BX.25, i.e., the physical layer, the multi-link layer, and the session layer.

8. PROVING IN THE BX.25 CERTIFICATION FACILITY

Assuring that the BX.25 Certification Facility is itself free from errors is of prime importance if the users of the facility are to have confidence in the reports produced by the facility. The following steps have been taken to achieve this goal:

- Many different implementations of BX.25 were tested on a "trial" basis. The purpose of the trial was to verify the correctness of the reports issued by the facility. As a result of the trial some problems with the facility were discovered, and were subsequently corrected. In the great majority of cases though, reports were accurate.
- Require that "certified" implementations report protocol problems discovered in the field to the group responsible for operation of the facility. In this manner the facility can be appropriately enhanced/modified, so that in the future the presence of such problems in implementations can be detected and reported.

Implementors of BX.25 may disagree with the correctness of the reports issued by the facility. This could happen, for instance, because of different interpretations of the BX.25 specification. In such cases, arbitration will be performed by those within Bell Telephone Laboratories responsible for the BX.25 specification. It should be noted that this has not yet occurred.

9. RELATED WORK

There has been other work in the area of certification and testing of data communication protocols. Bartlett and Rayner [4] discuss certification in general, and describe research underway at the National Physical Laboratory (NPL). Fong [14] describes NTS, a test system for DECnet. NTS is concerned with testing the user interface to DECnet. Weaving [15] presents the reference and test center of Euronet. The center provides a reference implementation, and a test and debug service for high-level protocols implemented on top of X.25. Weir et. al. [16] describe X.25 test facilities available on Datapac. The major test facility available is a X.25 protocol tester, which must be operated manually. Other facilities available include network generation of alarms, line monitoring, and X.25 diagnostic codes. Piatkowski [17] discusses the feasibility of testing ADCCP. Piatkowski concludes that a "complete and rigorous" test on an ADCCP station is impractical because of the length of time it would take to complete the test. Piatkowski sketches an ad hoc approach to ADCCP testing.

10. CONCLUSIONS

The BX.25 Certification Facility, a facility to automatically test that a BX.25 implementation conforms to the BX.25 specifications, has been described. The facility issues reports to indicate to what degree an implementation meets the BX.25 specifications.

The facility has the following major features:

- Certification is performed with minimal human intervention (i.e., the facility is automated).
- The tests systematically test all normal conditions, and error conditions (i.e. purposeful injection of errors) of BX.25.
- Adding tests is easy, since tests are stored in a computer file system (UNIX).
- Tests are parameterized, so the same tests can be used for all implementations.

- The reports issued by the facility feature tracing, time-stamping, numbering of tests for easy reference, an indication of which tests passed/failed, and for failed tests, the reason that test failed. A summary of the total number of tests, together with a list of the failed tests, is also included in the report file.

The BX.25 Certification Facility is a powerful tool. In addition to certifying an implementation of BX.25, it may be utilized by developers for testing during the development process.

11. ACKNOWLEDGMENTS

The author gratefully acknowledges the assistance of the following persons: A. Cline, B. Dickman, M. Lee, L. Mulraney, T. Peterson and T. Ryan. The author would also like to thank the referees for their valuable suggestions and comments. The high-level design of the BX.25 Certification Facility was influenced by a similar system developed at Bell Canada.

REFERENCES

[1] The BX.25 Operations Network Communication Protocol Specification, Bell System Technical Reference, pub. #54001.

[2] Amoss, J. J., "Planning for the Bell Operation Systems Network," Proceedings of the Fifth International Conference on Computer Communications (ICCC), Atlanta, Georgia, October 27-30, 1980. pp. 559-563.

[3] CCITT Recommendation X.25, CCITT Orange Book, Vol. VIII.2, "Public Data Networks," International Telecommunication Union (ITU), Geneva, Switzerland, 1977.

[4] K. A. Bartlett and D. Rayner, "The Certification of Data Communication Protocols," Proceedings of Trends & Applications: 1980, Computer Network Protocols, National Bureau of Standards, Gaithersburg, Md., May 29, 1980, pp. 12-17.

[5] C. Sunshine, "Formal Techniques for Protocol Specification and Validation," Computer, Vol. 12, No. 9, September, 1979, pp. 20-27.

[6] P. E. Green, "An Introduction to Network Architecture and Protocols," IBM System Journal, Vol. 18, No. 2, 1979, pp. 202-222.

[7] M. S. Solomon, "X.25 Explained," Computer Communications, Vol. 1, No. 6, December, 1978, pp. 310-328.

[8] D. F. Lee & J. Pasqua, "Internal Data Communications", Bell Laboratories Record, Jan., 1980, pp. 21-27.

[9] S. Leung, "The Concept and Implementation of BANCS", Computer, Jan., 1980, pp. 80-92.

[10] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Comm. of the ACM, Vol. 17, No. 7, July, 1974, pp. 365-375.

[11] International Packet Technology, "X.25 Protocol Tester PRO/TEST, Revision B," Document XT.0002.0114, September, 1980.

[12] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Prentice-Hall Inc., Englewood Cliffs, NJ, 1978.

[13] K. Jensen and N. Wirth, "PASCAL: User's Manual and Report," Second Edition, Springer-Verlag, New York, N.Y., 1978.

[14] Ng Pin Fong, "NTS -- A Protocol Test and Development System," Proceedings of Computer Networking Symposium, National Bureau of Standards, Gaithersburg, Md., December 13, 1978, pp. 124-127.

[15] K. Weaving, "Euronet Reference and Test Center," Computer Communications, Vol. 3, No. 5, October, 1980, pp. 221-223.

[16] P. F. Weir, W. E. Prater, X. N. Dam, "X.25 Test Facilities on DATAPACK," Evolutions in Computer Communications, Proceedings of the Fourth International Conference on Computer Communication (ICCC), Kyoto, Japan, September 26-29, 1978, pp. 273-279.

[17] T. F. Piatkowski, "Remarks on the Feasibility of Validating and Testing ADCCP Implementations," Proceeding of Trends & Applications: 1980, Computer Network Protocols, National Bureau of Standards, Gaithersburg, MD, May 29, 1980, pp. 94-109.

Appendix 1

PRO/TESTER COMMANDS

This appendix lists those PRO/TESTER commands which cause the transmission of frames or packets. The other PRO/TESTER commands have to do with setting up various initial conditions in the PRO/TESTER. For complete information concerning the PRO/TESTER refer to the PRO/TESTER manual [11].

LINK LAYER COMMANDS

Below are the PRO/TESTER commands to transmit frames. The general format is command, followed by parameters, separated by commas. Angle brackets are metasymbols used to identify those parameters which are optional. The values of the fields in the transmitted frame are automatically set by the PRO/TESTER, except when modified by a parameter. The meaning of each parameter is:

Parameter	**Meaning**
*	Set the poll/final bit.
a= hh	Set the address to hh (h represents a hexadecimal digit).
c= hh	Set the rejected control field of a FRMR to hh.
d= hh	Set the diagnostic field of a FRMR to hh.
nr= hh	Set the receive sequence number to hh.
ns= hh	Set the send sequence number to hh.
h= hex	Set the I field of an I frame to the hexadecimal value given by hex.

The commands are:

Command	Meaning
disc<,*><,a= hh>	Transmit a DISC (disconnect) frame.
sabm<,*><,a= hh>	Transmit a SABM (set asynchronous balanced mode) frame.
ua<,*><,a= hh>	Transmit a UA (unnumbered acknowledgement) frame.
dm<,*><,a= hh>	Transmit a DM (disconnect mode) frame.
frmr<,*><,a= hh><,c= hh>,<d= hh>	Transmit a FRMR (frame reject) frame.
crr<,*><,a= hh><,nr= hh>	Transmit a RR (receive ready) command frame.
crnr<,*><,a= hh><,nr= hh>	Transmit a RNR (receive not ready) command frame.
crej<,*><,a= hh><,nr= hh>	Transmit a REJ (reject) command frame.
rr<,*><,a= hh><,nr= hh>	Transmit a RR response frame.
rnr<,*><,a= hh><,nr= hh>	Transmit a RNR response frame.
rej<,*><,a= hh><,nr= hh>	Transmit a REJ response frame.
i<,*><,a= hh><,nr= hh> <,ns= hh><,h= hex>	Transmit an I (information) frame.
crc	Transmit a frame with a Cyclic Redundancy Check (CRC) error.
abort	Transmit an aborted frame.
o,h= hex	Transmit a frame whose hexadecimal representation is given by hex.

For example the command:

i,*,ns= 5,h= 4444

causes the transmission of an I frame with the poll bit set, the send sequence number equal to five, and an I field of 4444. The values of the other fields (address, receive sequence number) would be automatically set by the PRO/TESTER according to its current state.

PACKET LAYER COMMANDS

Below are the PRO/TESTER commands to transmit packets. The general format of the commands is the same as that for link layer commands. The meanings of the parameters are:

Parameter	Meaning
l= hhh	Set the logical channel identifier (LCI) to hh.
c= hh	Set the cause field to hh.
q	Set the qualifier bit in the general format identifier (GFI).
fi= hh	Set the GFI and the first four bits of the LCI to hh.
db	Set the delivery confirmation bit in the GFI.
m	Set the more data bit.
pr= hh	Set the packet receive sequence number to hh.
ps= hh	Set the packet send sequence number to hh.

h= hex	Set the User Data field of a DATA packet to the hexadecimal value given by hex.

The following are the PRO/TESTER commands to transmit packets:

Command	Meaning
r<,fi= hh><,c= hh>	Transmit a RESTART packet.
rc<,fi= hh>	Transmit a RESTART CONFIRMATION packet.
c<,fi= hh><,l= hhh>	Transmit a CALL packet.
acp<,fi= hh><,l= hh>	Transmit a CALL ACCEPTED packet.
int<,fi= hh><,l= hhh><,q>	Transmit an INTERRUPT packet.
intc<,fi= hh><,l= hhh><,q>	Transmit an INTERRUPT CONFIRMATION packet.
rst<,fi= hh><,l= hhh><,c= hh>	Transmit a RESET packet.
rsc<,fi= hh><,l= hhh>	Transmit a RESET CONFIRMATION packet.
rr<,fi= hh><,l= hhh><,pr= h>	Transmit a RECEIVE READY packet.
rnr<,fi= hh><,l= hhh><,pr= h>	Transmit a RECEIVE NOT READY packet.
d<,fi= hh><,l= hhh><,q><,m> <,db><,pr= h><,ps= h><,h= hex>	Transmit a DATA packet.
clr<,fi= hh><,l= hhh><,c= hh>	Transmit a CLEAR packet.
clrc<,fi= hh><,l= hhh>	Transmit a CLEAR CONFIRMATION packet.

p,h= hex	Transmit a packet whose hexadecimal representation is given by hex.

For example the command:

```
d,l= 001,ps= 4,h= 44
```

causes the transmission of a DATA packet over LCI 001, with a packet send sequence number of four, and a User Data field of "44". The other fields (packet receive sequence number, GFI, the M bit) would be set according to the current state of the PRO/TESTER.

The c command has additional parameters not listed here, having to do with the user facilities selected. Moreover, there exists eacp, and eclr commands, for "extended" acp, and "extended" clr, respectively. These commands also have additional parameters, not listed here. Refer to [11] for more information.

THE DESIGN OF THE CSNET NAME SERVER

(Preliminary Report)

Marvin Solomon
Lawrence H. Landweber
Donald Neuhengen

University of Wisconsin--Madison

1. INTRODUCTION

CSNET is a new computer communications network being constructed that will link together University Computer Science departments and other groups doing computer science research in the United States. An important component of CSNET will be a directory service called the CSNET Name Server, which is implemented by a central database at the University of Wisconsin and by software running at Wisconsin and on the computers of member institutions. This paper describes the architecture of the name server facility.

In early stages of CSNET, the principal use of the name server will be to facilitate sending of electronic mail by providing such services as directory assistance in locating addresses of mail recipients, and aiding in forwarding mail and establishment of nicknames and aliases. It is on this aspect of the name server that this paper focuses. In later stages of CSNET, the name server will also help support other facilities such as file transfer and remote access to computing resources.

In the next section, we briefly describe CSNET and explain how its characteristics have influenced design of the name server. The structure of CSNET is described in more detail in [1,2].

We have designed the name server to be implemented in a series of phases, progressing from facilities that already exist, through more and more sophisticated structures, to a system that will eventually provide all desired features. In doing so, we have attempted to be conservative in early phases, using the simplest structure that will fulfill the immediate needs of CSNET users, while leaving the door open for more ambitious enhancements in the future. While the services described here will be implemented with available CSNET staff and resources, we expect the project to identify several challenging research areas.

Section 2 describes CSNET and discusses project characteristics which have influenced the name server design. Section 3 describes name server design requirements and implementation performance goals. Section 4 includes definitions of terms and an outline of the various phases. The four phases of the name server implementation are described in Sections 5-8. Section 9 briefly addresses the issue of mailing lists, and Section 10 provides a summary and comparison to related work.

2. OVERVIEW OF CSNET

CSNET is a logical network which uses communications services provided by ARPANET [3], the commercial value-added network Telenet, and a telephone-based mail relay service called PhoneNet. Member institutions access the services of CSNET by connecting a computer ("host") to ARPANET or Telenet, or if their budget is limited and they are willing to accept reduced service, by arranging for their host to exchange mail periodically with a PhoneNet relay machine which is directly connected to ARPANET and Telenet. CSNET will provide electronic mail, file transfer, and remote login (virtual terminal) services to directly-connected hosts. PhoneNet hosts will only have access to electronic mail services. In addition, CSNET maintains a Public Host, which is a VAX computer connected to ARPANET and Telenet, running the UNIX operating system, and providing mail-only accounts to individuals who do not have access to any other CSNET member host. Although CSNET is being subsidized in its initial stages by the National Science Foundation, it is expected to become self-supporting in a few years, with all members paying a fair share of the costs.

One of the challenges of CSNET is to reconcile the differences between characteristics of these communications media and provide users with as uniform an interface as is possible. ARPANET provides a high-bandwidth, low-delay communications path between computers connected to it. Telenet provides similar (but lower bandwidth) service, but whereas the cost of an ARPANET connection is fixed, Telenet charges are highly dependent on the amount of traffic. PhoneNet charges are even more dependent on traffic, since the only fixed charges are the cost of a modem and a telephone line. However, a much more important difference between PhoneNet and direct connection is delay. CSNET clients not directly connected to ARPANET or Telenet must rely on periodic exchanges of mail with a PhoneNet relay machine for their connection to the network. The frequency of such exchanges may be as low as once daily. We shall see that these wide variations in delay (from minutes or seconds to days) is an important consideration in the design of the name server.

3. GOALS

The name server facility is designed to satisfy the following service requirements:

1. The system must be simple to use. While most CSNET users will be computer science researchers, many will be theoreticians who have little experience with computer-based mail systems.

2. A sender of mail may identify a recipient in a variety of convenient ways. A user may refer to frequent correspondents by nicknames of his own choosing. In addition, a host may make available to its users aliases for other hosts and users.

3. A receiver of mail may supply the information others use to identify him. For example, he can supply his full name, organization, location, title, nicknames, common misspellings of his name, etc.

4. Mail may be sent to any user of any host in CSNET, without prior explicit effort on the part of the receiver, although reduced services will be available for communication with "unregistered" users. Similarly, CSNET users must be able to communicate with others "outside" the network, in particular users on hosts in the DARPA Internet address space but not running CSNET-specific software.

5. The mail system will never force a user to use more than one "mailbox" to receive mail, although a user may choose to establish more than one mailbox to reflect differing roles. In the latter case, each mailbox may be thought of as representing a different "virtual user".

6. A user can move his mailbox to a different host computer with a minimum amount of difficulty. Senders need not be explicitly notified; mail will be automatically forwarded.

7. Support will be provided for mailing lists.

In addition to these service requirements, the implementation is designed to satisfy the following additional performance and utility goals:

1. The system should expand gracefully to include more member sites, additional users, and even additional networks. In particular, anyone able to send electronic mail to the University of Wisconsin should be able to gain access to at least some of the name server services.

2. The system design should provide for phased implementation so that basic services can be put into place immediately, while more sophisticated facilities may be added incrementally until all desired features are available.

3. Network traffic should be minimized. Control messages should be infrequent and user text should be sent over the most efficient route. In particular, relaying of messages should be minimized.

4. The system should continue to function, perhaps in a degraded mode, if components fail.

5. Delay between the submission of a message by a sender and its delivery to a recipient should be minimized. In particular, if the sender is on a machine that is only periodically connected to the rest of CSNET (a PhoneNet host), the number of interactions between that host and the rest of CSNET which are required to dispatch the message should be minimized.

6. The system should work with a minimum of human intervention, either on the part of users or of administrative staff.

4. DEFINITIONS

Throughout this paper, we will be talking about users and hosts. For our purposes the term "user" always refers to a human being (and will not, for example, be used to mean a "user program"). A host is a computer connected to a communications network. Users gain access to network facilities through accounts on hosts. For our purposes, hosts can be classified as CSNET member hosts, that subscribe to CSNET defined conventions and run CSNET provided software packages, and other hosts, which are capable of exchanging mail with CSNET member hosts, but do not necessarily run CSNET software. There are also CSNET run hosts including the Service Host, a computer at the University of Wisconsin that maintains a central database and programs for accessing it, PhoneNet relays, computers (initially at the University of Delaware and the Rand Corporation) that periodically connect to other hosts to pick up and deliver mail, and the Public Host, a computer at the University of Wisconsin that is run by CSNET but otherwise is treated exactly like any other CSNET member host. Hosts may also be classified as ARPANET, Telenet, or PhoneNet hosts depending on the principal method used to exchange information with the rest of CSNET. The name server relies, for many of its functions on a mail transport system, which is a collection of protocols and programs that run on hosts and provide the mechanism for transferring messages from sources to destinations. Users normally interact with the mail transport system through a user-interface program (UIP), which is a program that interacts with users for composing, sending, receiving, reading, and filing messages.

The various services and mechanisms described in this paper comprise the name server facility. It is provided by a combination of files and programs residing on the Service Host and on other CSNET member hosts. The name server database is a database which includes directory information for registered CSNET users and hosts and which is distributed among a central directory database that resides on the Service Host, per-host tables that reside on hosts that originate mail, and per-user tables maintained by local mail systems on behalf of individual users. The postmaster general is a collection of software that runs on the Service Host and mediates access to and modification of the central directory database.

Users may access the name server database by sending messages directly to the postmaster general. However, users will normally compose their queries by interacting with a name server agent program, copies of which reside on CSNET member hosts. A copy of the agent program will also reside on the Service Host for the convenience of users on non-CSNET hosts who have virtual-terminal access to the Service Host. The agent programs communicate with the postmaster general using the best means available, either by direct connection of by exchange of messages through the mail transport system. In the latter case, there is necessarily a large delay, so users will receive a limited level of service.

The name server facility is specified (and will be implemented) in four phases. As new phases are implemented, all features provided by earlier phases will remain available to users. Phase 0 provides basic services and is compatible with current addressing and naming schemes employed in the DARPA Internet. Phase 1 introduces a centralized directory database at the Service Host and a directory assistance service that users may access by exchanging mail with the postmaster general. In Phase 2, user interaction with the directory assistance service will be further automated. Phase 3 adds support for automatic forwarding of mail and for mailing lists.

5. PHASE 0: BASIC SERVICES

Phase 0 provides services which are very close to those currently available in the DARPA Internet environment. Each host in CSNET has an unambiguous name, such as "UWISC", "UDEL", or "WASHINGTON". A site with a local network may choose to designate a particular computer to serve as a gateway host to CSNET and assign it a name which designates the site. Arpanet hosts already have unambiguous names. Hosts that currently exchange mail using the Bell Laboratories "uucp" system also use unambiguous names. As sites join CSNET, they will register their hosts with the CSNET administration, which will certify that names are not duplicated. (By "unambiguous" we mean that no two hosts will have the same name; there is no reason to prevent a host from having more than one name.)

Users interact with the mail system through accounts on hosts that are assigned <u>user names</u>. Each host will guarantee that a user name unambiguously identifies one mailbox on that host. In other words, "user name" represents some name for a mailbox that is printable, is assigned by a host administration, and identifies a unique mailbox on that host. Hence the pair "user-name@host", which we call a <u>mailbox address</u>, can be used to uniquely identify any mailbox in CSNET. Current mail transport systems deliver messages based on this pair. A user who knows the mailbox address of a recipient may always use it to specify the destination of a message although, as we shall see, other more convenient methods will be available in later phases.

6. PHASE 1: DIRECTORY ASSISTANCE

Phase 1 augments the basic facilities described in the previous section with a "directory assistance" service. A central directory database on the Service Host contains information about users. Each entry in this database contains the address of one mailbox, together with information identifying the owner of that mailbox. This information is supplied by the owner and includes, at a minimum, his full name and the name of his organization (e.g., university or research lab). In addition, it may include other keywords such as titles, aliases, and common misspellings of the user's name, postal address, phone number, and any other information the user wishes to provide. Registration in this database is entirely voluntary and it is possible to communicate with non-registered users even if their local site has not installed the CSNET name server software.

The database is accessed by transmitting properly formatted queries to the postmaster general on the Service Host. Users will not normally communicate directly with the postmaster general but rather with an agent program that formats the request and forwards it to the Service Host by a direct connection or by the mail transport system. However, users of non-CSNET computers may also query the database by mailing requests directly to the postmaster general.

Since we intend that each user have the ability (and responsibility) to maintain the database entry describing him, certain access controls must be in place from the very beginning to maintain the integrity of the database.

6.1 Registering

The user adds or modifies entries in the database by interacting with his local agent using the commands "register", "update", and "unregister". The local agent creates a message containing the user request to insert, modify or delete a central database entry and sends it to the postmaster general either directly, or by mailing it to the address "REGISTRAR@CSNET-SH". The postmaster general will parse the message and perform the requested operation.

6.2 Authentication

An important issue is authentication of a user requesting insertion or modification of an entry. Each member organization will provide a key (password) when it joins CSNET. This key will be kept in encrypted form at the Service Host. The administration at the site will be responsible for controlling its distribution and for changing it when appropriate.

A user at a member host who wishes to register himself in the database interacts with his local agent. This agent runs as a privileged program that has access to the site password. The agent engages in a dialogue with the user to authenticate his identity (for example, by asking for a password) and verifies that the proposed database entry is appropriate to the user (in particular, that the "mailbox address" field properly identifies the user). Having satisfiea itself of the validity of the request, the agent formats it, encrypts it using the site password, adds an unencrypted header identifying the local site, and forwards it to the postmaster general. The postmaster general decrypts the request and installs the information in the database.

This scheme delegates authority for authenticating users to the member sites. Each site is held responsible for all database entries that identify a mailbox located at that site. The agent program (which is provided by CSNET) gives a mechanism for controlling these entries. A user cannot bypass this mechanism and send a registration request directly to the postmaster general because he does not know the site password. A user of a nonmember computer may not mail a request for a new entry directly to the postmaster general, nor may he add an entry by interacting with the copy of the agent program that resides on the Service Host (since the latter has no way of authenticating him). In other words, only users of csnet member hosts can add entries to the database, and they may only do it from their home machines.

Upon registration, a user may provide a password to be used by him when modifying or deleting his directory database entry. This password will only be required if the user initiates a change request from a host other than his home machine. The purpose of this feature is to allow users to modify their database entries from hosts other than the one specified in their mailbox address. It is particularly useful when an individual moves to a new site and changes his mailbox address. The postmaster general will inform the host specified in the original database entry that the entry has been changed. This notification will provide an additional check to insure that the change is authorized.

To perform housekeeping functions, particularly deleting of defunct entries, a site may authorize a special trusted user named "csnet" to perform commands on behalf of other users at that site.

This authentication mechanism is not "airtight", but should provide an adequate level of protection at modest cost. More importantly, it delegates authority for security, so that if breaches are detected, the responsible party may be identified.

An interesting example of a fraud that is not prevented by this scheme might be called "false advertising". The owner of mailbox vesco@costa-rica inserts an entry addressing his mailbox but including keywords that match some other user, with the intent of fooling users into sending mail for the other user to the perpetrator's mailbox. This ruse would be particularly pernicious when lookup is automated so that human users don't normally even look at the mailbox address returned (see Phase 2). The situation is comparable (in the non-electronic world) to Marvin Solomon putting an advertisement in the newspaper saying that the address of the First National Bank of Madison is 850 Terry Place (Solomon's home address). The name server mechanism cannot prevent such a fraud without understanding the semantics of all the keywords in an entry. But the injured party, if he discovers that mail is being misdirected, can query the central database to find the bogus entry. Similarly, a sender might notice that certain queries identify two entries, one of which looks suspicious, and report the fraud. Once the fraudulent entry is found, the culprit can be traced, at least as far as his host.

Authentication is a difficult but important area. Further study will be required if a more elaborate scheme than that described above is found to be necessary.

6.3 Using the Database

To access the central directory, a query is delivered to the postmaster general or mailed to it at the address NAMESERVER@CSNET-SH. Normally, users will use the "whois" command of the agent to compose such a request. The query will include lists of mandatory and optional keywords. Only entries that contain all mandatory keywords will be selected. If more than one entry matches, the optional keywords may be used to select the entry with the most matches, or the postmaster general may be instructed to return only entries containing at least *k* of the optional keywords.

Keywords can be parameterized so as to allow specification of pattern matching. Keywords may also contain "wild cards" to allow inexact matches. For example, the keyword "landwe*er" can be used by those not knowing whether his name is spelled as "landweber", "landwever", or "landwebber". Upper and lower case are considered equivalent for matching purposes, but the entries will be displayed to the requester in the same case as they were originally specified at registration. The requester can then select the appropriate entry (if there is more than one match) based on other information in the entries, and use the mailbox address included in the entry to send mail.

The provision of mandatory and optional keywords is primarily for the benefit of the user of a PhoneNet host, to maximize the chances of him getting the right answer on the first try. Too few keywords will flood him with bogus matches, but too many mandatory keywords may exclude valid matches. The ability to get a unique match on the first try becomes particularly beneficial in phase 2 (as we shall see).

Incidentally, directory assistance could be useful for services outside CSNET proper, such as looking up a user's phone number or (U.S. Post Office) mailing address.

6.4 Example

Here is a sample use of the name server in phase 1: To make it easier for others to find Marvin Solomon, he issues the "register" command to the agent, which engages in a dialogue with him to authenticate his identity and gather information about him. It then composes and encrypts a message to REGISTRAR@CSNET containing text something like this:

```
register solomon@uwisc Marvin Solomon
     University of Wisconsin Madison
     Computer Sciences Department
     1210 W. Dayton St. Madison WI 53706
     608-262-1204
     soloman csnet contractor service host public host
     computer science
```

A user who wishes to send mail to Solomon might issue the command

```
whois solomon [csnet implementor]
```

where the keyword "solomon" is mandatory, but the keywords "csnet" and "implementor" are optional. There may be several entries containing the keyword "solomon", but the one shown above is the only one containing either of the words "csnet" or "implementor". He would receive the response:

```
In response to <whois solomon [csnet implementor]>:

solomon@uwisc
Marvin Solomon
University of Wisconsin Madison
Computer Sciences Department
1210 W. Dayton St. Madison WI 53706
608-262-1204
soloman csnet contractor service host public host
computer science .
```

(The response might be abbreviated in an interactive setting.) He could then send mail to Solomon by addressing it to "solomon@uwisc". Existing mail user interface programs generally have a nickname (also called "alias") facility that allows the user to say something like:

```
nickname marv=solomon@uwisc
```

to avoid having to remember the address.

Solomon included "soloman" as a keyword, since he knew that people frequently misspell his name that way. The user querying the database can also protect himself from misspelling by using a combination of wildcards and optional keywords. For example, he could say

```
whois s* wisconsin [soloman soloman salemon]
```

7. PHASE 2: AUTOMATED LOOKUP

Phase 2 adds services to decrease the amount of interaction required between the human user and the central database. In particular, the mail uip and the local agent are integrated.

7.1 Automatic Nickname Establishment

In Phase 1, responses resulting from central database lookup queries are always returned directly to the user. In Phase 2, facilities will be added to automate establishment of local nicknames.

Continuing the previous example, the interaction with the name server and the establishment of a local nickname could be combined by issuing the command

```
nickname marv = whois(solomon [csnet implementor])
```

to the mail uip, which would format a request and send it to the postmaster general. (No authentication is required since no change to the nameserver database is being requested.) If the query matches exactly one entry, the nickname is installed in the user's private nickname table. If no entries or more than one entry are returned, the response is returned to the user requesting more information. In the PhoneNet environment, the user receives notification of success or failure of nickname establishment in the form of a message mailed to him. A facility will also be provided by which a local administrator can install commonly used aliases in a table accessible to all users at a site.

Finally, the user will be able to combine query of the database, establishment of a nickname, and sending of the first message with a command such as

```
send_to marv = whois(solomon [csnet implementor])
```

The ability to combine these operations will be particularly advantageous to PhoneNet users. The initial message will be delivered to the PhoneNet relay containing the keyword information instead of the mailbox address in the "To" field, together with an indication to the PhoneNet relay that a lookup is required. The relay composes the query to the postmaster general and intercepts the reply. Assuming that a unique match is found, the relay updates the message header to include the mailbox address (leaving the keyword information in as a comment) and delivers the message as usual. It also forwards the reply from the postmaster general back to the originating host so that the private nickname table of the sender can be updated. The advantage of this scenario is that the message can be delivered after only one interaction between the sending host and the relay. For example, if the sending host is only polled once each day but the destination host is directly connected, this scheme provides next-day delivery, assuming that the list of keywords uniquely identifies one mailbox. The reason for requiring each database entry to include the user's full name and institution (CSNET member organization), is to give the careful sender a reasonable chance of constructing a query that will match uniquely on the first try.

We are deliberately requiring the sender to specify both a list of keywords <u>and</u> a nickname with the first (or only) message to a given recipient rather than allowing a syntax such as "send_to marvin solomon". The reason is efficiency: If a nickname is established, subsequent transmissions are much cheaper. If it were too convenient to send using only keywords, users would be encouraged to use keywords every time, even to send to users they communicate with frequently. On the other hand, a special syntax might be provided for sending to a user just once (that is, avoiding the establishment of a local nickname), provided it was sufficiently awkward as to discourage its use if there is a reasonable chance of sending to the same user again. An example syntax might be

```
send_to temporary-nickname = whois(marvin solomon)
```

with the nickname "temporary-nickname" given the special semantics that results of the lookup are not to be returned to the sending site unless there is an error (such as multiple matching entries).

8. PHASE 3: FORWARDING

Suppose an existing user is assigned a mailbox on a new host. Under some circumstances, he may want that mailbox to be considered different from his previous mailbox. For example, he changed jobs. Under existing mail transport systems, a message sent to the old mailbox (assuming it was deleted) will be returned to the sender with an indication that the mailbox no longer exists. A user who is really interested in sending to him as a person, rather than in his official capacity at his old job, could query the database to determine his new address and resend the mail. This situation corresponds closely to the telephone system (where "address" corresponds to phone number). However, under other circumstances, the user would like the change of address to be invisible to his correspondents. For example, suppose he is moved to a different host on the basis of an administrative decision such as load-leveling, or he is temporarily visiting another site and finds it more convenient to have mail forwarded there (compare with the phone company's new "call forwarding" service).

8.1 The Forwarding Mechanism

To simplify various aspects of forwarding, each nameserver database entry will contain a registration ID that uniquely and unambiguously identifies the database entry. This ID is included in database entries from the start, but only comes into play in Phase 3. (This idea was inspired by a suggestion of Denning and Comer [4].) The mail uip will be modified to include the registration ID in per-user nickname tables. For example, if a user types

```
nickname marv = whois(solomon [csnet implementor])
```

the nickname table will store, under the entry "marv" not only the mailbox address (solomon@uwisc), but also the registration ID for the associated database entry.

The forwarding mechanism is best described by an example. Suppose Pat Kane is at site A and has a mailbox with address "pat@sitea". He moves to site B and is refused the name "pat" as his mailbox name since there is already a pat there, so he chooses the name "pkane". The mailbox pat@sitea is deleted from site A. Users who bypass the CSNET name service entirely and send to "pat@sitea" will have their messages returned as undeliverable. They must learn from channels outside of CSNET (such as word-of-mouth) that mail to Pat Kane must now be addressed to "pkane@siteb". However, Pat Kane may inform the postmaster general that he has moved. (Authentication of the information will use a similar mechanism to that described above for registering users.) His entry in the central database is updated to indicate his new mailbox address, so that new correspondents looking for him by keyword search will find his new address. Old correspondents will still have mail returned, but now senders who use the name server can have their local mail systems recover without manual intervention.

Suppose the sender has established an alias for Pat Kane by the command

```
nickname pk = whois(pat kane)
```

When the nickname was established, the local tables for the sender received an entry such as

```
pk : pat@sitea (CSNET-ID: 0012345)
```

When the sender tries sending to "pk" after Pat has moved, a message addressed to "pat@sitea (CSNET-ID: 0012345)" is sent to SITEA, refused, and returned to sender. The sender's uip can query the postmaster general to find out if there have been any changes in entry 0012345. In this case, the postmaster general sends the new address "pkane@siteb", the sender's uip updates its tables to read

```
pk : pkane@siteb (CSNET-ID: 0012345),
```

and re-sends the letter to "pkane@siteb (CSNET-ID: 0012345)". The sending user is never bothered, all his future mail to "pk" will be sent directly to the correct address.

One additional complication arises. Continuing the previous example, suppose after Pat Kane leaves site A, Pat Able appears and wants to be known locally as "pat". She might well be unhappy about being told that she couldn't use the name "pat" because there once was man named Pat Kane who already reserved that same name. On the other hand, SITEA will have no way of knowing whether mail addressed to "pat@sitea" was intended for Pat Kane or Pat Able. Once again, senders who bypass CSNET software can still send mail, but they receive reduced services. In this case, they run the risk of a message going to the wrong person.

If SITEA is a CSNET member site, however, it will store the registration ID of all its local users who are registered. If an incoming message contains a registration ID that does not match the registration ID of the addressee, the message will be rejected. When Pat Kane changed his address to "pkane@siteb", the postmaster general informs SITEA the the user id "pat:0012345" is no longer valid.

8.2 Optimizations

The update message to SITEA could include the forwarding address, and SITEA could cache forwarding addresses for recently moved mailboxes. When the letter addressed to "pat@sitea (CSNET-ID: 0012345)" arrives at SITEA, SITEA could then send it directly to pkane@siteb rather than returning it to the sender. It should still inform the sender of the change, and the sender may well wish to check the new address with the postmaster general rather than trusting SITEA, but the delay in delivering the letter would still be reduced from five message-transition times (sender to SITEA; SITEA to sender; sender to Service Host; Service Host to sender; sender to SITEB), to two (sender to SITEA; SITEA to SITEB). The possibility of this sort of forwarding raises difficult problems in billing, however (e.g., who pays for the forwarding hop and how is he billed), which are beyond the scope of this paper.

Another optimization is based on the observation that it is common for several users at one site to correspond with the same person. If they all have obsolete nicknames for him, the overhead of a misdirected message can be moved from the first time each of them sends to him to the first time <u>any</u> of them sends to him. Instead of storing the entry "pk : pat@sitea (CSNET-ID 0012345)" in the nickname table for a user, we can store an entry like "pk : 0012345" in the per-user table and maintain a per-host table with the entry "0012345 : pat@sitea".

9. MAILING LISTS

All the mechanisms described thus far are techniques for discovering the address of one mailbox. There is nothing to prevent them from being used repeatedly and in combination to develop multiple addresses on a single message, such as

```
send_to marv,
    larry = whois(lawrence landweber wisconsin),
    donn@uwisc
```

which names the three authors of this note by a nickname, a keyword search, and a mailbox address, respectively.

A related facility is the mailing list which is a name for a list of mailboxes that are often used together. Existing user interface programs often provide a mailing list function using the nickname facility to do a macro expansion of a mailing list name. In Phase 3, the CSNET name server will allow users to register mailing lists in the central directory database. A mailing list entry is similar to other entries in that it contains a list of keywords and a mailbox address of a user responsible for the entry. But it also contains a list of mailbox addresses. A request to add a mailing list to the directory contains the keywords as well as descriptions of the members, specified by any convenient means (i.e., keywords, nickname, or mailbox address). On receiving such a request (which must pass the usual access checks), the postmaster general resolves each member to an address and stores the list of addresses. If any member specification fails to resolve to a unique address, the request will be returned to the sender for correction.

When a mailing list is installed, the postmaster general will send a message containing the names of all the members to each member. (There might be circumstances under which this notification should be suppressed.) Similarly, a change in the mailing list will generate a notification to all parties involved. In Phase 3, any change-of-address notice to the postmaster general will also cause changes to all lists that contain the obsolete address.

10. COMPARISON TO OTHER WORK

Several reports have been published on "name servers" for computer networks [5,6,7]. A particularly interesting related name server design is the Xerox Clearinghouse [8], which may use mail transport services provided by the Grapevine [9] system. While Grapevine is primarily a mail transport system, it also provides for naming, authenticating, and locating people, machines, and services in a multi-network environment. The Clearinghouse is a system for naming and locating objects in a distributed environment. Both of these systems are designed to operate in an inter-network environment with associated databases and services distributed among different machines on different networks. Therefore, many of the complications that concern the authors of these papers, such as how to find a name server, do not arise in our context, in which there is a unique name server at a well-known address. On the other hand, these systems are designed for environments in which message-passing is cheap and quick, and in which broadcast is a viable means for locating services. It is not clear how to apply their techniques in an environment in which a single message "hop" can take more than a day.

Another difference has to do with how the sytems are to be used. Clearinghouse objects (including mailboxes) have a three-part "name" of the form object@domain@organization. Our strategy for mailbox addresses is similar, unambiguous host name together with a username that is unambiguous relative to the host. In the Clearinghouse system, a three-part name is used to obtain information about the associated object. Besides wildcard characters, which may be used to aid in matching a name, the user is not provided with any assistance in locating the desired database entry. In our system, the primary goal is to facilitate lookup of mailbox addresses based on incomplete information. Hence, it is not necessary to know any particular piece of information such as the user´s complete name to locate his entry.

11. SUMMARY

We have presented a detailed specification of a name server facility for CSNET and have sketched out the algorithms for implementing it. The facility is implemented by a postmaster general program running on the Service Host and local agent programs running on local hosts. The facility will be implemented as a series of enhancements to existing services, each adding more convenience for users. It assumes a mail transport system that can deliver a message when presented with a list of destination addresses. It also allows for interactive database access in cases in which the user or the user´s host is capable of direct connection to the Service Host.

We have deliberately avoided discussing issues involving the mail transport system, such as routing, mail relays, multicast delivery, or reply-to-sender, except as they are directly related to the name server, but we do not believe that the name server facility creates any new problems in these areas since address specifications ultimately resolve to addresses in the style already in use. We have also not tied the name server specification to a particular mail interface program.

The techniques for implementing the algorithms described here are well understood, and tools (such as a flexible filesystem, inverted indices, and encryption programs) are already in place in the operating system for the Service Host. Therefore, we feel that the name server can be implemented quickly and begin to provide services to users soon. Phase 1 of the name server is currently being implemented with test release scheduled for January 1982.

12. ACKNOWLEDGMENTS

We gratefully acknowledge helpful comments from Vint Cerf, Keith Lantz, and Jon Postel. Continuing discussions with Doug Comer, Peter Denning, and Mike Litzkow have been have been very valuable in crystalizing many of the ideas in this paper.

13. REFERENCES

[1] Lawrence H. Landweber, "CSNET - A computer research network," Proposal to the National Science Foundation, (October, 1980).

[2] L. Landweber and M. Solomon, "The structure of CSNET," CSNET-DN-81-1, University of Wisconsin--Madison Computer Sciences (August, 1981).

[3] L. G. Roberts and B. D. Wessler, "Computer network development to achieve resource sharing," Proceedings of SJCC, pp. 543-549 (1970).

[4] P. Denning and D. Comer, The CSNET user environment, Computer Science Department, Purdue University (July, 1981) unpublished note.

[5] J. R. Pickens, E. J. Feinler, and J. E. Mathis, "An experimental network information center name server (NICNAME)," IEN 103, SRI International, Menlo Park, California (May 1979).

[6] J. R. Pickens, E. J. Feinler, and J. E. Mathis, "The NIC Name Server--A datagram based information utility," Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks, (August 1979 on Distributed Data Management and Computer Networks)).

[7] J. Postel, Internet Name Server, Information Sciences Institute, University of Southern California (May 1979).

[8] D. C. Oppen and Y. K. Dalal, "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment," Technical Report OPD-T8103, Xerox Office Products Division (October 1981).

[9] A. Birrell, R. Levin, R. Needham, and G. Schroeder., "Grapevine," Proceedings of the 8th ACM Symposium on Operating Systems Principles,, (To appear, December, 1981).

ON THE CORRECT AND EFFICIENT SCHEDULING OF TRANSACTIONS IN A HIGHLY PARALLEL DATABASE MACHINE*

Ravindran Krishnamurthy
IBM Thomas J. Watson Research Center
Yorktown Heights, New York, 10598

Umeshwar Dayal
Computer Corporation of America,
Cambridge, Mass.

Abstract

This paper proposes a two-step technique for producing correct and highly parallel schedules for MIMD, (multiple instruction stream, multiple data stream) database machine. A parallel program schema model for transaction systems is presented. The concept of correct (i.e. serializable) executions existing in cocurrency control theory for the sequential model is extended to this parallel model. The model is used to derive minimally constrained schemas for optimal scheduling. This constitutes the first step of the two-step technique. In the second step, the transactions are partially interpreted to enhance parallelism. A high level query language is chosen, for which a set of transaction modification rules are presented. A practical scheduling algorithm is proposed to obtain a highly parallel schema.

1. Introduction.

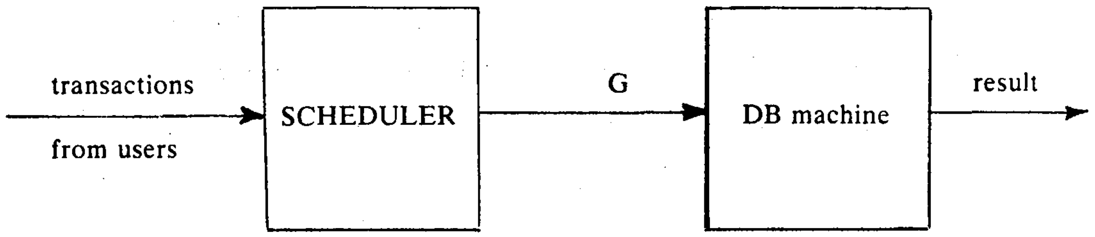
Parallel (i.e. multiple instruction-stream, multiple data stream) database machines such as DIRECT, have been proposed with the objective of enhancing processor utilization and

* Most of this work was done while the authors were associated with the Computer Science dept. at The University of Texas at Austin, Austin, TX.

achieving high transaction throughput [DeW78]. Improving processor utilization requires the efficient scheduling of transactions (for parallel execution) on available processors. But a parallel execution of transactions requiring access to shared data, can lead to race conditions and inconsistent states of the database, unless some synchronization (concurrency control) mechanism is used [EGLT76]. This underscores the importance of both synchronization and efficient scheduling to achieve correct (i.e. serializable) and maximally parallel executions.

The DIRECT machine uses locking in the front-end as its synchronization mechanism. However, this seems unduly restrictive and may even be prohibitive for very high throughput machines. No other synchronization mechanisms for parallel database machines have been proposed in the literature. On the other hand, a wealth of concurrency control theory has been developed for centralized and distributed data base systems, assuming a sequential model of execution [BSW80, Papa79, BSR80]. In this paper, we extend this theory to a parallel model of execution and use it to derive the minimal set of precedence constraints for scheduling.

Our model of a database system is shown in figure 1.1. Users submit transactions (each consisting of several steps) to the system. A set of these transactions, called a transaction system, is input to the scheduler, which examines the transactions for potential conflicts and imposes a partial order on the transaction steps. The scheduler outputs a precedence graph (called a schema) corresponding to this partial ordering, for execution on the database machine. The transaction system is then executed by the machine using some low-level processor allocation policy. We are concerned here with the problem of representing a transaction system using minimal precedence constraints so that any execution satisfying these constraints is correct. An execution is correct if and only if it is serializable, i.e. equivalent to some serial execution. In the existing concurrency control theory for the sequential model of execution, both in centralized and distributed systems, a transaction is modelled as a sequence of operations. The execution of a transaction system in the centralized case is also modelled as a sequence of operations, perhaps with steps of different transactions interleaved in it. This sequence is called a history. A given history is serializable if and only if it is computationally equivalent to a serial history, which defines a total ordering on the transactions in the system [BSW79, Papa79]. In the distributed case, an execution is modelled as a set of histories, one for each site. A distributed execution is serializable if and only if the history at each site is serializable and the equivalent serial histories at all sites impose the same total ordering of transactions.



$\mathcal{T} = \{T_1, T_2, \ldots T_n\}$ = Transaction system.

$T_i = (t_{i1}, t_{i2}, \ldots t_{ik_i})$ = a sequence of transaction steps.

T = set of all transaction steps.

G = Schema output by the scheduler.

Fig. 1.1 Model of a database management system.

A sequential ordering of transactions steps does not exploit the full power of a parallel machine. Gouda [Gou80] has extended this sequential model of a transaction and a transaction system to a directed acyclic graph (DAG). The edges of the DAG impose precedence constraints on conflicting pairs of transaction steps. In the first part of this paper, we formalize this model using parallel program schemata theory [Kell73] and extend the notion of serializability to this model. We then derive a minimal set of precedence constraints for a transaction system to satisfy this correctness criterion.

Most proposed concurrency control mechanisms use uninterpreted transactions, i.e. they use only syntactic information such as the read and write sets of the operations in the transaction steps. It has been shown in [KP79] that greater concurrency can be achieved if semantic information is utilized in addition to purely syntactic information. In the second part of this paper we show how to exploit semantic information to modify transaction steps and thus increase parallelism even further. For example, in an airline reservation system, if there are five (simultaneous) requests for seats on a particular flight, most systems of today will satisfy these requests one by one. We present an algorithm that modifies these requests so that instead of executing sequentially, they can run in parallel. This is accomplished by making one request to update the number of reserved seats in the database by 5 and making all the others only read the database. Thus, all the five requests can be made to run in parallel.

Thus, this paper proposes a two-step technique for producing correct and highly parallel schedules: first, construct a schema with minimal precedence constraints; then, modify it using semantic information to increase parallelism.

Section 2 presents a parallel program schema model of a transaction and defines the notion of syntactic and semantic equivalence of schemata. Using this, the scheduler, the schema for a transaction system, and an execution of a transaction system are formally defined. In section 3, correctness criteria are presented for each of these, together with a method for syntactically constructing a correct execution schedule with minimal precedence constraints. Section 4 extends the theory to incorporate semantics by partially interpreting the transaction steps. Here, a high level query language like QUEL0 [HSW73] and a class of query processing strategies is chosen; a set of transaction modification rules are proposed for transforming the schema (i.e. the execution schedule) in this environment. A viable scheduling algorithm is proposed to obtain a highly parallel schema.

2. Model of a Transaction System

In this section we adapt the general model of parallel program schemata developed in [Kell73] to database transaction systems. First section reviews the terminology, followed by the presentation of the formal model of a transaction system. We first review the terminology and then present the formal model of a transaction system and discuss the properties of the model.

2.1. Terminology

The database is viewed as a shared memory DB, consisting of a countably infinite number of cells or data items. Transactions and transaction systems are modelled as schemata, defined over a set of operations. Each transaction step is an operation, and henceforth, we use these two terms synonymously. Associated with each operation t_{ij},(jth step of the ith transaction) are two finite sets $D(t_{ij}) \subseteq DB$ the domain of t_{ij} and $R(t_{ij}) \subseteq DB$, the range of t_{ij}. Intuitively, each operation reads the elements of its domain, performs some computation on them, and writes into the elements of its range. Also associated with t_{ij}, is a terminator, an atomic event (i.e., indivisible and mutually exclusive) that represents the commitment of the operation.

The domain and range of an operation provide purely syntactic information. To express the semantics of the operation (i.e. the actual computation performed by it), an interpretation is required. An interpretation for an operation set defines a universe of values and an initial assignment of values to the database DB; and, for each operation t_{ij}, a set of functions which map $D(t_{ij})$ into $R(t_{ij})$.

2.2 Formal Model

As we are interested initially in developing a model based purely on syntactic information, a schema is defined independent of any specific interpretation. In section 4 we extend this concept to incorporate partially interpreted operations.

A parallel program schema, (or simply a schema) over a transactions system $\mathscr{T}$, is a directed acyclic graph G=(T,E), where T is the set of all transaction steps in $\mathscr{T}$, and the edges in E represent precedence constraints on the transaction steps in T. The schema specifies those steps which may be executed concurrently. An operation can be enabled for execution only after all its predecessors in G have been terminated. So we see from figure 1.1, that the database machine executes the transaction steps in any order, consistent with the precedence constraints of G. So a schema is to be viewed as a set of directives for the database machine.

Given that the database machine has executed a transaction system $\mathcal{T}$, we represent that <u>execution</u> as a directed acyclic graph. Intuitively, the execution graph depicts the order in which the transaction steps of $\mathcal{T}$ were executed. So for every transaction step t_{ij} that preceded another transaction step $t_{k\ell}$, there is an edge $(t_{ij},t_{k\ell})$ in the execution graph X. We observed earlier that any <u>execution X permitted by a schema G</u> satisfies the precedence constraints in G. So, G must be a subgraph of X. Furthermore, note that an execution is a partial order, because some of the transaction steps were executed in parallel.

In particular, we are interested in a special kind of non-parallel execution that imposes a total ordering of transactions and whose graph is the transitive closure of a chain. In this execution, only one transaction step is executed at any time and each transaction is run to completion before the next transaction is started. Such an execution depicts a sequential execution of transactions in a uni-processor environment and is called a serial execution. This is defined as follows: For a transaction system $\mathcal{T}$, and a permutation p of $\{1,2,3,...,n\}$, (where p is viewed as a function), a <u>serial execution</u> corresponding to p, (denoted SX_p), is an execution in which all transaction steps of $T_{p(j)}$ are executed before any transaction step of $T_{p(k)}$ is executed, iff $p(j)<p(k)$. Further, if a schema $G^+ = SX_p$ for some permutation p, then G is called a <u>serial schema</u>.

The serial execution is a non-interleaved execution of transactions. Therefore, it is said to be correct because it preserves database consistency (assuming that each transaction executed by itself, starting from a consistent state leaves the database in a consistent state) [EGLT76]. If we can show that a given execution X is computationally equivalent to some serial execution, then we can guarantee database consistency. First we define the notion of computational equivalence.

2.3. Computational equivalence

To formally define the notion of computational equivalence, we must first define computaton sequences. Properties of a schema (or an execution) can be inferred from the properties of the associated set of computation sequences. This technique was used in parallel program schema theory and we adopt the same technique here. A <u>computation sequence</u> (or <u>comp</u>) for a schema G (or an execution X), is a string $x \in$ HIST such that x is a total order consistent with the partial order defined by the schema G (or the execution X). Intuitively, a comp is a sequence of terminations representing a permissible order in which the effects of the transaction steps could be committed.

For an execution X (or a schema G), the set of computation sequences corresponding to X (or to G) is denoted by COMP(X) (or COMP(G)). If X is an execution permitted by G, then G is a sub-graph of X. Consequently, any comp of X is a comp of G, i.e. COMP(X)$\subseteq$COMP(G). Therefore, COMP(G) represents all possible committing sequences of any execution permitted by G.

2.3.1. Semantic Equivalences: Intuitively, we expect two computation sequences x,y$\in$HIST to be equivalent if for every interpretation, they behave identically; i.e. starting with the same state of the database DB, x assigns the same sequence of values to every data item d in DB as y does. Since this must be true for all interpretations, we can formally define this equivalence using the notion of an Herbrand interpretation [Mann74]. Herbrand interpretation for the dth data item, (denoted $H_d(x)$), is an encoding of the final value of the d^{th} data item after the termination of the computation sequence x, under the Herbrand interpretation. Given two comps: x,y$\in$HIST, they are said to be related by the equivalence relation E_C iff the sequence of values stored in data item d in both cases are the same. This can be formalized by defining a trace for each data item as follows:

$$TRACE_d(x) = C_d({}_0x)C_d({}_1x)C_d({}_2x)...C_d({}_{|x|}x)$$

where

$$C_d({}_ix) = \begin{cases} \lambda & \text{if } d \notin R(x_i) \\ H_d({}_ix) & \text{if } d \in R(x_i), \end{cases}$$

${}_ix$ is that prefix of x whose length is i, x_i is the i^{th} symbol in the string x and λ is the empty string.

Informally, $TRACE_d(x)$ represents a trace of the values stored in data item d during x under Herbrand interpretation. Two comps x,y$\in$HIST are said to be related by the equivalence relation E_C iff for every data item d$\in$DB, $TRACE_d(x) = TRACE_d(y)$. This is denoted by $x \equiv y(E_C)$. Thus we have a notion of computational equivalence relating any two computation sequences of HIST. We would like to infer properties of the schema based on the properties on COMP(G), using this notion of computational equivalence. But it would be difficult to use this semantic definition of the equivalence relation. Algorithmic checking requires a syntactic characterization.

2.3.2. Syntactic equivalence: An equivalence relation E_d that corresponds to E_C is adapted from [Papa79]. Two transaction steps t_{ij} and t_{kl} are said to be conflict free if

$$D(t_{ij}) \cap R(t_{k\ell}) = \phi \text{ and } R(t_{ij}) \cap D(t_{k\ell}) = \phi \text{ and } R(t_{ij}) \cap R(t_{k\ell}) = \phi$$

Intuitively, if two transaction steps are conflict-free then their order of execution relative to each other is immaterial. We define a symmetric relation $\Longleftrightarrow$ as follows. Let $x,y \in HIST. x \Longleftrightarrow y$ iff y is obtained from x by switching two adjacent conflict-free operations. We define the syntactic equivalence relation E_d to be the reflexive-transitive closure of $\Longleftrightarrow$. (Since $\Longleftrightarrow$ is symmetric, E_d is an equivalence relation.) It has been shown in [Kris81] that two comps are equivalent under E_C if and only if they are equivalent under E_d. Equivalence of comps under E_d (and hence under E_C) can be tested in $O(n^2)$ time.

2.4 Properties of a Schema

We use the equivalence relations E_C, E_H to infer properties of a schema G from the properties of COMP(G). This is motivated by the fact that every execution X permitted by G, is associated with a set of computation sequences that is a subset of COMP(G). For instance, if we can show that every computation sequence in COMP(G) is in some sense correct, then every execution permitted by G is associated with computation sequences that are all correct. Two properties proposed in [Kell73] are of interest. These are equivalence and determinacy of schemata.

Given any equivalence relation E on HIST and two schemata G_1 and G_2 on a transaction system $\mathscr{T}$, G_1 and G_2 are said to be E-equivalent, (written $G_1 \equiv G_2(E)$), iff
$\forall x \in COMP(G_1)\ \exists y \in COMP(G_2)\ [x \equiv y(E)]$ and $\forall y \in COMP(G_2)\ \exists x \in COMP(G_1)\ [x \equiv y(E)]$

Intuitively, both the schemata represent sets of computations that are equivalent under E. The equivalence of schemata also implies that their COMPs represent the same equivalence classes.

The other property of interest is the determinacy property of a schema. Given any equivalence relation E on HIST, a schema G is said to be E-determinate if for all $x,y \in COMP(G)$, $x \equiv y(E)$. In other words, COMP(G) is contained in a single E-equivalence class of HIST. Intuitively, E_C-determinacy ensures that all comps produce the same result under any interpretation of the transaction steps. In particular, if G_1 or G_2 is determinate then $G_1 \equiv G_2(E)$ implies both are determinate and are contained in the same equivalence class of HIST.

We are particularly interested (as we shall see in the next section), in detecting whether a schema is determinate. This problem is solved as follows. A schema G is said to be conflict-preserving if for every pair of conflicting transaction steps t_{ij} and $t_{k\ell}$, either $(t_{ij},t_{k\ell})$ or

$(t_{k\ell}, t_{ij})$ is an edge of E^+ (where $G^+ = (V, E^+)$, is the irreflexive, transitive closure of G). Intuitively, conflict preserving implies that all conflicting transaction steps are totally ordered; consequently, they cannot be executed in parallel. Then, as shown in [Kris81], Schema G is conflict preserving iff it is E_C-determinate. Thus, E_C-determinacy and therefore, E_C-equivalence with a determinate schema, can be checked efficiently.

3. Correctness criteria for executions, schemata, and schedulers

The concept of serializability has been used as the correctness criterion in the sequential model of execution [EGLT76,Papa79,BSW79,PBR77]. A given history is said to be serializable iff it is computationally equivalent to some serial history. In this section, this correctness criterion is extended to executions in the parallel model. Recall in the parallel model, the execution of a transactions system by a database machine is represented by a DAG, X = (T,Ex). We first define correct (i.e. serializable) executions. Then, we define correct schema to be that which permits only correct executions. Finally, a correct scheduler is one which presents the database machine (see Figure 1.1) with a correct schema.

3.1 DSR-class

An execution X is said to be DSR-serializable (or-in DSR-class)† iff there exists a serial execution, SX_p such that $X \equiv SX_p$ (E_C). Intuitively, DSR-serializable implies that there is a serial execution SX_p to which every computation sequence in COMP(X) is equivalent. So, irrespective of the specific order of commits that may have occurred in the database, we are guaranteed serializability. Furthermore, every execution X in DSR-class is a determinate execution, as SX_p is determinate (trivially!). Therefore, the recognition problem for DSR-class: given an execution X, is X a member of DSR-class, is polynomially sovable. This can be seen as follows. First check if X is determinate, (as shown in the previous section, check if X is conflict preserving). If not, then X is not in DSR-class; otherwise, determine if for any computation sequence $x \in$ COMP(X), $x \equiv SX_p$ (E_D) for some SX_p, using the algorithm given in [Papa79]. Thus we have shown that there is an efficient solution to the recognition problem for DSR-class.

† In [Kris81], serializability has been extended to SR-class, and is shown that most scheduling problems are intractable. As most schedulers in practice use a proper subset of the DSR-class, we restrict our attention to this class.

3.2 Correctness criteria for a schema

A schema G is said to be DSR-serializable if every execution permitted by G is DSR-serializable. We observed in the previous section that for any execution X permitted by a schema G, COMP(X) must be a subset of COMP(G). Further, that a schema G can itself be viewed as an execution that satisfies all the precedence constraints. It follows directly from these observations that a schema G is DSR-serializable iff G (when viewed as an execution), is a member of DSR-class. As there is an efficient solution for the recognition problem for DSR-class, the schema recognition problem is also polynomially solvable.

3.3 Correct and efficient schedulers

We now turn to the problem of designing correct and efficient schedulers. The scheduler, as shown in figure 1.1, takes as input a transaction system $\mathscr{T}$, and outputs a schema G that is to be used by the database machine. Intuitively we would like to guarantee that every schema produced by the scheduler is correct. If correctness were our only goal, it would be easy to design schedulers. For example, a trivial scheduler that guarantees serializability is one that outputs only a serial schema. This is, however, too restrictive a mechanism for practical value and defeats the very purpose of parallel execution in the database machine. Clearly, therefore, in addition to correctness, there is the requirement of efficiency. We would like to design a scheduler that, for a given transaction system, produces a schema that is the "best" (with respect to some performance criteria) of all the schemata in the DSR-class. In this subsection we discuss the problem of constructing a correct schema that imposes a minimal set of precedence constraints. Lastly we dicuss the problem of efficient scheduling.

3.3.1. Schema for a transaction system: Here we present a sytactic procedure to obtain a serializable schema, corresponding to a given serial execution SX_p. Given a serial history SX_p, we define a serialization graph (abbreviated SR-graph) of SX_p, $G_p = (V_p, E_p)$, where $V_p = T$ and $E_p = \{(x_i,x_j) \mid i<j$ in SX_p and x_i and x_j are not conflict free.$\}$ It follows from the definition of G_p that every comp in COMP(G_p) is E_C-equivalent to SX_p, or in other words that G_p is DSR-serializable. (This is formally proved in [Kris81].) Further it is also shown in [Kris81] that none of the precedence constraints in G_p is unnecessary. That is, if any non-transitive edge of G_p is deleted, then G_p is not DSR-serializable. Therefore, G_p represents a set of necessary and sufficient constraints to assure correctness.

The reader should note that in the above discussion we assume that we are given a serial execution, SX_p to which all the scheduled executions must be equivalent, and we construct a G_p that is both necessary and sufficient. But there are n! such serial executions. In [Kris81] it is shown that the problem of choosing a p that procuces the "best" shcema, for all interesting optimality measures is intractable. As the motivation for finding the "best" schema is efficiency, (but finding the "best" schema is a hard problem), we present in the next section a method for transforming a given schema to a "better" schema, by partially interpreting the transactions. We also show that the transformed schema will have better processor utilization characteristics.

4. Scheduler Using Partially Interpreted Transactions

In figure 1.1 we presented a model in which the scheduler was defined to output a schema G for the transaction system $\mathscr{T}$ using only syntactic information, namely readsets and writesets. Thus, the two parameters of the model are the restriction on the output and the level of information in the input. The assumption that the scheduler outputs a schema G, restricts the scheduler's domain of optimization to the DSR-class and this optimization problem was shown to be intractable in [Kris81]. It is also shown there that relaxing this assumption (by increasing the domain of optimization to SR-class) does not simplify the problem. In this section, we relax the other parameter of the model namely the level of information in the input to the scheduler. We had assumed that the scheduler uses purely syntactic information (i.e. readsets and writesets of the transaction steps) for input to the algorithm. We now redefine the scheduling problem by modifying the input to the scheduler. This section shows how semantic information can be used to make the problem simpler. This approach is motivated by a result from [KP79] that a scheduler using semantics is less restrictive than a scheduler using syntactic information alone.

Our approach is to construct a schema that is serializable and then to optimize it using semantic information. Semantic information is incorporated by partially interpreting the operations. Thus, each transaction step is assumed to be a statement in a high level query language. We develop a set of rules for transaction modification. Each modification transforms a schema (for a given transaction system) into an equivalent schema which is better according to some performance measure. Initially, this measure is assumed to be the diameter of the schema (i.e. the longest path in the schema). This measure is significant because it represents the response time, assuming an unlimited number of processors and unit cost per transaction step. In practice, however, the number of processors in the database machine are limited and the transaction steps have varying costs. Hence, in section 4.5 we present an algorithm that takes both of these factors into account.

4.1 Environment

The database is assumed to consist of a single relation R. In a database with more than one relation, R can be thought of as the product of all the relations. Each tuple of R has a unique identifier (TID) and corresponds to a data item of the memory. Transaction steps are assumed to be statements written in some high-level relational calculus based language, such as QUEL0 [HSW75]. As the exact syntax is not of concern here, we represent each type of statement in the following manner:

1. MODIFY: MOD($targ_m$, q_m, R)
2. INSERT: INS($targ_i$, q_i, R)
3. DELETE: DEL(q_d, R)

In each statement the qualification q is a predicate that selects a subset of the relation R to be the operand of the operation; the target list, targ, defines the computations to be performed on the operand. In QUEL0, the qualification is a boolean combination of the form <term><op><term>, where a term is an attribute, a constant, or an arithmetic function (e.g. +,*) of other terms; and <op> is an arithmetic comparison operator (e.g. $=, \leq$). The target list is a list of (attribute, term) pairs. Observe that we have not included a RETRIEVE statement. A retrieval can always be treated as the insertion of new tuples into a part of the database. This is achieved by including the user's terminal to be part of the database [PBR77,BSW79].

Before we formally present the transformation, let us consider an example that illustrates the goal of the transformation. The original schema (figure 4.1) consists of three statements to be executed sequentially. Our objective is to transform this schema into the parallel schema of figure 4.1. For the two schemata to be equivalent, the transformed schema must be determinate. This implies that the transformed schema must be conflict preserving. The reader will readily discern that in the form shown in figure 4.1, TS is not conflict preserving. In the sequel, we describe a query processing strategy under which TS is in fact conflict preserving.

Without loss of generality we assume that a target list specifies every attribute in R. This assumption and our earlier assumption that the database contains only one relation do not restrict the applicability of the theory; they are made merely to simplify the formalism.

Let Q_x be the set of tuples selected by qualification q_x; and $TARG_x$ be a function corresponding to target list $targ_x$, such that, $TARG_x(Q_x)$ is the set of tuples generated by evaluating $targ_x$ on the selected tuples Q_x. For a single tuple t, $TARG_x(t)$ is defined in the obvious manner. Figure 4.2 tabulates the effect of each operation.

ORIGINAL SCHEMA

t_1: DEL ((age=16), R)

t_2: MOD ((age>=16), (salary=salary+1k), R)

t_3: INS ((salary=25k), (name=name), R)

$$\begin{tikzcd} t_1 \arrow[d] \\ t_2 \arrow[d] \\ t_3 \end{tikzcd}$$

TRANSFORMED SCHEMA

$$\begin{tikzcd} t_1 & t_2 & t_3 \end{tikzcd}$$

t_1: DEL ((age=16), R)
t_2: MOD ((age>16), (salary=salary+1k), R)
t_3: INS ((((salary+1k>=25k) and (age>16)) or ((salary>=25k) and (age<16))),
(name=name), R)

Figure 4.1 An example of the transformation.

The following table lists the effects of each operation. R and R' are the relations before and after the execution. Also let $Q_x = \{t \mid t \in R$ and t satisfies $q_x\}$. $TARG_x$ is the function corresponding to $targ_x$. R_{OS} and R_{TS} are the resulting relations after an execution permitted by OS and TS respectively.

MOD ($targ_m$, q_m, R)	$R' = (R-Q_m) \cup TARG_m(Q_m)$
INS ($targ_i$, q_i, R)	$R' = R \cup TARG_i(Q_i)$
DEL (q_d, R)	$R' = R-Q_d$

Figure 4.2 Effects of the operations.

We consider a general class of query processing strategies in which each operation is performed in two steps (as shown in figure 4.3); a qualification step q that selects the tuple ids (TIDs) of the R-tuples satisfying the qualification q; then, an effect step e that performs the operation specified by the target list on the tuples whose TID's were selected by q. For this class of query processing strategies, the time taken to evaluate the qualification (step q) is likely to be much greater than the time taken for step e. Also, step q does not update the database. These two observations make it a prime candidate for execution in parallel with other qualification steps (appropriately modified, as described below).

Consider an ordered pair of operations with the precedence constraints shown in figure 4.4 Suppose this pair can be modified to an equivalent schema (shown in figure 4.5) such that Te_1, Te_2, Tq_1, and Tq_2 can be syntactically determined. Then, the modified operation pair has smaller diameter and so is better according to our measure of parallelism.

In developing the transformation, we use a canonical representation of the update operation (see figure 4.6). In this representation qd and qi are the qualifications that select the tuples to be deleted and inserted respectively; ed and ei are the corresponding effects. It is obvious that the insert and delete operations can be modelled by choosing appropriate q's. To model modify operations, we require two assumptions. (1) The terms in a target list are constants. (2) every insert (ei) creates a new tuple for relation R. Thus, every new tuple that is inserted into R is assigned a new TID. Under these assumptions, a modify operation is modelled as the deletion of old tuples and the insertion of new (modified) tuples with new TID's. These assumptions assure the conflict preserving property of the transformed schema. We argue in section 4.3 that an implementation can infact relax these two assumptions, and still ensure the conflict preserving property.

With this canonical model of an operation, we define an elementary schema called an <u>operation</u> <u>pair</u> as shown in figure 4.7 For this elementary schema, we derive in the next subsection, an equivalent schema with greater potential for parallelism (i.e. smaller diameter).

4.2. Transformation of an Operation Pair in Series

Given a schema OS, (original schema), for an operation pair in series, shown in figure 4.7, we can find an equivalent schema TS, (transformed schema), as shown in figure 4.8 In this transformed schema Ted_1, Ted_2, Tei_1, Tei_2, and $Tei_{2/1}$ are the transformed operations whose readsets/writesets and target list functions are defined in figure 4.10. This transformed schema evaluates the qualifications based on the original relation R, to get TQd_1, TQi_1, $TQd_{2/1}$, $TQi_{2/1}$, TQi_2 and TQd_2. (These symbols are defined in figure 4.9). Using

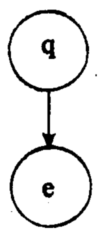


Fig 4.3 Model of an operation

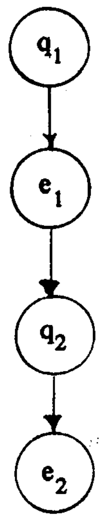


Fig 4.4 Schema for an ordered pair of operations.

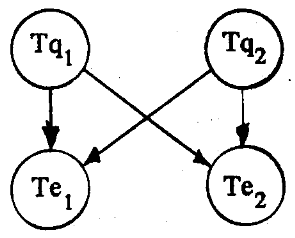


Fig 4.5 Modified schema for an ordered pair of operations.

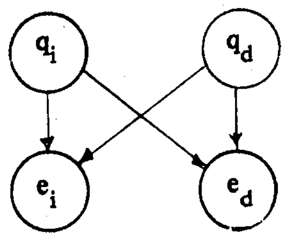


Fig 4.6 Canonical representation of an operation graph.

these sets of TIDs, the readsets and writesets of the effect steps (as shown in 4.10) can be determined.

The algorithm for the transformation is as follows. This algorithm takes input parameters from the original schema and outputs the parameters of the transformed schema.

Input:	oqi_1, oqi_2, oqd_1, oqd_2, $TARG_{i1}$, $TARG_{i2}$.
Output:	tqi_1 ,tqi_2, tqd_1, tqd_2, $tqi_{2/1}$, $tqd_{2/1}$, $TARG_{i1}$, $TARG_{i2}$, $TARG_{i2/1}$.

It is easy to see that each of tqi_1, tqi_2, tqd_1, tqd_2, $TARG_{i1}$ and $TARG_{i2}$ is identical to the corresponding input parameter. To calculate $tqi_{2/1}$ and $tqd_{2/1}$, we note the following. Each qualification can be viewed as a predicate calculus formula. Obtain a new formula $tqd_{2/1}$ from oqd_2 by substituting for each attribute the corresponding term given in $TARG_{i1.}$ It is straightforward to see that the set of tuples selected by this modified formula, $tqd_{2/1}$, is, infact the set, $TQd_{2/1}$ defined in figure 4.10. Similarly, we can obtain $tqi_{2/1}$ from oqi_2 by substituting from $TARG_{i1}$. We can also obtain $TARG_{i2/1}$ by substitution; as $TARG_{i2/1}$ is a composition of functions, we can substitute for every attribute in $TARG_{i2}$ the corresponding term in $TARG_{i1}$ to get the required target list.

Thus, we have described a syntactic procedure for transforming a serial schema to a more parallel schema. To explain the intuition behind this (seemingly complicated) transformation, four examples are presented below. In the subsequent discussion the reader will find it helpful to keep in mind the terminology given in figures 4.9 and 4.10, along with the observations recapitulated below.

1. The two schemata are E_C-equivalent if the net effect in both the cases is to delete the same set of tuples and insert the same set of tuples.
2. The set of tuples deleted in each schema is the union of the writesets of the delete (effect) steps.
3. The set of tuples inserted in each schema is the union of the writesets of the insert (effect) steps. Each tuple written by an insert (effect) step has new a TID and the set of all tuples written is determined by the readset.

<u>4.2.1.</u> <u>Case</u> <u>1:Delete-delete</u> <u>pair</u>: A pair of delete operation in series is shown in figure 4.11 and the corresponding transformed schema is shown in figure 4.12 For the two schemata to be E_C-equivalent, we have to guarantee that the net effect in both cases is to delete the same set of tuples from the relation R. In the schema given in figure 4.11 the set of tuples deleted is the union of the set of tuples deleted by the two steps (i.e. OQd_1 and OQd_2).The set of tuples deleted by the transformed schema (given in 4.12) is the union of the tuples deleted in the two

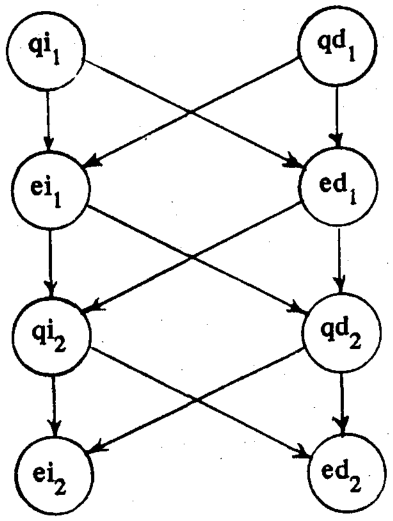


Fig 4.7 Schema (OS) for an operation pair.

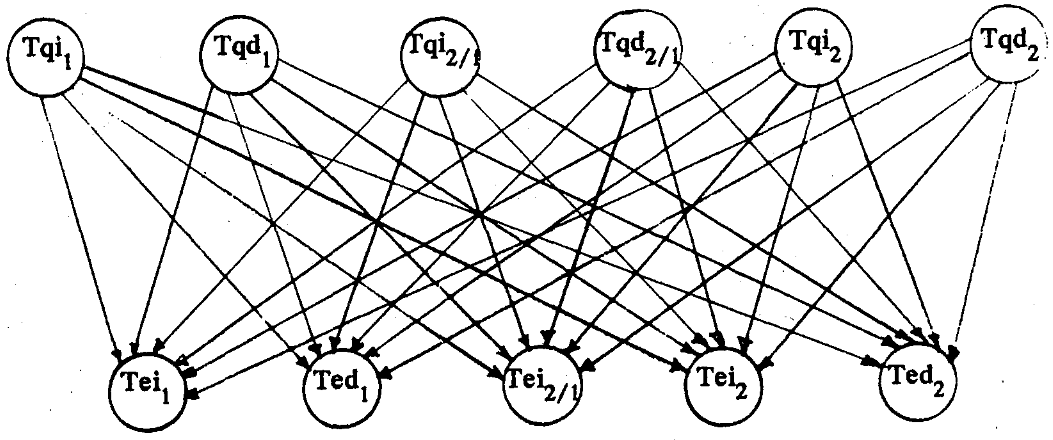


Fig 4.8 Transformed schema (TS) for an operation pair.

QUALIFICATION STEPS

	stepname	associated prediacte	read-set	write-set
orig. schema				
	Oqi_1	oqi_1	OQi_1	ϕ
	Oqi_2	oqi_2	OQi_2	ϕ
	Oqd_1	oqd_1	OQd_1	ϕ
	Oqd_2	oqd_2	OQd_1	ϕ
trans. schema				
	Tqi_1	tqi_1	TQi_1	ϕ
	Tqi_2	tqi_2	TQi_2	ϕ
	$Tqi_{2/1}$	$tqi_{2/1}$	$TQi_{2/1}$	ϕ
	$Tqd_{2/1}$	$tqi_{2/1}$	$TQi_{2/1}$	ϕ
	Tqd_1	tqd_1	TQd_1	ϕ
	Tqd_2	tqd_2	TQd_2	ϕ

EFFECT STEPS

	stepname	associated target list	read-set	write-set
orig. schema				
	Oei_1	$TARG_{i1}$	OEi_1	*
	Oei_2	$TARG_{i2}$	OEi_2	*
	Oed_1	-	ϕ	OEd_2
	Oed_2	-	ϕ	OEd_2
trans. schema				
	Tei_1	$TARG_{i1}$	TEi_1	*
	Tei_2	$TARG_{i2}$	TEi_2	*
	$Tei_{2/1}$	$TARG_{i2/1}$	$TEi_{2/1}$	*
	Ted_1	-	ϕ	TEd_1
	Ted_2	-	ϕ	TEd_2

* corresponding to the readsets there is a writeset of new TID's

Figure 4.9 List of symbols and their associated meanings.

Tuples selected by q's in the original schema:

$OQi_1 = \{t \mid t \in R \text{ and } t \text{ satisfies } oqi_1\}$
$OQd_1 = \{t \mid t \in R \text{ and } t \text{ satisfies } oqd_1\}$
$OQi_2 = \{t \mid t \in (R - OQd_1) \cup OQi_1 \text{ and } t \text{ satisfies } oqi_{i2}$
$OQd_2 = \{t \mid t \in (R - OQd_1) \cup OQi_1 \text{ and } t \text{ satisfies } oqd_{i2}$

Read/write sets for e's in the original schema:

$OEi_1 = OQi_1$	$OEi_2 = OQi_2$
$OEd_1 = OQd_1$	$OEd_2 = OQd_2$

Tuples selected by Tq's in the transformed schema:

$TQi_k = \{t \mid t \in R \text{ and } t \text{ satisfies } tqi_k\} \quad k = 1,2$
$TQd_k = \{t \mid t \in R \text{ and } t \text{ satisfies } tqd_k\} \quad k = 1,2$
$TQi_{2/1} = \{t \mid t \in R \text{ and } TARG_{i1}(t) \text{ satisfies } tqi_2\}$
$TQd_{2/1} = \{t \mid t \in R \text{ and } TARG_{i1}(t) \text{ satisfies } tqd_2\}$

Read/write sets for Te's in the transformed schema:

$TEi_1 = TQi_1 - TQd_{2/1}$	$TEd_1 = TQd_1$
$TEi_2 = TQi_2 - TQd_1$	$TEd_2 = TQd_2 - TQd_1$
$TEi_{2/1} = TQi_1 \cap TQi_{2/1}$	

Target lists for the Tei's in the transformed schema:

The target lists for TEi_1 and TEi_2 are the same as in the original schema. The target list for $Tei_{2/1}$ is the composition of $TARG_{i1}$ with $TARG_{i2}$.

Figure 4.10 OE's, OQ's, TE's, and TQ's

steps Ted_1 and Ted_2. The set of tuples deleted by Ted_1 (i.e. TEd_1, which is the same as TQd_1) is the same as OQd_1. But the set of tuples selected by Tqd_2 is not necessarily the same as the set selected by Oqd_2, as the step Tqd_2 might have selected some of the tuples in R that are deleted in step Oed_1. So the writeset for the transformed step is TEd_2=TQd_2-TQd_1. All the tuples selected by oqd_2 will also be selected by tqd_2; and hence, will also be in the set TQd_2; so the same set of tuples are deleted in both cases.

4.2.2. Case 2:Delete-Insert Pair: A pair of operations, consisting of a delete followed by an insert is shown in figure 4.13 and the corresponding transformed schema is shown in figure 4.14 In the original schema, the set of tuples OQd_1 is deleted and the new set of tuples $TARG_{i2}$ (OQi_2) is inserted. We have to guarantee that the same sets of tuples are deleted and inserted by both schemata. It is easy to see that TQd_1=OQd_1 so the same set of tuples is deleted. But tqi_2 selects all the tuples selected by oqi_2, and some more of the tuples in R that are deleted by the step Oed_1. Hence, the writeset for Tei_2 is TQi_2-TQd_1. Thus the sets of tuples inserted in the two schemata are the same; i.e. $TARG_{i2}(OQi_2)=TARG_{i2}(TQi_2-TQd_1)$.

4.2.3. Case 3:Insert-Delete Pair: A pair of operations consisting of an insert followed by a delete is shown in figure 4.15, and the corresponding transformed schema is shown in figure 4.16 We have to guarantee that the same sets of tuples are inserted and deleted in both the schemata. First we observe that every tuple selected by the step Oqi_1 is also selected by Tqi_1. But some of the tuples inserted by the step Oei_1, (i.e. in the set $TARG_{i1}(OE_{i1})$), are selected by step Oqd_2 and subsequently deleted by the Oed_2. This set of (deleted) tuples is $TARG_{i1}$ ($TQd_{2/1}$); as $TQd_{2/1}$ is the set of tuples, $t \epsilon R$ such that $TARG_{i1}(t)$ satisfies oqd_2. Therefore, the net effect of the insertion operation is to insert the set of tuples $TARG_{i1}(TQi_1-TQd_{2/1})$. Thus, the two schemata insert the same set of tuples.

The set of tuples in the original relation R that is selected by oqd_2 (i.e. OQd_2) is also selected by tqd_2; i.e. $OQd_2 \supseteq TQd_2$. The extra tuples in OQd_2 are those inserted by step Oei_1 and these tuples are not inserted at all in the transformed schema. Therefore, the net effect is that the same set of tuples is deleted in both the schemata.

4.2.4. Case 4:Insert-Insert Pair: A pair of insert operations is shown in figure 4.17, and the corresponding transformed schema is shown in figure 4.18 To show the equivalence of the two schemata, we need to guarantee that the same set of tuples will be inserted in both cases. We first observe that the set of tuples selected by oqi_1 is also selected by tqi_1; consequently the set of tuples inserted by Oei_1 is inserted by Tei_1. The set of tuples selected by oqi_2 can be partitioned into two sets: those tuples that belong to the original relation R and those that are

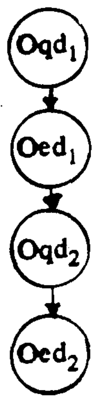
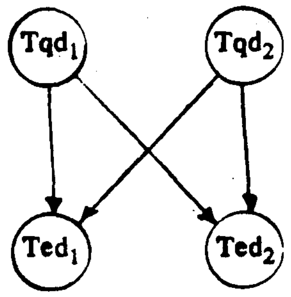


Figure 4.11: Delete-delete pair



$TEd_1 = TQd_1$, $TEd_2 = TQd_2 - TQd_1$

Figure 4.12: Transformed delete-delete pair

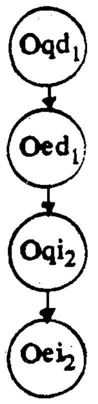
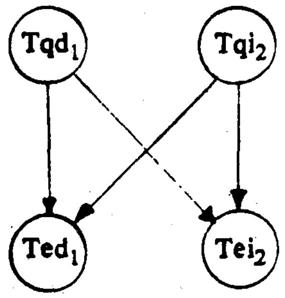


Figure 4.13: Delete-insert pair



$TEd_1 = TQd_1$, $TEi_2 = TQi_2 - TQd_1$

Figure 4.14: Transformed delete-insert pair

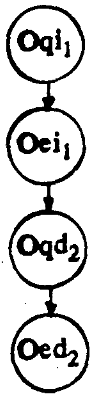
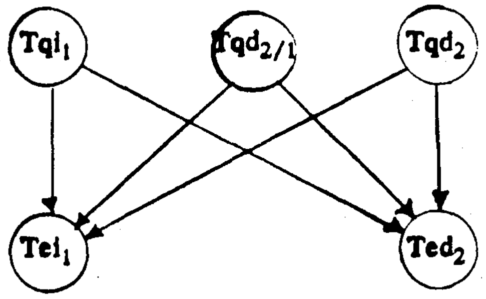


Figure 4.15: Insert-delete pair



$TEd_2 = TQd_2$, $TEi_1 = TQi_1 - TQd_{2/1}$

Figure 4.16: Transformed insert-delete pair

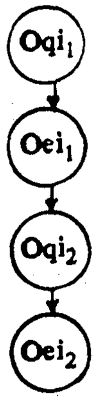
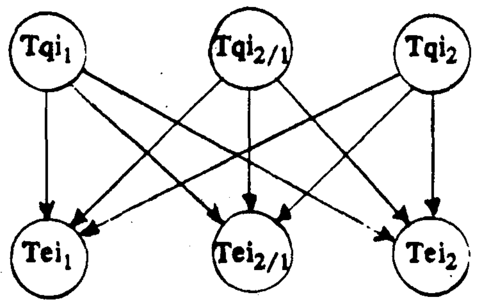


Figure 4.17: Insert-insert pair



$TEi_1 = TQi_1$, $TEi_2 = TQi_2$,
$TEi_{2/1} = TQi_1 \cup TQi_{2/1}$

Figure 4.18: Transformed insert-insert pair

inserted by the step Oe_{i1}. The former set of tuples is exactly the set TQi_2 and the latter is the same as the set of tuples selected by step $Tqi_{2/1}$. Therefore the set of tuples inserted by Tei_2 is $TARG_{i2}$ (TQi_2), and that inserted by $Tei_{2/1}$ is $TARG_{i2}$ $(TARG_{i1}(TQi_{2/1}))$. So, it is easy to see that the same set of tuples is inserted by both the schemata.

4.2.5. Informal proof of the transformation: In this section we present an informal proof of the transformation. Given a schema OS, (original schema), for an operation pair in series, shown in figure 4.7, we can find an equivalent schema TS, (transformed schema), as shown in figure 4.8 The validity of the transformation is proved formally in [Kris81]. Here, we attempt an intuitive justification. (The reader will find it helpful in following this explanation if he/she refers to the example in figure 4.19) $TEd_1 = TQd_1$ is seen from cases 1 and 2. The justification for $TEi_1 = TQi_1 - TQd_{2/1}$ is given in case 3 and is not contradicted by case 4. $TEi_2 = TQi_2 - TQd_1$ is shown in case 2 and is not contradicted by case 4. The expression for TEd_2 is due to case 1 and is not contradicted by case 3. Lastly, the validity of the expression for $TEi_{2/1}$ is given in case 4. Thus the transformed schema performs the same deletions and insertions on R as the original schema. And, significantly, the transformed schema allows greater parallelism amongst the qualification steps, which, by our assumption, are more time consuming than the effect steps.

4.3 Implementation considerations

The reader may have noticed that the figure 4.19 does not satisfy the two assumptions that were made in developing the above theory. These were:

1. Only constants were allowed as terms in a target list;

2. All inserted tuples had new TID's.

Both these assumptions were used to ensure that the schema for the transaction system $\mathscr{T}$ was conflict preserving. We show here that we can relax these assumptions by adopting the following implementation. Associated with each tuple are two flags: a delete flag and an insert flag, whose use is described below. Step ed sets the delete flags of all those tuples which are selected for deletion but do not have their insert flags set. These tuples are then deleted during the execution of the next transaction system. Step ei always creates new tuples for insertion and sets their insert flags. If ei is part of a modify operation, then it reuses the old TID; otherwise, it generates a new TID. (This implies that at any point in time, there might be two versions of a tuple both having the same TID). In reading an existing tuple x, in order to compute a tuple y for insertion, ei uses that version x which does not have its insert

TID	AGE	SALARY		
1	17	10K		qd_1 = (salary = 10K)
2	16	10K		qi_1 =(age = 16 and salary = 30K)
3	18	30K		qd_2 =(age = 16)
4	19	30K	Tuples in	qi_2 =(age = 17)
5	17	20K	the old	$TARG_{i1}$ = (salary = 25K)
6	17	30K	relation	$TARG_{i2}$ = (salary = 27K)
7	16	30K		
8	16	15K		
9	16	25K		
10	17	25K	Added	
11	18	25K	Tuples	
12	17	27K		R = {1, 2, 3, 4, 5, 6, 7, 8}
13	17	27K		R_{os} {3, 4, 5, 6, 10, 11, 12, 13, 14}

OQd_1 = {1,2} OQd_2 = {7, 8, 9}OQi_1 = {3, 6, 7}OQi_2 = {5, 6, 10}

OEd_1 = {1,2} OEd_2 = {7, 8, 9}OEi_1 = {9,10,11}OEi_2 = {12,13,14}

TQd_1 = {1,2} TQd_2 = {2,7,8,9}TQi_1 = {3, 6, 7}TQi_2 = {1, 5, 6}

$TQi_{2/1}$ = {7} $TQd_{2/1}$ = {9}

TEd_1 = {1,2} TEd_2 = {7, 8} TEi_1 = {3, 6} TEi_2 = {5, 6}

$TEi_{2/1}$ = {2}

Fig. 4.19 An example of the transformation of an operation pair

flag set. Further, the insert flag is reset in these tuples at the beginning of the next transaction system.

So any step ed writes only delete flags, and therefore the writesets of ed's are pairwise disjoint with the read/write sets of ei's; we already know from the transformation that the writesets of ed's are mutually disjoint. The insert flag ensures that the readsets of ei's are pairwise disjoint with the writesets of ei's. Creating new tuples ensures that the writesets of ei's are mutually disjoint. Therefore, the schema is conflict preserving and the theory of section 4-2 still holds.

Before we proceed to generalize this transformation, we present the (airline reservation) example cited in the introduction. Two (simultaneous) requests for seats on flight TWA101 of August 11, 1981 is made. Each request is modelled as a modify operation given in figure 4.20 As both of them are updating the no__seats, they must be executed sequentially; so this set of two requests is modelled as a pair of operations in series. The canonical form of the modify operation is the deletion of the old tuple and the insertion of the new tuple (with no__seats decremented by 1). The parameters of the original and the transformed schema are shown in figure 4.20. Intuitively, the transformed schema does the following: If there are two seats available, the old tuple is deleted and a new tuple (with no__seats = no__seats-2) is inserted; if there is only one seat available then the old tuple is deleted and a new tuple (with no__seats decremented by 1) is inserted; lastly, if there are no seats available then no change is made. This is achieved by computing the appropriate read/write sets dynamically. If there are two seats available then we see that TEd_1 and $TEi_{2/1}$ are the only nonempty sets and the target function for $Tei_{2/1}$ is to decrement no__seats by 2. If there is only one seat or no seats at all then appropriately the correct read/write sets are calculated.

4.4. Generalized transformation

We have shown how to transform a pair of operations into a schema having greater parallelism. The transformed schema is not quite in the canonical form of an operation (figure 4.6) because the set of effect steps has more than one target list. Thus, if this operation pair is part of a bigger schema and we want it to participate in further transformation with successor nodes, the transformation of section 4.2 cannot be directly used. We now show how to generalize the transformation. First we generalize the concept of an operation pair. This generalization is shown in figure 4.21 The first stage has k insert nodes and ℓ delete nodes. The second stage has m-k insert nodes and n- ℓ delete nodes. The transformation and the associated read/write sets are given in figure 4.22 The intuitive justification for the read/write

Reservation request:

We have two requests for reservation on flight TWA101 on the 11 August, 1981. One such request is shown below.

Mod(((flight=TWA101)and(date=081181)and(no__seats>0)), (no__seats=no__seats-1),R)

This is modelled as two such operations in series.

Original schema:

Schema is the graph shown in 4.7. The parameters of the schema are shown below.

$oqi_1 = oqi_2 = oqd_1 = oqd_2 =$ ((flight = TWA101)and(no__seats>0))
$TARG_{i1} = TARG_{i2} =$ (no__seats = no__seats−1)

Transformed Schema:

Schema is the graph shown in figure 4.8. The parameters of the schema are given below.

$tqi_1 = tqi_2 = tqd_1 = tqd_2 =$ ((flight = TWA101)and(no__seats>0))
$tqd_{2/1} = tqi_{2/1} =$ ((flight = TWA101)and(no__seats − 1>0))

Using the above predicates, we can calculate the read/write sets as follows (assuming that there are two seats available).

$TEi_1 = TEi_2 = TEd_1 = TEd_2 = \phi$
$TEd_1 = TEi_{2/1} =$ The tuple for flight TWA101
$TARG_{i2/1} =$ (no__seats = no__seats − 1 − 1)

The reader can easily verify that the transformation is correct for the case when there is only one seat available.

Figure 4.20 Airline reservation example.

sets is the same as before. TEd_a, $a = 1,2....\ell$, are the same as OEd_a since both TQd_a and OQd_a select tuples from the original state of R. TEi_a, a=1,2...k, is the set of tuples in TQi_a and not in $TQd_{b/a}$ for any $b \in \{\ell + 1, \ell + 2,....n\}$, because the tuples in $TQd_{b/a}$ are to be subsequently deleted in the step Oed_b of the second stage of OS. Hence, those tuples which are inserted and subsequently deleted in OS are not inserted at all in TS. Consequently, TEd_a, $a = \ell + 1, \ell + 2,....n$, are the sets of tuples to be deleted from R and contain only those tuples which are not already deleted by the first stage. TEi_a, a=k+1,k+2,...m, are the tuples in R from which the steps Oei_{k+1}, Oei_{k+2},....Oei_m created new tuples. So TEi_a consists of only those tuples which were not deleted in the first stage. As before, $TEd_{a/b}$, a = k + 1,......m ; $b = \ell + 1$,.,n, is the set of tuples that were inserted in the second stage based on tuples that were inserted in the first stage. Thus the transformed schema performs the same insertions and deletions as the original schema, but has greater parallelism. Furthermore, the transformed schema is in the generalized canonical form of figure 4.21, and so can be used in subsequent transformation.

4.5 Algorithm for scheduling using transformation

Until now we used the diameter of the schema to be the performance measure. This was justified on the basis that we assume that there are unlimited number of processors and each transaction step takes unit time. In this section we relax these two assumptions.

First let us relax the infinite processors assumption. Suppose that the database machine has k processors. Once the scheduler has constructed a schema representing minimal precedence constraints, it now remains to assign available processors to execute the nodes of the DAG. To maximize processor utilization, it is important to ensure that at every point in time, as many of the k processors as possible are busy. We shall show how to transform the schema to meet this objective.

We have to select k nodes to execute on the k processors. Assuming that each node takes unit time to execute, all k nodes will complete at the same time. At some point in this process, if there are m<k nodes ready for execution, then we can use the generalized transformation developed above to obtain k nodes in parallel. To do this, let n be the number of nodes that can be enabled for execution after the m nodes have been executed. From these n nodes choose n'= minimum(n,k-m) nodes. Then we can view the DAG as shown in figure 4.23 We see that there are no edges from level 1 to level 0, and from level 2 to either level 1 or 0. To this graph, add edges between every node at level 0 to every node at level 1 (this is not necessarily how the algorithm will be implemented). It is obvious that the graph is still

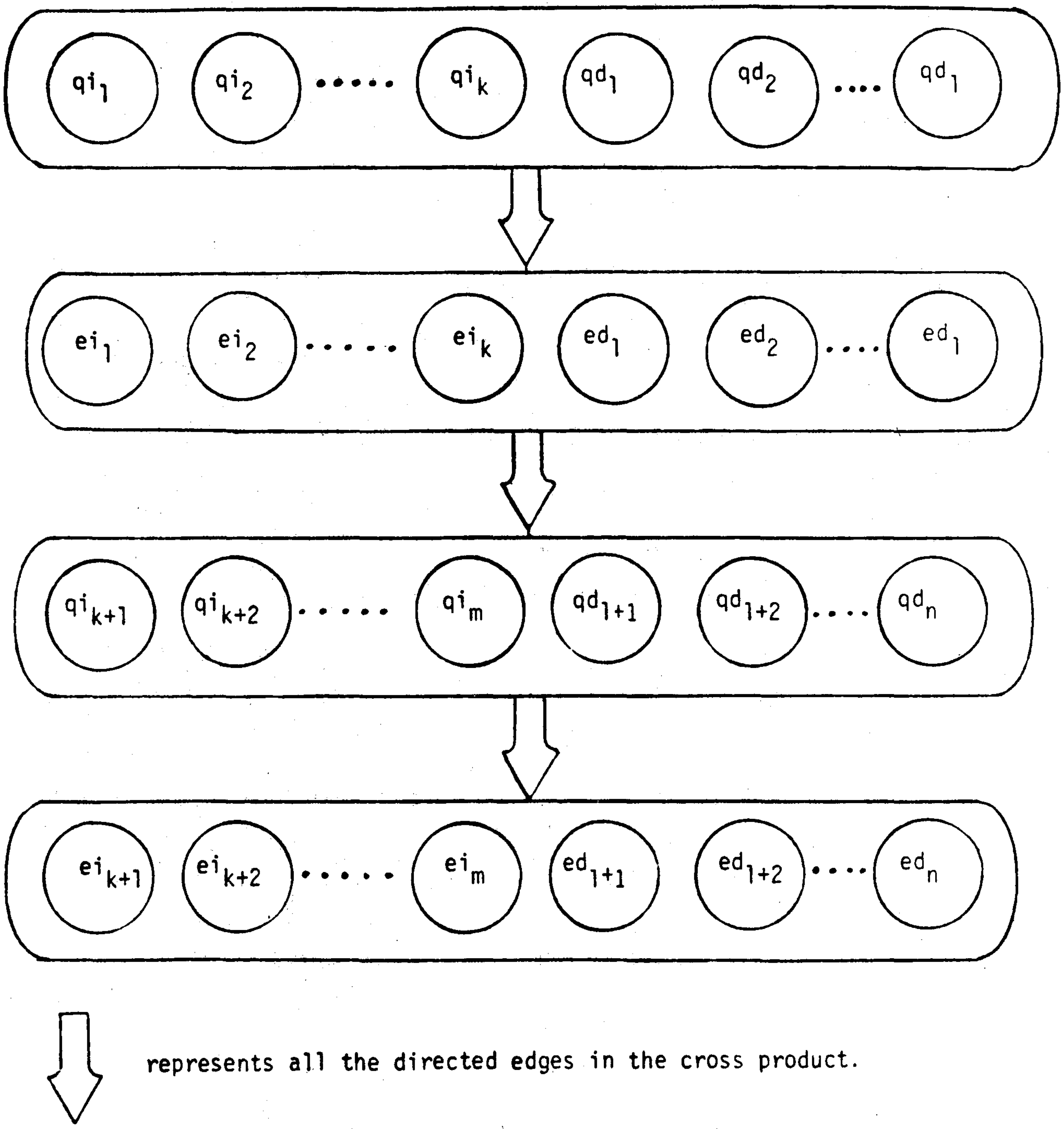
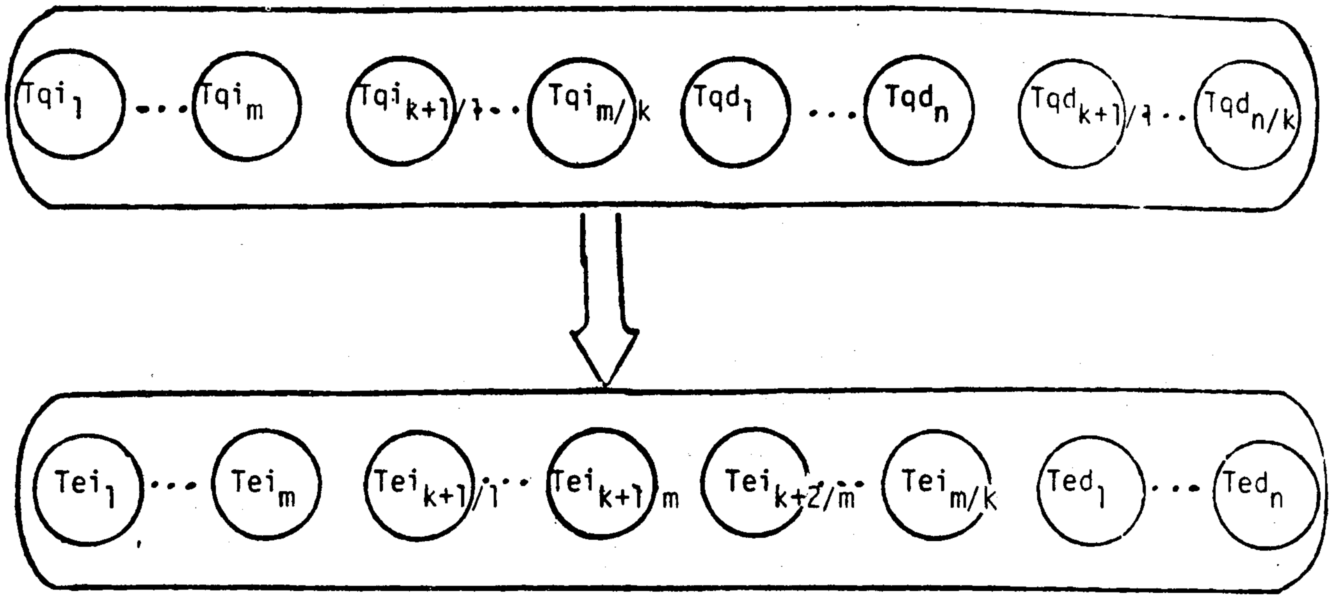


Fig. 4.21 Schema for generalized operation pair



Tqi_1 ... Tqi_m $Tqi_{k+1/1}$ ... $Tqi_{m/k}$ Tqd_1 ... Tqd_n $Tqd_{k+1/1}$ ... $Tqd_{n/k}$
Tei_1 ... Tei_m $Tei_{k+1/1}$ ... $Tei_{k+1/m}$ $Tei_{k+2/m}$ ... $Tei_{m/k}$ Ted_1 ... Ted_n

$TEi_a = TQi_a - \bigcup_{b=1+1}^{n} TQd_{b/a}$ $\quad a = 1,2,\ldots k$

$TEd_a = TQd_a$ $\quad a = 1,2,,,,1$

$TEi_a = TQi_a - \bigcup_{b=1}^{1} TQd_b$ $\quad a = k+1,k+2,\ldots m$

$TEd_a = TQd_a - \bigcup_{b=1}^{1} TQd_b$ $\quad a = 1+1,1+2,\ldots,n$

$TEi_{a/b} = TQi_b \cap TQi_{a/b}$ $\quad a = k+1,k+2,\ldots,m$
$\quad b = 1,2,\ldots,k$

Fig. 4.22 Transformed schema for generalized operation pair along with the read/write sets.

acyclic and the added edges do not contradict any existing precedence constraints. Now the set of nodes at level 0 and level 1 conforms to the generalized canonical form of figure 4.21 So we can apply the generalized transformation to get m+n' nodes in parallel. The transformed graph is shown in figure 4.21 If m+n'<k, then this process of transformation can be repeated until k parallel nodes are available.

Thus, these transformation can be used repetitively to reduce the diameter of the schema. Infact, the diameter can be reduced to 2 for any schema. But it is clear that reduction in the diameter may not be without cost; the number of nodes increases and so does the complexity of each node. It might be more appropriate to pick an optimality measure that takes into account the processing costs of the nodes (instead of the unit cost per node that we have assumed so far). Several cost measures for query processing have been proposed in the literature [HY79,Yao79]. These are based on physical parameters such as file sizes, attribute selectivities, storage and access methods. Given any cost measure that imposes a total ordering on the set of schemata, we apply the schema transformation described in this section only if it is beneficial to do so, i.e. only if the estimated cost of the transformation is less than that of the original schema. For example, let the number of disk accesses be a true estimator of time taken by an operation. Also, let us assume that we have a technique for calculating the number of accesses required by an operation; i.e. that, we can estimate the time required for each operation. Using this estimate we can now modify the algorithm as follows: Once again let the original DAG be viewed as shown in figure 4.23. In this figure, let

τ_1 = time estimated for the longest operation among the m nodes.

τ_2 = time estimated for the longest operation among the n nodes.

τ_3 = time estimated for the rest of the schema.

So the total time estimated for the original schema is $\tau_1 + \tau_2 + \tau_3$. We apply the transformation to get k>=m+n'>m nodes in parallel, only if the longest operation in the resulting set of (n'+m) nodes (that are ready for execution), takes no more than τ_1 units of time. Therefore, the total time estimated for the transformed schema to execute is $\tau_1 + \tau_3 < \tau_1 + \tau_2 + \tau_3$. Hence, this algorithm transforms the schema only if it can reduce the response time of a transaction system. Thus, it is shown that we can devise a practical algorithm to schedule the transactions efficiently and attain high processor utilization.

5. CONCLUSION

In this paper, we developed a parallel program schema model of transaction systems for parallel database machines. The concept of serializability, which is generally accepted as the

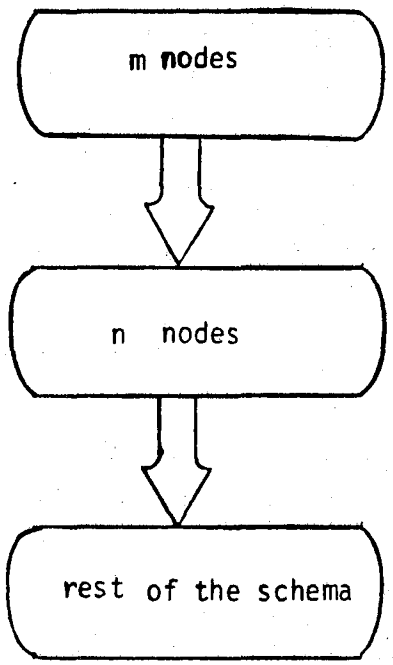


Fig. 4.23 Original DAG

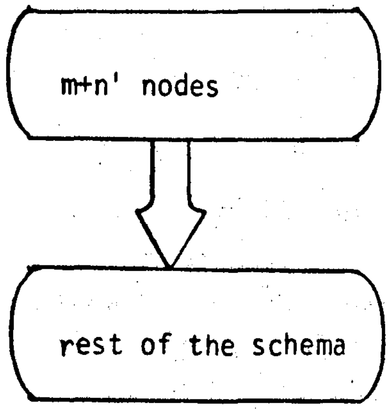


Fig. 4.24 Transformed DAG

correctness criterion in the existing concurrency control theory for the sequential model, was extended to our model. We proposed a two-step technique for producing correct and highly parallel schedules: first, obtain a schema that imposes a minimal set of precedence constraints on correct executions; then, transform the schema using semantic information to increase parallelism. Although the model developed in this paper is theoretical, we believe it to be of practical utility -- the proposed scheduling technique can be applied to any MIMD machine such as DIRECT (DeW78).

Several interesting performance related questions may be posed here. We described the scheduler as a single, centralized process. Will this become a bottleneck? Alternatively, given ample resources and the parallelism inherent in the system, will it be beneficial to partition the system and distribute the scheduling activity over several processes. Our theory is independent of whether the scheduler is centralized or distributed. Further, we have implied a batched mode of operation for the machine. Each transaction system can be thought of as a batch. This has the advantage that while one transaction system is being executed, the scheduler can be working in parallel on the next transaction system. Clearly, the selection of transactions to comprise a transaction system is a crucial factor affecting performance. An alternative to batching is to dynamically schedule transactions as they arrive. Will this improve performance? Simulation studies or queueing analysis can provide the answers to these questions.

The transformation presented in section 4 produces nodes that must be capable of evaluating arbitrarily complicated set expressions. The complexity of some of these nodes may be reduced by refining the nodes (i.e. replacing each by a more detailed subgraph) and then detecting common subexpressions across nodes of the subgraphs. As we pointed out before, a cost based on physical database parameters, must be attached to each node. When this is done, it can be determined when it is beneficial to transform a given schema.

Lastly, in section 4 we ignored the problem of eliminating duplicate tuples when an insert or modify operation is executed. We treat this as a special case of integrity checking. Integrity checking could be implemented as part of the effect step of an update operation. However, a more intriguing possibility is to use query modification (as suggested in [ston75]), together with the schema transformation of section 4, to perform integrity checking in parallel with the execution of the update (for example, tuples which are being duplicated can be flagged for subsequent deletions). Working out the details of this modification is a topic of future research.

REFERENCES

BSR80	P.A. Bernstein, D.W. Shipman, J.B. Rothnie, "Concurrency control in a System of Distributed Databases (SDD-1)" ACM TODS, vol. 5, no. 1, 1980.
BSW79	P.A. Bernstein, D.W. Shipman, W.S. Wong, "Formal Aspects of Serilaizability in Database Concurrency Control", IEEE-TSE, vol. 5, no. 3, 1979, pp. 177-187.
DeW78	D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems" Proc. of the 5th Annual Symposium on Computer Arch., Apr. 78, pp. 182-189.
EGLT76	K.P. Eswaren, J.N. Gray, R.A. Lorie, I.I. Traiger, "On the Notions of Consistency and Predicate Locks in a Relational Database System", CACM, vol. 19, no. 11, 1976
GOU80	M.Gouda, "Simultanity in Distributes Databases", Technical Report, Dept. of Computer Sciences, Univ. of Texas, Austin, TX, Oct. 1980.
HSW75	G.D. Held, M.R. Stonebraker, E.Wong, "INGRES - A Relational Database System", Proc. AFIPS NCC, 1975, pp. 409-416.
HY79	A.R. Hevner, S.B. Yao, "Query Processing in Distributed Database System", IEEE-TSE, vol. 5, no. 3, 1979, pp. 177-187.
Kris81	R.Krishnamurthy, "Concurrency Control and Transaction Processing in a Highly Parallel Database Machine", Doctoral dissertation in Dept. of Computer Sciences, Univ. of Texas (in preparation).
Kel173	R.M. Keller, "Parallel Program Schemata and Maximal Parallelism. Part 1: Fundamental Results", JACM, 1973, vol. 20, no. 4, pp. 696-710.
KP79	H.T. Kung, C.H. Papadimitriou, "An Optimality Theory of Concurrency Control for Database", Proc. of 1979 SIGMOD conf., Boston, Mass., May 1979.
Mann74	Z.Manna, "Mathematical Theory of Computation", McGraw Hill,1974
Papa79	C.H. Papadimitriou, "Serializability of Database Updates", JACM, vol. 26, no 4, 1979, pp. 631-653.
PBR77	C.H. Papadimitriou, P.A. Bernstein, and J.B. Rothnie, "Some Computional Problems related to database Concurrency Control", Proc. of Theoretical Computer Science, Waterloo, Aug. 77.
Ston75	M.R. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification", Proc. of ACM-SIGMOD Intl. Conf. on Management of Data, San Jose, 1975, pp. 65-78.
Yao79	S.B. Yao, "Optimization of Query Evaluation Algorithms", ACM-TODS, vol. 4, no. 2, 1979.

PROMISING APPROACH TO DISTRIBUTED QUERY PROCESSING

by

C.T. Yu, K. Lam, C.C. Chang and S.K. Chang

Department of Information Engineering
University of Illinois at Chicago Circle
Chicago, Illinois 60680
U.S.A.

Author's present addresses:
C.C. Chang, S.K. Chang and C.T. Yu, Department of Information Engineering, University of Illinois at Chicago Circle, Chicago, Illinois 60680, U.S.A.
K. Lam, Department of Statistics, Hong Kong University, Hong Kong

This research was supported in part by a grant from NSERC of Canada and in part by a grant from US Army under contract MDA 903-78-C-0293

I. INTRODUCTION

A distributed database management system (DBMS) allows data to be stored at multiple locations and to be accessed as a single unified database. A survey of problems related with distributed DBMS and its advantages over a centralized database can be found in [ACDG,CHAK,ROG1].

An important problem in distributed DBMS is to find an efficient strategy to process queries referencing data in different sites. Algorithms have been suggested by Wong [WONG], Hevner and Yao [HEYA], Yu, Lam and Ozsoyoglu [YLOZ] and Chiu and Ho [CHHO]. The algorithm by Wong obtains a local optimal solution; that by Hevner and Yao obtains the optimal solution for single domain relations and is a heuristic for more general queries, the algorithm by Yu et. al. and that by Chiu and Ho are very similar and obtain optimal solutions for tree-queries which is a subclass of queries, first studied by Bernstein and Chiu [BECH]. The recognition of such queries have been studied in [BECH,BEGO,YU01,YU02]. Other algorithms have been suggested by Goodman et. al. [GBWR], Kerschberg, Ting and Yao [KTYA] and Epstein and Stonebraker [EPST]. The algorithm by Goodman et. al. serves as a heuristic algorithm for general queries; the algorithm by Epstein and Stonebraker emphasizes on joins instead of semi-joins; the algorithm by Kerschberg is applicable on a star network configuration.

In this paper, we provide a promising approach to distributed query processing. The approach yields optimal strategies for a subclass of commonly issued queries in fully connected networks and is applicable to general queries.

In section 2, we discuss the difficulties to answer even the simplest type of queries optimally. This motivates us to attack the problem of distributed query processing in a specific way as outlined in section 3. In section 4, optimal strategies to fully reduce a relation for frequently issued queries are obtained in a fully connected network. The algorithm is generalized to other types of queries in section 5. Finally, in section 6, a comparison is done with the algorithm given in SDD-1. Experimental results show that a significient average improvements ranging from 14% for 3 relations to 60% for 11 relations are achieved by our algorithm.

A relational database model [COD1,COD2,DATE] is assumed throughout this paper.

II DIFFICULTIES OF THE PROBLEM

In this section, we will explain the difficulties in processing a distributed query optimally. This will motivate the approach we take in later sections.

In processing a distributed query, transmission cost is usually very significant. The transmission cost is the summation of all the costs involved in transferring data from one site to another. Specifically, the cost for tranferring X amount of data from one site to another is co+c1*X, where the total transmission cost given above is the same as the total cost referred to in [HEYA]. An important factor that needs to be considered is: choosing a copy of each relation referenced by the query. In a distributed database, some relations may be duplicated for efficiency and for reliability reasons. During the processing of a distributed query, it is necessary to select a copy of each relation referenced by the query so as to reduce the transmission cost. It is now shown that processing optimally the simplest type of distributed queries referencing relations with multiple copies in a distributed database is a NP-hard problem [HOSA,KARP,COOK]. Thus, it is extremely likely that any algorithm which guarantes optimal processing of even the simplest type of distributed queries will run in exponential time.

Details are specified as follows.

Let {R1,...,Rm} be single-domain relations referenced by a query, {S1,...,Sn} be computer sites and each Ri may have one or more copies in the n sites. The query has the qualification $\bigcap_{i=1}^{m-1}$ (Ri.A = Ri+1.A) where A is the common joining domain between the relations and "." denotes the projection operation, i.e. the query requests all common tuples among the m relations.

The total transmission cost problem (T-T problem) can be stated as follows: Given relations $\{Ri \mid 1 \le i \le m\}$, sites $\{Si \mid 1 \le i \le n\}$, each Si containing some relations, (if a relation appears in two or more sites, identical copies of the relation exist at the sites), the average size of a value in the common joining domain Av, the total number of possible distinct values in domain A, |A|, and the transmission parameters c0 and c1, find a strategy which minimizes the expected total transmission cost.

The above problem can be reduced to the Minimum-subset-problem (M-S problem) which is known to be NP-complete [AHUL].

Lemma 2.1: The total transmission cost problem is NP-hard.

Proof: We now show that for any given instance of the M-S problem, there is a corresponding instance of the T-T problem such that a solution for the latter problem provides a solution to the former.

For any given m elements {e1,e2, ...,em} and n subsets {T1, ...,Tn} in the M-S problem, the following instance of the T-T problem is constructed: m relations {R1, ...,Rm}, n sites {S1, ...,Sn} such that there is a 1-1 correspondence between ei and Ri, $1 \leq i \leq m$ and Ti and Si, $1 \leq i \leq n$ and set Ti contains element ej iff site Si contains a copy of relation Rj. Furthermore, c0 and c1 are chosen such that c0 > m*c1*Av*|A|, i.e., the start-up cost dominates the transmission cost. The selectivity of each relation is arbitrary.

Let g be the minimum number of sites that contain all the relations and h be the number of sites in an optimal strategy for the T-T problem. Then, we claim g=h. Suppose not, then g<h. Consider the following strategy: find the common tuples of the relations in one of the g sites before transmitting the resulting relation to the next site and repeat this process until all the g sites are visited. (Note: the data at the last site is not transmitted.) This strategy has expected total transmission cost $\leq$ $(g-1)*c0 + c1*|A|*\sum_{l=1}^{g-1}(Av) < g*c0$.

The optimal strategy has expected total transmission cost > $(h-1)*c0 \geq g*c0$, a contradiction.

Thus, an optimal strategy for the instance of the T-T problem must visit the minimum number of sites containing at least one copy for each relation. Since a site and a relation in the T-T problem corresponds to a set and an element in the M-S problem, an optimal solution for the T-T problem yields the minimum number of sets for the M-S problem. $\square$

In general, a query has two components: the output component and the qualification component. The qualification component selects the tuples of the referenced relations that satisfy the qualification, while the output component specifies the attributes of the selected tuples to be outputted to the user. Specifically, queries discussed in this paper are of the form similar to those in [BECH,HEYA,etc.].

$\{(Ri.Ai1, Rj.Aj1, \ldots.) \mid \bigcap (Rk.Ak1=Rt.At1)\}$

where the qualification component is a conjunction of equality clauses of the form $\bigcap$ (Rk.Ak1=Rt.At1) and the output component is (Ri.Ai1, Rj.Aj1, ...).

It has been shown [HEVN] that even if each relation referenced by the query has a single copy in the distributed database and the attributes of the relations referenced in the qualification of the query are the same attribute (i.e. when restricted to the qualification of the query, the relations are single-domain relations), the problem of finding an optimal strategy under this restriction is still NP-hard.

III OBJECTIVE

In view of the difficulties discussed in the last section in processing a distributed query optimally, the following restrictions are imposed on the remaining part of this paper:

(1) It is assumed that some algorithm pre-selects one copy of each relation referenced by a query before our algorithm (to be described) is invoked to produce a query processing strategy. The rationale for making this assumption is due to Lemma 2.1.

(2) The processing of each query is divided into two stages: The first stage is to concentrate on the qualification component of the query. Specially, we eliminate data from the referenced relations by making use of semi-joins (Ri _semi-join_ Rj is the result of joining Ri with Rj and then projected back on the attributes of Ri) [BECH,BEGO]. The second stage is to decide which relations, that are referenced in the qualification, are really needed to be sent to an assembly site to produce the answer specified in the output component of the query. The reasons to separate the

processing into two stages are: (i) the ideas to be presented will be easier to understand and the separation is natural since one stage deals with the qualification while the other deals with the output, (ii) if the stages are not separated, then the problem is NP-hard, even if the qualification part involves simple queries only [HEVN]. On the other hand, the qualification part can be performed optimally in polynimial time for many common queries, as demonstrated in the next section.

(3) Different relations referenced by a query reside in different sites. If two or more relations reside in one site and if these relations have common joining attributes, then they will be joined together using local processing. Since local processing cost is likely to be small compared to transmission cost, it is usually cost-effective to merge the relations together. If the relations in a site do not have common joining attributes, they will be treated as if they are in different sites. Clearly, the cost estimate for such a situation will not be higher than that as if the relations are in the same site. It is further assumed that attributes of relations which do not appear in the query are eliminated by local processing before the strategies to be described in later sections act on the relations.

We believe that most users, in particular the casual users, will not submit highly complicated queries as it may be beyond their means to formulate such queries. As a result, we propose an optimal strategy to fully reduce a relation (a relation is <u>fully reduced</u> with respect to a query if all the tuples that do not satisfy the qualification of the query are eliminated) for a simplified type of queries in a fully connected network in Section 4.

The approach we suggest is applicable to more general queries. It is hoped that the approach will yield optimal or close-to optimal query processing strategies for many common queries. We believe that an exhaustive enumeration algorithm to yield optimal strategies will not be feasible as the number of possible configurations is much more than 2^n for n relations [CHIU].

IV OPTIMAL STRATEGIES FOR COMMON QUERIES ON FULLY CONNECTED NETWORK

In this section, a fully connected network is assumed. We seek an optimal strategy to fully reduce a relation which appears in the qualification of a query. In other words, the first stage of query processing discussed in (2) of section 3 is given in this section. The qualification of the query under consideration is

$$[\bigcap_{i=1}^{m-1}(\mathrm{Ai.A=Ai+1.A})] \cap [\bigcap_{j=1}^{n-1}(\mathrm{Bj.B=Bj+1.B})] \cap (\mathrm{I.A=Am.A}) \cap (\mathrm{I.B=Bn.B})$$

i.e. there are m+n+1 relations, m single-domain relations have a common joining domain A, n single-domain relations have a common joining domain B and I is a 2-domain relation whose A domain joins with the m A-relations and whose B domain joins with the n B-relations.

A strategy which fully reduces one of the m+n+1 relations and which incurs minimum communication cost is sought: the cost of transmitting X amount of data from one site to another is c0+c1*X where c0 and c1 are constants [HEYA] and the total communication cost of a strategy is the summation of the costs of transmitting data in the strategy. Let OPTS(m,n,I,Y) denote an optimal strategy where m is the number of A-relations, n is the number of B-relations, I is the 2-domain relation and Y is one of the relations to be fully reduced. Each strategy, including an optimal strategy, can be considered as a directed graph, where the vertices of the graph are the relations, and each edge of the graph, say (Ri,Rj) denotes the transmission of the relation Ri to the site containing relation Rj. When Ri reaches the site, the relations Ri and Rj are joined according to the qualification. If some relation, say Rk, is sent to the site containing Ri before Ri is transmitted, then only the part of Ri which joins with Rk is transmitted to Rj.

Example 4.1: Consider the qualification of a query: $\bigcap_{i=1}^{2}$ (Ai.A=Ai+1.A). The following strategy A1-->A2-->A3, which sends A1 to the site containing A2, eliminates tuples from A2 which do not satisfy A1.A=A2.A to obtain a smaller relation $\overline{A2}$, sends the reduced $\overline{A2}$ to the site containing A3 and eliminates tuples from A3 which do not satisfy $\overline{A2}$.A=A3.A to obtain $\overline{A3}$. This $\overline{A3}$ is a fully reduced relation as any tuple in A3 which does not satisfy the qualification of the query will not appear in $\overline{A3}$.
The cost of A1-->A2 is c0+c1*|A1|*a where |A1| is the number of tuples in A1 and a is the size of a tuple. |A1| can also be written as a1*|A| where a1, the selectivity of A1 is defined to be |A1|/|A| and |A| is the number of possible distinct tuples in domain A. $|\overline{A2}|$ is estimated to be a1*a2*|A|, where a2 is the selectivity of A2. It is assumed in the estimation that distinct values of A1 and A2 are distributed independently. Thus, the cost of A2-->A3 is c0+c1*a1*a2*|A|*a. The total communication cost of the strategy is (c0+c1*|A1|*a) + (c0+c1*a1*a2*|A|*a). The estimation of costs given here is consistent to those given in [HEYA,YLOZ]. □

An exhaustive search to find an optimal strategy among all possible strategies can be very expensive as the number of strategies is highly exponential. As a first step, we find same inherent properties of an optimal strategy so that any strategy which does not satisfy these properties cannot be an optimal strategy and can therefore be eliminated from consideration.

Some inherent properties of an optimal strategies are (Proofs of these properties can be found in [YLOZ]):

Property 4.1: All A′s and B′s appear exactly once, while I may appear once or more.

Suppose Y is the first relation to be fully reduced in a strategy. In order to fully reduce Y, each relation appearing in the qualification must be sent to the site containing Y directly or indirectly via sites and merging with relations contained in these sites. Thus, for each relation, there is a directed path from that relation to Y.

Property 4.2: All A-relations and the I-relation must lie in a single path leading to Y; similarly, all B-relations and the I-relation must lie in a single path leading to Y. The path containing the A′s and the path containing the B′s may intersect at I or they may be the same path.

Example 4.2:

(a) In
$$\begin{tikzcd}
A2 \arrow[r] & A1 \arrow[r] & I \arrow[r] & B2 \arrow[r] & B3 \arrow[r] & Y, \\
& B1 \arrow[ur] & & & &
\end{tikzcd}$$

there are 2 paths A2-->A1-->I-->B2-->B3-->Y and B1-->I-->B2-->B3-->Y. They intersect at I, then merged into one path.

(b) In A1-->A2-->I-->B1-->B2-->B3-->I-->Y, the two paths containing the A′s and the B′s are actually only one path.

(c)
$$\begin{tikzcd}
A1 \arrow[r] & A2 \arrow[r] & I \arrow[r] & Y \\
& B3 \arrow[ur] & & \\
B2 \arrow[ur] & & B1 \arrow[uu] &
\end{tikzcd}$$
cannot be an optimal strategy,

because the B′s are in 2 paths, violating Property 4.2. □

Property 4.3: The A′s must appear in ascending order of their sizes in the path leading to the first fully reduced relation Y; Similarly, the B′s must appear in ascending order of their sizes also. Starting from this point, we order the A′s and the B′s such that $|A1|<|A2|< \dots <|Am|$ and $|B1|<|B2|< \dots <|Bn|$.

Example 4.3: The strategy in Example 4.2(a) violates Property 4.3, because A1 and A2 are in descending order of size; the strategy in Example 4.2(b) satisfies Property 4.3. □

Property 4.4: Every vertex has out-degree of one (one edge leading away from the vertex) except Y whose out-degree is 0.

Property 4.5: Only the vertex representing the first occurrence of I may have in-degree (the number of edges going into the vertex) greater than one.

Example 4.4:

$$\begin{tikzcd} A1 \arrow[r] & I \arrow[r] & A2 \arrow[r] & I \arrow[r] & B2 \arrow[r] & Y \\ B1 \arrow[ur] & & & & & \end{tikzcd}$$

is a possible optimal strategy

while

$$\begin{tikzcd} A1 \arrow[r] & I \arrow[r] & A2 \arrow[r] & I \arrow[r] & B2 \arrow[r] & Y \\ & & B1 \arrow[ur] & & & \end{tikzcd}$$

is not possible to be an optimal strategy since there are two edges (one due to B1, the other due to A2) going into the second occurrence of I. □

By Property 4.3, the first fully reduced relation in an optimal strategy is Am or Bn or I. Thus, an optimal strategy is one of the following 3 forms:

OPTS (m,n,I,Am)

OPTS (m,n,I,Bn)

OPTS (m,n,I,I).

Consider OPTS(m,n,I,Am). The vertex immediately preceding Am cannot be a B-relation since a B-relation cannot merge with Am directly. In fact, this vertex must be either I or Am-1, by Property 4.3.

Subcase 1: If the vertex is Am-1, then the set of relations preceding Am-1, together with Am-1, form a substrategy involving the m-1 A-relations {A1,A2, ...,Am-1}, the n B-relations {B1,B2, ...,Bn} and the I-relation. This substrategy is optimal among all substrategies ending at Am-1 and involving the same subset of relations (otherwise a better substrategy followed by the data transfer to Am will produce a better strategy) by dynamic programming principle and is denoted by OPTS(m-1,n,I,Am-1).

Subcase 2: If the vertex is I, then again we have an optimal substrategy involving the same subset of relations. This substrategy is denoted by OPTS(m-1,n,I,I), since the last vertex in the substrategy is I.

Both substrategies process the same set of relations and the relation immediately following each of these substrategies is Am. Thus, the amount of data transmitted from each of those substrategies to Am is identical and can be denoted by X. Thus, OPTS(m,n,I,Am) is either Am preceded by OPTS(m-1,n,I,I) or Am preceded by OPTS(m-1,n,I,Am-1). Let C(strategy) be the cost of the strategy. Then,

$$\begin{aligned} C(OPTS(m,n,I,Am)) &= (c0+c1*X) + \\ &\quad \min \{C(OPTS(m-1,n,I,Am-1)),C(OPTS(m-1,n,I,I))\} \end{aligned}$$

Pictorially, OPTS(m,n,I,Am) is

$$Am \longleftarrow \min \{OPTS(m-1,n,I,I),OPTS(m-1,n,I,Am-1)\} \qquad (4.1)$$

where the cost functions are not explicitly written. Similarly, OPTS(m,n,I,Bn) is

$$Bn \longleftarrow \min \{OPTS(m,n-1,I,I), OPTS(m,n-1,I,Bn-1)\} \qquad (4.2)$$

Consider OPTS(m,n,I,I). If the first fully reduced relation I has in-degree 1, then the relation immediately preceding I can be either Am or Bn. The two subcases are respectively

$$I \longleftarrow Am \longleftarrow \min \{OPTS(m-1,n,I,I), OPTS(m-1,n,I,Am-1)\} \qquad (4.3)$$

$$I \longleftarrow Bn \longleftarrow \min \{OPTS(m,n-1,I,I), OPTS(m,n-1,I,Bn-1)\} \qquad (4.4)$$

If the first fully reduced relation I has in-degree 2, then by Property 4.2 the optimal strategy is

$$\begin{tikzcd} & OPTS(m,0,0,Am) \arrow[dl] \\ I & & (4.5) \\ & OPTS(0,n,0,Bn) \arrow[ul] \end{tikzcd}$$

From (4.1)-(4.5), OPTS(m,n,I,Y) can be computed in constant time if OPTS(m-1,n,I,I), OPTS(m-1,n,I,Am-1), OPTS(m,n-1,I,I), OPTS(m,n-1,I,Bn-1), OPTS(m,0,0,Am) and OPTS(0,n,0,Bn). This suggests the following method to obtain the optimal strategy.

Consider the 2-dimensional figure in Figure 4-1, where the point (i,j) denotes 3 optimal strategies involving {A1,...Ai,B1,...Bj,I} ending in Ai, Bj and I. From equation (4.1)-(4.5), the optimal strategies at (m,n) are obtainable from those at (m-1,n), (m,n-1), (m,0) and (0,n). Thus, if we compute all optimal strategies at (x1,x2), x1+x2=t, and at the boundary points (i,0), (0,j), $1 \leq i \leq n$, $1 \leq j \leq m$, (the optimal strategies at the boundary points involving essentially single-domain relations are easily computable [HEYA]), then the strategies at (y1,y2), y1+y2=t+1 are easily computable. Starting from t=1, we progress to t=m+n when the optimal strategy for the query is obtained. This can be shown to take O(mn) time [YLOZ].

The algorithm can be generalized to obtain optimal strategies to reduce relations for tree queries (see [CHHO, YULO]). However, the algorithm generalized in that direction is not as easy as the one given as follows to program and does not take into consideration the cost of sending relations to the assembly site.

V GENERAL ALGORITHM

Before an algorithm to process a general query is presented, a query graph will be defined to facilitate the description of the algorithm.

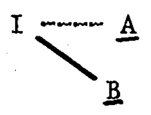
A set of relations are of the <u>same kind</u> if the set of relations have the same set of joining attributes in the qualification of the query. For example, relations A1, A2, ..., An are of the same kind, because they have the same joining attribute A in the qualificaton of the query discussed in the last two sections. Vertices of a query graph denote relations of the same kind while edges of the graph denote the joining of the relations appearing in the qualification of the query.

A query graph G=(V,E) defined here is similar to that defined in [BECH], except that each vertex consists of the set of relations of the same kind. In other words, the edges between relations of the same kind in the [BECH] definition of a query graph are not shown in the present query graph and all relations of the same kind are merged into a single vertex.

<u>Example 5.1</u>: A query graph as defined in [BECH] can be

$$\begin{tikzcd}
I \arrow[r, dash] \arrow[dr, dash] & A1 \arrow[r, dash] & A2 \\
 & B1 \arrow[d, dash] & \\
 & B2 \arrow[r, dash] & B3
\end{tikzcd}$$

According to the present definition, the query graph is

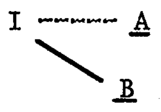


where A represents the set of relations of the same kind having the same joining attribute A, i.e., A1 and A2 and

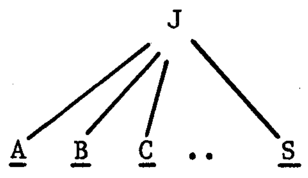
B represents B1, B2 and B3. □

<u>Notation</u>: A vertex, if underlined, represents a set of relations of the same kind, while a vertex which is not underlined represents a single relation.

Section 4 gives optimal algoritms to fully reduce a relation for query graphs of the form



The same method is applicable to query graphs of the form



where J is a relation having attributes A,B,C,...,S.

Let query graphs of this form be called <u>tree of height 1</u> (where J is the root and each vertex in the tree is one edge away from J). We now describe a query processing method making use of the optimal strategies to process subqueries whose tree-query graphs are trees of height 1 as substrategies. The method consists of 2 key steps.

In the first step, a relation, say R, is reduced as much as possible by semi-joins. If the query is a tree-query [BECH,BEGO,YU01,YU02], then the relation will be fully reduced otherwise it is only partially reduced.

In the second step, a set of relations which are needed to construct the answer as defined in the output component are identified. Then, this set of relations are possibly reduced by R and used to produce the answer at the result site. Details of the steps are given as follows.

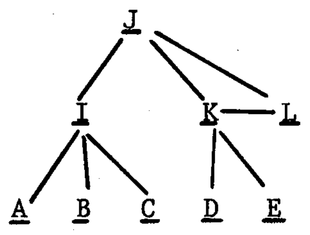
Step 1: To reduce a relation as much as possible.

1.1 Construct a query graph G=(V,E) from the qualification part of the query. /* each vertex is a set of relations of the same kind */

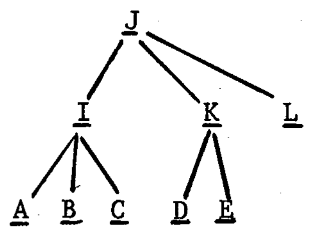
1.2 If G is cyclic, choose a spanning tree T=(Vt,Et), otherwise G is a tree, set T=G /* see for example [YU01,YU02,BEG0] */. Designate a vertex R whose joining attributes are not subset of the joining attributes of any other vertex in Vt as the root of T.

1.3 Decompose T into a number of subtrees of height 1, plus a set of relations of the same kind.

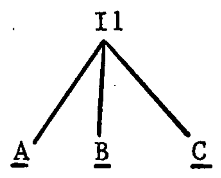
Example 5.2: Suppose we have the following query graph,



we can choose the spanning tree T as,

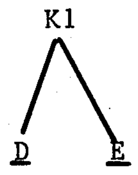
T:


suppose the root of T is J, then T can be decomposed into subtree 1



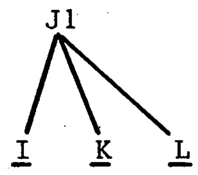
where I1 is the smallest relation in I

subtree 2



where K1 is the smallest relation in K

subtree 3



where J1 is the smallest relation in J

and relations of the same kind in J, namely {J1,J2,...,Js}. □

```
1.4 Starting from the bottom of tree T toward root R,
    For each subtree,
        a) for every leaf f of the chosen subtree,
           if f has a joining attribute outside the joining attributes
              of its father r1, assume the smallest relation in f is f1,
           then (i) if the cost of f1-->other relation  is greater than
                        the cost of r1-->f1-->other relation
                    then  do r1-->f1  first,
                (ii) send the relations in f in ascending order of sizes.
                     (in step b, whenever f is referenced, only the last
                      reduced relation in f is used)
        b) apply the optimal strategy given in section 4 to the chosen
           subtree.
```

When root R is reached, all relations of the same kind in the root are sent in ascending order of sizes /* as given in [HEYA], sending relations of the same kind in ascending order of sizes is optimal */

Example 5.3: Using Example 5.2, assume that only the joining attributes of J1 of subtree 3 do not contain that of some of its sons, say, the joining attributes of J1 is ABCDG; of I is ABC; of K is DE; of L is EG, we apply the optimal strategy to subtree 1 to reduce I1; then to subtree 2 to reduce K1; as for subtree 3, we consider whether (J1-->the smallest relation in K-->other relation) has lower cost than (the smallest relation in K-->other relation), if yes, (J1-->the smallest relation in K) is done first. Then, Hevner-Yao's algorithm is applied to reduce K to one relation. The same process is done for L. And then, we apply the optimal strategy to subtree 3; finally, Hevner-Yao's algorithm is applied to reduce the relation in J, namely Js. At the end of the reduction, Js is the smallest relation in J. □

Step 2: Identify the set of relations needed to be sent to assembly or result site to produce the answer, and use the relation which is reduced as much as possible in Step 1 (Js in Examples 5.2 and 5.3), to reduce the set of identified relations, then produce the answer at result site.

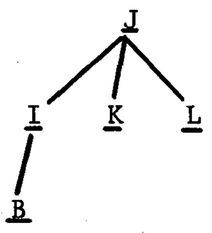
2.1 All output relations are identified (a relation is an output relation if it contains one or more output attributes). Then, all vertices that contain output relations and those vertices that appear in cycles /* only if the query is cyclic */ are identified. A minimum connected graph G1=(V1,E1), which is a subgraph of the spanning tree T, connecting the identified vertices and R is formed. Designate R as the root of G1. Finally, the smallest relation in each vertex of G1 /* except root vertex */ is identified. /* only the output relations and the smallest relation in every vertex of G1 are needed to produce the answer, see Example 5.5 */

2.2 For the identified relations of the same kind as the first relation which has been reduced as much as possible (say Js),

```
if no relations are identified, Js is sent directly
        to assembly site or result site.
else (1)  for every identified relation, say Ji, has an
        output attribute (or attributes) outside the joining
        attributes of Js,
        if   Js-->Ji-->some site    has a lower cost than
               Ji--------->some site,
        then Ji is reduced by Js before being sent to
               assembly site or result site.
     (2) if no sending of Js is involved for all Ji's,
        then send Js to the largest Ji.
```

Example 5.4: Using the same query graph as Example 5.2, suppose the output relations are B2 and J1, then the minimal connected graph G1 can be

G1:



and the set of identified relations is the output relations J1, B2, and the smallest relation of I, say I3; of B, say B4; of K, say K4; of L, say L2. Suppose Js is the relation that has been reduced as much as possible in step 1, since there is only one identified relation J1 in J, J1 is reduced by Js (check Step 2.2).

2.3 Consider the subtree of height 1 with the root of the tree as Js and the leaves as I, K, ... etc..

For each identified relation I1,I2,... of I,

K1,K2,... of K, etc..

```
if     Js-->Ii--->some site      has a lower cost than
       Ii---------->some site,
then   Ii is reduced by Js before being sent to
       assembly site.
```

2.4 Repeat Step 2.3 for subtrees of height 1 with roots being the smallest relation of I, the smallest relation of K, etc.. and the leaves of the subtrees are the immediate descendants of I, K, etc.. in tree G1 of the query graph, until all identified relations in V1 has been considered. /* in effect, the identified relations are reduced from root Js towards the leaves of tree G1 */

2.5 /* decide whether an assembly site other than the result site is worth having */ Let X be the largest identified relation after the reduction up to and including Step 2.4, and M1, M2, ..., Mt are the other identified relations

if

$$\begin{tikzcd} M1 \arrow[r] & X \arrow[r] & \text{result site} \\ \cdot & & \\ Mt \arrow[uur] & & \end{tikzcd}$$

has a lower cost

than

$$\begin{tikzcd} M1 \arrow[rr] & & \text{result site} \\ \cdot & & \\ Mt \arrow[uurr] & X \arrow[uur] & \end{tikzcd}$$

then the former strategy is used with the site containing X being the assembly site,

otherwise the latter strategy is used with the result site as the assembly site.

Example 5.5: Using Example 5.4, suppose the output relations (those containing one or more output attributes) are B2, I2 and J1, we find the minimal connected graph Gl is the same as in Example 5.4. Suppose B4, I3, K4 and L2 are the other identified relations.

First, the strategy (Js--->J1--->some site) is chosen. Then the strategy with the lower cost among (Js--->I3--->some site) and (I3--->some site) and among (Js--->I2--->some site) and (I2--->some site) are choosen. And the strategy with lower cost among (Js--->K4--->some site) and (K4--->some site) and among (Js--->L2--->some site) and (L2--->some site) are chosen. Let Ii be the smallest I-relation, which can be either I2 or I3. Then the strategy with smaller cost among (Ii--->B2--->some site) and (B2--->some site) and among (Ii--->B4--->some site) and (B4--->some site) are chosen. We will not reduce any of the relations in $\underline{A}$, $\underline{C}$, $\underline{D}$ and $\underline{E}$, because at the time Js was reduced, relation I3 already satisfied the part of qualification involving $\underline{A}$, $\underline{B}$, $\underline{C}$ and $\underline{I}$ and relation K4 already satisfied the part of qualification involving $\underline{D}$, $\underline{E}$ and $\underline{K}$ (please refer to the algorithm given in [BECH] which sends the relation from the leaves to the root such that intermediate results satisfy the subqueries of the given query). Suppose the result site is different from that containing J1, I2, I3, B2, B4, K4 and L2 and B2 is now the largest relation. Then the strategy with the lower cost among

$$\begin{tikzcd}
J1 \arrow[dr] & & \\
I2 \arrow[r] & B2 \arrow[r] & \text{result site} \\
I3 \arrow[ur] & & \\
B4 \arrow[uur] & K4 \arrow[uu] & L2 \arrow[uul]
\end{tikzcd}$$

and

$$\begin{tikzcd}
J1 \arrow[drr] & & & \\
I2 \arrow[rr] & & \text{result site} & \\
I3 \arrow[urr] & & & \\
B2 \arrow[uurr] & B4 \arrow[uur] & K4 \arrow[uu] & L2 \arrow[uul]
\end{tikzcd}$$

is chosen.

In effect, our algorithm is similar to that given in [BECH] where reductions are performed from the leaves to the root and backwards, except that we perform optimization for subtrees of height 1 and only some relations (the identified relations in Step 2.1) are reduced and then sent to the assembly or result site.

VI COMPARISON WITH SDD-1

SDD-1 is a distributed database system [TODS]. We now compare the performance of our algorithm with the query processing algorithm given in [p. 34, GBWR].

Our algorithm and theirs are not entirely compatible. As a result, the following conditions are added to make the comparison meaningful:

(1) When a multiple domain relation, say I, is reduced by a single-domain relation, say A, the projection of I on another domain, say B, can be estimated by a number of methods [GBWR,HEYA,YLOZ]. We make use of the estimation method given by SDD-1.

(2) In SDD-1 all relations referenced by the query are sent to an assembly site and the result site is not mentioned, while our algorithm sends only some of the relations to the result site. In the comparison, it is assumed that all relations are sent to a site containing one of the relations. This will usually incur higher transmission costs to strategies produced by our algorithm.

Only queries whose qualifications of the form given in section 4 are considered, because we believe most users, especially casual users, do not usually submit queries involving relations of more than three kinds (or queries having height greater than 1). Different number of A-relations and

B-relations are tried, where for each set of the same number of A & B-relations and a single I-relation, 50 different combinations of relation sizes (which are randomly generated) are experimented. It is found that (1) the average improvements of our algorithm over that of SDD-1 vary from 14% to 60%, and (2) when the number of relations increased, the improvement also increases. Figure 6-1 plots the improvements against the relations used. Improvement is defined to be

$$\frac{\text{(total amount of data transferred by SDD-1)} - \text{(total amount of data transferred by our algorithm)}}{\text{(total amount of data transferred by our algorithm)}} * 100\%.$$

VII CONCLUSION

We have shown that the process of selecting one copy for each relation for the simplest type of queries in order to minimize the cost of transmission is NP-hard. We have presented an approach to distributed query processing, making use of dynamic programming. The processing of queries is broken down into two stages. (1) eliminate useless data from a relation as much as possible by semi-joins, then, (2) the relation obtained from the first stage is used to reduce relations that are really needed to produce the answer to the query.

We believe our approach is rather promising, because the reduction of a relation using semi-joins for common queries (trees of height 1) is performed optimally by our algorithm, which is applicable to general queries. Thus, optimal strategies are provided for stage 1 for many common queries. Although the algorithm may not yield optimal strategies for all tree queries, the algorithm is easy to program. In stage 2, our algorithm avoids sending unnecessary relations to the assembly site or the result site. As a result, transmission cost is cut down. Rather signifiecient improvement of our algorithm over SDD-1 is provided experimentally, even when the benefit of our algorithm in stage 2 is not considered in the experiments.

REFERENCES

[ACDG] Adiba, M., Chupin, J.C., Demolombe, R., Gardarin, G. and Bihan, J.L., "Issues in distributed database management systems: a technical overview," International Coference on Very Large Databases, Berlin, pp. 89-110, 1977.

[AHUL] Aho, A., Hoperoft, J. and Ullman, J.D., "The Design and Analysis of Computer Algorithms," Addison-Wesley, 1974.

[BABB] Babb, E., "Implementing a relational database by means of specialized hardware," ACM TODS, Vol. 4, March 1979, pp. 1-29.

[BECH] Bernstein, P.A. and Chiu, D-M.W., "Using semi-joins to solve relational queries," JACM.

[BEGO] Bernstein, P.A. and Goodman, N., "Full reducers for relational queries using multi-altitute semi-joins," Centre for Research in Computing Technology, Harvard University, July 1979.

[CHAK] Chandy, K.M., "Models of distributed systems," International Conference on Very Large Databases, Tokyo, pp. 105-120, 1977.

[CHEU] Cheung, T.Y. "Two methods of resolution for general equi-join queries in distributed relational database," Tech. Report, University of Ottawa, Department of Computer Science, 1981.

[CHHO] Chiu, D.M. and Ho, Y.C., "A methodology for interpreting tree queries into optimal semi-join expressions," Harvard University, Dec. 1979.

[CHIU] Chiu, D.M., "Optimal query interpretation for distributed databases," Ph.D. Thesis, Division of Applied Sciences, Harvard University, 1980.

[COD1] Codd, E.F., "A relational model for large shared databases," CACM, pp. 377-389, 1970.

[COD2] Codd, E.F., "Further normalization of the database relational model," in Database Systems, Prentice Hall, Englewood Cliffs, N.J. pp. 33-64, 1972.

[COOK] Cook, S.A., "The Complexity of theorem-proving procedure," Proc. of third ACM Symposium on Theory of Computing, 1971, pp.151-158.

[DATE] Date, C.J., "An Introduction to Database Systems," Addison Wesley, Reading, MA, 1977.

[EPST] Epstein, R. and Stonebraker, M., "Analysis of distributed database processing strategies," IEEE, 1980, pp. 92-101.

[GBWR] Goodman, N., Bernstein, P.A., Wong, E., Reeve, C. and Rothnie, J.B., "Query processing in SDD-1: A System for Distributed Databases," Computer Corporation of America, 575 Technology Square, Cambridge, MA, Oct. 1979.

[HEVN] Henver, A.R., "The optimization of query processing on distributed database systems," Ph.D. Dissertation, Department of Computer Science, Purdue University, Lafayette, Indiana, 1980.

[HEYA] Hevner, A.R. and Yao, S.B., "Query processing in distributed database system," IEEE Transactions on Software Engineering, May 1979, pp. 177-187.

[HOSA] Horowitz, E. and Sahni, S., "Fundamentals of Computer Algorithms," Computer Sciences Press, 1979.

[KARP] Karp, R., "Reducibility among combinatorial problems," Complexity of Computer Computations, Plenum Press, N.Y., 1972, pp. 85-104.

[KTYA] Kerschberg, L., Ting, P.D. and Yao, S.B., "Optimal distributed query processing," Bell Laboratories, Holmdel, N.J..

[KULE] Kung, H.T. and Lehman, P.L., "Systolic (VLSI) arrays for relational database operations," Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pennsylvenia 15213.

[LULU] Luk, W.S. and Luk, Lydia, "Optimal query processing strategies in a distributed database system," Department of Computer Science, Simm Traser Uni., Burneby B.C., Canada.

[ROG1] Rothnie, J.B. and Goodman, N., "A survey of research and development in distributed database management," International Conference on Very Large Database, Tokyo, pp. 48-62, 1977.

[ROG2] Rothnie, J.B. and Goodman, N., "An overview of the preliminary design of SDD-1: A system for Distributed Databases," Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, 1977.

[TODS] ACM, TODS, March 1980, pp. 1-68.

[WONG] Wong, E., "Retrieving dispersed data from SDD-1: A System for Distributed Databases," Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, 1977.

[YLOZ] Yu, C.T., Lam, K. and Ozsoyoglu, M., "Distributed query optimization for tree queries," Dept. of Information Engineering, University of Illinois at Chicago Circle, Oct. 1979, revised July 1980.

[YUO1] Yu, C.T. and Ozsoyoglu, M.Z., "An algorithm for tree-query membership of a distributed query," IEEE Compsac, Chicago, Nov. 1979, pp. 306-312.

[YUO2] Yu, C.T. and Ozsoyoglu, M., "On determining tree-query membership of a distributed query," Department of Computing Science, University of Alberta, Nov. 1979.

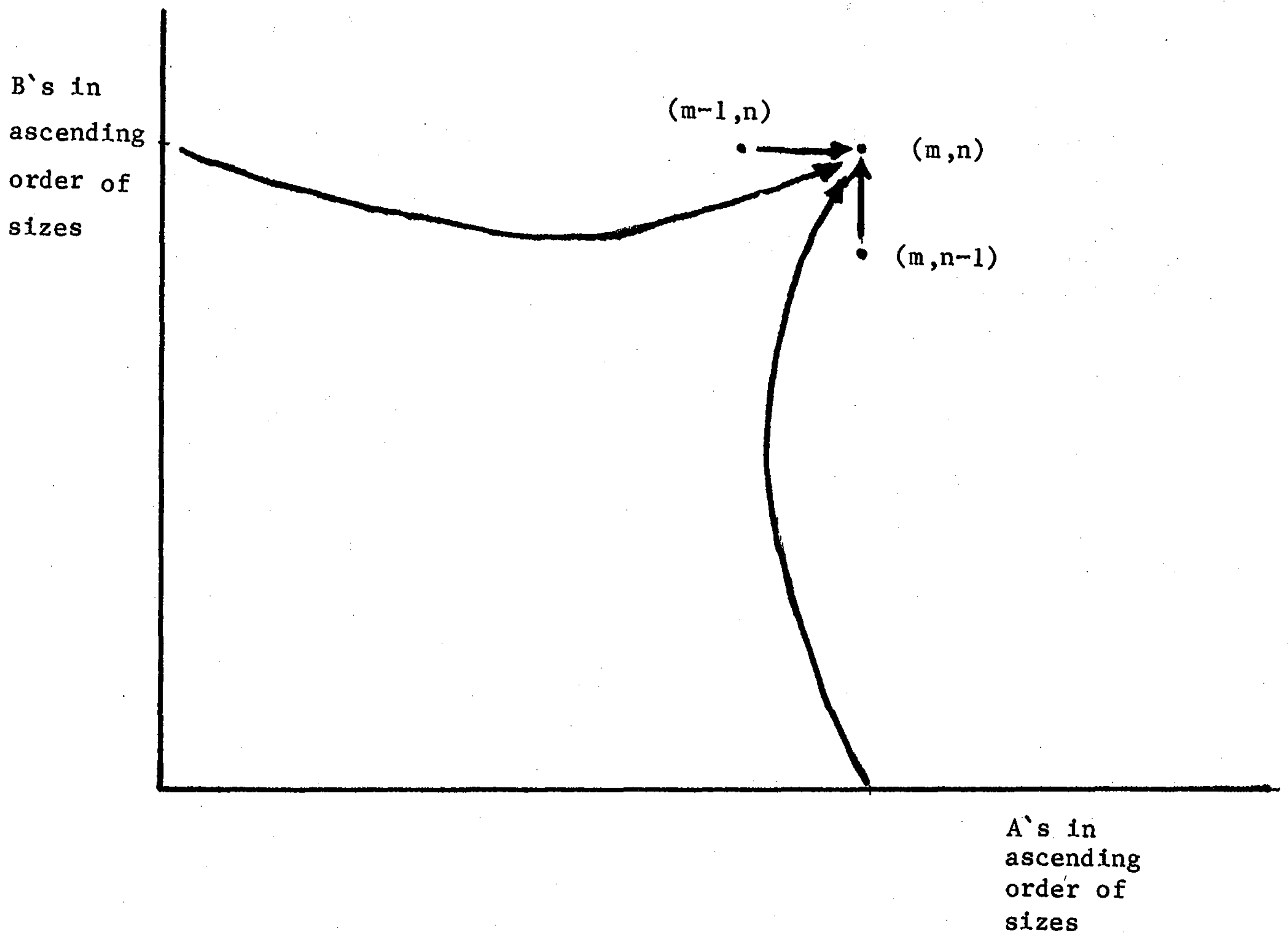
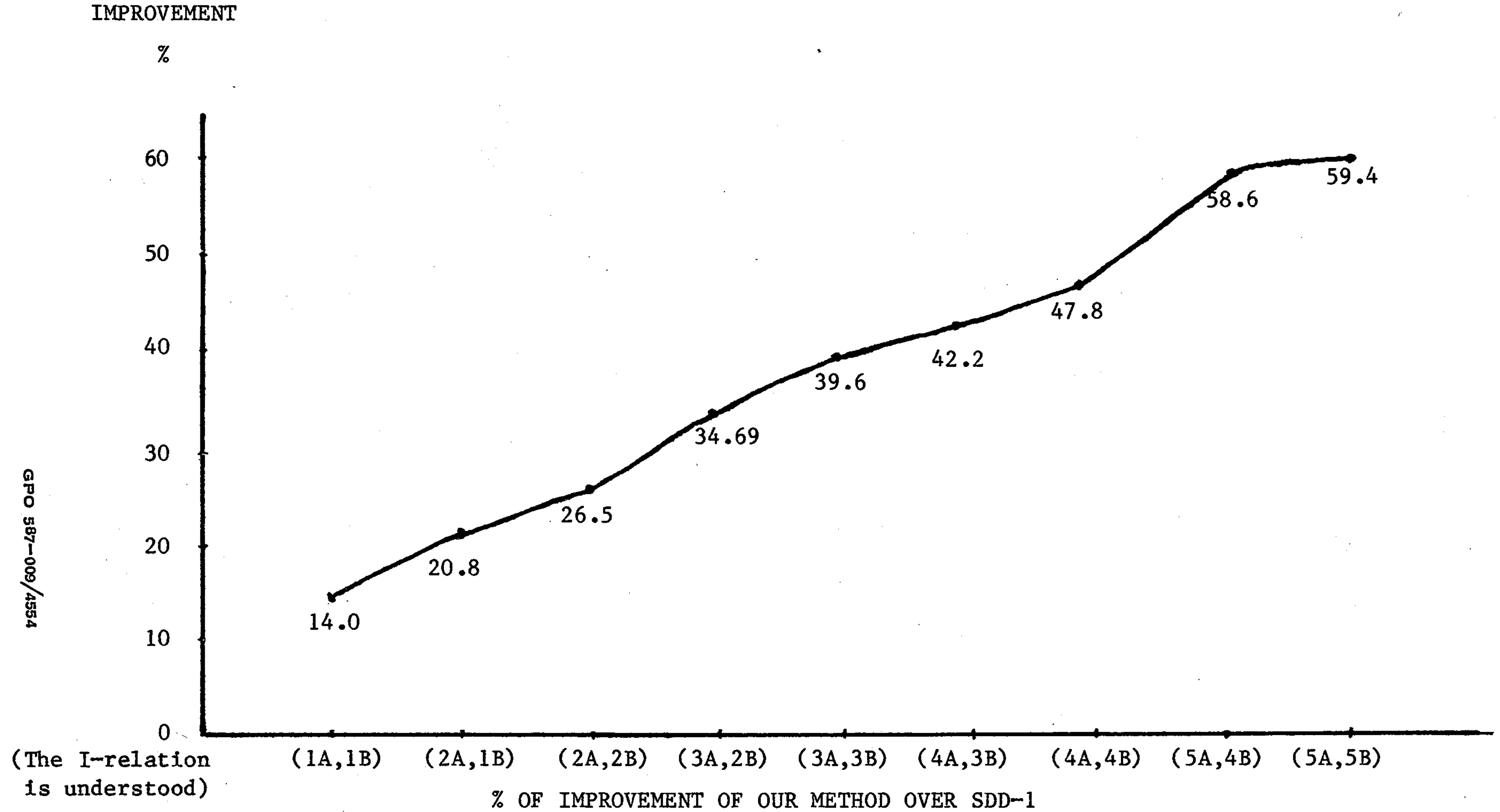

B`s in ascending order of sizes
(m-1,n)
(m,n)
(m,n-1)
A`s in ascending order of sizes

Figure 4.1 Illustrates how the optimal strategy is obtained



% OF IMPROVEMENT OF OUR METHOD OVER SDD-1

$$\text{Improvement} = \frac{\text{Amount of data transferred by SDD-1} - \text{Amount of data transferred by our algorithm}}{\text{Amount of data transferred by our algorithm}}$$

Figure 6.1

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720