0 0 0 0 5 1 0 4 0 3 3

LBL-7953
UC-32

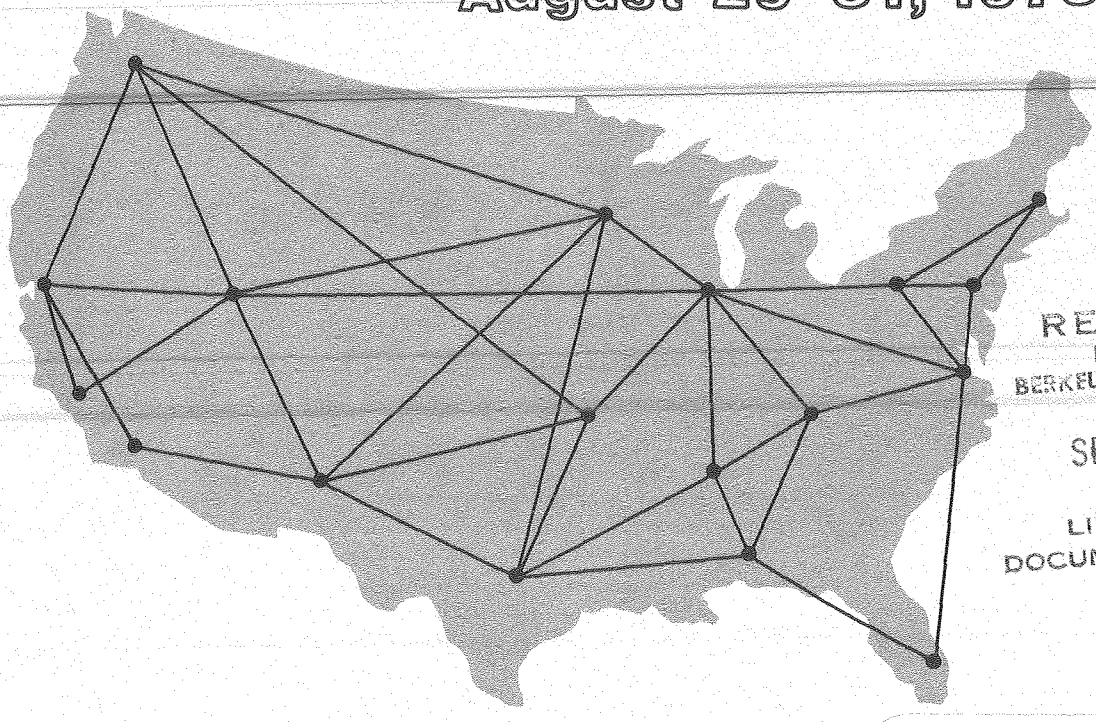
PROCEEDINGS of the

THIRD BERKELEY WORKSHOP

on

DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS

August 29-31, 1978



RECEIVED
LAWRENCE
BERKELEY LABORATORY

SEP 25 1978

LIBRARY AND
DOCUMENTS SECTION

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA, BERKELEY

For Reference

Not to be taken from this room

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

0 0 0 0 5 1 0 4 0 3 5

LBL-7953
UC-32

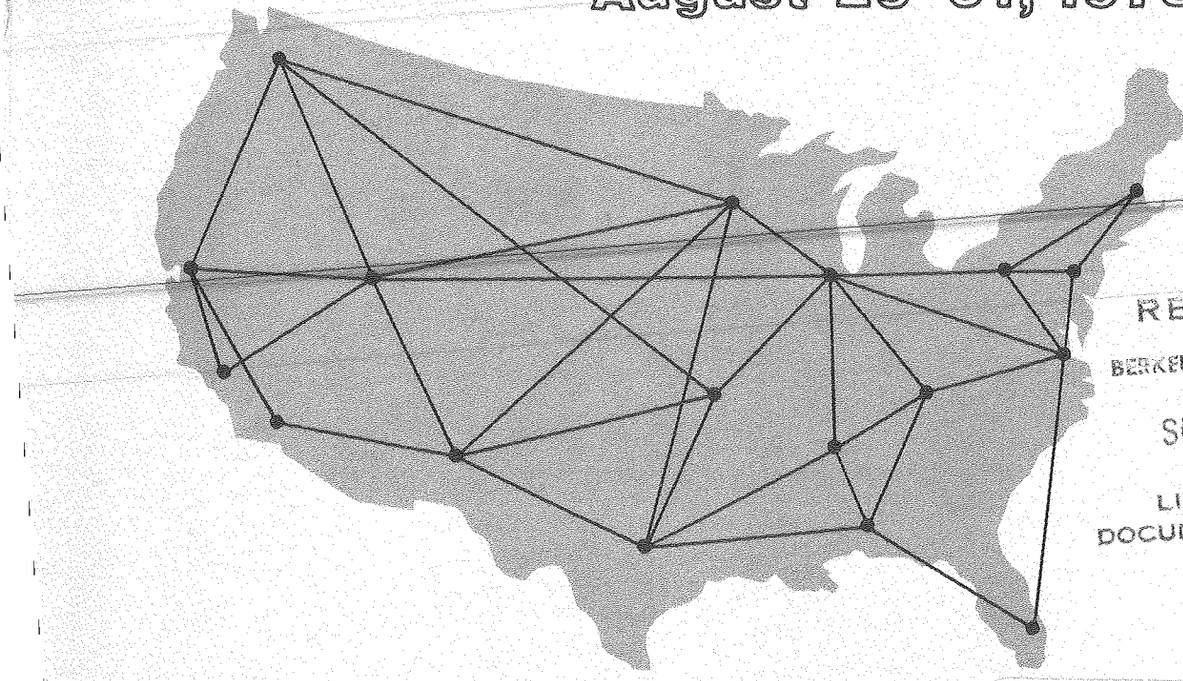
PROCEEDINGS of the

THIRD BERKELEY WORKSHOP

on

DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS

August 29-31, 1978



RECEIVED
LAWRENCE
BERKELEY LABORATORY

SEP 25 1978

LIBRARY AND
DOCUMENTS SECTION

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA, BERKELEY

For Reference

Not to be taken from this room

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

LBL-7953

LEGAL NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or 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.

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: 376-400

- - - 0 3 1 0 4 0 3 4

LBL-7953

PROCEEDINGS OF THE THIRD BERKELEY WORKSHOP ON DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS

Sponsored by

Computer Science & Applied Mathematics Department
Lawrence Berkeley Laboratory
University of California

Applied Mathematical Sciences Research Program
Office of Energy Research
U.S. Department of Energy

And in cooperation with ACM and IEEE

AUGUST 1978

General Chairman: Arie Shoshani, Lawrence Berkeley Laboratory

Program Cochairmen: Steve Kimbleton, National Bureau of Standards
Dennis Tsichritzis, University of Toronto

Program Committee:

Peter Alsberg, Digital Technology Incorporated
Mani Chandy, University of Texas
James P. Fry, University of Michigan
Dennis Hall, Lawrence Berkeley Laboratory
Peter Kreps, Lawrence Berkeley Laboratory
Frederick H. Lochovsky, University of Toronto
Jon Postel, Information Sciences Institute, USC
James B. Rothnie, Computer Corp. of America
Carl Sunshine, RAND Corporation
Robert W. Taylor, IBM Research Center, San Jose
Peter Ting, Bell Laboratories
Richard Watson, Lawrence Livermore Laboratory
Bea Yormark, Interactive Systems Incorporated

ACKNOWLEDGMENTS

We would like to thank Dr. James C. T. Pool of the Office of Energy Research in the Department of Energy for his continued support of the workshop.

The program committee members greatly appreciate the help of the following people who assisted in refereeing the papers.

H. Aghili
P. Arraya
D. Austin
D. Batory
J. Bitner
W. A. Burkhard
P. Chan
P. Chang
T. C. Chiang
D. Corneil
S. Crocker
Y. Dalal
J. Day
M. Deppe
J. Donnolley
R. Epstein
J. Fletcher
L. Garlick
N. Goodman
G. S. Graham
M. Graham
J. Gray
B. Greiman
P. Griffiths
G. Grossman
P. Hawthorn
S. Holmgren
R. Hudyma
R. Johnson
K. Kim
T. Klug
I. Kornatowski
S. Lam
Y. E. Lien
B. Lindsay
I. Ludd
J. Mathis
J. Misru
J. Mullin
D. Nessett
P. Neumann
F. Olken
J. Popek
R. Rom
L. Schneider
D. W. Shipman
S. Sluizer
D. Swartwout
S. Tepper
R. Thomas
F. Tobagi
Y. Vassiliau
J. Weiner
J. White
D. Willcox
E. Wong
Y. Ying

CONTENTS

Acknowledgments iii

INVITED TALK

Provisional Model of Open System Architecture 1
Charles W. Bachman

NETWORK PROTOCOLS AND ARCHITECTURE
Chairperson: *Carl Sunshine*

Software Architecture for a Microprocessor-Based Packet Network 21
Holger Opderbeck, Jay H. Hoffmeier, and Roy L. Spitzer

Protocol Converters and User Interface in the CYCLADES Network 36
N. Naffah and H. Zimmerman

An Architecture for Secure Packet-Switched Networks 54
M. A. Padlipsky

PERFORMANCE ISSUES IN DISTRIBUTED DATABASES
Chairperson: *Peter Kreps*

Analysis of Update Synchronization for Multiple Copy Data-Bases 69
Erol Gelenbe and Ken Sevcik

Query Processing on a Distributed Database 91
Alan R. Hevner and S. Bing Yao

Performance Comparison of Two Update Algorithms for Distributed Databases 108
Hector Garcia-Molina

LOCAL NETWORKS
Chairperson: *Dennis Hall*

Interaction Between Protocol Levels in a Prioritized CSMA Broadcast Network* 123
James E. Donnelley and Jeffry W. Yeh

The Pacific Northwest Laboratory Minicomputer Network 144
L. H. Gerhardstein, J. O. Schroeder, and A. J. Boland

COMMUNICATION TECHNIQUES
Chairperson: *Peter Alsberg*

Inter-Network Fragmentation and the TCP* 161
John F. Shoch

Dynamic Management of Packet Radio Slots 169
Glenn Ricart and Ashok K. Agrawala

Failsafe Distributed Algorithms for Routing in Communication Networks 178
Adrian Segall

CONCURRENCY CONTROL I
Chairperson: *Bea Yormark*

A Formal Model of Concurrency Control Mechanisms for Database Systems† 189
Philip A. Bernstein and David W. Shipman

Maximally Concurrent Transaction Processing 206
Toshimi Minoura

Locking and Deadlock Detection in Distributed Databases† 215
Daniel A. Menasce and Richard R. Muntz

CONCURRENCY CONTROL II
Chairperson: *Peter Ting*

Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES† 235
Michael Stonebraker

Algorithms for Distributed Data-Sharing Systems Which Use Tickets 259
Gérard Le Lann

A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems 273
D. Z. Badal and G. J. Popek

0 0 4 0 5 1 0 4 0 3 5

NETWORK MODELS AND PERFORMANCE
Chairperson: *Jon Postel*

On the Design of Elementary Distributed Systems* 289
Randolph T. Yeh and K. Mani Chandy

Synchronization in Distributed System Modules* 322
Gregor V. Bochmann

Performance Limits of the TMM Protocol: Modelling and Measurement 340
Erol Gelenbe, Jean-Louis Grangé, and Patrice Mussard

Distributed Simulation Using a Network of Microcomputers* 358
J. K. Peacock, J. W. Wong, and Eric Manning

*These papers (or an improved version) were selected for publication in *Computer Networks Journal*.

†These papers (or an improved version) were selected for publication in *IEEE Transactions on Software Engineering*.

0 0 0 3 5 1 0 4 0 3 6

PROVISIONAL MODEL OF OPEN SYSTEM ARCHITECTURE

Charles W. Bachman
(Honeywell Information Systems)

Introduction:

The attached report is a lightly edited copy of the document ISO/TC97/SC16/N34, by the same title, which was produced at the first meeting of SC16 (February 27 - March 1, 1978 in Washington D.C.). SC16 - Open System Interconnection, was formed in early 1977 to examine the needs for standards in open systems (distributed systems). ANSI created a study group to prepare the U.S. position on this subject.

The document N34 has been edited to reflect certain minor defects in original document which were the result of its rapid preparation. The document N34 was the result of joining a number of contributions to SC16 by its member delegations and liaison representatives. Appendix 1 lists these contributions. The editor is responsible for all changes, as the revised document has not been subject to SC16 review.

INTERNATIONAL STANDARDS ORGANIZATION

(WORKING PAPER)

SOURCE: ISO/TC97/SC16

SUBJECT: Provisional model of open-systems architecture

PURPOSE: Initial contributions received by SC16 on the reference model for open systems interconnection (See Appendix 2), enlighted by discussions during its first meeting in Washington, February 1978, have clearly shown a high degree of commonality between the views expressed by all member bodies on this subject.

The various models which have been proposed all conform with the principles of layered architecture. Similar functions tend to be grouped into layers in slightly different ways while terminology differs from one proposal to the other.

The purpose of this document is to provide member bcdies with common basis for work on the model of architecture during the coming period of time, by:

* establishing the basic reference model in the form of layered architecture

* defining the number and names of the specific layers composing the reference model

describing, for each layer describing, its purpose, the concepts it supports and the services/functions it provides

It is expected that the use of this common basis for discussion, will first help to reach a common understanding of the problem and then a consensus on the model to be adopted.

0 0 0 0 5 1 0 4 0 3 7

TABLE OF CONTENTS

1.0 Introduction

2.0 Layered Architecture

2.1 Layering techniques

2.2 Basic Concepts for the description of layers

3.0 Reference Model

4.0 Application Level

5.0 Presentation Control

6.0 Session Control

7.0 Transport End-to-End Control

8.0 Network Control

9.0 Link Control

10.0 Physical Control

Appendix 1 List of initial contributions in the reference model

1.0 INTRODUCTION

The concept of "open-ness" refers to a set of commonly agreed standards that makes possible meaningful interactions between any combination of computing systems, data processing systems, or human operators which are connected together in some way.

The purpose of the model is to provide a complete reference structure for the totality of standards necessary for open system interconnection. It will allow to:

- position existing standards in perspective
- identify areas in existing standards requiring additional developments
- identify areas where standards should be developed
- expand without disrupting previously defined protocols and interfaces
- allow easy mapping onto potential practical implementations
- be subject to formalization and modelling for determining completeness and correctness of functionality.

2.0 LAYERED ARCHITECTURE

A model is an abstraction or simplification that makes a concept more understandable. In order to comprehend models of complex systems, it is important to partition the structures into easily comprehended parts. Communication systems are often envisioned in terms of "layers" of functions.

The rationale for structuring a system into layers is to provide a convenient partitioning of functions which will ensure:

- sequentiality in time from layer to layer
- independence of activities between layers (a change in one layer will only affect that layer)
- hiding of specific implementation of each layer from users of that layer
- sharing of common services by different applications

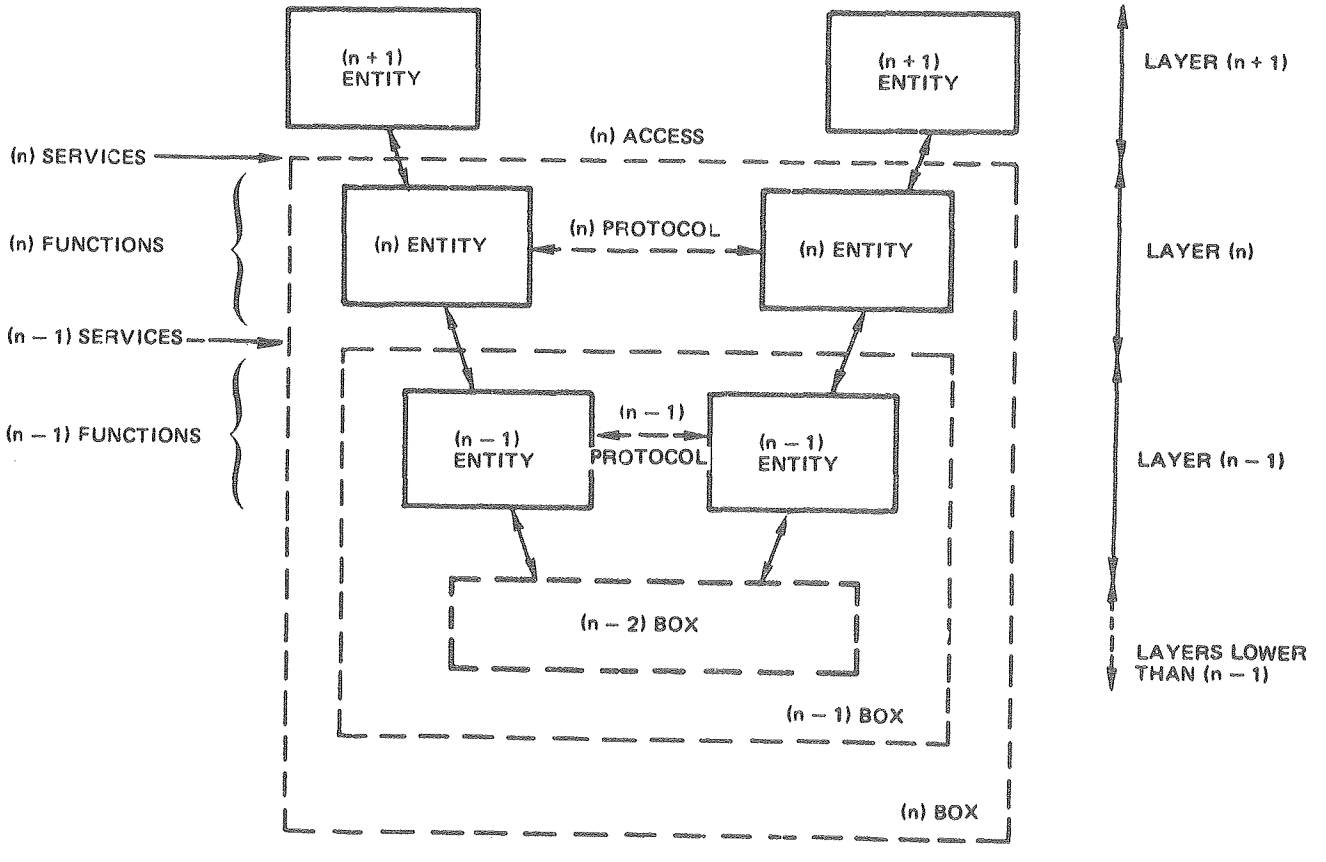
This approach is further explored in this section in terms of distributed or open systems. Section 3.1 describes the notion of layering and Section 3.2 introduces additional concepts that are useful in the discussion of layered architectures.

2.1 LAYERING TECHNIQUES

The basic structuring technique for building the architecture of the reference model consists of a hierarchical assembly of layers, as shown in figure 1. It can be more formally defined as follows:

- Layer (n) of the structure makes use of type (n-1) services provided by the lower layers through the type (n-1) access.
- The structure of these lower layers is not known by layer (n) which considers only the services provided by a type (n-1) box.
- Layer (n) is made of type (n) entities which cooperate according to a type (n) protocol.
- The type (n) entities perform type (n) functions using its own capabilities and the type (n-1) services, to provide type (n) services to layer (n+1).

- The specification of layer of the architecture must in some way refer to the set of services provided by lower layers. This will be done by making use of access functions. The set of access functions to a service should be viewed simply as a means to describe the logical structure of the network. It does not necessarily imply the existence of the corresponding interface in any implementation of a piece of the network.



This definition does not imply a one to one correspondence between type (n) and type (n-1) entities. For instance, a type (n-1) entity can very well serve several type (n) entities.

This model is sufficient for representing simple configurations such as two systems connected by a dedicated line. In order to cover more sophisticated configurations another type of assembly must be used consisting of chained boxes. A chain of boxes may be concatenated to form a new box that can in turn be used in another assembly as exemplified in figure 2.

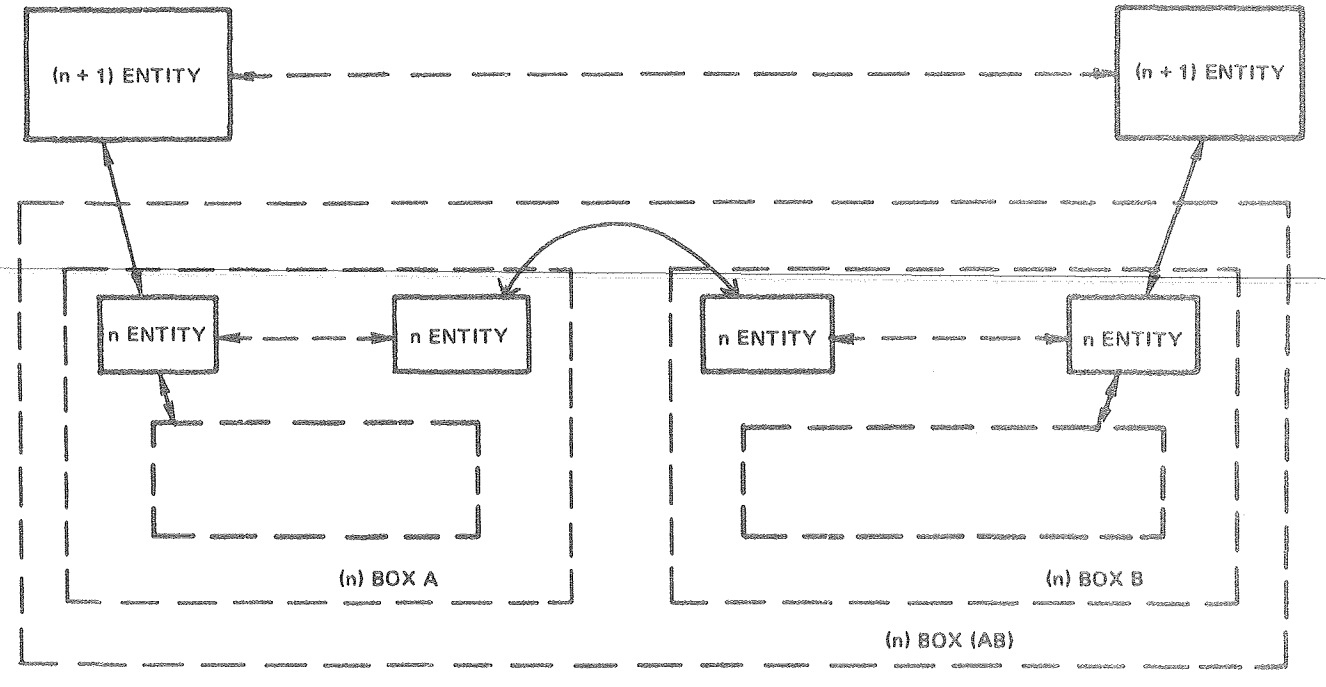


FIGURE 2. CHAIN ASSEMBLY COMBINED WITH HIERARCHICAL ASSEMBLY

2.2 BASIC CONCEPTS FOR THE DESCRIPTION OF LAYERS

Some concepts tend to be applicable to all layers while others apply only to specific layers. This section introduces concepts of the former category, proposing a name and a definition for each of them.

The concepts of layer, service, access, box, entity, protocol, function and interface have already been introduced in Section 3.1.

NOTE ON FUNCTION/PROTOCOL

The same type of function can occur at multiple layers in an architecture, and in many cases even simultaneously. Even though the same function appears a number of times in an architectural description of a system, it may be implemented by a different protocol in each layer. Some examples of functions which occur multiply in a system are: blocking and segmentation, and encryption and decryption. The protocols that support a function are not visible from outside the layer.

Functionality:
Functionality is an attribute of a layer which corresponds to the set of dunctions being performed in this layer of a protocol. Each layer in an architecture ossesses a general functionality or purpose. It is also possible to collapse a layer and nullify its effect on the flow of data through an architecture. Such a nullification is termed "zero functionality".

location:
A place where things happen.

termination:
An addresssable endpoint at a location.

connection:
A logical association capable of transferring data between terminations

data-assurance-unit:
A quantity of data whose successful transfer over a connection is acknowledged.

data flow-unit:
A quantity of data whose transfer over a connection has been authorized.

data-unit:
A quantity of data and control information transferred as a unit over a connection.

data-mapping-unit:
An entity used to map a data-unit of the next higher level onto a data-unit of the current level.

error control: (across a connection):
the function by which errors in the transfer of data across a connection are detected and recovered.

flow control: (across a connection):
the function by which a piece of data is accepted only when it can be transferred across the connection.

establishment, maintenance, termination (of a connection)

transfer:
(across a connection)

mapping:
(next higher layer connection into current layer connection)

blocking:
The function by which several data units of the next higer layer are grouped into one data unit of the current layer.

segmentation:
The function by which one data unit of the next higher layer is split into several data units of the current layer.

NOTE: The question of the ultimate adequacy of the model to account for supervisory functions is still pending.

3.0 REFERENCE MODEL

In the discussion of the reference model, the transport service has been identified as a major boundary between layers. The corresponding first step in the definition of the model is illustrated in figure 3.

TRANSPORT SERVICE

The variety of transmission media provide different levels of service. The transport layers implement the necessary functions to provide network independent services to its users. The functions at that logical interface offer data transmission security and flow control in an end-to-end basis. Data security implies that data units will be delivered correctly and, when required, in the original sequence to the remote transport service interface. The transport service relieves the higher levels of the burden of non-uniform network access. Four gross levels have been identified as making up the transport service architecture.

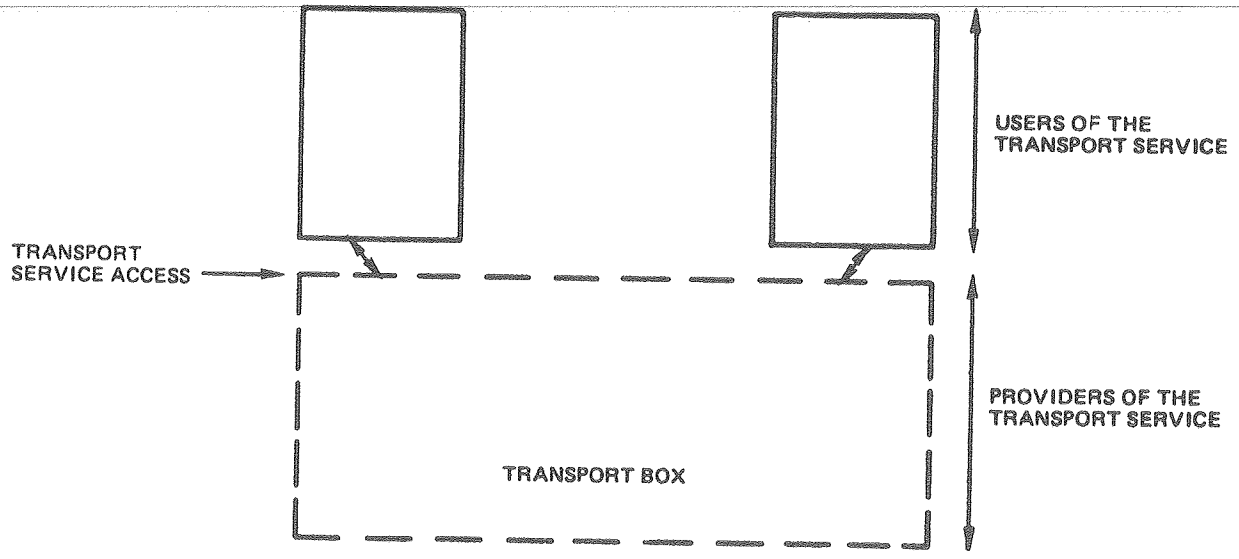


FIGURE 3. TRANSPORT SERVICE

Further analysis of functions to be performed on both sides of the transport service tends to suggest the following subdivision into layers as illustrated in figure 4.

An initial description of each of these layers is given in the following sections.

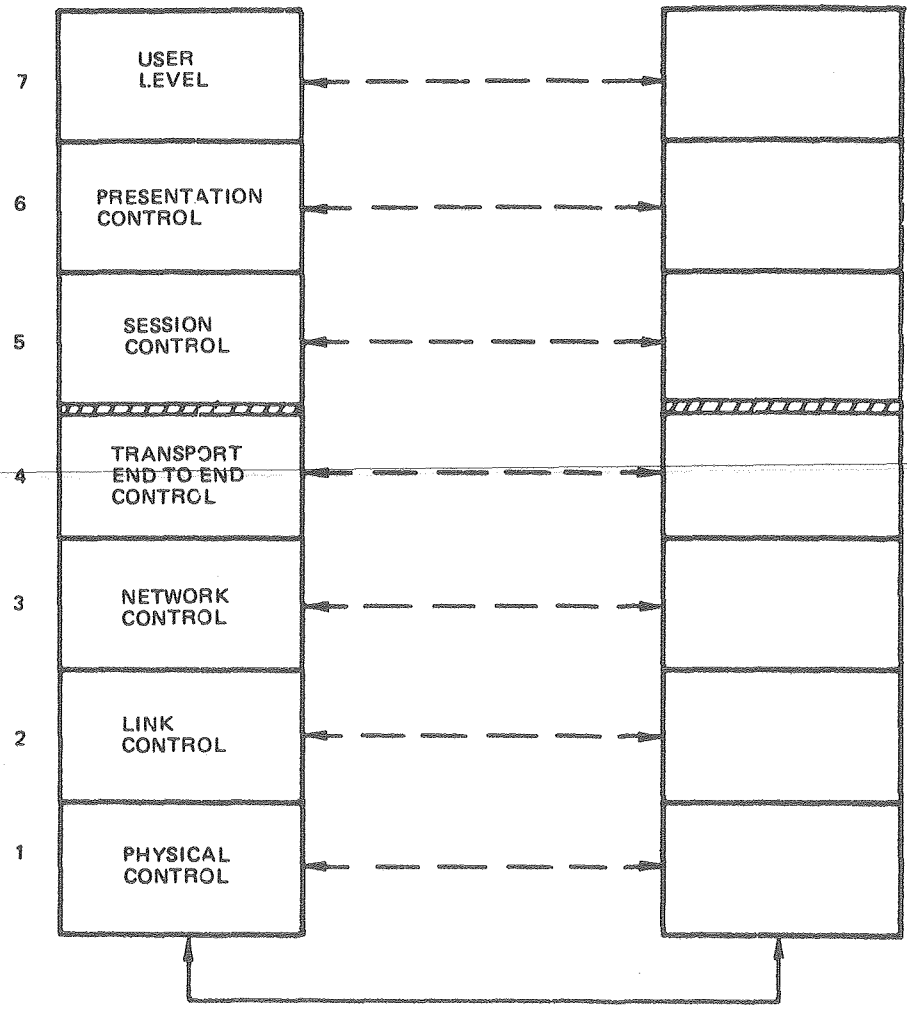


FIGURE 4. LAYERS IN THE REFERENCE MODEL

It should be noted that while layers 4 to 7 correspond to a hierarchical assembly (i.e., perform end-to-end functions), those functions performed in layers 1 to 3 may be chained, as exemplified in figure 5.

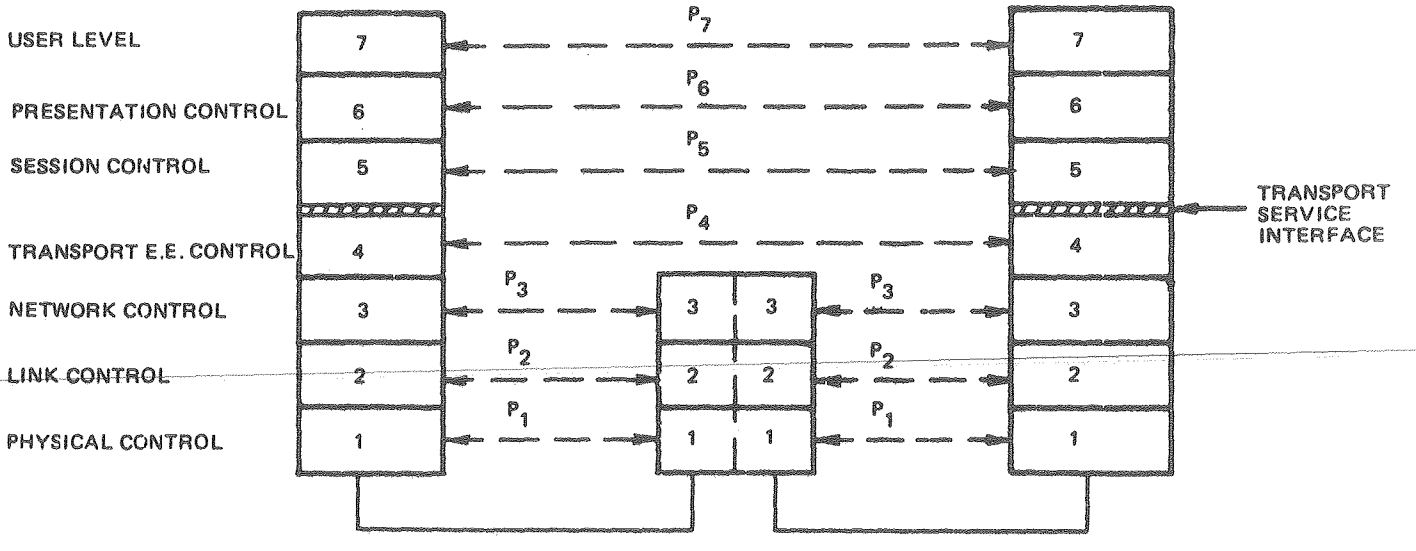


FIGURE 5. LAYERS UP TO NETWORK CONTROL MAY BE CHAINED

4.0 USER LEVEL

4.1 Purpose

Perform application/system activities to provide/support the information processing function of the enterprise.

4.2 Examples of Level 7 Protocols

- Systems protocols:
  - remote batch processing protocol
  - distributed database access protocol
  - file transfer protocol
  - workstation control protocol
  - terminal interaction protocol
- Applications protocols:
  - electronic funds transfer protocol
  - electronic mail protocol
  - civil aviation protocol
  - manufacturer X's order entry protocol

4.3 Unresolved Issues

- Is it appropriate to isolate a set of system services to be provided to all user processes as a separately identified layer?
- How are virtual terminal functions distributed over levels 6 and 7?
- Is level 7 to be considered the highest level appropriate to the scope of the architecture?

0 0 - 0 3 1 0 4 0 4 2

5.0 PRESENTATION CONTROL (LEVEL 6)

5.1 Purpose

Provide required transformations of information being transferred.

5.2 Concepts and Their Relationships

Bracketed terms require further definition.

- [data representation]
- [data template]
- [process-data-unit]

5.3 Services/Functions Provided

- Specify transformations of information to be applied to [process-data-units]
- Apply transformations of information to [process-data-units]

5.4 Examples of Level 6 Protocols

- encrypting/decrypting
- compacting/expanding
- transforming network virtual terminal format to/from local terminal format
- transforming data formats, codes and representations

6.0 SESSION CONTROL (LEVEL 5)

6.1 Purpose

Support a dialog (i.e., structured exchange) between processes.

6.2 Concepts and Their Relationships

- session (a logical connection between two processes)
- [session-commitment-unit]
- [session-recovery-unit]
- [session-interaction-unit]
- [session-quarantine-unit]
- [session-data-unit]
- [mailbox]
- [process]
- [resource] (e.g., terminal, file, ...)
- [job], [task]
- [database]
- [user]
- [workstation], etc.

6.3 Services/Functions Provided

- addressing [workstations] via [mailboxes]
- session establishing, maintaining and terminating including requesting process creation when specified
- sending data and delimiting data enclosures
- notifying processes upon receipt of data and data enclosure delimiters
- segmenting and blocking
- buffering data and controlling flow
- [quarantining]
- supporting checkpoint, recovery and [commitment]
- Initiate, execute and terminate processes
- Initiate, use and terminate interprocess connections (sessions)
- Assign, access and release systems resources
- Assign, access and release data objects

6.4 Unresolved Issues

- Is it appropriate to multiplex sessions onto transport control and points.
- What are the precise semantics of mailboxes and of address.
- Are multiple sessions allowed to be related to each other (e.g., broadcast)?
- What are the requirements for a programming language

interface to session control.

6.5 Required Interface with Transport Services

Provide a network independent means for transporting session-data-units reliably between transport end point.

7.0 TRANSPORT END-TO-END CONTROL - LEVEL 4:

To provide a network independent interface to transport services users. The interface provides for end-to-end control and information exchange across simple or complex networks. Reliability, data assurance and operational attributes are examples of the interface characteristics.

8.0 NETWORK CONTROL - LEVEL 3:

To provide the functions required for intra-network operation such as addressing, routing, etc., above those individual link functions provided by the Level 2 facilities. Level 3 facilities may be viewed as those directing the control of switching points, rather than providing for the transfer of data between the switching points.

9.0 DATA LINK - LEVEL 2:

To provide for the reliable interchange of data between equipments connected by Level 1 facilites.

10.0 PHYSICAL LINK - LEVEL 1:

The physical, electrical, functional, and procedural characteristics to connect, maintain, and disconnect the physical circuits between equipments.

APPENDIX 1

LIST OF SC16 DOCUMENTS, INITIAL CONTRIBUTIONS ON THE REFERENCE MODEL

DOCUMENT NUMBER	TITLE	DATE
1	Report on Open Systems Interconnection prepared by TC9 (ECMA)	December 1977
3	USA contribution to 97/16 - Draft Reference Model on Distributed Systems	January 1978
7	Source: UK - Conceptual Model for Communication within Open System	January 1978
10	Source: Canada - Working Paper on Communication Systems Architecture	February 1978
13	Source: France - Further Definition of the Model for Open System Architecture	February 1978
16	Japanese Report on the Actions and Status in the Relevant Topic	February 1978
17	Source: Japan - Initial Consideration of Providential Model	February 1978
25	Source: AFNOR - Basic Architecture for a Terminal to Application Protocol	February 1978
28	Source: UK - Conceptual Modular Integrity Control in Open Systems	February 1978

APPENDIX 2

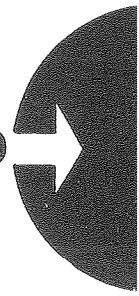
INITIAL LIST OF TERMS (could be turned later into a glossary)

- Open-ness
- Model
- Layer
- Service
- Access
- Box
- Entity
- Protocol
- Function
- Interface
- Hierarchical Assembly
- Chain Assembly
- Concept
- Objects
- Functionality
- Location
- Termination
- Connection
- Data-Assurance-Unit
- Data-Flow-Unit
- Data-Unit
- Data-Mapping-Unit
- Error Control (across a connection)

- Flow Control (across a connection)
- Establishment/Maintenance/Termination (of a connection)
- Transfer (across a connection)
- Mapping
- Blocking
- Segmentation

0 0 0 3 5 1 0 4 0 4 5

NETWORK PROTOCOLS AND ARCHITECTURE



00-0310406

SOFTWARE ARCHITECTURE FOR A MICROPROCESSOR-BASED PACKET NETWORK

Holger Opderbeck, Jay H. Hoffmeier, and Roy L. Spitzer

Telenet Communications Corporation

Abstract

Continuing improvements in the cost effectiveness of digital electronics have resulted in progressive step-function improvements in the cost effectiveness of packet-switched data communications networks. First and second generation packet switching systems utilized minicomputers for switching and network control functions. Third generation packet switching networks will be based upon multimicroprocessor systems which offer greater flexibility, reliability, and cost effectiveness than minicomputer-based systems, particularly in an environment in which traffic is growing rapidly and a variety of communication interfaces must be accommodated.

This paper describes the motivation for and the architectural design of a family of multimicroprocessor-based systems, developed for and currently installed throughout Telenet Communication's public packet switching network. The hardware architecture of these systems, called Telenet Processors (TPs), was described in detail in an earlier paper (Reference 1). This paper describes the software architecture of the Telenet Processors, the relationship between the software structure and the hardware structure, and the special facilities incorporated in the software to accomplish the design objectives.

1. INTRODUCTION

First generation packet switching systems are exemplified by the ARPA network, developed under the auspices of the Defense Department's Advanced Research Projects Agency. The ARPANET, which was implemented in the late 1960's and the early 1970's utilized Honeywell 516 and 316 minicomputers. Second generation packet networks are exemplified by Telenet Communications Corporation's public packet switching network, designed in 1974. The Telenet network utilizes Prime 200 minicomputers, at a cost of approximately one-third of the original ARPANET equipment.

In 1976, Telenet undertook the design and development of a new family of 'Third Generation' packet switching systems, to be phased into the Telenet network over a period of several years. The first of these systems, which are called Telenet Processors, was installed in August 1977, and the deployment of TPs throughout the network and at Telenet customer sites is currently proceeding at a rapid pace.

The Telenet Processors were specifically designed to serve as network access and switching systems within Telenet's public packet network. As such, the design objectives for the TPs were (1) maximum flexibility in configuration and capacity, (2) high inherent reliability and operational maintainability, and (3) maximum cost effectiveness.

In order to serve a broad variety of customer computer and terminal devices, and in order to accommodate the constantly growing, but unpredictable, number of line interfaces and traffic volume, a primary objective was the ability to configure systems with a wide range of line terminations (from 5 to 500), with a mixture of line speeds and protocols, and with sufficient memory and processing capacity to accommodate an increasing load without degradation of performance. To achieve this objective, each additional group of line interfaces in the TPs contains its own microcomputer, thereby adding memory and processing power as the number of lines per system grows. Also, since each group of lines contains its own microcomputer, it is quite straightforward to accommodate a wide mixture of interface protocols. Finally, to facilitate network evolution and installation of new customer network access facilities, the ability to down-line load software systems and customer-specific network control parameters was incorporated.

The second primary objective was maximum inherent reliability and maintainability. The resulting TP system design provides for the optional redundancy of all subsystems, with automatic switchover and recovery. No single hardware component or subsystem failure can cause a TP system failure. To facilitate maintenance of TPs, constant status reporting to a network control site is provided, as well as remote diagnostic program-based facilities.

The third objective was cost effectiveness. The use of low-cost microcomputers, coupled with the ability to install additional capacity in small increments, provides for a significant improvement in cost effectiveness over earlier minicomputer-based packet switching equipment. Additionally, to minimize operations and maintenance costs, many remotely controlled functions are provided to reduce the number of field trips to modify and/or repair installed TP systems.

2. HARDWARE ARCHITECTURE

The major design goals for the TP system were high processing power, large memory capacity, high reliability, low cost, modular structure, and flexibility. These goals were realized through a multimicroprocessor architecture where each microprocessor has its own dedicated local memory to operate independently as well as full access to a large common (or main) memory to facilitate the exchange of information. A dual bus architecture with dual memory banks is provided to achieve the desired redundancy goals. Access to main memory is controlled by a special arbitrator board which can also be duplicated in redundant systems. Figure 1 shows a block diagram of the

0 0 - 0 5 1 0 : 0 4 7

TP hardware architecture.

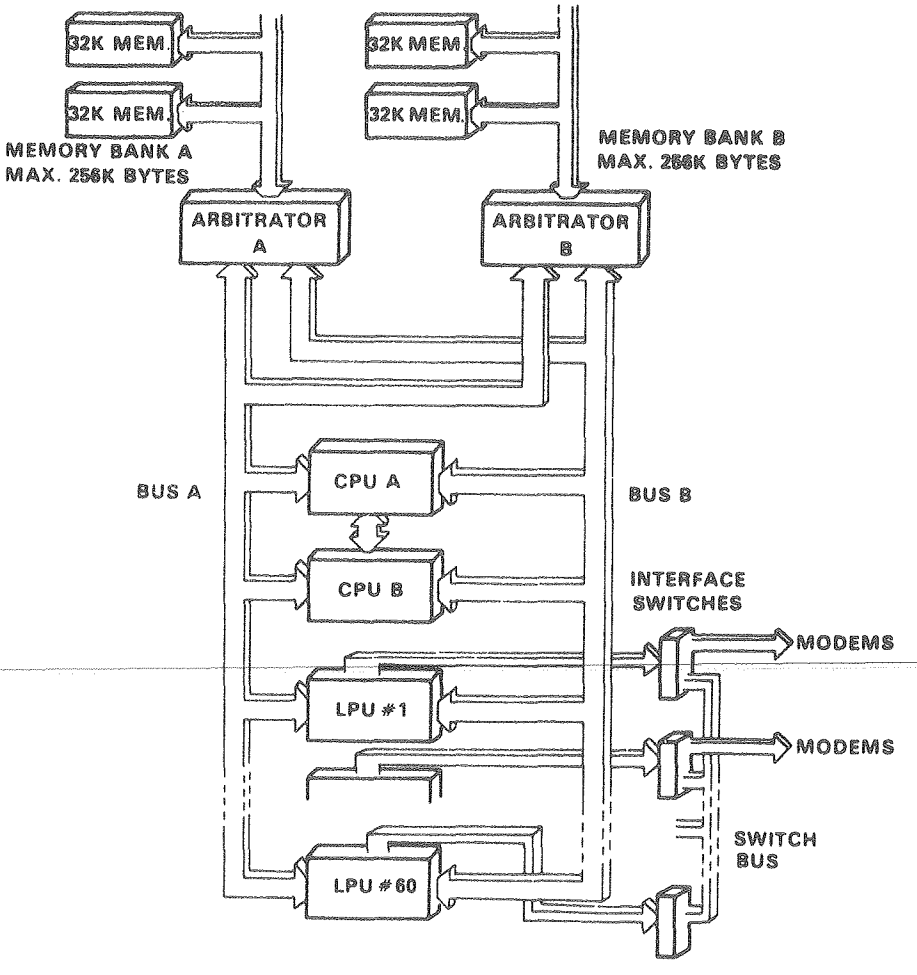


Figure 1 TP HARDWARE ARCHITECTURE

A microprocessor (the MOS Technology 6502A) is used on two types of boards to obtain the required processing power: 1. Line Processing Units (LPUs) which handle between 4 and 8 communication lines and 2. Central Processing Units (CPUs) which are used to control the entire system. A TP may contain up to 60 LPUs and 2 CPUs. Each LPU is similar except for the number of line interfaces and the types and speeds of the interfaces. Current units are designed to handle 4 or 8 asynchronous lines (LS cards), 4 binary synchronous lines (MS cards), or 8 SDLC lines. An LPU typically has 8KB of RAM and 1/2KB of ROM as dedicated local memory. Each CPU has 16KB of local memory and 1K ROM. The CPU can start, halt, and reset, as well as exchange interrupts with any of the LPUs or the other CPU.

Main memory is organized into two separate banks each having up to eight 32KB modules giving a total main memory capacity of 512KB. Each memory bank is further subdivided into 2KB blocks which are accessed through page maps by the CPUs and the LPUs. Changes in the page map allow all of main memory to be accessed. The page map also provides a write protect mechanism.

A more detailed discussion of the TP hardware architecture is given in Reference 1.

3. TELENET PROCESSOR SOFTWARE STRUCTURE

a) Introduction

The X.25 interface protocol (see Reference 2) is the foundation on which the Telenet processor software is built. This protocol is used in two different ways: 1. As a standard interface to connect TPs to a network and 2. as an internal interface in self-contained TP networks. Figures 2, 3, and 4 show various configurations in which TPs can be used. In figure 2, the TP acts solely as a network access machine. It

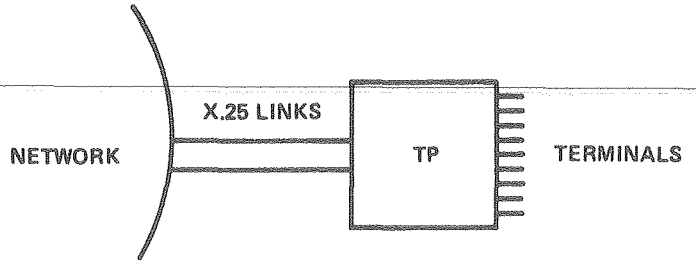


Figure 2 NETWORK ACCESS MACHINE

translates the various terminal protocols into X.25 and multiplexes the traffic over one or more high speed lines to the network. Conversely, data received from the network is demultiplexed and a protocol translation from X.25 to the specific terminal protocol is performed.

In figure 3, the TP connected to the network (TP1) supports not only local terminals, but also other TPs (TP2 and TP3) through an X.25 link on the demultiplexing side. By mapping virtual channels between X.25 lines one can now build hierarchical subnetworks. Call routing in such subnetworks is accomplished through address tables provided by the Network Control Center (discussed below). These address tables permit automatic load sharing over multiple high-speed lines, as well as automatic transfer to alternate lines in case of line or node failure. Figure 4 shows a self-contained TP network with its own Network Control Center. Here calls are routed through the network by means of dynamically updated routing tables. To overcome line and node failures an automatic reconnection facility is provided.

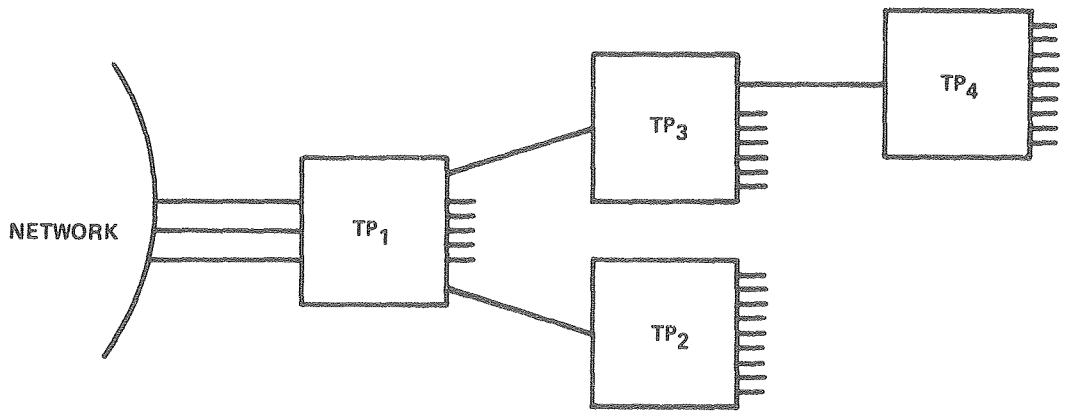


Figure 3 HIERARCHICAL SUBNET ARCHITECTURE

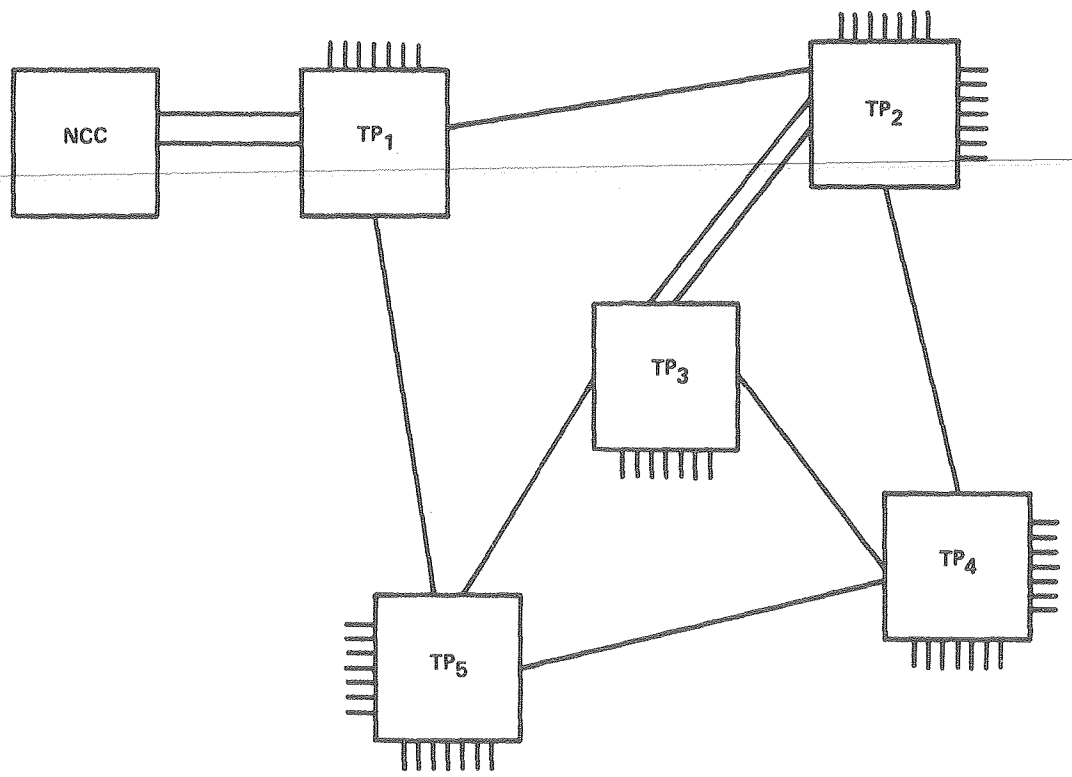


Figure 4 SELF-CONTAINED TP NETWORK

b) Software Structure

The X.25 interface protocol consists of three distinct levels of control procedures: 1. The Physical Interface, 2. the Link Access Procedure and 3. the Packet Level Logical Interface. The Physical

Interface specifies the use of a duplex, point-to-point synchronous circuit, thus providing a physical transmission path. The Link Access Procedure (level 2) ensures that packets provided by the packet level are accurately exchanged between both ends of the transmission line. The Packet Level Logical Interface (level 3) specifies the manner in which control information and user data is structured into packets. The structure of the X.25 interface protocol is directly reflected by the structure of the software that implements this interface in the TPs.

The TPs use a 'system master' architecture where one card acts as the central controller of the system. All other line processing units (LPUs) are called slave cards. In the smallest configuration, an MS card is the master of the system (figure 5). In addition to running the major part of the TP operating system (TPOS), the master in this configuration is also processing the X.25 level 2 (Link Access Procedure) and level 3 (Packet Level) code. Each slave contains a smaller part of TPOS through which it is interfaced to the remainder of the system. The main function of the slave is the protocol translation from whatever type of terminal it is supporting to X.25 and vice versa (see section 4 for more detail).

The capacity of the system in figure 5 is limited by the processing power of the master card. This bottleneck can be overcome by introducing the CPU card as the master of the system (figure 6). In this configuration, the master is only responsible for processing TPOS and the X.25 level 3 protocol, while the X.25 level 2 protocol is now taken care of by the MS card which is a slave within this system.

With the software modules shown in figures 5 and 6, the TP can only act as a network access machine (figure 2). In order to build hierarchical subnets and self-contained TP networks another "switching" module needs to be added to the master. This is shown in figure 7 where the master now provides the packet level procedure for the locally attached terminals as well as the packet switching function for packets going from one X.25 line to another X.25 line.

Figures 5 through 8 show the subdivision of main memory into five major areas. The communication area is used for the exchange of commands between master and slave cards. A copy of the local code of all slave cards is also kept in main memory. This allows the slave cards to reload a fresh copy without the help of the Network Control Center. The next area contains all the code that is directly executed out of main memory by master or slave cards. All code in main memory is write protected and shared by all cards that run on the same software system. The remaining RAM code is used for buffer and table space. The 4K of ROM code is discussed in section 6.

Figures 5 through 7 show how the extraordinary modularity of the TP hardware is matched by a similarly modular design of the TP software. This is further demonstrated by the configuration shown in figure 8. Here the packet level processing for the local terminals has

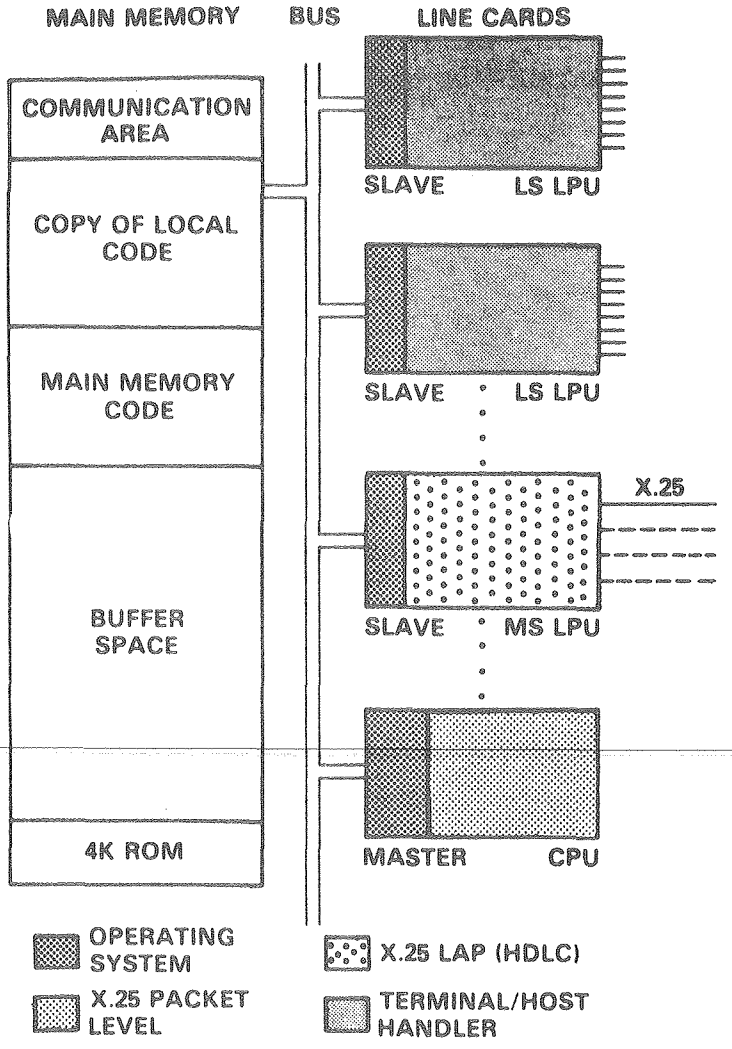


Figure 6 TP 4000 WITH CPU MASTER

3. Redundancy control
4. Status reporting

The information exchange between the various line cards is organized in terms of commands that are sent from the master to the slaves and vice versa. For each slave there exists an information exchange area in main memory which is used to send and receive commands. To send a command the proper information is written into this area. Then a flag is set to signal the receiver that there is a new command that needs to be processed. When the command processing is complete, the flag is turned off and the next command can be sent. To overcome the case where the information exchange area is in use when the master wants to send a command to a slave, TPOS automatically

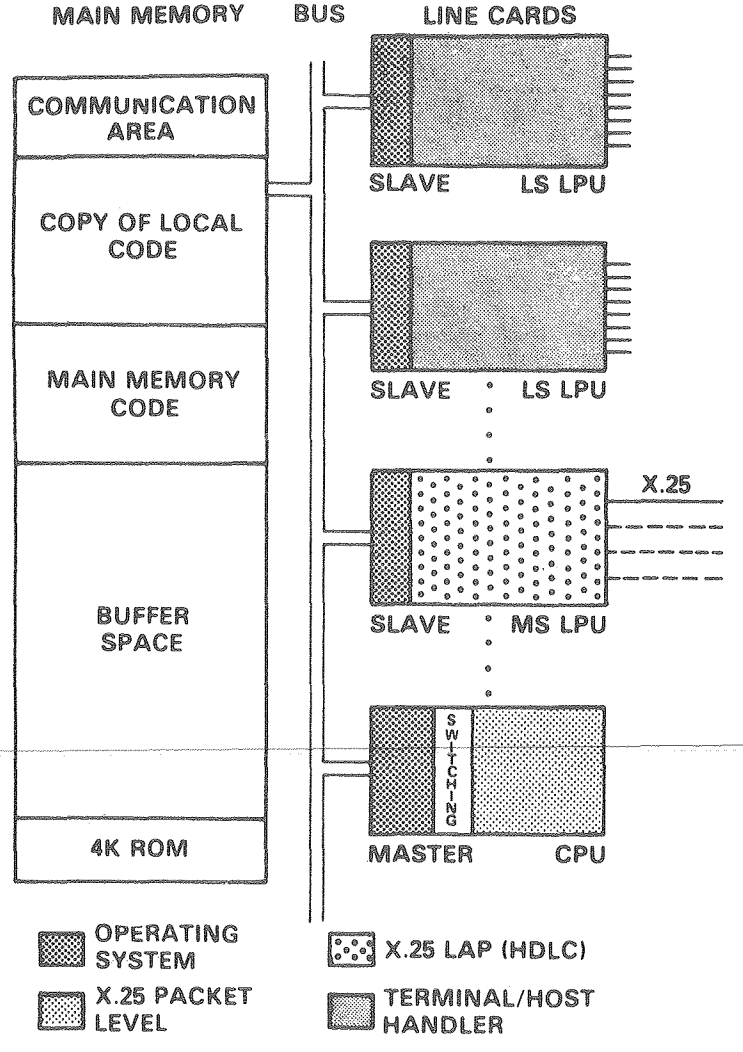


Figure 7 TP 4000 PACKET SWITCH

buffers the commands and delivers them to the slave as fast as possible.

In the case when the master code is running on an MS card, commands are received by continuously polling the flag bit in each exchange area. When the CPU is the master of the system the sending of a command is accompanied by the sending of an interrupt which reduces the polling overhead and speeds up the command transfer.

Up to four parameters can be passed with a command to the receiver. Typical parameters are: the address of a buffer in main memory, or the number of one of the up to eight lines a slave card is handling. The command exchange facility is used by the operating

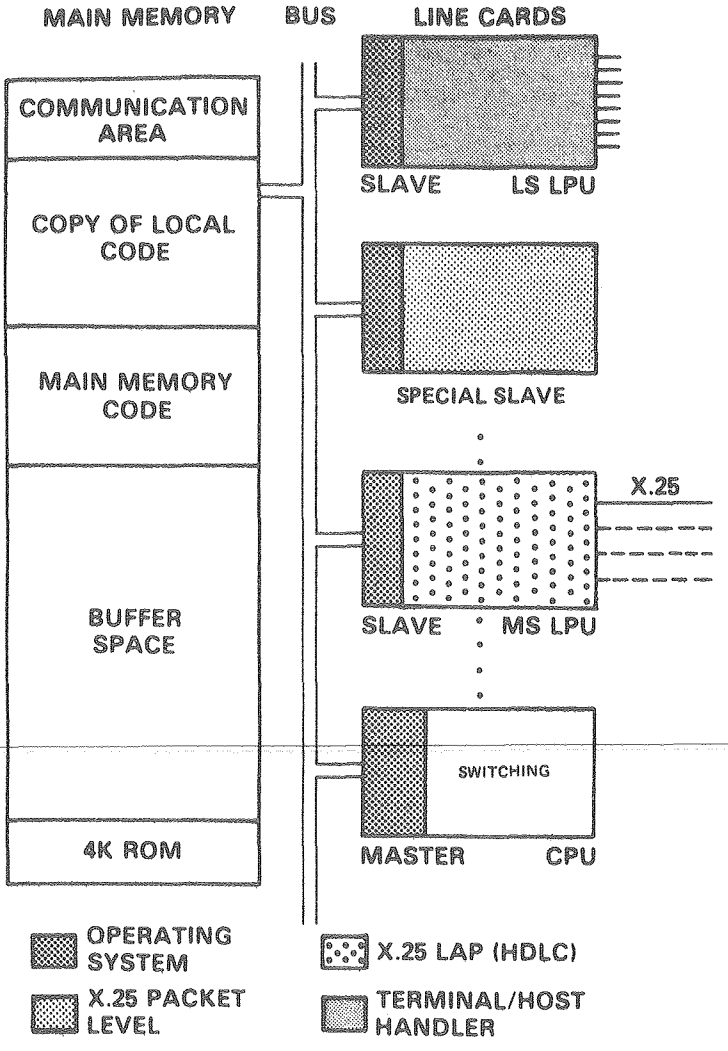


Figure 8 TP 4000 PACKET SWITCH WITH SPECIAL SLAVE

system to perform its internal functions as well as by the terminal handling code to communicate with the master card. Typical operating system commands are initialization commands, buffer request commands, and buffer transfer commands. There is also a HELLO command which is sent periodically from all slaves to the master to verify that the slave is still operating.

The syntax and semantics of the commands exchanged between the terminal handler and the X.25 packet level processor reflect the translation from the specific terminal protocol to X.25 and vice versa. There are commands for sending data, "qualified" data, interrupts, and resets. Other commands allow the establishment and clearing of virtual connections.

Buffer management is the second major function that TPOS is providing. The entire buffer pool is subdivided into blocks of 144 bytes of which 128 bytes can be used to transfer data. The data area of such a "full buffer" can be further subdivided into 4 "quarter buffers". Full buffers and quarter buffers are distributed by the master to the slave cards as necessary. Quarter buffers are automatically reassembled into full buffers whenever possible. A garbage collector is used to reclaim buffers that were allocated to a slave but were never deallocated due to a failure of the slave card.

Redundancy control, the third major function of TPOS is discussed in the next section. The fourth major function of TPOS is concerned with continually providing the Network Control Center with all information and control it needs to remotely operate a large-scale network. This function will be discussed in more detail in Section 6.

5. REDUNDANCY

The TP system has been designed to overcome single unit failures without loss of capabilities. This objective is achieved through the use of two types of redundancy: 1. Central logic redundancy, and 2. Back-up card redundancy.

In a system with central logic redundancy all centrally used resources are duplicated. In particular, there are two arbitrators, two sets of memory boards, two power supplies, and two separate busses (see figure 1). All line cards are attached to bus A or B and will normally switch only in case a failure occurs. Both halves of the redundant system have their own master and run in normal mode as two independent systems. A central logic failure or a failure of the master card will cause the slave cards to switch to the alternate bus. The slave cards are treated on the alternate bus like any other slave. If the master card detects a central logic failure it also switches to the alternate bus. On this bus it will normally have to run as a slave. Should, however, the master on bus A (B) fail while the central logic of bus B (A) is down, then the normal master of bus B (A) can become master on bus A (B). Thus, normally no combination of central logic and master card failure will incapacitate the TP. Slave cards will periodically try to switch back to their "home" bus if here is no established connection to be served. Bus switch-over can also be effected by commands from the Network Control Center.

The second type of redundancy, called back-up card redundancy, allows the TP to overcome line card failures without loss of capabilities. Whenever the master detects that one of the slaves has stopped operating, the master switches over all lines handled by that slave card to a back-up card. The back-up card then loads the proper line handling code out of the main memory and starts operating. Thus, a spare card can back-up several other cards of the same type independently of the software it is running. Such a back-up may be on a one for one or one for N basis. Since the CPU is required for the switching of the lines, only TPs with a CPU provide back-up card

redundancy.

6. NCC SUPPORT

There are major functions that the Network Control Center (NCC) performs with respect to the TPs: 1. it provides the TP with all its code and configuration information (tables); and 2. it provides the network operations staff with all the necessary information to remotely control the operation of a large-scale TP network; and 3. it collects status, performance, and traffic data for analysis and accounting purposes.

The arbitrator of each TP contains 4K of ROM code which is used to load the system. This ROM code contains a complete implementation of the X.25 protocol. To load the system the TP sends a CALL REQUEST packet to the NCC and thereby establishes an X.25 connection. The TP's own address and the address of the network control center are encoded in strap bytes that are read by the ROM code. Thus any host on the same or some other network that is accessible through an X.25 connection can act as a loading device for the TP.

From the address of the originator of the CALL REQUEST packet the NCC determines which software system and which tables are requested. The software loaded into a TP system consists of the following four parts: 1. software for the master card, 2. software for the slave cards, 3. a local memory software loader, and 4. a table processing program. A program (the 'System Builder') has been developed at the NCC which merges these four software packages into one loadable system and automatically determines the initial page maps for all processors.

Following these four programs, the NCC loads the tables defining the TP configuration. As the TP receives these tables, the table processing software expands the tables, stores them into main memory and links them through pointer chains. Control is then turned over to the master card. The master card proceeds to initialize the system and then indicates to the slave cards that they can now load their local memory using the Local Memory Software Loader. (A slave card can reload its local memory from main memory any time it so desires or is instructed to do so by the NCC). The TP system is now operational. The entire loading process takes two to four minutes depending on the speed of the access line.

Several other programs have been developed to assist the network operators in controlling a TP network. The most important ones are the Table Builder, TDT (Telenet Debugging Tool), and the Status Monitor. These programs reside in the NCC computer.

To construct the tables necessary to define the TP configuration, an operator-oriented, interactive program has been developed. This program prompts the operator in an English-like question-answer session to provide the necessary information. Default values are automatically

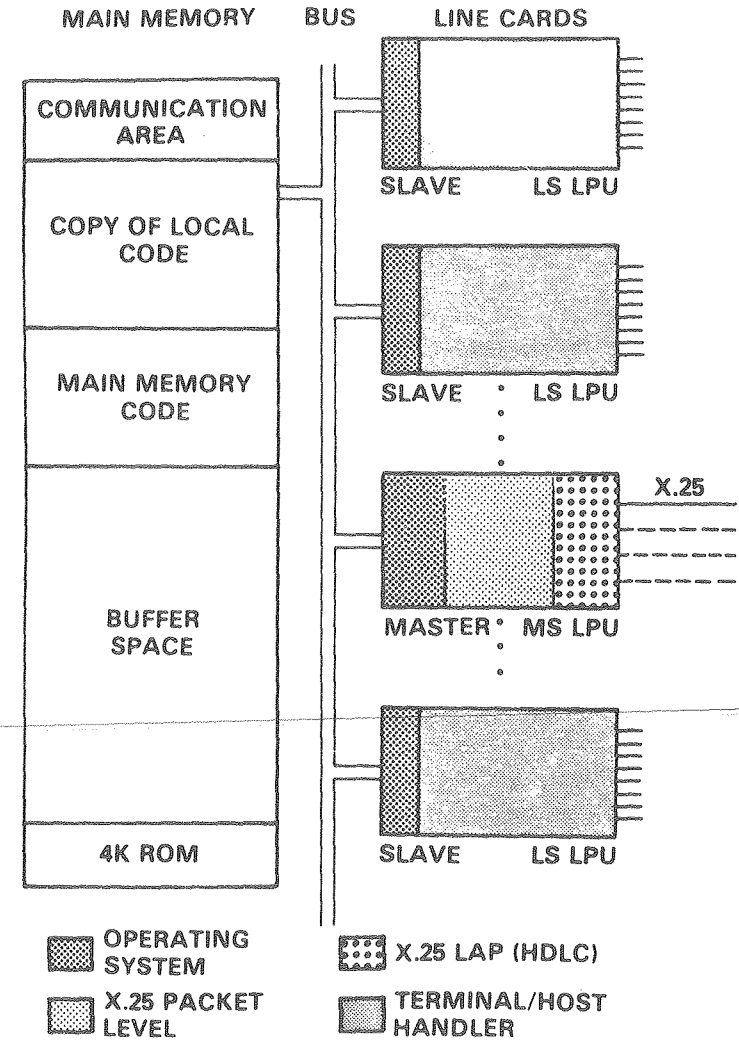


Figure 5 TP 4000 WITH LPU MASTER

been moved to a separate line card to relieve the master from this burden and thereby increase the overall capacity of the system. The additional line card, which could also be a second CPU, runs as a special slave whose only task is to increase the processing power of the system.

4. TELENET PROCESSOR OPERATING SYSTEM

The Telenet Processor Operating System (TPOS) provides four major functions:

1. Inter-card communication
2. Buffer management

inserted whenever possible. When all the information has been collected, the Table Builder formats the data properly and stores it on disk at the NCC where it is then available for the next reload of the TP.

The tables contain, among other things, information about the

* system redundancy
* passwords
* asynchronous port definitions (address, speed, code, dial vs. leased, single line vs. rotary, privacy list, autoconnect address, padding, etc.)
* synchronous port definitions (address data, capacity, etc.)

TDT allows the operator, through English-like commands, to invoke a large number of functions which are implemented by the use of cooperating software in the NCC computer and in the TPs. In particular, it allows the user to inspect any memory location in any TP in the network. TDT is typically being used to analyze reported hardware or software problems or simply to monitor the operation of a TP. For this purpose it provides the operator with statistics about line performance (e.g. CRC and parity errors), commands to loop-back lines, commands to activate and deactivate ports, etc. There are also commands to change most of the configuration data (e.g. port speed and code). Other commands allow the operator to reset, halt, or reload any slave LPU, and to initiate a bus switch-over or a line switch-over to a back-up card, as well as a reload of the entire system.

The Status Monitor program provides the network operator with an overview of the status of a particular TP or a specified set of TPs. It displays the status of all lines, cards, and the attached devices. The status information is obtained from the TP's through periodic status reports, as well as alarm reports to inform the operator of exceptional conditions in the network. The Status Monitor also collects statistics of various sorts. In particular, it collects data on line loading, peak utilization, etc.

7. SOFTWARE DEVELOPMENT TOOLS

All tools for the TP software development have been designed and built by Telenet and were optimized for the specific application. They include: TML (Telenet Microprocessor Language), a linking loader, a debugger, and a simulator.

TML provides the programmer with higher level language constructs required in data communications programming. In particular, facilities for easily accessing control blocks and data buffers as well as setting and clearing control bits are available. Programming is further simplified through the use of IF-ELSE-FI and DO-WHILE statements. TML is fully compatible with assembly language, which can be interwoven with TML wherever the programmer deems it advantageous. Experience has shown that more than 90 percent of the code is usually written in TML.

Assembly language is used for the remaining code to optimize the execution of statements that are critical for the performance of the system (e.g. interrupt level routines).

The linking loader provides a means by which several independent modules can be linked together to provide the load module of a single system. In this way several programmers can write and debug individual modules that will be linked to form a much larger system. The actual debugging of independent modules and entire systems is done with the use of the simulator and the debugger which will now be discussed.

The debugger is the single most important tool that was developed for the TPs. It extends the functions that are normally available in debuggers (e.g. the ability to examine and set memory locations, breakpointing, single stepping through programs) into the multiprocessor environment. This means that the programmer has complete control over each line card in the system. In particular, there are commands to attach to, load, stop, and reset any given card.

The main code of the debugger is executed by the CPU. This CPU-resident part of the debugger uses the hardware interrupt facilities between the line cards and the CPU to talk to cooperating code that is executed by the line cards. The user can logically attach to any line card and debug the code on that card as if it were running as a single-processor system. The physical connection to the debugger is made through the asynchronous port on the CPU board.

The simulator is the second most important software development tool for the TPs. It was developed in FORTRAN to run on an in-house time-sharing system. The simulator emulates not only the detailed operation of each of the microprocessors, but also all the details of the line interface hardware as well as the interaction between the various cards. Thus the same programs that run on the hardware can be loaded into and executed by the simulator.

Aside from all the normal debugger facilities the simulator provides the programmer with additional controls that are normally not available. In particular, the programmer has the ability to execute the program not only in a forward direction, but he can also back-track through the program for up to 1000 instructions, restoring all locations to their original contents. This turns out to be extremely valuable in debugging software. The simulator also collects extensive statistics on the execution of programs (e.g. number of accesses to local memory, number of accesses to main memory) which allow the user to, for instance, calculate the processor utilization under various conditions.

The simulator enables the user to debug a large number of programs before the TP hardware was available. Thereby the start-up time for bringing up the first system on the TP was shortened considerably. The simulator is now used in essentially two modes: 1. to test small subroutines or modules under as many different sets of input data as

possible, and 2. to test entire software systems through the use of command files for the simulator.

8. CONCLUSIONS

While minicomputers have been used for nearly a decade in packet switching networks, they were not specifically designed for such applications, and consequently do not offer the system operator the optimal mix of flexibility, reliability, and cost effectiveness. Multimicroprocessor based systems offer the system operator considerably enhanced capabilities in each of these areas. The software architecture for such systems is structured in a way which reflects the hardware architecture. Due to the nature of the functions performed within a packet switching system (e.g. line interfacing, protocol conversion, status reporting, routing, etc.), these functions can be 'compartmentalized' into separate physical processing units. Consequently, multimicroprocessor-based packet systems achieve a high degree of parallel processing and are therefore extremely cost effective. It is expected that these systems will find wide application in a broad range of data communication systems and in many other fields as well.

9. ACKNOWLEDGMENTS

We gratefully acknowledge the help of other members of the Telenet technical staff in the development of the TP software system. In particular, we would like to thank Christine Bjorklund, Ben Lisowski, Howard Seid, and Marc Seriff for their contributions. We would also like to thank Stu Mathison for his numerous valuable suggestions for the preparation of this paper.

REFERENCES

1. Newport, C. B. and Kaul, P.
'Communications Processors for TELENET's Third Generation Packet Switching Network'
Proceedings, EASCON - 77, 8-2A to 8-2L

2. Rybczynski, A., Wessler, B., Despres, R., and Wedlake, J.
'A new communication protocol for accessing data networks - The international packet-mode interface'
Proceedings National Computer Conference 1976, 477-482

PROTOCOL CONVERTERS AND USER INTERFACE IN THE CYCLADES NETWORK

N. NAFFAH and H. ZIMMERMANN
IRIA - 78150/ROCQUENCOURT - France

Abstract

In 1975, the CYCLADES computer network moved from experimental to operational stage. Since then two factors in particular have influenced new developments and studies in CYCLADES :

1 - The drastic reduction of the cost of the microprocessors and LSI memories,
2 - Adoption of recommendation X25 in the CCITT as a standard interface for public packet switching networks.

The first factor opened up new possibilities for interconnection products hardware. New concentrators for connecting TTYs and Interface Converters for connecting hosts were designed around a multi-microprocessor system (MLV). A new microprogrammed multiplexer (MUMI) was also developed in order to extend the network interface near the host site. An Intelligent Terminal (TIPAC) was microprogrammed for connecting three data-entry display-stations to the network. The second factor brought above the need to study the problem of interconnection of Datagram Networks with Virtual Circuit (X25) Networks. CYCLADES conducted a study to solve such a problem.

The aims of this paper are to give the general design of a "Customized User Interface" (CUI) for connecting any type of Informatic equipment (terminal, host, ...) to the network, and to present an overview of the different Protocols Converters that were developed and studied in CYCLADES. The principal factors which affected the design of these products are discussed. Emphasis is given to products based on microprocessor technology.

I - INTRODUCTION

Packet-switching technology is now well stabilized. Many networks based on this technology are already operational in different countries. Public packet-switched service will soon be offered in the majority of industrial countries. Networks could not reach this stage of maturity without the joint effort of many research centers in different countries, international working groups (IFIP and ISO), and the results of experimental networks such as ARPANET (1), CYCLADES (2), EIN (3) and DDX-2 (4). Experiments were conducted in different fields :

1°) - in transmission, subjects like topology design, adaptive routing, queuing and buffering were covered.
2°) - in inter-process communication, end-to-end protocols (Transport and Virtual Terminal) were defined and implemented.
3°) - in interconnection, "Concentrators" were designed for connecting terminals, "Adapters" were developed for connecting host computers, and "Gateways" were studied for internetting.

In this paper, we focus upon the third field. After giving very briefly the Basic Architecture of Computer Networks and Protocols (section II), we consider the general design of Interconnection Products that can apply to any case of interconnection in one Network or between two Networks (section III). The notion of the Customized User Interface (CUI) is then introduced (User means Terminal of Host using the network). The CUI consists of Processors performing Protocol Conversion (section IV).

It provides a quick and simple method for connecting Users to the Network without making any software or hardware change in the terminal or in the host. In section V, we present the different products constituting the CUI in CYCLADES : Concentrators (CT) for teletype-like terminals and remote-Job-entry terminals, Interface Converters (IC) for host computers, and microprogrammed Multiplexer (MUMI) for host connected through point-to-point asynchronous links. This will be followed (section VI) by a description of some microprocessor - based solutions (MLV) for the CUI.

II - NETWORK ARCHITECTURE AND PROTOCOLS

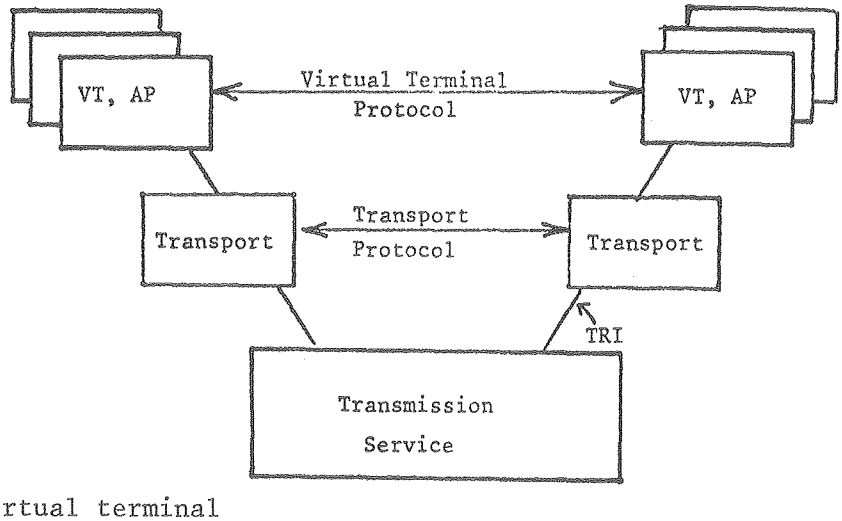
It is now widely accepted that the Computer Network Architecture must be defined in a structure of hierarchical and independent layers according to the reference model studied in ISO (5). Each one of these layers :
1°) - consists of entities communicating according to a Protocol,
2°) - uses the service offered by the lower layer,
3°) - offers a service to the upper layer.

Adjacent layers communicate by means of a simple and well defined interface.

In already operational Computer Networks (e.g. CYCLADES) we find (Fig. 1) :
- At the bottom level, a packet switching subnetwork offering a transmission service (Datagram (6) or Virtual Circuit (7)). The subnetwork (8) in the case of CYCLADES, performs the simple function of switching packets independently and transparently. Those packets, or "Datagrams" are self-contained. They include all the information necessary for their routing. The use of such a service requires only :
  1°) - the handling of a Line Control Procedure (e.g. HDLC (9)) to exchange frames carrying packets,
  2°) - formatting packets to be sent, and interpreting received packets.
- On top of the transmission box (Fig. 1), the transport service is provided by Transport Stations (10) generally located in Data Processing Equipments (DPE) connected to the transmission box. They

permit the handling of "Liaisons" between "Ports" assigned to the Processes running in DPEs.
The transport protocol deals with functions like addressing, liaisons initiation, fragmentation and reassembly of messages (when needed), error and flow control on whole blocks of information (called letters) exchanged between processes. Work to define a transport service independent of the various transmission media is undertaken by different working groups (INFOREP, the French Association of Distributed Informatics Users (11), IFIP TC.6.1 - INWG (12), ...). In CYCLADES, the transport station that has been implemented is completely defined in (13).

- At the top level, the Virtual Terminal Service is the set of conventions used for dialogue between Application Programs (AP) and a standard terminal representing one class of terminals (e.g. teleprinter-oriented terminals). This standard terminal called the Virtual Terminal (VT), has the most current functional characteristics of terminals in any one class. The Virtual Terminal Protocol (VTP) defines all the rules for the standard representation and the meaning of the messages exchanged between VTs and APs. The advantage of the VT concept is that it facilitates the task of adapting one application environment to a variety of terminals connected to the network. A unique VT handling module needs to be implemented on the host side in the Application environment (e.g. Access method, teleprocessing monitor). On the terminal side, an adaptation is needed for converting non-standard real terminals to Virtual Terminal.
In CYCLADES, the VTP that has been implemented covers the scroll-mode terminal class (14). Another VTP was defined by INFOREP for Data-Entry terminals (15). Work is continuing in this field in different standardization working groups (ISO, AFNOR (Association Française de Normalisation - ISO French member body) , INFOREP, ...).



VT = Virtual terminal
AP = Application program
TR I = Transmission interface

Fig. 1 - Network Architecture

III - GENERAL DESIGN CONSIDERATIONS

When two informatics systems (Network, Computer, Terminal) wish to communicate, the first criterion is that in both systems each level concerned with the communication must respect a set of rules defined in a Protocol. For example, when a start-stop terminal has to be connected to a computer system (Fig. 2), they must have two compatible levels : physical (bit rate, character code, ...) and logical (alphabet, control characters, ...). Between 2 computers (Fig. 3), interworking requires compatibility at the physical level, line control procedure level, and user level (Application and terminal).

All the problems related to protocols in a homogeneous environment (e.g. built by one manufacturer) were covered by the traditional teleprocessing systems supporting different terminals and line procedures (e.g. polling) according to specific protocols that are manufacturer dependent.

With the advent of the network, and the possibility of resource sharing between a variety of computers connected to the network, the problem changes completely, for the following reasons :

- Packet switched technology used in the new network requires new interfaces (e.g. X25) not supported yet by different manufacturers.
- Heterogeneous systems need to communicate through the available networks but their teleprocessing systems already implemented are not suitable for that purpose.
- Networks of different types (Datagram, Virtual Circuit) may need to interwork.

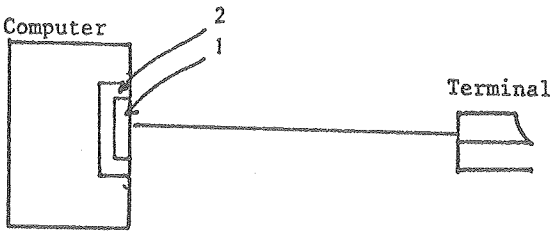
One general approach to define a method for interworking in one network, and between networks, taking into account the three factors mentioned above, is to adapt the different systems components to a standard Model which is defined in terms of Layers and Protocols ruling the communications between entities of the same layer.

When a system (network or computer) wishes to connect to another system, both have to build the adaptation necessary to comply with the Model. Two kinds of adaptation will then be possible :

- The first consists of an internal adaptation and is worked out when the system (or all systems of a Network) is ready to modify its components (e.g. access method) and implements the standard protocols (Fig. 4a).
- The second corresponds to the case where systems are not modifiable or not ready to be modified. An external adaptation will be implemented. We will give to this adaptation the generic name of "Converter" (Fig. 4b).

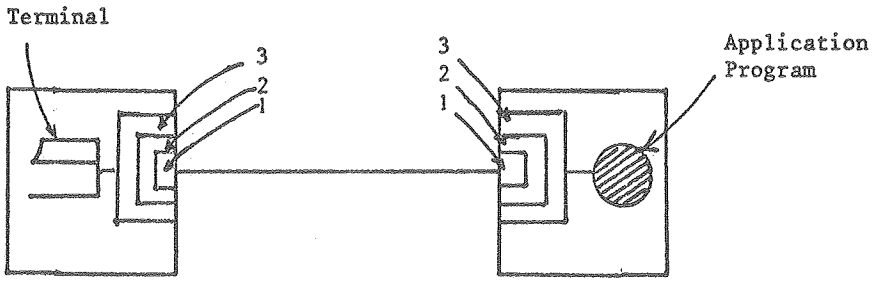
We will see later, that this "Converter" will take different forms : a Concentrator (CT) when we want to connect a dumb terminal to the network, a Gateway (GW) in the case of networks interconnection, an Interface Converter (IC) when a host is to be connected to a network.

In the following sections, we will review the design concepts and development principles of some adaptations in one network environment. Products used in CYCLADES will illustrate the different solutions adopted.



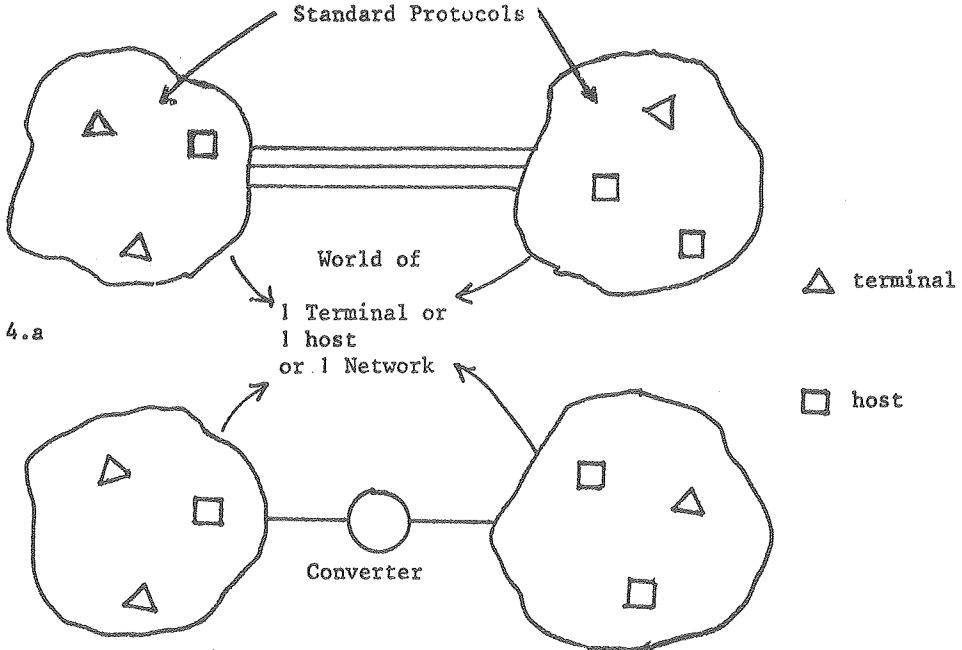
1. physical level
2. logical level

Fig. 2 - Computer-terminal interworking



1. physical level
2. line control procedure
3. user level

Fig. 3 - Interworking between 2 computers



4.a

4.b

Fig. 4 - General Interworking

IV - INTERCONNECTION IN ONE NETWORK : The CUI approach

a) - Possible adaptations

Conforming to one possible standard model that was presented in section 2, the ideal solution for interconnecting informatics systems in one network is to build Internal Adaptations which means to integrate all the network protocols inside those systems (Fig. 5). Modules to be implemented are :

- The transmission interface which depends on the type of transmission network being used (e.g. X25 if public PSN (16), Datagram for CIGALE (8)).
- The transport station which provides for end-to-end Liaisons handling.
- The virtual terminal which allows communication between terminals and applications.

Three categories of informatics systems should be considered for interworking : Host Computers, Intelligent Terminals, Simple Terminals. While the solution for the third category is inevitably based on Converters performing External Adaptation, the solution for the first two categories, can be carried out in both ways externally or internally (Fig. 6).

The choice between the two solutions is in general based on economic considerations. Whereas Intelligent Terminals manufacturers, can implement easily in PROMs the network protocols (17), the large computer systems manufacturers will wait until the network protocols reach a large consensus, since the introduction of new protocols will have an important impact on all their products.

However, the major manufacturers are preparing the integration of the transmission interface X25 (7) and the line control procedure HDLC (18), which were recently agreed by CCITT and ISO. Some other work has been undertaken for the implementation of high level protocols such as Transport Station (13) and Virtual Protocol (14) in the IRIS 80 by CII (the French Computer Manufacturer*) and in the IBM 360/67 by a research team of IMAG at Grenoble (France).(19) and (20)give all details about the specification of those implementation.

In the present paper we will concentrate on external adaptations for simple terminals and hosts. The special case of microprogrammable terminals which can include in their internal logic the network protocols, is covered in another paper (17).

b) - The customized User Interface (CUI)

If we examine the functions that must be performed by the "Converters" which connect the users (terminals and host computers) to the network, we can identify three common functions : the Transmission Interface, the Transport and the Virtual Terminal. Only the "peripheral" function which interfaces the Host and the Terminal is respectively Host and Terminal dependent (Fig. 7). All "converters" constituting

---

* Now linked with HONEYWELL-BULL.

this kind of ring will be built around Processors chosen according to some parameters such as the number of users to connect, the traffic, the type of applications...

These processors will include on the one hand the Network Protocols and on the other hand either the Terminal Emulation function if they interface Hosts only, or the Terminal Handling function if they interface terminals only, or both functions if they are shared by hosts and terminals. These specific functions : Terminal Emulation and Terminal Handling will be described in details when we analyse the different products developed in CYCLADES for that purpose. The basic principles of the Terminal Emulation is to create an adaptation to the protocols of the host computer at all levels concerned with the communication : the physical level, the line control procedure, and the terminal-to-application dialogue level. Thus, this adaptation will have to behave like a real terminal or terminal controller that the host computer already supports. For this reason, we designate the peripheral function of the converter for host connection, the Terminal Emulation function.

On the other hand, the peripheral function for connecting a terminal will perform different tasks concerning the physical attachment of terminal, the commands introduced by the operator, and some other facilities for end-user dialogue (ref. § 5.b). We call this ring of processors the Customized User Interface (CUI). A similar notion has been introduced in another paper (21).

The main advantages offered by this approach are :

1°) - The unifying of all CUI Converters around the transmission networks which simplify the maintenance and further development.

2°) - The network is permanently ready to connect to any type of terminal or host. The only piece of software to be developed is within peripheral function.

3°) - Since the Converters perform dedicated tasks, their design over a multi-microprocessors system working in parallel will greatly improve their performance. In the following sections, this approach will be illustrated by the different interconnection products developed within CYCLADES.

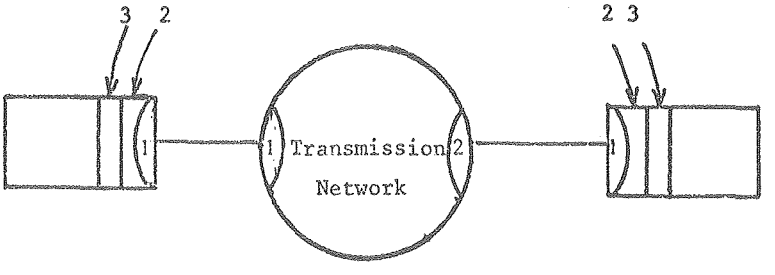


Fig. 5 - Basic scheme

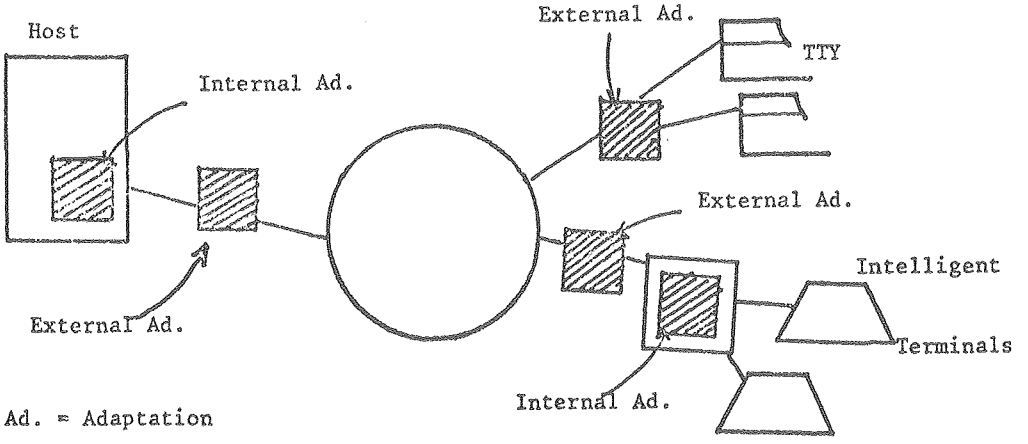
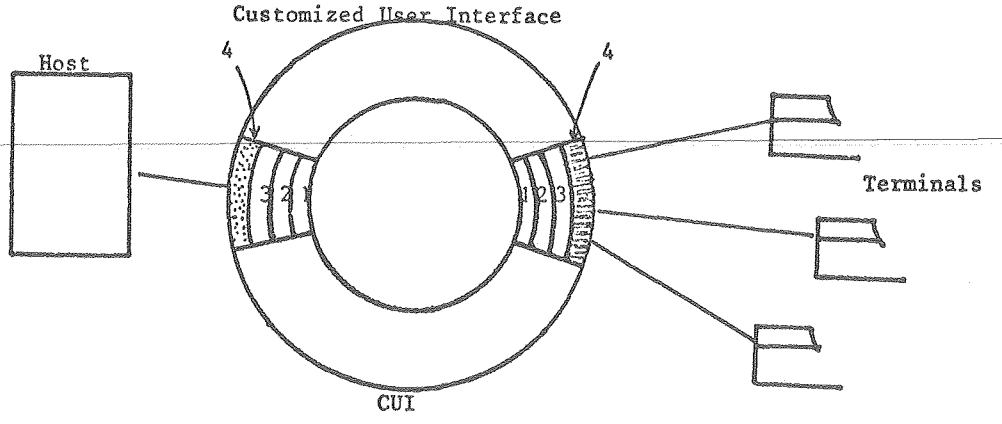


Fig. 6 - Variety of Adaptations



1. Transmission Interface
2. Transport
3. Virtual Terminal
4. Peripheral function(s)

Fig. 7 - Customized User Interface approach

V - THE FIRST APPROACH FOR BUILDING THE CUI CONVERTERS IN CYCLADES

a) - The hardware

The CUI converters in CYCLADES were built around the MITRA 15 minicomputers (manufactured by CII in France), which are identical to the nodes of CIGALE. This minicomputer has a microprogrammed CPU, 800 ns

cycle time, 16 bit word length and 32 levels of interrupt. It includes also 32 asynchronous line-adapters, and 14 synchronous line-adapters. Terminals may be connected through leased or dedicated lines, and via the Public switched telephone network. Hosts may be connected via leased lines. Different speeds can be used : 110, 150, 300, 600, 1200 b. , for asynchronous ports, and up to 48 Kbits/sec. for synchronous ports. The main memory can be expanded to 64 Koctets by adding cards of 4 Koctets.

b) - The software

The principal decision for the design was to build the system around the basic software of the switching node of CIGALE. The following reasons lay behind this choice :

- CIGALE Node contains all the system mechanisms for cooperating processes which are needed in the CUI converters.
- The CUI converter must perform the function of switching packets between External and Internal Transport Stations. This function is the main task performed by the node.
- Internal Transport Stations representing the services offered by every node in CIGALE : ECHO, Time, Debug, (22) are very useful in the CUI Converter.
- Lastly, a compatible software in the CUI will permit the Network Control Center (NCC) to supervise both the nodes and all the CUI converters (Teleloading, teledump, measurements ...). The architecture of a "Combined Converter" is shown in Fig. 8. It consists of :
  . a supervisor part which performs, the queueing management and the process synchronization.
  . immediate tasks for handling the lines. The line control procedure is the TMM-UC (23). These tasks are attached to interrupt levels that are triggered by external events (line adapter, clock, ...).
  . permanent tasks are attached to interrupt levels triggered by software. These tasks are :
    . The Packet Switch
    . The Transport Station (TS)
    . The Letter Switch performing the switching of messages between terminals and Transport Station. Terminals attached to one Converter can exchange messages (Lettergram) locally without passing through the network.
    . The connection control for "liaisons" supervision.
    . The Terminal Adaptations perform the mapping between real terminals and Virtual Terminal.
    . Terminal Handling applies only to terminal connection. Great care has been given to this function because of its important impact on the end-users of the network.

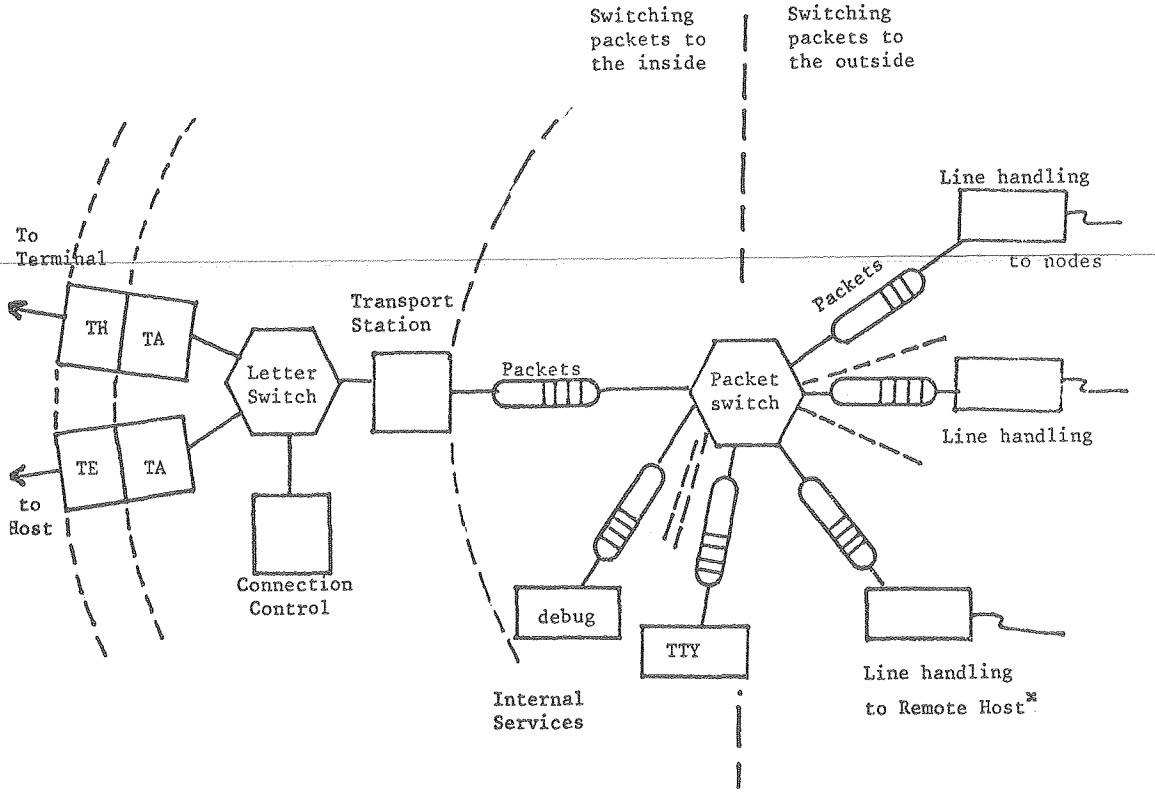
The main facilities offered by the CYCLADES Concentrator (the Terminal Converter) are :

- Symbolic addressing,
- Type-ahead,
- Editing facilities : cancel character, cancel line,
- Stop output

- Escape and Return mechanism : to disconnect temporarily the "liaison",
- Terminal tailoring for Virtual Terminal Adaptation,
- Interrupts.

Many other facilities are also offered. All details can be found in (24, 25) :
- Terminal Emulation applies only to host connection.

In CYCLADES a variety of cases have been covered in the Converter, called also "Interface Converter" (IC). In the following we describe very briefly the different connections that were carried out, giving emphasis to the problems encountered in each case.



TH = Terminal Handling
TE = Terminal Emulation
TA = Terminal Adaptation

Fig. 8 - Converter Software Organization

* This host integrates the network protocols.

c) - The Terminal Emulation in the Interface Converter (IC)

In this type of connection, the IC emulates the behaviour of a terminal, or cluster of terminals, attached to the host or to its front-end according to a specific line configuration. Existing teleprocessing systems can handle in particular, two categories of line configuration :

1°) - Multidrop line to which terminals are connected. These terminals are continuously "Polled" by the central system, and can be simple (e.g. TTY) or more sophisticated.

2°) - Point-to-point line to which terminals of different types can be attached.

c.1) - Polled Terminal Emulation

Both cases were covered in CYCLADES. The Terminal type emulated was teleprinter oriented. When undertaking this type of emulation, the IC must perform the following tasks :

1°) - handle the line control procedure supported by the Host.

2°) - assign "polled address" to transport station "Ports" and "Liaisons".

3°) - map the messages exchanged over the 2 step-link (Virtual Liaison and Real Liaison), in converting the Transport Commands : Liaison Initiation (LI-INIT), Liaison-Termination (LI-TERM), Send and Receive Letter and Telegram, into commands of the polling procedure. For example, a select command received from the host is converted to Send-Letter, to the distant Transport Station. A Letter received from the distant Transport Station will be the answer to the next polling sequence received from the host. If no letter is received, a "negative" answer to the Polling is sent back to the host. No polling sequence crosses the network.

4°) - map between the contents of real text-messages and commands, into Virtual Terminal Text-items and Control-items.

Two host systems were connected in CYCLADES with the multidrop line configuration : the first is the IBM 360/65 of the European Space Agency located at Frascati (Italy). The line control procedure used is called RTC and is defined in (26). The second is the Unidata 7730 of the "Institut Textile de France" located in Boulogne (France) which uses the MSV1 (27) line control procedure of Siemens.

The major problems that were encountered concern : 1° the Transport mapping and 2° the Virtual Terminal mapping.

- The first category of problem results from the fact that the services offered by the transport protocol on a liaison cannot match completely with the services offered on the physical link by the line control procedure. For example, the RTC procedure lacks error and flow control which is performed by the transport stations. Thus messages can be lost between the IC and the host, or they can be ignored by the IC when they are received from the host if the remote transport station asks for the stopping of transmission on the liaison. This shows the unreliability of the external adaptation solution when emulating an inefficient line control procedure.

In the case of MSV1, the matching between transport station and the procedure logic is total. Error control is done in a step-by-step manner. Flow control is performed without anticipation. (Fig.9) shows a typical diagram of commands and messages exchange between terminal and host accross the CUI.

- The second category of problem concerning the virtual terminal mapping can be related to the "Dialogue mode compatibility" and the "Functional command adaptability". Since the virtual terminal is set to "alternate mode" which is synchronized by the YOUR-TURN mechanism, the method of mapping this mode is based on the "end-of-message" (EOM) indication sent by the application. EOM is mapped into YOUR-TURN indication in the VT message header. If EOM is not explicit in the host messages, the only way to avoid the conflict of accessing the terminal "presentation unit" between the operator and the host, is to set the "Free Running mode" in the VTP for this type of host. The mapping of the real commands will not lead to any problem when the Virtual Terminal model has the most current functional characteristics of the class of terminal covered by the VTP which is being implemented.

c.2) - Point-to-point terminal emulation

This method can apply to the connection of any host system which can handle start-stop terminals attached via point-to-point lines operating in character mode (n asynchronous host ports are linked to the IC). The value of n will depend on the number of simultaneous liaisons that the host wishes to distribute between the network end-users (Fig. 10).

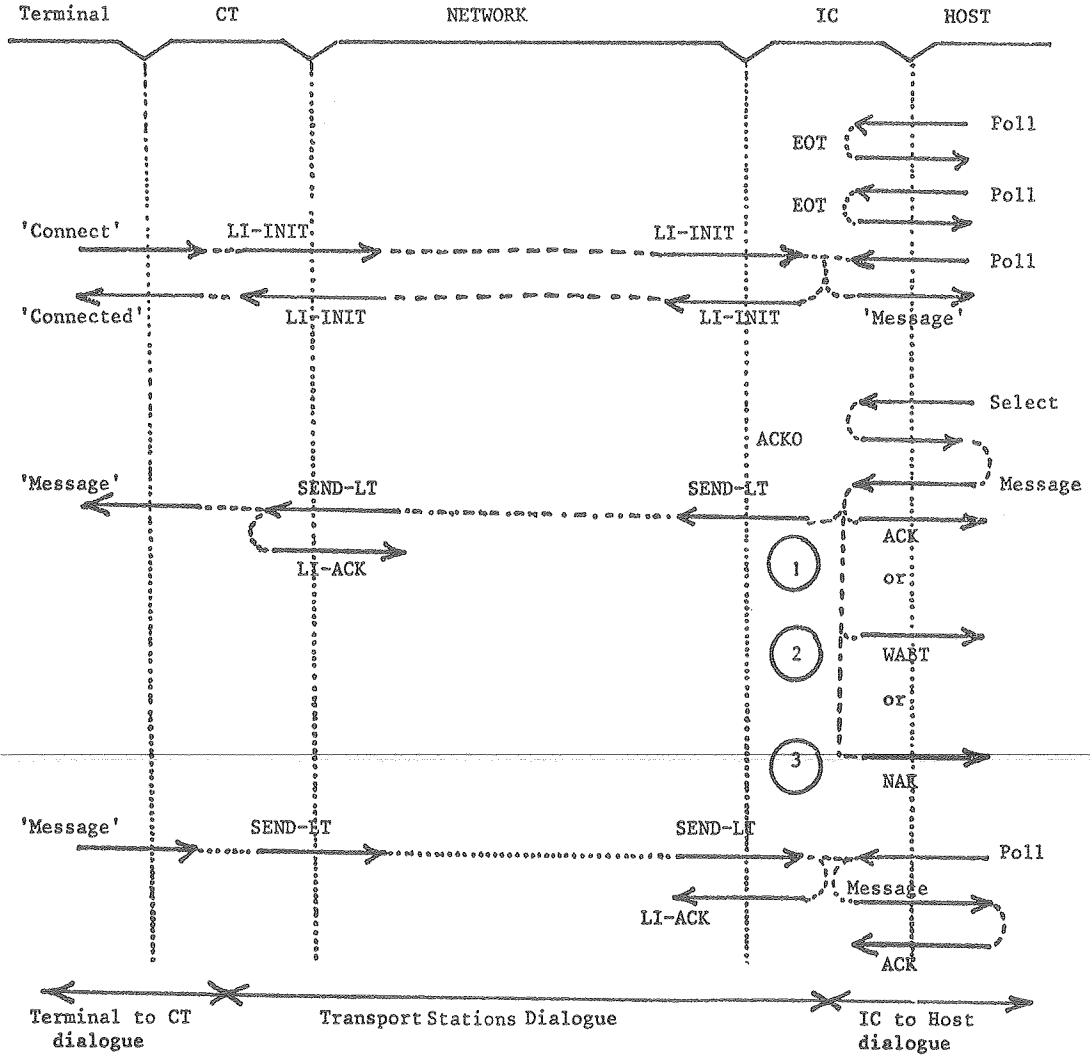
The same problem mentioned in the above sub-section, will be encountered in this type of emulation. The same solutions adopted for the Virtual Terminal mapping problem can apply here :

- if an end-of-message is indicated explicitly by the host-application, the VT will operate in the Alternate mode, and the End-of-message indication will be translated into a YOUR-TURN indication in the VT message header. When this is not the case, the VT will be set to a "Free-Running" operating mode.
- definition of a general Virtual Terminal that covers the most current characteristics of terminals supported by the host system.

With regard to the Transport, the problems are more complicated since the asynchronous transmission procedure lacks any error control and recovery. The quality of the transport service deteriorates when the physical link is juxtaposed with the Liaison. This can be made even worse when a long distance separates the host from the IC which is in most cases true because the IC is generally placed in a network operation center (with nodes and concentrator).

In order to improve the quality of transmission over the link between host and IC, one can bring the IC interface, which is the network interface, nearer to the host : by putting an Intelligent Processor at the host site, and linking this processor to the IC via a dedicated high speed line operating synchronously according to a Line Control procedure that offers Error Control with recovery. This was achieved

in CYCLADES by designing a microprogrammed system, called MUMI.



1. Queue not full, credit available
2. Queue full, or no credit
3. Terminal has disconnected

Fig. 9 - Examples of dialogue in multidrop Terminal Emulation

d) - The MUMI interface

The MUMI is linked to the IC via high speed line. The exchange of blocks is done according to the TMM-UC (23) synchronous procedure which offers error and flow control. It is also linked to the host ports locally according to standard modem interface (RS 232 C or V 24) - (see Fig. 11). The MUMI performs statistical multiplexing over the asynchronous host channels. The available information transmitted by one

channel will be sent as an item in the block of information exchanged over the synchronous link. The multiplexing scheme is block oriented and is shown in Fig. 12. The channel address is indicated in the header. When no information is available on one channel, the item is not present in the block. At the other end of the synchronous link, the multiplexing is performed in software by the IC, which assigns a "Port" and a "Liaison" to each active channel.

The scanning of channels is done continuously by the MUMI. The decision when to package the available information in one item is based on two criteria : an inter-character timer and a full buffer. The values of the timer and the buffer size are chosen according to the speed of the asynchronous ports (for 300^{b} channel typical values are 100ms for the timer, and 30 characters for the buffer). The MUMI was built around the motorola 6800, with 2 koctets of RAM and 4 koctets of PROM. The present version supports three asynchronous channels. All details about software and hardware of the MUMI can be found in (28).

In addition to the advantage of having a more reliable transmission between the host and the network, the MUMI approach permits an economy in the line cost. Furthermore the MUMI hardware costs half the price of a traditional TDM multiplexer performing the same task.

With this approach, we connected to CYCLADES hosts like PDP 10, PDP 11, and H66 located in Paris. Many other hosts will soon be connected. Each MUMI type of connection of a new host takes 2 weeks, including the Virtual Terminal adaptation study and implementation, and the testing.

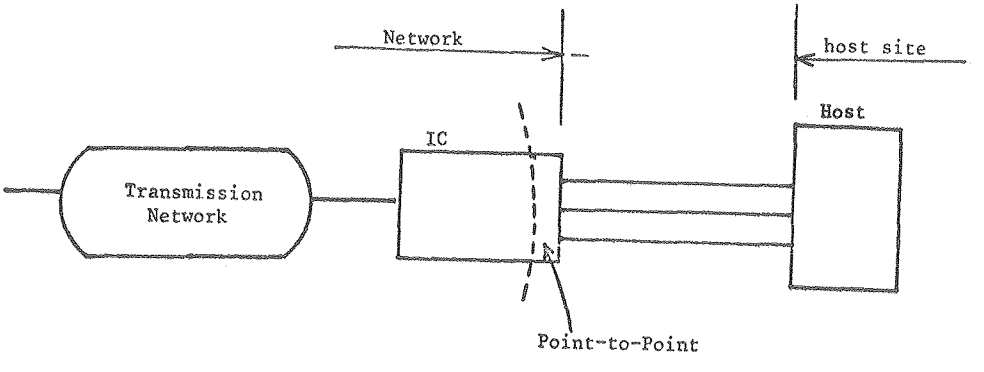


Figure 10

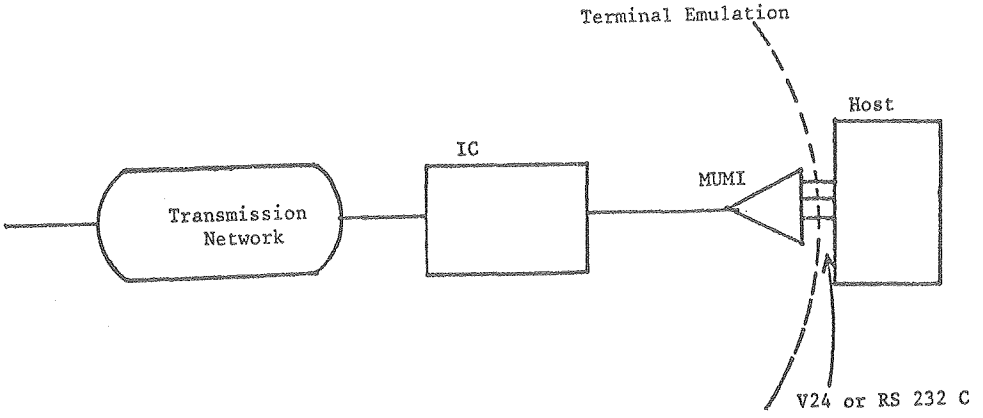


Figure 11

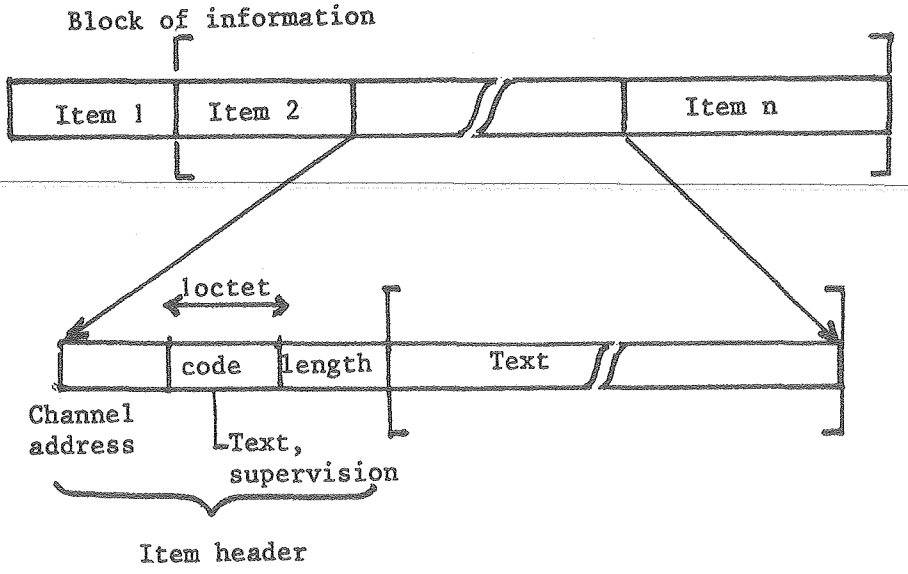


Figure 12 - Multiplexing scheme

VI - THE MULTI-MICROPROCESSOR APPROACH

When one examines the dedicated tasks executed by the CUI Converters (CT, IC) that were built over minicomputers shared by many users (terminals and hosts), and taking into account the technological advance of micro-systems, it appears evident that the solution of building all the CUI converters around microprocessors will be more economic and more flexible and reliable (Fig. 13).

Each user will have its own converter that will be tailored to his

need. For that purpose, the best design is to have a multi-microprocessor architecture that can handle the number of network entries wanted by the user. Additional boards will be included as the number of ports increases. Each host site will have its own micro-IC that is adapted to its traffic and each cluster of terminals will have its own micro-CT which is adapted to the number of terminals.

In CYCLADES, we developed a multi-microsystem to build the CUI converters.

The system, called MLV (Multiplexeur de Liaisons Virtuelles) is based on 2 Intel 8080. One is intended for Datagram interface and has 2 koctets of PROM and 1 koctets of RAM, the other offers the Transport Station (13) and the Virtual Terminal Protocol (14) and has 4 koctets of PROM (2 koctets : TS and 1,5 koctets : VTP) and 4 koctets of RAM. The number of ports supported is 4.

The operator working with the MLV can open a Liaison with another MLV and indicate the mode of operation at the terminal level : Transparent or Virtual. The transparent mode is used when connecting terminal not covered by the VTP (e.g. graphic). More details on MLV can be found in (29).

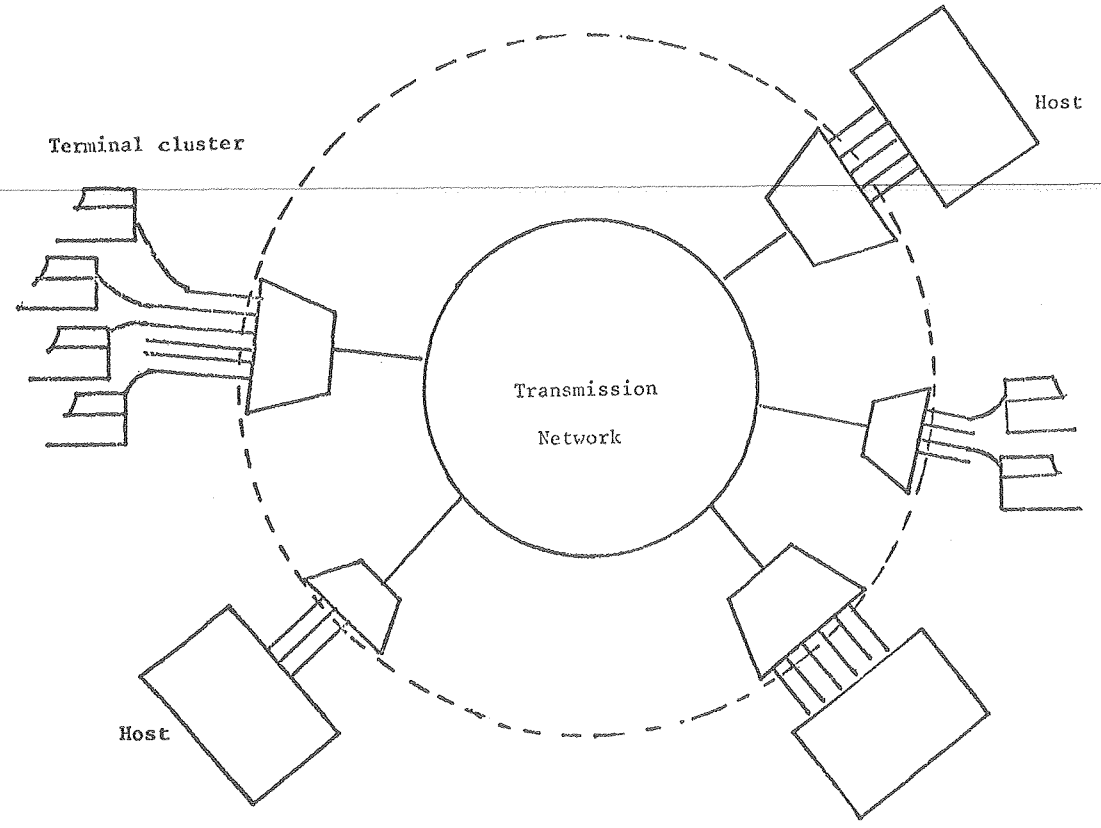


Figure 13 - The multi-microsystem CUI

VII - CONCLUSION

In the new and changing world of distributed informatics, where rapid technological advances occur regularly new transmission services are being installed in all countries. Heterogeneous systems need to communicate. Prime emphasis must therefore be given to defining a "standard model" for informatics networks. CYCLADES has been built according to a layered structure similar to the one proposed for standardization in ISO. In order to connect existing worlds (terminals, host systems, networks) to CYCLADES, a long-term solution would be the integration of the model's protocols into these worlds. Since such a long-term solution would require substantial investment, a temporary solution is to build external adaptation between the existing worlds and the model.

In this paper, we have presented an outline of the design principles of the Customized User Interface (CUI) built around CIGALE the packet switched network of CYCLADES, and incorporating such external adaptations. These principles are sufficiently general to be applied to any network with the same architecture.

REFERENCES

(1) - POUZIN L. - "The CYCLADES network. Present state and development trends" . IEEE Symp. on Computer Networks Trends and Applications. Gaithersburg (USA) - June 1975.

(2) - ROBERTS L. and WESSLER B. - "The ARPA Network".Chap. 13 of "Computer Communication Networks" by Prentice-Hall, 1973.

(3) - BARBER D.L.A. - "A European Informatics Network : Achievment and Prospects" - Proceedings of ICCC 76 - Toronto, pp. 44-50.

(4) - NAKAMURA R. et Al - "Some Design Aspects of a Public Packet Switched Network", Proceedings of the ICCC, Toronto, Aug. 76, pp. 317-322.

(5) - ISO - TC 97 - SC6 - N1369 - "Basic Architecture for End-to-End protocols' - French contribution - Proj. 17 - Jan. 77, 8 p.

(6) - AFNOR - 76/SC6 - N255 - Working document. "Datagram Service" - Nov. 77, 17 pages.

(7) - CCITT - "Recommendation X25" - Doc. APVI, n° 55, Geneva, May 76, pp. 80-144.

(8) - POUZIN L. - "CIGALE, The packet switching machine of the CYCLADES Computer network", IFIP Congress, Stockholm, (Aug. 74), 155-159.

(9) - ISO - TC97 - SC6 - N1340 - "HDLC proposed Balanced Class of Procedures" - 11 p.

(10) - CERF V. et AL. "Proposal for an International end-to-end protocol" - ACM, SIGCOMM. Comp. Comm. Review - Vol. 6, N° 1, Janv. 76, pp. 63-89.

(11) - INFOREP - Groupe Normalisation "Definition du Service de Transport et Fonctions à réaliser par une Station de Transport" - Working paper - Oct. 77 - (IRIA).

(12) - IFIP - TC6.1 - "Proposal for an Internetwork End-to-End Transport Protocol" - INWG Note N° 96 - Rev. 1, Dec. 77.

0 0 - 0 3 1 0 4 0 6 2

(13) - ZIMMERMANN H. - "The CYCLADES End-to-End Protocol" - 4th Data Comp. Symp. - Québec C. (Oct. 75), pp. 7.21-7.26.
(14) - ZIMMERMANN H. - "Proposal for a VTP" - Jan. 76 - 16 p. CYCLADES Doc. Ref. TER 533.1.
(15) - NAFFAH H. - "Protocole d'Appareil Virtuel Type Ecran" - INFOREP, doc. Ref. IFR 503.2, Nov. 76, 61 p.
(16) - PLATET F. - TRANSPAC "A public network packet data-transmission" COMNET 77 - Budapest, October 1977 - pp. 153-169.
(17) - NAFFAH H. - "TIPAC : an approach for Intelligent Terminal interconnection to packet switched networks" - CYCLADES, doc. TER 545, Oct. 77 - 14 p.
(18) - ISO - TC97 - SC6 - N1501 - "HDLC classes of procedures" - Oct. 77, (WG1 - draft) - 21 p.
(19) - GARCIA C. et al. - "Specifications de Réalisation de la Station de Transport ST2 portable" - CYCLADES doc. SCH 536.2, May 75, 84 p.
(20) - ANSART J.P. - "Système interactif dans un environnement réseau. Connexion d'une machine à mémoire virtuelle 'IBM 360/67' au Réseau CYCLADES" - Thesis, (Feb. 76), 223 p. - CYCLADES Doc. SCH 578.
(21) - BARBER D. - "The Thing-Ring Approach to Network Architecture", Sept.,77, IFIP - IC6.1 - INWG, Note n° 147 - 28 p.
(22) - GRANGE J.L. - "CIGALE implementation, tools and techniques", SHARE European Association, Dublin, (Sept. 75), 10 p.
(23) - NAFFAH N., LAROCHE J.P. - "Spécification d'Interface TMM avec le Réseau CIGALE" - CYCLADES doc. TRA 521 - Dec. 75, 7 p.
(24) - GIEN M. - "Spécifications de l'Interface Utilisateur des Terminaux Asynchrones sur le CT CYCLADES". CYCLADES Doc. TER 541.1, Fev. 77, 30 p.
(25) - WEBER S. - "Concentrateur CYCLADES, Manuel de références" - CYCLADES Doc. EXP 505, Apr. 77, 30 p.
(26) - VIVIER A. - "Description de la procédure RTC de l'ESA " - CYCLADES doc. TRA 524, Nov. 75, 8 p.
(27) - SIEMENS - "Système de transmission de données du BS 1000" - SIEMENS 4004 - Ref. Unidata - S. 5806 E/FR.
(28) - NAFFAH N. - "Utilisation des microprocesseurs pour l'interconnexion des ordinateurs dans un réseau informatique" - Convention Informatique, Sep. 1977, Fas. B, pp. 40-44.
(29) - NAFFAH N. - "Multiplexeur Microprogrammé de Liaisons Virtuelles MLV" - CYCLADES Doc. MP X-502.1 - Nov. 76, 23 p.

AN ARCHITECTURE FOR SECURE PACKET-SWITCHED NETWORKS

M. A. Padlipsky
The MITRE Corporation

Abstract

The paper offers a broad approach to making packet-switched networks secure by means of judicious application of security kernel technology. The central insight is that the expedient of having a single uncertified process for each Host homed on a given kernelized packet switch can, in conjunction with a single uncertified Network Daemon process on each switch, lead to an architecture which is at once secure and possessed of such other desirable characteristics as requiring less verification than other kernelized approaches, avoiding inordinate process overhead, preserving the integrity of networking protocols, and being readily comprehensible.

INTRODUCTION

We present here an apparently rather attractive high level architecture, which we have not seen before, for making packet-switched networks secure. Our primary intent is to submit the approach to the consideration of workers in the secure computer systems field, but we wish to avoid becoming enmeshed in the technicotheological disputes of Security. Thus, we will not align ourselves with any particular terminological or technological camp. Also, in the interests of conciseness and to avoid introducing potentially sensitive material, scholarly apparatus will be minimal.

A SIMPLE COMPROMISE PARADIGM*

Limiting the sense in which "secure" is meant to convey something along the lines of "a system in which no sensitive information is compromised" ("denial of service", e.g., being beyond our scope), we consider it axiomatic that for compromise to occur there must be implicit or explicit senders of information, implicit

*If there weren't so many mathematicians in the field, we'd call it a "model".

or explicit channels for the information to traverse, and explicit receivers of the information.

We state that senders may be implicit in order to suggest that, owing to design inadvertence presumably, information in a given system may simply be accessible in some direct or indirect fashion to unintended receivers. Similar considerations apply to channels, but here another distinction is useful as well: whether they be viewed as "covert", "storage", "modulation", and/or "confinement", channels fall into two broad categories, which we choose to term "obvious" and "devious". For example, if any process (or address space in execution) in a given system can read the text editor command's working file there exists an obvious channel - even if the system has a sophisticated file system such that a user's permanent copy of the file in question is perfectly "secure". On the other hand, if the text editor itself contains maliciously inserted code to copy the contents of the file to somebody else's file directory as well as to the invoker's we'll call this an instance of a devious channel. The point at issue here is not intended to be profound; rather, we seek to avoid the somewhat confusing terminology of the literature by substituting an admittedly imprecise, but workable, alternative.

SEVERAL SIMPLE SECURITY PARADIGMS*

To counter the threats implied by the simple compromise paradigm, the following broad courses are open - irrespective of whether single systems or networks are at issue:

Eliminate Senders

If the design of the system were such that there were no (implicit or explicit) senders of information to other than "proper" receivers, there would, tautologically, be no compromise possible. Presumably, this would entail exhaustive verification of the code. As far as we know, this is beyond the state of the art.

Eliminate Receivers

For receiving programs, similar considerations apply as to

*We can't be sure the mathematicians have gone away, even after the previous paragraph.

senders. For networks, however, literal and figurative wire-tappers are also potential receivers; we posit encryption as the means of defeating such wire tappers.

Block Channels

It being too difficult to eliminate senders and receivers until and unless exhaustive software verification techniques become available, what about blocking the channels over which they communicate? Without going too far afield into computer system security, let us simply observe that there are approaches extant for dealing with obvious channels, but devious channels are still a problem. For example, in single system software security, there is the so-called "confinement problem" [1,2]. Also, for networks "end-to-end encryption" has been advanced as a means to block sender-receiver channels; however, it cannot easily deal with the devious channels which arise from the modulation of such "clear" parameters as the length and address of transmissions [3]. (Trusted software or hardware could be interposed to garble such channels - padding lengths and generating fictitious addresses - but this not only would detract from throughput, it would also, to a purist, only add "noise" to the channels.*)

Block Receivers

The other alternatives being more or less unsatisfactory even at the cursory level of detail above, a final alternative becomes inherently attractive, if only by contrast: What about making it impossible for any receiving software to get illicitly-acquired information out of "the system" to a human confederate? In a network in particular, this alternative has intuitive appeal, for we in some sense "own" the communications subnetwork processors (CSNP's). Therefore, even though it is incontrovertible that we cannot dictate that all subscribers eliminate potential senders of information to potential receivers in the CSNP, and even though we cannot afford the performance penalties of blocking the channels,

*We acknowledge that if the network in question can afford to require the interposition of crypto gear "between" every Host and any switch, the presumed malicious switch software's spontaneous transmissions would be garbled before reaching any receiver. This is, however, expensive - and would appear to allow a sort of second-order devious channel still. In any event, we are far more comfortable in the software realm, and will concentrate on it henceforth, without prejudice to the meta-issue of whether end-to-end encryption "really works".

and, finally, even though we cannot guarantee the absence of potential receivers in our CSNP software, we can perhaps arrange matters such that the receivers are unable to pass the illicitly acquired information along, "outside the net".

TWO APPROACHES TO BLOCKING RECEIVERS

With the focus of attention on preventing not the acquisition of illicit information by CSNP software but rather on preventing its further use, we see that it is straightforward to protect the transmitted data themselves (obvious channels): The data can be protected either by (end-to-end) encryption or by some sort of "security kernel". Now, again, we run the risk of digressing into deeper waters than we want to wade in, so let us merely characterize a kernel as trusted software which furnishes the primitives necessary (and sufficient) for process isolation and address space management, and avoid taking up the cudgels for either the MITRE/ESD mathematical model, the SRI Provably Secure Operating System, ARPA's Kernelized Secure Operating System, UCLA's virtual machines, or even the University of Texas work we're not really familiar with. The crucial point of a (generic) kernel is that we can assume trustworthy process isolation (which assumes address space management - and control of Input/Output).

So let's posit the protectability of the transmitted data within the CSNP. The potential receiver software could still obtain information over devious channels, via "headers" (which might only be addresses and lengths, if end-to-end encryption is in play--ignoring the possibility of modulating the timing of transmissions because that's easily hidable from untrusted software). We have seen two broad approaches to the problem of blocking the receivers from passing this sort of information on:

Ad Hoc Methods

One approach is simply to make trusted all the CSNP code which could transmit at its own behest. Somewhat more sophisticated is the alternative which attempts to identify - and certify - only the code which can transmit "outside the net" (i.e., to subscribers) at its own behest. There are problems with any such ad hoc approach, however. In the first place, the identifying of the code to make trusted is not a rigorous proceeding; rather, it depends on the ingenuity and acuity of the designer. Also, the amount of code which needs certification (an expensive undertaking, we note) is high. Finally, although it is admittedly a subjective consideration, this approach "is mere engineering" which of course

is not intuitively satisfying to the scientists of the protection racket.

By Design

Finally, we come to the desirable approach: Consider how to design into the CSNP the blocking of receivers. There are two "obvious" sub-approaches. If each security level is handled by a separate process, or if each transmission is so handled, any good kernel can prevent such a process from writing to any inappropriate place. Unfortunately, however, for a military network, say, when both categories and compartments must be taken into account, there are a lot of security levels. There are also a lot of transmissions. And the CSNP is driven by economic considerations to be as small as possible. So there is a fear of swamping the CSNP inherent in process per level and process per transmission approaches.* Further, there is a fear that the handling of the implied multilevel data bases (e.g., for buffers for the Switch-Host Protocol, and for "routing") would place an inordinate burden upon the kernel. Dealing with these fears will be the burden of the next section, which sketches our proposed architecture.

A RELATIVELY SIMPLE, APPARENTLY SOUND ARCHITECTURE

Premise

Our premise is that there should be one process in the CSNP for each Host "homed" on the CSNP. The crucial underlying observation is that you have to trust Hosts to maintain separation of data to whatever extent they are legally permitted to. That is, ordinary Hosts - which contain untrusted operating systems - deal with one level at a time (unless under dispensation to deal with multiple compartments). They can be allowed to communicate with Hosts at their level(s) freely, but with no others (unless network security policy permits "writing up" - a complication which we ignore for present purposes, although it might alter somewhat the implementation of one particular kernel primitive). "Multilevel Secure" Hosts, on the other hand, will, when they are certified, be by definition capable of enforcing separation of their processes

*In fairness, it should be noted that we have seen no rationale other than intuition offered for this fear; let us assume, though, that we share it.

according to security levels.* They can, then, communicate anywhere within the range of security levels for which they are certified. But with a kernel imposing separation on the per-Host processes within the CSNP and allowing transmissions (indelibly associated with level codes by the kernel itself on entry to the CSNP) to be output only to like Hosts (whether locally or remotely homed) there is no risk of broad-band compromise by the per-Host processes.**

That takes care of the "too many processes" fear, we submit, for typical CSNP's home only four to eight Hosts. As to the implied multilevel data base fear, we turn to, eventually, the kernel, but first to an indispensable ancillary process.

"Network Daemon"

Shifting our perspective momentarily from network security to network protocols, we note that the CSNP must interpret, in addition to the Switch-Host Protocol we've just relegated to per-Host processes, some sort of Switch-Switch Protocol. At the least, the Switch-Switch Protocol deals with subnet flow control; it might also address "accountability" and "routing", and perhaps some other items we've overlooked (reports to a Network Control Center are a good possibility). Here, though, the key observation is that all transactions are internal to the network, and the kernel can keep them so by the simple expedient of indelibly associating a special level code with any transmissions initiated by the single process - which, for sentimental reasons, we call the Network Daemon - that interprets Switch-Switch Protocol in the broad sense for the CSNP. Thus, even if addresses and lengths of transmissions are being

*Note that this implies "proper" integration of network software into the Multilevel Secure Host, in the sense of performing the same sort of demultiplexing and isolation about to be described, in order to preclude a devious channel from a CSNP per-host process back to an inappropriate process on its "own" Host (as opposed to on a foreign Host). In particular, the process in which the Switch-Host Protocol interpreter resides must be isolated.

**We touch upon devious channels later. It should be noted, though, that Host-switch devious channels involving parameter modulation appear to be at least two orders of (decimal) magnitude down from the rates of direct Host-switch transmission, and, at least intuitively, any overlooked devious channels out of the switch ought to be two further orders of magnitude down.

clandestinely modulated to comprise a devious channel, the kernel only makes them known to the Network Daemon process, which can't use the information derived to communicate with anything other than another Network Daemon process. (Note also that lengths need not be made known at all in many circumstances, and can be judiciously lied about in all cases; further, addresses only need be of target CSNP, not Host, thus reducing the number of bits available for modulation.)

This strategy appears to eliminate all but one of the feared data bases, at the cost of a single, untrusted process.

Kernel

The other data base which is in some sense multilevel is the buffer pool for incoming transmissions (which will also be the source of outgoing data transmissions). That is, for example, a devious channel between per-Host processes (PHP's) could be created by means of filling up the pool with "garbage" by PHP A in order to communicate a bit to PHP B, if B can come along and request a buffer from the same pool A is using. The straightforward solution to this situation is to have the kernel manage the pool and only dole out buffers to PHP's according to some algorithm which places ceilings on any PHP's share of the pool. (This is, we believe, what's known as "virtualizing shared resources" in some circles.)

So, aside from performing the functions common to kernelizing, we see a few other actions the kernel for a CSNP with a PHP architecture must perform:

As just noted, it must manage the buffer pool. As previously stated, it must indelibly associate levels with transmissions, ban spontaneous transmissions by PHP's, do all I/O multiplexing / demultiplexing, and, of course, forbid output to a (physical) line of a transmission the level of which is not in a proper relationship according to the policy of the network to that of the line. (We assume the latter condition is equivalent to saying the transmission is in the access range of the recipient, but want to avoid tripping over the treacherous ground of "partial ordering".)

Two additional services seem desirable (assuming that mediation of all interprocess communication is within the spectrum of common kernel functions, and that this sort of kernel refuses to let PHP's talk to one another - but they can, and must, indirectly invoke a primitive or two of the Network Daemon): The kernel should be conversant enough with the header structure of the given network to indelibly associate the originating address with the transmission

(on initial input). This is needed to allow eventual multilevel secure Hosts to make their own discriminations about whom they're talking to for purposes of internal access control. Also, in networks where there is sub-net "accountability" (not, for example, in "pure datagram" nets) the kernel must permit the Network Daemon to convey a few bits (three seems to be enough) to a PHP to report on transmission disposition.*

"Scenario"

The foregoing is at best concise, and may well be terse. Perhaps a rough scenario of the normal routes through the subnet will help clarify the architectural sketch, even though it does not attempt to turn the sketch into a blueprint.

When a packet comes into a CSNP, it comes over either a Host line (where terminal support mini-Hosts are still, after all, Hosts) or a trunk line. It is first dealt with by - and always retained "in" - the kernel, which must perform a level check before causing either the appropriate PHP (if a Host line) or the Network Daemon (if a trunk line) to be invoked. For Host lines, the level check is to assure that the level alleged for the packet in the header is "within range" for the originator (based on a kernel-internal line table, and according to the network's policy in this area); the kernel also writes the originator's address into the header; then it invokes the appropriate PHP, which in turn interprets Switch-Host Protocol as necessary (the relevant header command field, at least, being an invocation parameter**). If necessary, the PHP requests more buffers from the kernel (so it can do Switch-Host flow control); it then (assuming a data packet) calls the kernel for purposes of causing the Network Daemon to route and send the packet. The kernel gives the Daemon the foreign CSNP address (along with a buffer registration number, probably, and a rounded up length, if necessary) and subsequently responds to the Daemon's request to send

*Purists might argue that this could constitute a devious channel. They might be right. (But it's very slow.) And we do know of one network wherein the documentation says that five completion statuses can be conveyed in only two bits. (It doesn't say how, though.)

**We have been accused by one colleague of implicitly subscribing to what we term "The Unclassified Headers Heresy". We prefer to think of our approach as following the "Need-to-Know Header Rite", according to the "Principle of Least Privilege Orthodoxy".

the packet over a given trunk or Host line (doing a level check if it's a Host line). For trunk lines, the kernel invokes the Network Daemon, which in turn routes tandem traffic or traffic for locally-homed Hosts (and must, of course, have the kernel output the packet on the appropriate line), or performs Switch-Switch Protocol. In the latter case, any responses again go out of the CSNP via the kernel, with the "network internal" level inserted by the kernel. (Note that similar considerations apply to a PHP which needs to generate output to its own Host as part of Switch-Host Protocol: here, the kernel either refuses "new" buffers addressed anywhere other than to the PHP's own Host or allows the PHP to overwrite the header, as implementation considerations dictate.)

Of course, the crafting of such functions as buffer "registration" and the demultiplexing of transmissions which come in from the net (as opposed to over a Host line) will require some care. For present purposes, however, it is not particularly interesting to worry about them, as they do not appear to be broad architectural issues. Therefore, we will turn instead to a brief discussion of the advantages and disadvantages of our proposed PHP architecture.

ADVANTAGES

We perceive the following advantages to the PHP approach:

Minimize Verification

Only the kernel need be trusted code. Although the kernel is somewhat "bigger" than a single system's need be, the additional primitives clearly do not impose so heavy a verification burden as would alternatives wherein, at the least, the Switch-Host Protocol Interpreter and various internal-to-the-net message generators must be verified.

Optimize Process Population

Although it is likely that a function-oriented process assignment scheme would lead to numerically fewer processes within the CSNP, the five to nine processes dictated by the PHP approach should not be unduly burdensome.

Maintain Protocol Integrity

In another context, the major advantage of the approach would be that by keeping protocol interpreters as unary modules it spares

protocol designers the onerous burden of attempting to find a rational basis for functional decomposition of protocols into "secure" and "security unrelated" sub-sets. Note that nothing in the approach debars Host-Host and/or process level protocols from taking whatever steps are appropriate to introducing security considerations as germane, but also that the approach makes no unsupportable assumptions about the actions - or even veracity - of protocol interpreters external to the CSNP.

Comprehensibility

Finally, we believe the architecture to be comprehensible by designers/implementers of given secure networks. Despite the fact that prior familiarity with the field was assumed by the concise presentation here, if the premises of kernel technology are valid it would seem that PHP "holds water" as a way to make networks secure - and, it should be noted, a way which is not a priori inefficient.

A corollary to the comprehensibility issue is that any devious channels which might exist should at least become more nearly visible, in like manner to the "accountability" point raised above.

DISADVANTAGES

We perceive the following potential disadvantages to the PHP approach:

"Piggybacking"

It is not clear whether the additional kernel complexity (in the area of having to "know" too much about the relevant protocols) is worthwhile in order to support the "piggybacking" of Switch-Switch Protocol control information onto data transmissions destined for an appropriate CSNP. (It is not even clear to us, however, whether it's worthwhile to go to the trouble of identifying suitable traffic on which to piggyback even in a non-secure net.) The simple solution is to ban piggybacking, which may, of course, cost some throughput.

Verification Ease

If it be the case that "trusted processes" are more amenable to automatic verification techniques than is a kernel per se, then the argument that PHP offers less to verify has correspondingly lighter impact.

CONCLUSIONS

Granting without equivocation that the foregoing reasoning is far from rigorous, it still seems to us that a packet-switching network based upon per-Host processes in the Communications Subnet Processors (with "any good kernel" - and, of course, link encryption to defeat literal wire-tappers) has sufficient conceptual appeal to be worth pursuing further.

An analogy to the proposed approach might prove of interest, and will prove to be our final observation: Picture each per-Host process and the Network Daemon in a given CSNP painted with a distinct color.* The kernel's job, then, is to allow - in cooperation with the other CSNPs' kernels - metaphorical wires of the only same color as the end points to be "strung" across the network.

It should work.

ACKNOWLEDGMENTS

The basic PHP insight arose during a conversation with Grace Nibaldi, who deserves much of the credit (but none of the blame) for such refinement of the idea as this paper might represent. Stanley Ames and Kenneth Biba also provided useful comments.

*We ourselves would be strongly disinclined to choose either red or black, however.

REFERENCES

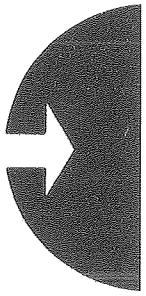
[1] Lampson, B. W., "A Note on the Confinement Problem", CACM, Vol. 16, No. 10, October 1973, pp. 613-615.

[2] Lipner , S. B., "A Comment on the Confine Problem", MTP-167, The MITRE Corporation, November 1975.

[3] Padlipsky, M. A., et al., "Limitations of End-to-End Encryption in Secure Computer Networks", USAF Electronic Systems Division Technical Report, as yet unpublished.

0 0 0 5 1 0 4 0 6 7

PERFORMANCE ISSUES IN DISTRIBUTED DATABASES



0 0 0 5 1 3 4 9 7 3

ANALYSIS OF UPDATE SYNCHRONIZATION FOR MULTIPLE COPY DATA-BASES

Erol Gelenbe
Laboratoire de Recherche
en Informatique-Université
Paris-Sud, 91405 Orsay
France

Ken Sevcik
IRIA-LABORIA
BP.105, 78150 Le Chesnay
France

Abstract

A formal model allowing a precise definition of coherence and promptness in a multiple copy information system is presented and used to analyze a class of update synchronization techniques.

I - INTRODUCTION

There are an increasing number of computer applications in which many people at different locations use a communication network to access and update a single data base. In such systems, there are several motivations for storing copies of the same information at more than one center in the network. First, responses to queries can be made more quickly if the information requested is stored at the center where the query is received. Second, the breakdown of certain centers or communication links will not cause everyone to lose access to the information if copies are stored at several centers.

If updates to a multiple copy data base originate at various centers, several major problems arise. Each center becomes aware of update requests originating at other centers only after some unpredictable delay. Thus, a person may make an update request based on information currently available at his center when actually the information has already been modified by someone at another center. If not controlled, such situations can cause the internal consistency constraints of the data base to be violated even if each update maintains consistency when applied alone. In single copy data bases, mutual exclusion can be provided by mechanisms such as locks, but in the multiple copy context, the volume and duration of the communication required to set global locks is often prohibitive An alternative is to check that all variables read by the update remain unchanged when the update is ready to write new values in the data-base. The update must be rejected if the value of some variable it read has changed. A second problem is that the order in which updates are applied affects, in general, the contents of a data base copy. Thus all centers must somehow agree both on which updates are to be rejected in order to maintain internal consistency and on the order in which the accepted updates are to be applied.

Finally, each center must decide when to apply an update request it has received to its copy of the data base. The most conservative strategy is to wait until later communication with all other centers has guaranteed that the update need not be rejected and that all previous updates have already been applied in the agreed upon order at the center. Alternatively, a center may choose to apply the update sooner on the

assumptions that it will not eventually be rejected and that other updates not yet applied at the center but earlier in the agreed upon order will neither access nor modify any of the information that it accesses or modifies. If the update is applied and this assumption is later discovered to have been false, then the data base copy must be backed up and the updates reapplied in the proper order.

A variety of techniques for controlling concurrent updates in multiple copy data bases have been proposed recently. The simplest of these rely on a "central" node [AD,B]. All update requests are passed from the center where they originate to the central node where they are either rejected or else receive a sequence number. The updates are then sent to all centers, where they are applied in the order of the sequence numbers. In a refinement of this scheme, the central node assigns batches of sequence numbers periodically to each center [GB]. In this way, updates receive their sequence number at the center where they originate, and then are sent directly to all other centers. Another possibility is to allow updates to originate from only one center at a time, with centers explicitly passing the right to admit updates from one to the next [LE].

Other update control techniques are based on a time stamp by a local clock assigned to each update when it is created. The time-stamps can be made globally unique if at most one update can be created in any tick of the clock and if the center identification number is appended to the low order end of the time stamp. Such globally unique time stamps cen be used directly to define the order of updates [JT], or they can play a supporting role in a more complex scheme. Thomas has proposed an update control method based on "majority concensus" [Th]. Each update request is passed from center to center, and each center may "reject" or "approve" the update based on what decisions it has made on updates seen earlier. An update is accepted when it has been approved by a majority of the centers. Accepted updates are applied in the order of their time stamps. This technique is robust in the presence of failures of centers and communication links. As long as more than a half of the centers are active and able to communicate, updates from these centers can be applied to the data base.

Another approach involving time stamps has been developed for use in the SDD-1 distributed data base system [BSRG]. Updates are classified according to the data-base variables they access and modify. A particularis station assigned the responsability of sequencing a given class. Potential interference between updates of different classes is resolved by a "conflict graph". Pre-analysis of the conflict graph reveals the (hopefully few) situations in which careful coordination of updates of different classes is necessary.

The primary responsibility of an update control scheme is to guarantee that all centers reject the same requests, and apply the accepted updates in the same order (at least as for as can be detected from the data base state). There are, however, several secondary goals that should be considered in selecting an update control technique for a specific data base environment. These include (i) reducing delays in applying updates, (ii) avoiding favoring updates from some centers, (iii) giving similar responses to identical queries submitted simultaneously at diffe-

rent nodes, (iv) minimizing the need to undo updates, and (v) insuring that the order in which updates are applied conforms with their actual order of submission. Because these goals are not all mutually compatible , the selection of an update control policy requires that trade-offs be made.

Our goal in this paper is to identify and illustrate the use of some quantitative performance measures for studying the trade-offs among the goals indicated in the previous paragraph. We introduce a general metric for measuring the difference between two copies of the data-base, then we point out instances of the metric that are useful in measuring the extent to which particular update control techniques satisfy the various goals.

In section 2, we define a formal model for studing the control of concurrent updates to a multiple copy base. Section 3 presents the analysis of an example of such a system (a simple inventory control system). Section 4 examines two parameterized families of update control policies. In section 5, we discuss the possibility of extending our approach to more general models. Finally, in section 6, we survey some additional uses of the performance measures we have identified.

2. THE FORMAL MODEL

Consider a distributed data base system composed of N centers. Each center maintains a copy of a set of data and replies to queries which it receives locally concerning the information contained in this data. Each center must also modify the data as a function of updates it receives. These updates may originate locally, or can be sent to a center by some other center. A transmission delay T_{ij} is incurred by an update as it travels from center i to center j.

The copy of the set of data at center i is completely characterized by its state $V_i(t)$, $t \geq o$. This state is modified by the introduction of updates via the update function f which is common to all of the centers and which is time-invariant.

An update is defined as a vector (j,k,τ,x) where j is the number of the center at which it originates, k indicates that is the k-th update originating there, τ is the instant at which center j recognizes the arrival of the update (but does not necessarily introduce it into its copy of the data), and x is its information content. Part of the information content is the specification of three sets of data base variables : the readset includes variables read by the update at its center of origination ; the reference set includes variables read at each center as the update is applied there ; the write set includes all variables assigned new values by the update. In order to preserve internal consistency, an update must be rejected if, between the time of its origination and the time it is applied at accenter, any of its readset variables are changed by another update.

An update control technique consists of two policies. The ordering policy specifies the order in which the updates are to be applied (in effect) at each center. The release policy specifies when an update received at a center should be applied to the local data base copy.

The time τ is the time-stamp which center j assigns to the update (j,k,τ,x). At that instant, center j also initiates its transmission to all the other centers. Notice that each update originates at exactly one center ; k is the sequence number assigned to it there. Note that, independent of the granularity of the clocks, time stamps can be made globally unique (and thus fully ordered) by using j and k.

The update function operates on the state of a center as follows. Suppose center i receives an update (j,k,τ,x) at some time $\tau+T_{ji}$. This update will wait in queue at center i for some duration ω (determined by the update control technique) before being processed. At time $\tau+T_{ji}+\omega$ its processing will begin and will end at $\tau+T_{ji}+\omega+S_i$. S_i is the processing time (or service time) of the update at center i. At the end of the processing of the update, the state of the center is modified ; that is

$$V_i(t^+) = f(V_i(t),x), \quad t = \tau + T_{ji} + \omega + S_i$$

We assume that only one update can be applied at a time at a given center ; this implies that at center i in the interval

$$(\tau + T_{ji} + \omega, \tau + T_{ji} + \omega + S_i]$$

only the update (j,k,τ,x) will be processed. This excludes the possibility of having multiple state transitions at the same time.

Initially, at t=0, the state of all of the centers is the same :

$$V_i(0) = e_o, \quad \text{for all } 1 \le i \le N.$$

Thus the system is started with identical copies of the data at each center.

Let Σ be the state-space of a copy of the data, and Γ the space of all updates. Thus

$$f : \Sigma X \Gamma \to \Sigma \quad .$$

A record of what updates have been applied to the data copy is retained as part of Σ.

In general, for $e \in \Gamma, x, y \in \Gamma$

$$f(f(e,x)y) \neq f(f(e,y)x) \quad \text{for } x \neq y.$$

Thus the order in which updates are performed will be of importance.

We define a reference copy V(t) as follows. V(t) represents an ideal system in which all updates are made at the instant of their time-stamp. Thus

$$V(0) = e_o$$

and for each $t \ge o$,

$$V(t) = f(f(\ldots f(e_o,x_1)x_2) \ \ldots \ x_{n-1})x_n)$$

where (i_1,k_1,t_1,x_1), $(i_2,k_2,t_2,x_2),\ldots,(i_n,k_n,t_n,x_n)$ are the updates which have arrived to the "real" (N copy) system prior to t, and

$$o \leq t_1 < t_2 < \ldots < t_n \leq t.$$

In order to measure the information gap existing between different copies of the data, or states of the system, we will introduce a metric or norm d(e,e'). This metric will allow us to formulate some concepts related to the physically intuitive notion of coherence of the distribution information system and which will be our main criteria for evaluating the "quality" of the system.

For exemples d(e,e') may be the number of records of different types in which e and e' differ, or the difference in the number of updates included in e and e'. More complex definitions of this metric which include the specific information content of the states may also be considered.

We shall introduce two performance criteria for the distributed information system. The promptness of a center i is defined as its distance from the reference copy. Let $\pi_i(t)$ denote the promptness of center i at time t ; then

$$\pi_i(t) = d(V_i(t),\ V(t)).$$

The promptness of the system at time t, $\pi(t)$, is the average promptness of the centers :

$$\pi(t) = \frac{1}{N} \sum_{i=1}^{N} \pi_i(t).$$

Thus $\pi_i(t)$ tells us how up-to-date center i is at a given instant of time. We shall often deal with the steady-state dustribution of promptness defined as

$$P[\pi_i \leq X] = \lim_{t\to\infty} P[\pi_i(t) \leq X]$$

or with its average (or expected) value $E\pi_i$.

Another measure of interest for which we shall give a precise definition is the coherence at time t. This quantity will provide a measure of the "dispersion" of the information states at each center. Thus a highly coherent system is one on which all centers have exactly the same information stored in the copies of the data. Notice that the system may be coherent, but not up-to-date (i.e. not prompt).

We define the coherence of two copies i and j at time t :

$$C_{ij}(t) = d(V_i(t),V_j(t)).$$

The coherence of the system is

$$C(t) = \frac{1}{N^2} \sum_{i,j} d(V_i(t),V_j(t)).$$

The stationary distribution

$$P[C_{ij} \le X] = \lim_{t\to\infty} P[C_{ij}(t) \le X]$$

$$P[C \le X] = \lim_{t\to\infty} P[C(t) \le X]$$

and the expected values in stationary state EC_{ij}, EC will be of interest. The system is mutually coherent at time t if $C(t) = 0$; copies i and j are mutually coherent at time t if $C_{ij}(t) = 0$.

We will say that the system is convergent if stopping all updates after time t guarantees that

$$\lim_{X\to\infty} C(t+X) = 0 \qquad \text{(with probability one).}$$

(i.e. all copies eventually become identical if all updates are stopped).

A system checkpoint is a time interval $[t,t+\Delta]$ during which the system is mutually coherent. We shall say that the system is stable if there exists an infinite sequence of non-overlapping checkpoint intervals

$$[t_1,t_1+\Delta_1],[t_2,t_2+\Delta_2], \ldots,[t_n,t_n+\Delta_n],\ldots$$

such that the expected length of each intercheckpoint interval is finite :

$$E[t_n-(t_{n-1}+\Delta_{n-1})] < \infty \quad \text{for all } n \ge 1.$$

The model we have presented in this section is quite general. For our purposes in the next two sections, we specialize the model as follows :

(i) Each center keeps a copy of the whole data base.

(ii) The clocks at each center are synchronized and are of sufficiently high resolution that two updates cannot originate at the same center with the same time stamp.

(iii) An update is sent from the center of origination by an individual message to each other center.

(iv) Every message sent is eventually received (although the transit delay may include times for re-sending lost messages).

(v) The ordering policy is to apply updates in the order of their time stamps (with ties broken by center number).

(vi) The release policy is either "release at arrival", or "release at τ+R" where R is a fixed delay, or "release only when it is guaranteed that all earlier updates have already been received and released".

0 0 0 0 5 1 0 4 0 7 3

Because some of these assumptions are sometimes unrealistic (e.g. (ii)) or restrictive (e.g. (i), (iv), (v), (vi) we will discuss how they can be relaxed in section 5.

In the remainder of this section we relate the formal model to our intuitive notion of how updates are achieved in a multiple copy data base. Updates are assumed to take the following form. The values of variables in the readset are acquired through data base queries and some computation is done to determine the desired update. Next, the update is requested and sent to each other center. At each center it is eventually released. Its application involves the current values of variables in the reference set, doing a limited amount of computation, and writing new values for variables in the write set. The amount of computation done involving the reference variables is limited both because extensive computation should not be replicated at each center and because each center must guarantee mutual exclusion in applying the updates.

Several types of updates have special properties. If all updates in a system have null readsets and null reference sets (i.e. pure assignments), then it is never necessary to reject updates or to undo updates. The value of a variable at a center is simply that value assigned by the update with the latest time stamp among those received. If all updates have null readsets and only increment or decrement variables in the reference set (as in an inventory system), then convergence of the system is guaranteed even if the updates are applied in different orders at different centers. Again, updates never need be rejected or undone. Some updates may require extensive computation involving the readset variables either because the computation is complex or because the readset is very large. Without some additional mechanism, such updates would be likely to encounter perpetual rejection due to the high probability that, during its computation, some other update would change one of its readset variables. One way to allow such updates to proceed would be to provide global locks in addition to another update control technique. Only updates risking perpetual rejection would incur the expense of setting, and releasing the global locks.

If the release policy chosen requires that updates be applied only when it is known that all earlier updates have between received and applied, then it is desirable that centers inform one another not only of updates that originate at the center, but alsoof time intervals in which no updates originate. This can be done by having each center periodically send a special message to each other center. The special message would consist of the identity of the center of origination, a time stamp, and the time stamp of the most recent update to have originated at the center. Another center, receiving such a message, could then determine from the sequence numbers of updates whether or not it has received all updates origination, at the other node with time stamps earlier than the one of the special message.

Updates must be rejected if, during their computation, any of their readset variables are changed by another update. The mechanism for detecting such a change is not simple. In principle , it requires that for each data base variable, not only its current value but also the time stamp of the update that gave it that value be retained. In practice, it is

necessary in most environments to group variables and let a single time stamp indicate the most recent change to any variable in the group. This increases the frequency with which updates must ne rejected, but it avoids having to retain a time stamp for each variable.

In general, the ordering policy will not necessarily preserve the actual order in which the update requests are made. Clearly, in many applications, the order of application of two updates originating at different nodes at nearly the same time is of little consequence. In fact, the problem of synchronizing clocks at the nodes makes it impossible to determine the exact order in which requests are made [C,Lam]. Nonetheless, there are potential applications for distributed data-bases in which the order of applying the updates may be of great significance (e.g. inventory control systems, funds transfer systems, and stock exchange systems). In these cases, it is highly desirable to minimize the extent to which the order of updates determined by the ordering policy differs from the true order in which the requests were made. For similar reasons, the frequency of having to reject update requests should also be minimized in such systems.

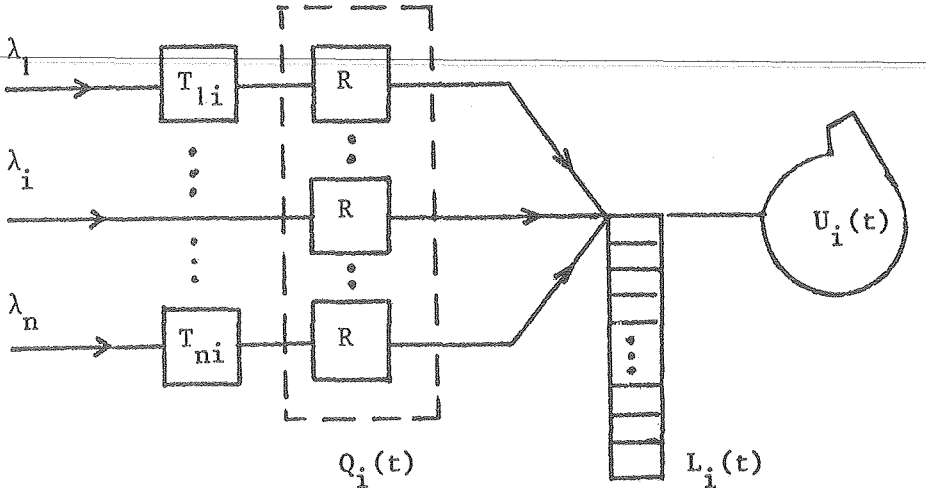


Figure 1. Center i of the distributed inventory control system

3. COHERENCE AND PROMPTNESS IN A DISTRIBUTED INVENTORY CONTROL SYSTEM

One of the typical applications of distributed information processing systems is in the area of inventory control. Several examples come to mind : distributed banking applications, reservation systems, inventory control systems in geographically distributed industrial plants, and so on. That is why we shall examine in this section an idealized instance of such a system. This example will serve to illustrate the concepts developped in section 2.

Again we have n locations. At each location i an image of the level of the inventory $L_i(t)$ at time t (i.e. the number of items in stock) is maintained. Let us stress that $L_i(t)$ refers to the same inventory of the same item. We exclude multiple item inventories (the case where $L_i(t)$ is a vector) in order to preserve the simplicity and clarity of the discussion.

Each center also keeps the variables $U_i(t)$ and $Q_i(t)$:

- $U_i(.)$ is the audit trail or time-ordered set of all updates which have been used to modify the inventory $L_i(t)$ in the time interval (0,t) at center i,
- $Q_i(t)$ is the set of updates which have been received at center i but not yet used to modify the inventory level.

Updates have the form $u = (j,k,\tau,x)$ defined in section 2 ; x will take values in the set {+,-} implying that the result of an update is to increase or decrease the level of the inventory by one item. The updates arrive to center i both locally, and from any other center j after a random delay T_{ij}, and the time S_i necessary for introducing the update after it is released is zero. Thus in all respects we are dealing with a special instance of the general model described in the previous section, and the state is represented by :

$$V_i(t) = (L_i(t), Q_i(t), U_i(t), t)$$

The behaviour of the i-th center with its update control policy is represented on Figure 1. The policy we consider here consists of holding any update arriving from center j to center i at time t^* up to time t^*+R before it is released in order to modify the level L_i, where R is fixed as a function of system parameters. The role of the holding time R becomes clear when one examines Figure 1 : this delay is used to reduce the disparity between the order induced by the time-stamp and the order in which updates are introduced into the data base (representing the level of the inventory), as well as the difference between the set of updates which have been applied at each center.

Consider the following set of updates, which we call the <u>difference set</u> associated with centers i and j :

$$D_{ij}(t) = \{u : [u \in U_i(t) \textit{ and } u \notin U_j(t)] \textit{ or } [u \notin U_i(t) \textit{ and } u \in U_j(t)]\}$$

Our distance function will be the number of updates in the difference

set :

$$d(V_i(t),V_j(t)) = |D_{ij}(t)|$$

Using this definition of the distance function we shall proceed to compute the promptness and coherence measures related to the system.

We shall make the following assumptions concerning the behaviour of the system :

- updates arriving locally to center i constitute a Poisson stream of rate λ_i, $1 \le i \le n$. These n Poisson streams are mutually independent.
- the transmission delay T_{ij} from center i to center j, for any $i \neq j$, is exponentially distributed of parameter μ_{ij} :

$$P[T_{ij} < X] = 1 - \exp(-\mu_{ij} X)$$

With these assumption we can obtain the following results. Their derivation is not given here because it calls upon some simple analysis.

Concerning the promptness $\pi_i(t)$ we have :

Result 1 : *The average promptness of center i at time t is given by (for $t > R$)*

$$E\pi_i(t) = \sum_{\substack{j=1 \\ j\neq i}}^{n} \lambda_j [R + \frac{1-\exp(-\mu_{ji}(t-R))}{\mu_{ji}}] + \lambda_i R$$

and asymptotically, as $t \to \infty$, we have

$$E\pi_i = \sum_{\substack{j=1 \\ j\neq i}}^{n} \lambda_j [R+\mu_{ji}] + \lambda_i R$$

Result 2 : *For $t > R$, the average mutual coherence of centers i and j :*

$$EC_{ij}(t) = \sum_{\substack{k=1 \\ k\neq i,j}}^{n} \lambda_k \frac{\mu_{ki} + \mu_{kj}}{\mu_{ki}\mu_{kj}} (1 - \exp(-\mu_{ki}(t-R)))$$

$$\cdot (1 - \exp(-\mu_{kj}(t-R)))$$

$$+ \lambda_i \frac{1- \exp(-\mu_{ij}(t-R))}{\mu_{ij}}$$

$$+ \lambda_j \frac{1- \exp(-\mu_{ij}(t-R))}{\mu_{ij}}$$

and asymptotically, as $t \to \infty$,

$$EC_{ij} = \lambda_i/\lambda_{ij} + \lambda_j/\mu_{ji} + \sum_{\substack{k=n \\ k \neq i,j}}^{n} \lambda_k \frac{\mu_{ki} + \mu_{kj}}{\mu_{ki}\mu_{kj}}$$

Result 3 : *For any fixed t, the promptness of the system increases linearly with R, while the coherence tends to zero as R approaches t.*

This last result presents clearly the role of the fixed delay update control technique : we see that this update control technique will be particularly valuable in conjonction with another technique which makes the system stable.

Result 4 : *The system being analyzed is convergent.*

This last result is easy to establish since all updates are guaranted to arrive and because the update control method considered simply delays updates for some finite time.

4. ANALYSIS OF TWO UPDATE CONTROL TECHNIQUES

In this section, we analyze the coherence and promptness of a multiple copy data base system under two parameterized families of update control techniques. We define $d(V_i(t),V_j(t))$ to measure the difference in the sets of updates applied at centers i and j by time t. As was shown in the inventory system example, sets of updates may often be commutative. In this section, we examine coherence and promptness assuming that this is not the case.

Taking time stamp order as the ordering policy and assuming that the clocks at the nodes are sychronized, we will predict the extent to which one update is applied at some node before an earlier update is applied at another node. Let τu and τv denote the time-stamps of updates u and v, and let $E_{ij}(u,t)$ be the set of all updates applied at i or j before u for $\tau u \leq t$. We define

$$d(V_i(t), V_j(t)) = \sum_{\forall u,v \in E_{ij}(u,t)} (\tau v - \tau u)^+$$

where $(X)^+ = \begin{cases} X \text{ if } X > 0 \\ 0 \text{ if } X \leq 0 \end{cases}$

The first parameterized family of release policies we analyze is based on postponing the release of each update for a fixed time with the intention of waiting notice of any nearly simultaneous updates. Update u is released at center i only after the time on the local clock is at least $\tau u+R$. In the extreme case when R is zero, each update is applied at a center as soon as it is received. The smaller R is, the more frequently it is necessary to "undo" updates in order to assure convergence. The second family of release policies avoids undoing updates entirely by releasing an update only when it is known to be the oldest update not yet applied at the center. Each center uses update messages and any special status messages it receives to keep track of how much earlier some update, not yet received, could have originated somewhere in the network. A parameter S is the time between broascasts of status messages from each center. The smaller S is, the better the centers can keep track of one another's status.

Intuitively, among the fixed postponment release policies, we expect that as S grows, both incoherence and the frequency of having to undo updates are reduced, but promptness is also diminished. Among the full knowledge policies, as S decreases, the traffic of status messages is increased but promptness is improved substantially and incoherence is reduced somewhat.

We will derive the expected values of both promptness and coherence at an arbitrary point in time under each of the families of release policies. To do this, we implicitly consider all possible patterns of updates originating at the nodes.

0 0 : 0 5 1 9 4 0 7 5

A)- Fixed postponment analysis

We wish to determine $C^R_{ij}(t) = d^R(V_j(t),V_i(t))$, the contribution to incoherence caused by the application of updates at node j prior to the application of older updates at node i if each update is released no sooner than R time units after its time stamp. For two other nodes, a and b, consider the execution of updates originating from a at node i and originating from b at node j. For two specific updates, u from a and v from b, assume that $\tau u < \tau v$. If v is applied at j but u is not applied at i by time t, then $C^R_{ij}(t)$ is increased by $\tau v - \tau u$. Because updates originate in Poisson streams at each center, with λ_a being the rate at center a, the probability of an update originating, at a in a short interval (y,y+dy) is $\lambda_a dy$. Considering all pairs of nodes, a and b, and all pairs of time stamps such that $\tau u < \tau v \leq t-R$ yields :

$$E[C^R_{ij}] = \sum_b \sum_a [\lambda_a \lambda_b \int_{X=R}^{\infty} W_{bj}(X) \int_{Y=X}^{\infty} (Y-X)\ [1 - W_{ai}(Y)]dydx]$$

where $X = t-\tau v$, $Y = t-\tau u$ and $W_{ij}(z)$ is the probability distribution function of T_{ij}.

The difference between the copy at center i and the reference copy has a somewhat more complex form because all updates less than R units old contribute to $\pi_i(t)$. Such updates have not yet been applied at any center, yet all subsequent updates are reflected in the reference copy. Hence :

$$E\pi_i = \sum_b \sum_a [\lambda_a \lambda_b \{\int_{X=0}^{R} \int_{Y=X}^{R} (Y-x) W_{ai}(Y) dydx + \int_{X=0}^{\infty} \int_{Y=X}^{\infty} (Y-X)\ [1 - W_{ai}(Y)]dydx]$$

which, after straightforward manipulation, becomes

$$E\pi_i = \lambda \sum_a \lambda_a [\frac{R^3}{6} + \frac{1}{2} \int_{Y=R}^{\infty} Y^2\ [1 - W_{ai}(Y)]dy]$$

where $\lambda = \sum_b \lambda_b$.

B)- Full knowledge analysis

Under the full knowledge release strategies, at each center at any time, there is some time stamp value such that the set of updates applied so far includes all updates with time stamps earlier than the value, and none with time time stamps later. Thus $C_{ii}(t) = 0$ for all i, and we concentrate on determinating $C^S_{ij}(t)$ where $i \neq j$. At node i, the latest upda-

te applied by time t is the one which satisfies the following criteria :

(i) all previous updates from the same source node have arrived and been applied at i,

(ii) the next update from the same source node has not yet arrived at i,

(iii) the newest status message or update from the same source node indicating no further updates is the oldest among all the updates that satisfy (i) and (ii).

Note that (i) and (ii) determine a unique candidate among the updates originating at a given node, while (iii) selects among these candidates. When no status messages are issued (i.e. $S \to \infty$), (iii) picks the candidate update with the earliest time stamp.

We will call the time stamp of the update selected according to (i), (ii) and (iii) the "release point" of node i at time t. The significance of the release point is that it is the most recent time such that communication with other nodes has guaranteed that center i knows of all updates that originated anywhere in the network before that time. Under the full knowledge release policies, at center i at time t all updates with time stamps at or before the release point have been applied to the data base copy while no updates with time stamps after the release point have been applied.

Once again, consider the application at centers i and j of updates originating at centers a and b respectively. Let $r_i(t)$ denote the difference between t and node i's release point time t. Updates u and v with time stamps τu and τv such that $\tau u < \tau v$ contribute $\tau v - \tau u$ to $C^S_{ji}(t)$. Considering all pairs of source nodes and all pairs of time stamps yields :

$$E[C^S_{ji}] = \sum_b \sum_a [\lambda_a \lambda_b \int_{X=0}^{\infty} p^b_j(X) \int_{Y=X}^{\infty} (Y-X)\,[1 - p^a_i(Y)]\,dy\,dx]$$

where $X = t - \tau v$, $Y = t - \tau u$

and $p^a_i(Y) = Pr[r_i(\tau+t) \leq Y \mid$ update from center a with time stamp $\tau]$

The meaning of $p^a_i(Y)$ is probability that, by time $\tau + t$ node i has received the appropriate update messages or status report messages to be able to apply an update originating at node a with time stamp τ. The value of $p^a_i(Y)$ is given by :

$$p^a_i(Y) = W_{ai}(Y) \left[\prod_{k \neq i} Z_{ki}(Y)\right] \left[\prod_{k \neq i,a} Q_{ki}(Y)\right]$$

where

$$Z_{ki}(Y) = \exp\left[-\lambda_k \int_{h=o}^{\infty} [1 - W_{ki}(Y+h)]dh\right]$$

and $$Q_{ki}(Y) = 1 - \exp\left[-\lambda_{kY} - \frac{1}{S}\int_{h=o}^{Y} W_{ki}(h)dh\right]$$

$$- \int_{g=o}^{Y} \lambda_k \, e^{-\lambda k(y-g)}(1 - W_{ki}(g)) \exp\left[-\frac{1}{S}\int_{h=g}^{Y} W_{ki}(h)dh\right]dg \ .$$

Intuitively, $W_{ai}(Y)$ is the probability that the update from centera with time stamp τ has itself been received at node i by time $\tau+Y$, $Z_{ki}(Y)$ is the probability that all updates with time stamps before τ originating at center k have arrived at node i by time $\tau+Y$ and $Q_{ki}(Y)$ is the probability chat center i known by time $\tau+Y$ all updates with time stamps before $\tau+Y$ originating at node k. The derivations of the expressions for $Z_{ki}(Y)$ and $Q_{ki}(Y)$ are given in the appendix.

The distance to the reference copy is simply the special case of $C^S_{ji}(t)$ where $p^b_j(X) = 1$, to signify the instantaneous effect of updates on the reference copy :

$$E\pi_i = \sum_b \sum_a \lambda_a \lambda_b \int_{X=0}^{\infty}\int_{Y=X}^{\infty} (Y-X)\ [1 - p^a_i(Y)]dydx$$

where $p^a_i(Y)$ is defined as before.

After some manipulations,

$$E\pi_i = \lambda \sum_a \frac{\lambda_a}{2} \int_{Y=0}^{\infty} Y^2\ [1 - p^a_i(Y)]dy]$$

where $\lambda = \sum_b \lambda_b$.

The formulae for promptness and coherence derived above for the two families of release policies are complex. An intuitition for their behavior can be developed only by assuming a particular distributional form for the transit delay time distributions and evaluating the formulae in various example networks. We have done this for deterministic and exponential transit delay distributions. In most networks the transit delay distributions are "between" deterministic and exponential in that their coefficient of variation is between zero and one.

Under the fixed postponement release policies, as the parameter R increases, the coherence measure generally decreases exponentially, while the promptness measure increases in proportion to R^3 once R is larger than most transit delays. Promptness is relativ ly insensitive to R when R is less than most transit delays . Under the full information policies,

both promptness and coherence decrease very rapidly as S is reduced from approximately the mean time between updates to approximately the mean transit delay time. In the former case, the value of R is chosen to reflect the relative importance of promptness and coherence in a particular application, while in the latter case, the choice of S balances good promptness and coherence against avoiding excessive status message traffic.

In symmetric systems, where all centers have the same rate of update origination and all pairs of centers have the same transit delay distributions, promptness and coherence depend on the network parameters as follows. As the number of centers increases (while maintaining the same total rate of updates), both promptness and coherence increase approximately linearly. The increase can be attributed to the fact that a lower and lower proportion of the updates applied at a center originate there. As the mean transit delay increases, both promptness and coherence tend to increase in proportion to approximately the cube of the mean transit delay. Finally, as the rate of update origination at each node increases, both promptness and coherence increase very rapidly.

Because the variety of non-symmetric network configurations is vast, their behavior with respect to promptness and coherence cannot be characterized in general. We mention only a few specific cases. Consider first two networks that differ only in that, in one, the update origination rate is the same at each center, while in the other, all updates originate at one center. Both promptness and coherence are marginally worse (i.e. larger) in the latter case. Next, consider isolating one center more and more by increasing its mean transit delay to each of the other centers. Both coherence and promptness increase rapidly as the mean transit delays for the one center are increased. A similar effect occurs when any number of centers, as a cluster, are moved further and further from the rest of the network with respect to mean transit delay.

While individual applications and different network configurations are important, our numerical studies have indicated some general guidelines for choosing values of R or S in using the release policies analyzed earlier. In the case of the fixed postponement release policies, the best value of R will generally lie between one and four times the mean transit delay. For the full knowledge release policies, the average time between status message broadcasts should be closer to the mean transit delay than to the mean time between update originations at a center.

5. EXTENSIONS TO OTHER CASES

The analyses of sections 3 and 4 were based on the assumptions made in section 2. Here, we discuss the extent to which our analytic approach can be extended when some of these assumptions are relaxed.

A)- Non-complete data base copies

Rather than having a copy of every part of the data base reside at every center, we might require only that each part reside at any (non-empty) subset of the centers. In this case, update message traffic could be reduced if each center could determine the subset of other centers affected by a particular update. Each center would have to treat each other center individually with respect to update sequence numbering. The application of an update involves only those centers storing the variables that are changed (or variables related by consistency constraints to those changed), so analysis of update control techniques proceeds as before, except that only a subset of nodes are involved with each update.

B)- Imperfect synchronization of clocks

While careful dynamic control can assure that the clocks at the nodes will be nearly synchronized, perfect synchronization is impossible [Lam]. This fact could be included in our model by introducing the quantity $\delta_i(t)$ which is defined to be the difference between the clock at center i and t when the "true" time (according to an ideal constant rate clock) is t. When all clocks run at the same constant rate (i.e. $\delta_i(t)$ does not depend on t), our analytic approach remains applicable. When $\delta_i(t)$ depends on t, the form of the dependence would determine the approach to analyzing update control techniques.

C)- Non-broadcast protocols for update requests

In some types of networks (store and forward packet-switching, for example), the total load on the network due to updates can be reduced by passing a single update request message from center to center rather than sending a separate message to each other center. In this case, the quantitative analysis of the performance of update control techniques is complicated by the fact that the taken by a particular update request to reach various centers are not independent. A similar dependence among the transit times of messages arises if the message passing protocol of the network is "send and wait", or something similar [Gel]. It seems that the benefits of non-broascast protocols can be acquired only by sacrificing some promptness and some coherence.

D)- Other reference copy definitions

Our belief that it is good to retain the "true" order of updates as much as possible led us to define a reference copy in which updates are applied at the instant they originate. As a result, those update control techniques that are based on time stamp order tend to do best with respect to promptness. With a different definition of the reference copy, however, promptness could measure the performance of update control techniques with respect to a different idea of the "ideal" system.

E)- Significant update computation times

Most computer networks pass messages with sufficient speed that the queueing and processing times for applying updates at a center are significant. We have assumed these times to be negligible for the sake of clarity, but the same analytic approach could be used without th assumption. Some parts of the generalized analysis might best be treated using approximations.

F)- Failures of centers or network links

The fact that messages are occasionally lost and must be re-sent can be represented in the model we have analyzed simply by adjusting the transit delay distributions. Failures of centers or communication links, however, can only be reflected in a more detailed model. These problems motivate the use of a "majority concensus" control scheme [Th] in order to allow the majority of centers to proceed despite the failure of some centers or links. A model in which all messages originating at given center or traversing a given network path in a time period are lost is needed in order to quantify the advantage of using Thomas' control algorithm over another.

G)- Discussion and conclusions

A number of other useful quantitative performance measures can be derived by analyses similar to those that we have presented here. Two examples are the "rejection rate" and the "inversion rate". The rejection rate is the frequency with which requests must be rejected. It depends more on the network parameters (transit delays and update rate) than on the update control technique chosen. The inversion rate is the frequency with which the ordering policy interchanges the order of two updates with respect to the true order of origination. Note that whenever the inversion rate is non-zero, promptness with respect to the reference copy as defined in section 2 grows without bound. Hence, if the ordering policy may have a non-zero inversion rate, promptness is only meaningful with respect to a reference copy in which updates follow the same ordering policy as for the centers.

Just as promptness and coherence represent a trade-off in choosing an update control policy, other performance measures reflect a trade-off in determining how to balance the priorities between updates and queries at a center. Two such measures are "query response time" (as traditionally defined) and "precision", which quantifies the extent to which a query response incorporates all updates with time stamps earlier than that of the query. Good precision requires that good response be sacrificied in order to await the application of all earlier updates before responding to the query.

In the presence of a release policy that allows the provisional application of updates before it is known for certain whether they might have to be rejected or undone, the responses to queries may also need to be "undone". Since responses once sent to a terminal or printer can only be retracted, not entirely undone, it is desirable to accompany each query response with an indication of the likelihood that the response will have to be retracted later. This indication can take the form of the recency of communication with other centers (perhaps the time stamps of the oldest possible unapplied update and the latest applied update), or an estimate of how many updates with time stamps earlier than the query have not yet been applied to the data-base copy. If a user chooses to do an update based on information in a query response that may have to be retracted, then he should incorporate the conditions that motivate him to do the update into a conditional clause on the update. Only in this way will this update be automatically undone if the query response is later retracted in such a way that the update is no longer appropriate. The alternative is for the user to monitor retractions and explicitly remove the effect of earlier updates when necessary.

We have concentrated on the use of promptness and coherence in choosing among update control techniques by calculating equilibrium expected values. There is, however, another way of using promptness and coherence. Each center can maintain a dynamic estimate of its promptness and coherence with other centers based on those messages that it has received as well as those that it infers are currently in transit. Appending these estimates to query responses (when requested) would help the users to assess the recency and soundness of the responses they receive.

We believe that quantitative tools can be helpful in identifying and resolving various trade-offs that arise in the design of update protocols in multiple copy distributed date bases. In this paper, we have established a formal model, proposed two particular performance measures and demonstrated the kind of analytic approach that can be taken. Deeper examinations of promptness and coherence and the search for other useful performance measures remain as work for the future.

APPENDIX - Derivations of $Z_{ki}(Y)$ and $Q_{ki}(Y)$

We denote by $Z_{ki}(Y)$ the probability that all updates originating at center k with time stamps before τ are received at center i by time $\tau+Y$. Because the number of requests with time stamps before τ is unbounded, we use the following limiting argument. We consider only the updates originating at k with time stamps in the interval $(\tau-T,\tau)$, then we let $T\to\infty$. This yields :

$$Z_{ki}(Y) = \lim_{T\to\infty} \sum_{n=o} \Pr[\text{n updates from k in } (\tau-T,\tau)] \Pr[\text{all n arrive at i by } \tau+Y]$$

Because the origination process is Poisson, the probability that n occur in a time period of length T is $\exp(-\lambda_K T)(\lambda_k T)^n/n!$ and each one of these may ne considered independently as occuring of a point with uniform distribution over the interval. Thus,

$$Z_{ki}(Y) = \lim_{T\to\infty} \sum_{n=o}^{\infty} e^{-\lambda_k T} \frac{(\lambda_k T)^n}{n!} \{ \int_{n=o}^{T} W_{ki}(h+X) \frac{1}{T} dh \}^n$$

which yields

$$Z_{ki}(Y) = \exp(-\lambda_k \int_{h=0}^{\infty} (1-W_{ki}(h+X))dh)$$

The probability that no status report from k with time-stamp in the interval $(\tau,\tau+g)$ has reached center i by $\tau+Y$ is given by

$$N_{ki}(g) = \sum_{i=0}^{\infty} \frac{(g/S)^i}{i!} e^{-g/S} \{ \int_{h=0}^{g} (1-W_{ki}(Y-h))dh/g \}^i$$

$$= \exp(-\frac{1}{S}\int_{h=0}^{g} W_{ki}(Y-h)dh)$$

Then considering all possible values of g we obtain

$$Q_{ki}(Y) = 1 - \exp(-\lambda_k Y - \frac{1}{S}\int_{h=0}^{Y} W_{ki}(Y-h)dh) -$$

$$- \int_{g=0}^{Y} dg \; \lambda_k e^{-\lambda_k g} (1-W_{ki}(Y-g))\exp(-\frac{1}{S}\int_{h=0}^{g} W_{ki}(Y-h)dh)$$

REFERENCES

1. [AD] Alsberg, P.A. and J.D. Day, "A principle for resilient sharing of distributed resources. Proc. 2nd Int. Conf. on Software Eng. (Oct. 1976).
2. [BSRG] P.A. Bernstein, D.W. Shipman, J.B. Rothnie and N. Goodman, The concurrency control mechanism of SDD-1 : A system for distributed data bases, Techn. Rpt. CCA-77-09, Computer Corp. of America, (Dec. 1977).
3. [B] S.R. Bunch, Automated backup, see [8]
4. [C] E. Chang, Decentralized control in distributed systems, Thesis in progress, University of Toronto.
5. [CO] W.W. Chu and G. Ohlmacher, Avoiding deadlock in distributed data bases. ACM Natl. Conf. (Nov. 1974), 156-160.
6. [EGLT] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger, The notions of consistency and predicate locks in a data-base system. CACM 19, 11 (Nov. 1976).
7. [GLPT] J.N. Gray, R.A. Lorie, G.R. Putzolo and I.L. Traiger, Granularity of locks and degrees of consistency in a shared data base. IBM Res. Rpt. RJ 1654, (Sept. 1975), San Jose.
8. [GB] E. Grapa and G.G. Belford, Techniques for update synchronization in distributed data bases. Center for Advanced Computation, Univ. of Illinois, (1977).
9. [Gel] E. Gelenbe, J.L. Grangé, P. Mussard, Performance limits of the protocol TMM, submitted for publication.
10. [JT] P.R. Johnson and R.H. Thomas, The maintainence of duplicate data-bases. Request for comments # 677, ARPA Network Inf. Cent. Doc. # 31507, (Jan. 1975).
11. [Lam] L. Lamport, Time, clocks and the ordering of events in a distributed system, Massachusetts Comp. Assoc., Wakefield Mass., (March 1976). Report CA-7603-2911. (submitted to CACM)
12. [Le1] G. Le Lann, Distributed systems - Towards a formal approach. Proc. IFIP Congress 77, p. 155-160.
13. [S] L.C. Sekino, Multiple concurrent updates. Proc. VLDB, (1975), p. 505-507.

14. [SRL] R.E. Stearns, P.M. Lewis, and D.J. Rosenkrantz, Concurrency control for data-base systems, Proc. IEEE 17th Ann. Symp. on Foundations of Computer Science (Oct. 1976).

15. [Th] R.H. Thomas, A majority concensus approach to concurrency control for multiple copy data bases. Rpt. 3733, Bolt, Beranek and Newman, Inc. (Dec. 1977). (submitted to TODS)

16. [GM] E. Gelenbe, R. Muntz, Probabilistic models of computer systems Part. I, Acta.Informatica 7, 1 (1976).

17. [G] E. Gelenbe, On the optimum checkpoint interval, Res. Rpt. 77-4, Laboratoire de Recherche en Informatique, Univ. Paris-Sud (March 1977) to appear in JACM.

QUERY PROCESSING ON A DISTRIBUTED DATABASE

Alan R. Hevner S. Bing Yao

Purdue University
West Lafayette, IN 47907

ABSTRACT

Query processing in a distributed system consists of an arrangement of local data processing and data movements within the system. Two cost measures are used to judge the quality of a data processing and data movement strategy for a given query. The cost measures are response time and total time of query processing. A graphing method is introduced in order to visualize the costs of distributed query processing. Under various system assumptions algorithms are developed that attempt to minimize data movement costs. Examples are given in which these algorithms find movement strategies for a given query. Costs are compared.

1. INTRODUCTION

A distributed database management system offers several attractive advantages over a conventional centralized DBMS. These advantages include increased reliability, localized access to data, and the potential for upward scaling of database capacity (5). The promising implications of distributed databases have led to an upsurge in research and development in this area.

One challenging research area concerns query processing on a distributed database. Accessing data that is stored at separate computers in a distributed system differs in two important ways from accessing data from a centralized computer. The necessary movement of data over communication lines introduces substantial time delays. An advantage of a distributed system is the ability to process and move data in parallel at separate points in the network. The distributed DBMS must determine a strategy for processing queries that takes into account these facts. Rothnie and Goodman (5) have demonstrated great variations in communication costs among feasible distributed query processing strategies. The importance of using efficient processing strategies on distributed databases is thus emphasized.

Previous research in the area of distributed query processing has been light. Wong (7) has proposed an algorithm that is currently being implemented in the SDD-1 system (3). Wong's algorithm produces a strategy of alternating local processing-data movement steps that process the query. A hill-climbing technique recursively finds cost improvements in the strategy. Other distributed processing methods that have been proposed (6) extend centralized query processing

methods. As such they overlook the key role of communication costs in a network of computers.

The goal of this paper is to approach the distributed query problem by synthesis modelling[2]. In Section 2 we outline the background concepts and assumptions of a distributed query environment. We discuss what is meant by a distribution solution to the query problem. We introduce two cost objectives that are used to evaluate proposed strategies and we give a cost graph method that illustrates query processing costs. In Section 3 we detail a combining algorithm that finds an efficient processing strategy for a specific query environment. An example compares the costs of the processing strategies derived by this algorithm and other algorithms on a given query. The paper closes with our conclusions and directions for further research.

2. BACKGROUND

2.1 A Distributed Database

We consider a network of inter-connected computers. Each computer, known as a node in the network, has a processing capability and data storage capacity. Each computer can send data to other computers by means of communication links. We assume that the data transfer rate between nodes is significantly slower than the data transfer rate between each computer and its disk storage.

The communication cost between nodes A and B is assumed to be a linear function, $C(X)$, of the amount of data, X, moved. Let $C(X) = C_0 + C_1X$, where C_0 and C_1 are constants. Communication cost is independent of the sending or receiving nodes. As a linear function the following equations are true.

i) $C(X+Y) = C(X) + C(Y)$

ii) $C(XY) = C(X)\ C(Y)$

iii) if $X \geq Y$ then $C(X) \geq C(Y)$.

In light of this cost assumption, it is clear that minimizing the cost of data movement is equivalent to minimizing the amount of data moved.

This assumption not only simplifies upcoming analysis but also can be justified by recognizing that advances in communication technology are rapidly eliminating distance and even physical connection as essential cost considerations (e.g., satellite communications).

We consider a collection of data organized as a relational database. Each computer contains a copy of the distributed DBMS and a possibly redundant portion of the database. We consider the unit of allocation to be a relation. Only relations or relation partitions can

be moved between nodes. We assume that the physical allocation of relations among nodes is given. A system directory is used to give each query a non-redundant mapping of its required data. The problem of maintaining consistent copies of redundant data (1) will not be considered in this paper.

2.2 Query Analysis

In order to interact with the distributed database system, a user introduces transactions at any node in the network. The fundamental concept of data independence being assumed, the data distribution is invisible to the user who writes the transactions in a high-level relational query language. A user transaction is submitted to the system as a set of interdependent queries. These queries are ordered so that each query depends upon the results of the previous queries. We assume that query analysis done by the system further refines these queries so that each query is connected (8) and in conjunctive normal form. Methods to do this type of query pre-processing are well known (8).

Query processing on a relational database can be performed with the use of the operations RESTRICT, PROJECT, and JOIN. A query can be fully analyzed in terms of these operations and the relation domains upon which the operations are performed. For a multiple query transaction, the system would select a result node for each query so that the result would be at the most cost beneficial node for its use in subsequent queries in that transaction. The final query of the transaction places the result at the user's node.

Query analysis gathers the following information:

i) The relations required by the query and their location in the network are found.

ii) The domains of these relations are designated as joining domains, restricting domains, output domains, or any combination of the three. Joining domains appear in a two relation join clause in the query. A join between two relations at separate nodes is given the special designation of a combining arc in the system. The presence of one or more combining arcs in a query indicates that data movement is required to find the query result.

iii) The result node is given. A constraint of all feasible query processing strategies is that the result must end up at the result node.

2.3 The Distribution Solution

Distributed queries are processed by bringing data together at a single node and performing the required operations. The distributed query problem is one of transforming a distributed query into a local

query by means of a combination of local processing and data movements. Wong calls this process distribution (7). Local processing reduces the size of relations by restrictions and projections. The size reduction decreases the subsequent movement cost of the relation. Data movement brings data together to allow local processing. The final data movement of all distribution solutions must move data to the result node for final processing. We designate destination arcs that link nodes containing required relations to the result node. For a given query, data movement in the network occurs only on combining arcs and destination arcs.

The distributed database as a static collection of data can be described by the following parameters.

M - number of unique relations in database,
N - number of nodes in system.

For each relation R_i, $i = 1, \ldots, M$:

n_i - number of tuples,

a_i - number of domains,

s_i - size of relation.

For each domain d_{ij}, $j = 1, \ldots, a_i$, of relation R_i:

v_{ij} - number of possible values,

p_{ij} - selectivity,

w_{ij} - size of each data item.

We assume that all domains are independent and that domain values are evenly distributed.

Define $s_i = n_i * \sum_{j=1}^{a_i} w_{ij}$ for each R_i.

Selectivity, $0 < p_{ij} \leq 1$, represents the fraction of possible values v_{ij} that domain d_{ij} currently contains. The size reductions that dictate cost beneficial moves in a processing strategy are estimated in the following manner.

Assume a joining domain d_{ij} is moved along a combining arc to joining domain $d_{k\ell}$. The subsequent join restriction on relation R_k produces the following parameter changes.

i) $n_k = n_k * p_{ij}$,

ii) $s_k = s_k * p_{ij}$,

iii) $p_{k\ell} = p_{k\ell} * p_{ij}$

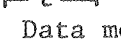
iv) $p_{k\ell'} = \min (p_{k\ell'}, n_k/v_{k\ell'})$, for all $\ell' \neq \ell$.

For the query environment that we analyze in the next section, local processing costs are assumed to be insignificant in comparison with data movement costs. From this assumption, we stipulate that the initial step of all distribution strategies is to perform all local intra-node processing that reduces the size of relations, before any data movements are made. This initial step performs restrictions and join restrictions for all clauses in the query for which all variables therein are located at the same node. Then the inter-node joining domains and output domains for each relation are projected. After initial processing let:

m = number of relations in query,

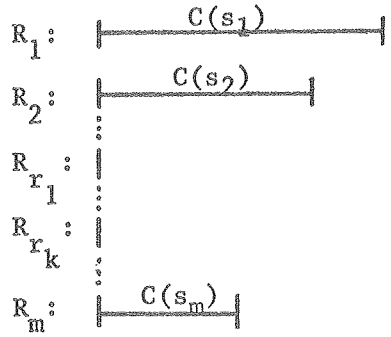
α_i = number of domains in R_i, i = 1, ..., m,

β_i = number of joining domains in R_i.

We will now describe a cost graphing method that will be used to analyze the costs of different distribution strategies. Our cost measure is given in time units. The graphing method illustrates the the timing of local processing and data movements found in a distribution strategy. (The complete graphing method is detailed in reference (4). We will present only a limited version here.) Local processing is represented in the graph as an ellipse () whose major axis is an analog measure of processing time. Data movement is represented by a horizontal line connecting processing symbols ( ). The length of the line corresponds to movement time, C(x). Since processing times will not be considered in the following analysis, a data movement between processing steps will be represented as ( ).

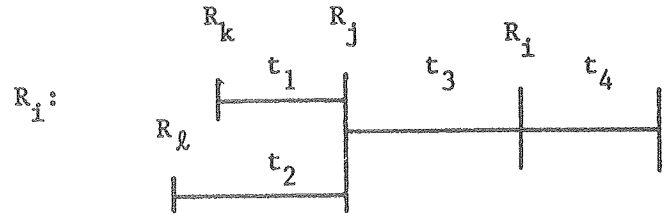
A query cost graph is composed of parallel lines of movement; one for each required relation. These parallel lines are viewed on a horizontal time axis. This allows us to recognize synchronization between different lines of movement. A line of movement for relation R_i represents the sequential and parallel data movements that are made in order to reduce the size of the relation before it is moved.

To demonstrate the cost graph consider what we will call the Initial Feasible Solution (IFS) for any query. After the initial local processing, all relations are moved directly to the result node along destination arcs. All processing is then done at the result node. This cost graph is shown below where the relations $R_{r_1},\ldots,R_{r_k}$ are assumed to be at the result node originally. Therefore their line of movement costs are zero.



Each line of movement is labeled by its relation R_i, i = 1,..., m.

Most algorithms use the IFS as a starting point from which to find efficient distribution strategies. The basic tactic used to recursively improve a distribution strategy is to find data movements on combining arcs that reduce relation sizes so that subsequent data movements are less costly. Thus, data movements are sequenced with local processing in serial and parallel arrangements. For example:



In this line of movement data is moved in parallel from R_k and R_ℓ to R_j. Relation R_j is reduced in size and data from it is then moved to R_i. Relation R_i is reduced in size and moved to the result node.

We define the line of time movement cost ($lincost_i$) for R_i as the time from the start of data movement until relation R_i is received at the result node. For this example $lincost_i = t_2 + t_3 + t_4$. The time t_1 is not included because it occurs in parallel with time t_2.

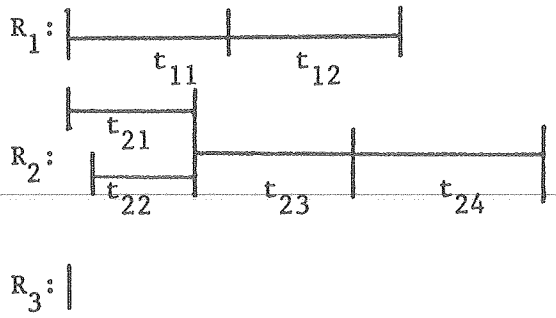
0 0 : 0 5 1 9 4 0 8 3

2.4 Cost Objectives

In order to understand what is meant by an efficient distribution strategy, we must define cost objectives of interest. We analyze two cost objectives for distributed query processing.

i) Response time cost. The cost objective is to minimize the time from the beginning of the initial processing step to the formation of the query result. In terms of the cost graph, the response time is the time represented by the maximum line of movement, i.e. response time cost = $\max_{i=1,\ldots,m} (\text{lincost}_i)$.

ii) Total time cost. The cost objective is to minimize the total time of data movements and local processing in the distribution strategy. All data movement lines in a cost graph are summed to find total cost. In the following cost graph,



Response time
$= t_{21} + t_{23} + t_{24}$.

Total time
$= t_{11} + t_{12} + t_{21} + t_{22} + t_{23} + t_{24}$

The choice of cost objective to be used depends upon the system. In a lightly loaded system with few queueing delays, response time minimization would appear to be preferable. In a more heavily loaded system, queueing delay consideration may make total time minimization the better objective.

3. ANALYSIS OF THE QUERY PROBLEM

3.1 System Assumptions

It is recognized that deriving an algorithm to find efficient distribution startegies for general distributed queries is a very difficult problem (5). In this section, we first describe a somewhat simplified query environment. A combining algorithm (Algorithm C) is then presented that finds an efficient distribution strategy for this environment in terms of response time for a given query.

We restrict the general query environment by making the following assumptions. Local processing costs are insignificant in comparison with data movement costs; and after the initial local processing, each

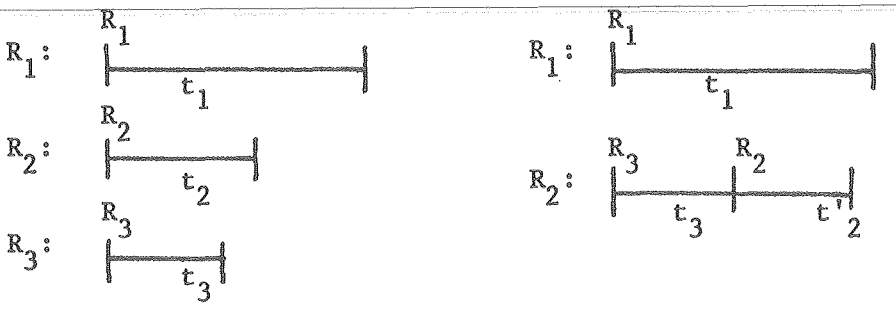
relation contains only a common joining domain. Thus

$$\beta_i = 1 \text{ for all } R_i,\ i = 1, \ldots, m.$$

Two implications of these assumptions are important to note. Since all relations are interconnected on a common joining domain each node containing query data would have only one required relation after initial local processing. Also once a relation is moved along a combining arc, then it need not be moved along its destination arc in the distribution strategy. This is because a move along a combining arc transfers the relation's complete information to the receiving relation, which may then be moved to the result node.

3.2 The Combining Algorithm

Recall that the minimum response time cost is defined as the maximum line of movement cost when all lines of movement costs have been minimized. It is clear that several distribution strategies may produce the same minimum response time, perhaps with differing total times. For example, both of the following cost graphs show feasible movement patterns for a query requiring data from three relations. Both graphs have equal response times although the distribution strategies and total times differ.



The response time is t_1 for both graphs.

We now propose Algorithm C to find an efficient distribution strategy for response time reduction. We start with the IFS and search for cost beneficial data moves in the current system state. The state of the system is given by the values of size s_i, selectivity p_i, and line of movement cost, lincost_i for each relation R_i. A cost beneficial move to reduce response time is defined as any data move to a relation R_i so that lincost_i is reduced. Relations are ordered so that after the initial processing step, $s_1 \geq s_2 \geq \ldots \geq s_m$. Algorithm C, then, uses this relation ordering to implement a tactic of moving relations with small size to relations with larger size if such a move is cost beneficial.

Algorithm C attempts to minimize lincost_i for each relation R_i in

the order i=m to i=1. Relations R_j, where $j > i$, are checked for potential cost beneficial data moves to R_i. The most cost beneficial of these moves is added to the distribution solution. The selectivity factor used to approximate the subsequent size reduction of relation R_i is $\prod_{k=j}^{m} p_k$ for the data move from relation R_j. Thus we consider that all relations R_k, $k > j$, can be moved in parallel to R_i along with R_j. In most cases, these parallel moves contribute no additional response time cost to the distribution strategy. The parallel data movement from relation R_k is added to the strategy, in the order k=j + 1 to k=m, if R_k does not already appear in a line of movement to R_i and if $p_k \neq 1$.

In Appendix A we detail Algorithm C.

Example: We can illustrate the use of Algorithm C by a numerical example. Assume a query such that four required relations at four different nodes have the following parameter values after initial processing. Let R_4 be at the result node, and let C(x)=x for data movement costs.

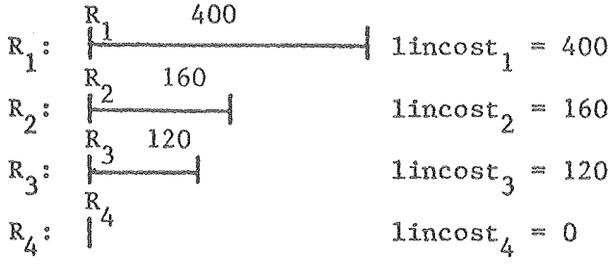
R_1: $s_1 = 400$ $p_1 = 1$

R_2: $s_2 = 160$ $p_2 = \frac{1}{2}$

R_3: $s_3 = 120$ $p_3 = 1$

R_4: $s_4 = 20$ $p_4 = \frac{1}{4}$

The IFS cost graph is:



Response time = 400 and total time = 680.

Algorithm C attempts to reduce lincost_i in the order R_3, R_2 and R_1.

Check R_3

R_4 to R_3: $\text{lincost}_3 = 20 + \frac{1}{4}(120) = 50 < 120$. Therefore,

Add R_4 to R_3 move to strategy.

Check R_2

R_3 to R_2: $\text{lincost}_2 = 50 + \frac{1}{4}(160) = 90$,

R_4 to R_2: $\text{lincost}_2 = 20 + \frac{1}{4}(160) = 60$.

Since $60 < 90$ and $60 < 160$, add R_4 to R_2 move to strategy.

Check R_1

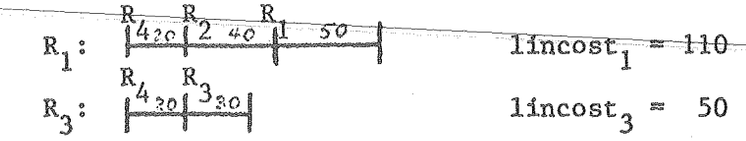
R_2 to R_1: $\text{lincost}_1 = 60 + \frac{1}{8}(400) = 110$,

R_3 to R_1: $\text{lincost}_1 = 50 + \frac{1}{4}(400) = 150$,

R_4 to R_1: $\text{lincost}_1 = 20 + \frac{1}{4}(400) = 120$.

Since 110 is minimum and $110 < 400$, we add the move R_2 to R_1 to the distribution strategy. Note that relation R_4 is already included in the line of movement of R_2 to R_1, and need not be moved in parallel to R_1. The selectivity of relation R_3 is 1, thus moving R_3 to R_1 in parallel with R_2 would cause no further size reduction.

The final cost graph using Algorithm C is:



Response time = 110 and
Total time = 160.

3.3 Minimum Response Time Cost

Of the two cost measures, response time and total time, we will concentrate on minimizing the response time per query. As such we assume that the distributed system has a light to medium communication load with short queueing delays. The response time cost of the query distribution strategy is minimized by Algorithm C if an ordering of the relations R_i exists such that $s_1 \geq s_2 \geq \ldots \geq s_m$ and $p_1 \geq p_2 \geq \ldots \geq p_m$. Note that this condition, while rather restrictive, nonetheless coincides with the intuitive observation that domains with few tuples usually have a small selectivity.

The formal proof of the above proposition can be found in (4). A sketch of this proof goes as follows. For each relation R_i we define the minimum possible cost of its line of movement to be mincost_i. We show that the ordering, $\text{mincost}_1 \geq \text{mincost}_2 \geq \ldots \geq \text{mincost}_m$, is satisfied. It can be shown that this result limits the search space

LEGAL NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or 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.

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: 376–400

LBL-7953

PROCEEDINGS OF THE THIRD BERKELEY WORKSHOP ON DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS

Sponsored by

Computer Science & Applied Mathematics Department
Lawrence Berkeley Laboratory
University of California

Applied Mathematical Sciences Research Program
Office of Energy Research
U.S. Department of Energy

And in cooperation with ACM and IEEE

AUGUST 1978

General Chairman: Arie Shoshani, Lawrence Berkeley Laboratory

Program Cochairmen: Steve Kimbleton, National Bureau of Standards
Dennis Tsichritzis, University of Toronto

Program Committee:

Peter Alsberg, Digital Technology Incorporated
Mani Chandy, University of Texas
James P. Fry, University of Michigan
Dennis Hall, Lawrence Berkeley Laboratory
Peter Kreps, Lawrence Berkeley Laboratory
Frederick H. Lochovsky, University of Toronto
Jon Postel, Information Sciences Institute, USC
James B. Rothnie, Computer Corp. of America
Carl Sunshine, RAND Corporation
Robert W. Taylor, IBM Research Center, San Jose
Peter Ting, Bell Laboratories
Richard Watson, Lawrence Livermore Laboratory
Bea Yormark, Interactive Systems Incorporated

ACKNOWLEDGMENTS

We would like to thank Dr. James C. T. Pool of the Office of Energy Research in the Department of Energy for his continued support of the workshop.

The program committee members greatly appreciate the help of the following people who assisted in refereeing the papers.

H. Aghili
P. Arraya
D. Austin
D. Batory
J. Bitner
W. A. Burkhard
P. Chan
P. Chang
T. C. Chiang
D. Corneil
S. Crocker
Y. Dalal
J. Day
M. Deppe
J. Donnolley
R. Epstein
J. Fletcher
L. Garlick
N. Goodman
G. S. Graham
M. Graham
J. Gray
B. Greiman
P. Griffiths
G. Grossman
P. Hawthorn
S. Holmgren
R. Hudyma
R. Johnson
K. Kim
T. Klug
I. Kornatowski
S. Lam
Y. E. Lien
B. Lindsay
I. Ludd
J. Mathis
J. Misru
J. Mullin
D. Nessett
P. Neumann
F. Olken
J. Popek
R. Rom
L. Schneider
D. W. Shipman
S. Sluizer
D. Swartwout
S. Tepper
R. Thomas
F. Tobagi
Y. Vassiliau
J. Weiner
J. White
D. Willcox
E. Wong
Y. Ying

CONTENTS

Acknowledgments iii

INVITED TALK

Provisional Model of Open System Architecture 1
Charles W. Bachman

NETWORK PROTOCOLS AND ARCHITECTURE
Chairperson: *Carl Sunshine*

Software Architecture for a Microprocessor-Based Packet Network 21
Holger Opderbeck, Jay H. Hoffmeier, and Roy L. Spitzer

Protocol Converters and User Interface in the CYCLADES Network 36
N. Naffah and H. Zimmerman

An Architecture for Secure Packet-Switched Networks 54
M. A. Padlipsky

PERFORMANCE ISSUES IN DISTRIBUTED DATABASES
Chairperson: *Peter Kreps*

Analysis of Update Synchronization for Multiple Copy Data-Bases 69
Erol Gelenbe and Ken Sevcik

Query Processing on a Distributed Database 91
Alan R. Hevner and S. Bing Yao

Performance Comparison of Two Update Algorithms for Distributed Databases 108
Hector Garcia-Molina

LOCAL NETWORKS
Chairperson: *Dennis Hall*

Interaction Between Protocol Levels in a Prioritized CSMA Broadcast Network* 123
James E. Donnelley and Jeffry W. Yeh

The Pacific Northwest Laboratory Minicomputer Network 144
L. H. Gerhardstein, J. O. Schroeder, and A. J. Boland

COMMUNICATION TECHNIQUES
Chairperson: *Peter Alsberg*

Inter-Network Fragmentation and the TCP* 161
John F. Shoch

Dynamic Management of Packet Radio Slots 169
Glenn Ricart and Ashok K. Agrawala

Failsafe Distributed Algorithms for Routing in Communication Networks 178
Adrian Segall

CONCURRENCY CONTROL I
Chairperson: *Bea Yormark*

A Formal Model of Concurrency Control Mechanisms for Database Systems† 189
Philip A. Bernstein and David W. Shipman

Maximally Concurrent Transaction Processing 206
Toshimi Minoura

Locking and Deadlock Detection in Distributed Databases† 215
Daniel A. Menasce and Richard R. Muntz

CONCURRENCY CONTROL II
Chairperson: *Peter Ting*

Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES† 235
Michael Stonebraker

Algorithms for Distributed Data-Sharing Systems Which Use Tickets 259
Gérard Le Lann

A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems 273
D. Z. Badal and G. J. Popek

0 0 4 0 5 1 0 4 0 3 5

NETWORK MODELS AND PERFORMANCE
Chairperson: *Jon Postel*

On the Design of Elementary Distributed Systems* 289
Randolph T. Yeh and K. Mani Chandy

Synchronization in Distributed System Modules* 322
Gregor V. Bochmann

Performance Limits of the TMM Protocol: Modelling and Measurement 340
Erol Gelenbe, Jean-Louis Grangé, and Patrice Mussard

Distributed Simulation Using a Network of Microcomputers* 358
J. K. Peacock, J. W. Wong, and Eric Manning

*These papers (or an improved version) were selected for publication in *Computer Networks Journal*.

†These papers (or an improved version) were selected for publication in *IEEE Transactions on Software Engineering*.

0 0 - 0 5 1 0 4 0 8 5

for cost beneficial moves in such a way that to reduce the size of relation R_j, only data moves from relations R_k, $k > j$, need be checked. Among these data moves it can be shown that only data moves from R_k for which $\text{lincost}_k = \text{mincost}_k$ need be considered. Thus, mincost_j can be found for each relation R_j in inductive steps from j=m to j=1. This optimal strategy to find a minimum response time is implemented by Algorithm C. The minimum response time $= \max_{i=1,\ldots,m} (\text{lincost}_i)$

$= \max_{i=1,\ldots,m} (\text{mincost}_i) = \text{mincost}_1$.

3.4 An Example

We now present an example query for which Algorithm C, and Wong's Algorithm (7) will be used to find distribution solutions. Response time costs and total time costs will be compared for the IFS, the Algorithm C solution, and the Wong Algorithm solution.

Consider a distributed relational database where each of the following relations is located at a separate node.

RELATION	VARIABLE
Department (D#, LOC)	D
Employee (E#, D#, SEX)	E
Manager (E#, D#, SAL)	M
Budget (BUD, D#)	B
Equipment (EQ#, D#, TYPE)	EQ

The example query is:

Find the department numbers (D#) of all departments that are located in the U.S. (LOC), have a budget over $100,000 (BUD), have managers that earn over $30,000 (SAL), employ at least one female (SEX), and have at least ten cars in their equipment inventory (TYPE).

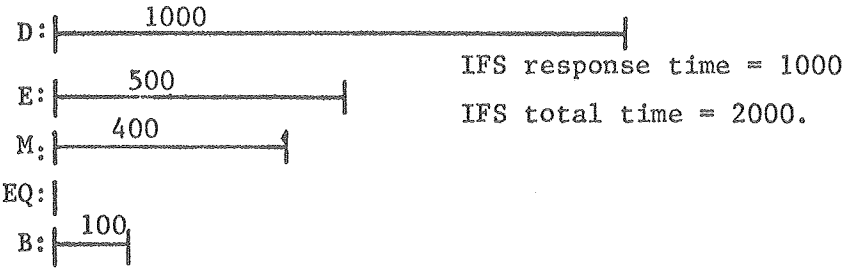
The first step of all distribution algorithms is to do all local processing. After the restrictions are performed on domains LOC, BUD, SAL, SEX, and TYPE, the unique D# values are projected for each relation. Thus each relation has only one common joining domain, D#. Assume the size and selectivities are:

RELATION	SIZE (s_i)	SELECTIVITY (p_i)
D	1000	½
E	500	½
M	400	½
EQ	180	½
B	100	½

This example satisfies the size and selectivity ordering

necessary to find a minimum cost distribution strategy for response time. We assume that relation EQ is at the result node. We also assume that C(x) = x in the example.

1. IFS cost. The IFS cost graph is:

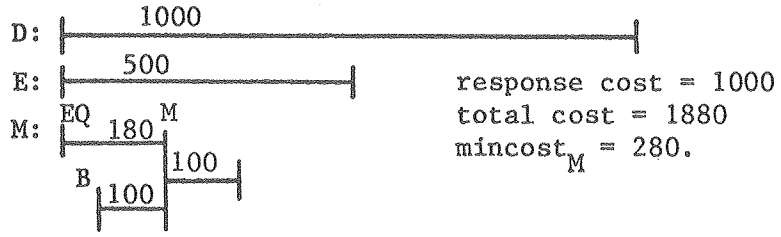


2. Algorithm C. We apply Algorithm C to the example. Recall that Algorithm C minimizes the line of movement cost for each relation from smallest to largest.

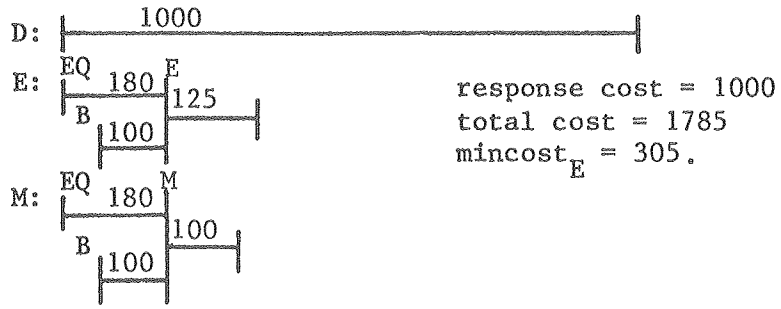
i) Minimize $lincost_B$. Since B is the smallest relation, no data movement can reduce B. $mincost_B = 100$.

ii) Minimize $lincost_{EQ}$. Even though EQ is at the result node, it still must be minimized since it may prove beneficial to move EQ to other relations. The line of movement EQ is eliminated from the cost graph at the last step of the algorithm. In this example it is not beneficial to move B to EQ, since $100 + 90 > 180 = lincost_{EQ} = mincost_{EQ}$.

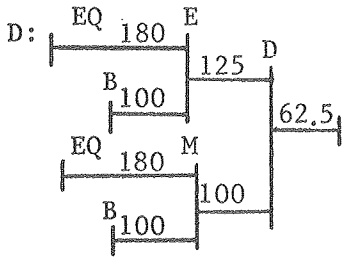
iii) Minimize $lincost_M$. Both the data moves from B and EQ are beneficial. Dropping the lines of movement for B and EQ, the new cost graph is:



iv) Minimize $lincost_E$. Parallel data moves from relations B and EQ are formed to be most cost beneficial. The new cost graph is:



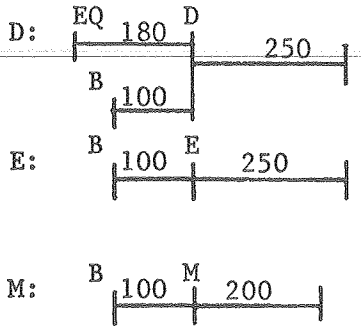
v) Minimize $lincost_D$. Parallel data moves from E and M are found to be most beneficial. Dropping the lines of movement for E and M, the final cost graph is:



$mincost_D$ = minimum response time
= 367.5
total time = 847.5 .

3. Wong's Algorithm. Using Wong's algorithm we find all data moves such that the cost of the data movement is less than the data reduction at the receiving relation. With this testing method we use the algorithm as described in reference (7).

The final cost graph is:



response time = 430
total time = 1180.

Table 1 shows the costs for the three derived distribution solutions for this example.

	RESPONSE TIME	TOTAL TIME
Algorithm C Solution	367.5	847.5
Wong's Algorithm Solution	430	1180
Initial Feasible Solution	1000	2000

TABLE 1

Algorithm C is seen to be superior in this restricted environment and for this particular query. This is to be expected since the algorithm is designed to use the ordering of relations to find minimum cost distribution solutions. Wong's algorithm and the IFS are applicable to much more general environments.

4. CONCLUSIONS AND FURTHER RESEARCH

This paper reports the preliminary results of a synthesis

modelling approach to the problem of query processing on a distributed database. We recognize that solving the distributed query problem in a general query environment is a difficult task. The goal of the first step of this research is to analyze and better understand several conceptually simple data movement tactics on a distributed database system.

For example, one simple data movement tactic in a distribution strategy is to move domains with small sizes and small selectivities to relations with larger sizes. An important part of analyzing such tactics is modelling query environments in which these tactics can be shown to be optimal in terms of minimizing some cost measure.

In this paper a simplified query environment is defined and an algorithm (Algorithm C) is presented that provides a minimum response time distribution strategy in the environment under a relation ordering condition. Algorithm C implements the basic tactic of moving small domains to large relations in order to find the most cost beneficial relation size reductions. As stated, Algorithm C is straightforward and simple to use.

When viewed from a realistic standpoint, the query environment assumptions in this paper appear quite restrictive. The relation ordering condition under which Algorithm C finds a minimum response time strategy can be partly justified by the observation that domains with small sizes usually have small selectivities. In addition, Algorithm C still can find a feasible distribution strategy even when this relation ordering is not satisfied. This is illustrated in the first example. The restriction of each relation to one common joining domain after the initial local processing limits the application of the algorithm. However, this assumption is required in order that Algorithm C may be stated and understood in a conceptually simple manner. It is believed that the simple case solution will provide insight to more complex situations.

Further research involves considering the general query environment. Heuristic algorithms can be modelled that extend and adapt the optimal movement tactics found in restricted environments to the general query environment. The modelling approach should also include developing methods to measure the quality of the distribution strategies derived by these heuristic algorithms.

```
      begin
         "move R_low to R_j";
         lm_j := lm_j + lm_low;          (* add relations in lm_low to lm_j *)
(* Add parallel moves *)
         for k:= low to m do
         begin
            drop:= drop + [k];
            if (k not in lm_j) and (p_k ≠ 1) then
            begin
               "move R_k to R_j";
               lm_j := lm_j + lm_k;
            end;
         end;
         "perform local processing at R_j";
         s_j := selopt * s_j;            (* new size *)
         lincost_j := sizopt + C(s_j);(* mincost_j *)
      end;
(* minimize next relation line of movement *)
      j:= j + 1 ;
   end; (* of while j > 0 loop *)
   "eliminate lines of movement R_ℓ where ℓ in drop";
end; (* of Algorithm C *)
```

APPENDIX A

Algorithm C is presented in a Pascal-type language. We use Pascal sets to record the relations found in the line of movement to relation R_i (ℓm_i) and to store the relations that need not be moved along destination arcs to the result node (drop). We assume relation R_r is at the result node.

Algorithm C

```
begin
   sel_{m+1} := 1.0;                    (* sel_i holds cumulative selectivity
   for i:= m downto 1 do                 product *)
   begin
      lincost_i:= C(s_i);               (* IFS system state *)
      sel_i:= sel_{i+i} * p_i;
      lm_i:= [i] ;                      (* relations in R_i line of movement*)
   end;
   drop:= [r];                          (* lines of movement to be
                                           eliminated *)
   j:= m-1;                             (* current relation *)
(* Find most beneficial data move to R_j *)
   while (j > 0) do
   begin
      low:= j + 1;
      sizopt:= lincost_{j+1};           (* Initialize R_{j+1} as best move *)
      selopt:= sel_{j+1};
(* Compare each relation R_k move with best previous move *)
      for k:= j + 2 to m do
      if (lincost_k + C(sel_k * s_j) < sizopt + C(selopt * s_j)) then
      begin
         low:= k;
         sizopt:= lincost_k;
         selopt:= sel_k;
      end;
(* Check if best move is beneficial - if so, add to solution *)
      if (lincost_j > sizopt + C(selopt * s_j)) then
```

0 0 - 0 5 1 0 4 0 8 8

REFERENCES

[1] Bernstein, P. A., Rothnie, J. B., Shipman, D. W., and Goodman, N., The SDD-1 Redundant Update Algorithm (The General Case) , Tech. Report No. CCA-77-09, Computer Corp. of America, Aug. 77.

[2] Chandy, K. M., "Models of Distributed Systems," Proc. VLDB Conference, Oct. 77.

[3] Computer Corporation of America, A Distributed Database Management System for Command and Control Applications, Semi-Annual Tech. Report No. CCA-78-03, Jan. 78.

[4] Hevner, A. R., and Yao, S. B., "Analysis of Distributed Query Processing," in preparation, 1978.

[5] Rothnie, J. B., and Goodman, N., "A Survey of Research and Development in Distributed Database Management," Proc. VLDB Conference, Oct. 77.

[6] Stonebraker, N., and Neubold, E., "A Distributed Database Version of Ingres," 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, Univ. of Calif., May 77.

[7] Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, Univ. of Calif., May 77.

[8] Wong, E., and Youssefi, K., "Decomposition--A Strategy for Query Processing," ACM Transactions on Database Systems, Vol. 1, No. 3, Sept. 76.

PERFORMANCE COMPARISON OF TWO UPDATE ALGORITHMS FOR DISTRIBUTED DATABASES

Hector Garcia-Molina

Computer Science Department
Stanford University

Abstract

Several algorithms have been suggested for updating redundant data in a distributed database. This paper analyzes and compares the performance of two of these algorithms in the case of completely duplicated databases in a no failure, update only environment. One of the algorithms is a centralized locking algorithm while the other is a distributed voting algorithm. The algorithms are studied through detailed simulations. However, the results obtained can also be approximated by an analytic technique based on a queuing model.

INTRODUCTION.

One of the most active areas in the field of distributed databases is the design of concurrency control mechanisms. In particular, several algorithms for updating duplicate data in a distributed database have been recently published (e.g., 1,2). Some of these algorithms have been compared in a qualitative fashion or by only considering the number of messages transmitted (e.g., 3). However, we believe that more complete and comprehensive comparisons are still needed in order to fully understand the tradeoffs involved in the choice of a redundant update algorithm.

This paper will attempt to shed some light on the subject by comparing the performance of two particular update algorithms. The algorithms are only compared in the completely duplicated database case in an update only, no failure environment. These restrictions have been made in this initial study in order to make the problem more tractible. Relaxing each restriction introduces a new set of issues which would at this point make the comparison harder to manage. (Notice that completely duplicated databases do arrise in practice, for example in a replicated system directory.) However, we believe that a detailed performance comparison even with these restrictions can produce intersting results and insights useful to the designer of a distributed database.

0 0 1 3 5 1 0 4 0 8 9

The procedure used for this study was the following one. First, a model that would encompass the main features of a distributed database and the algorithms was chosen. Then detailed event driven simulations were written for each algorithm. Finally, the results obtained from the simulations were checked by performing an approximate queueing analysis of our model. The results of the analysis agreed with the simulation results for most of the cases of interest.

THE ALGORITHMS.

The distributed database system consists of a set of completely duplicated databases located at each node of a computer network. We assume that each database is a fixed set of resources called items (6). Each item has a name and a value associated with it and each database in the system contains the same items. We also assume that update transactions specify fully at their inception the set of items they will reference. The update algorithms must guarantee that all the nodes in the system perform the same updates and they must also guarantee database consistency (6).

The two update algorithms we consider in this paper are a distributed voting algorithm due to Thomas (1) and a centralized locking algorithm. The locking algorithm is a good representative of a group of centralized control algorithms, while the voting algorithm is a good representative of the distributed control algorithms. Due to space limitations, we will only give very brief descriptions of the algorithms. We refer the readers to (1,4) for the details of the algorithms.

The centralized locking algorithm.

For this algorithm, one of the nodes is chosen as a "central" node and is in charge of locking. Each item has a lock that can only be granted by the central node. When an update arrives at a node "x", the following steps are performed:

1) Node "x" requests from the central node locks for all of the items referenced by the update.

2) The central node checks all of the requested locks. If all can be granted, then a grant message is sent back to node "x". If some items are already locked, then the request is queued. There is a queue for each item and a request only waits in one queue at a time. To prevent deadlocks, the items are always requested in a predefined order.

3) Once node "x" gets all of the requested locks, it can proceed with the transaction. The items are read from the local database, and the update is computed. A message is sent to all other nodes informing them of the update. Node "x" updates the values stored in its local database.

4) When the other nodes receive update messages, they perform the indicated update on their copy of the database. When the central node receives the update message, it also releases the locks of the involved items. Requests that were waiting on those items are notified and can continue the locking process at the central node.

To prevent timing problems (e.g., updates arriving out of order at a node), the central node gives sequence numbers to all updates it grants locks to. Nodes must remember the sequence number of the latest update message they have processed and they must delay updates that are out of order.

The distributed voting algorithm.

In this algorithm there is no central controller; the nodes communicate among themselves and decide what updates can be performed. The algorithm is based on the use of timestamps for items and updates (8). The nodes of the network form a daisy chain. Transactions move along this chain, each node in turn voting on the transaction. Some nodes may decide to defer the transaction until they can vote. When the transaction gets a majority of "ok" votes, then the last node that voted sends out the actual update command to all nodes. On the other hand, a node may reject a transaction. It will then send rejection messages to all nodes. When the node that originated the transaction receives the reject message, it must queue the transaction for another attempt at a later time. (Retry is actually optional. However, here we assume that rejected updates are always retried because this makes comparison with the other algorithm fair.)

Some readers may complain that the centralized locking algorithm described is unreliable, that it is not really a distributed system or that it has a "bottleneck" at the central node. First, let us point out that in this paper we are only considering failure free systems and that neither one of the algorithms as we have described them is "failure proof". However, we believe that both algorithms can be made resilient to an arbitrary degree by adding the necessary checks and redundancy to the original algorithms. For example, in the centralized case, one or many backup central nodes can be included in the system (7). The central node can inform the backup nodes of all of its activities, so that in case the central node fails, one of the backup nodes can take its place. Similar ideas can be used for the distributed voting algorithm. Of course, it might be somewhat simpler to make one algorithm resilient than it would be to make the other. However, at the time being it seems that it will be just about as hard to add the resiliency to one algorithm as to the other, so that it seems fair to compare the centralized algorithm with the distributed one.

It is also important to notice the difference between a distributed system and distributed control. The centralized update algorithm is a case of centralized control in a distributed system. Such a system can still be categorized as a distributed since it provides all of the features desired in a distributed system: reliability, modularity, independent nodes, etc.

Finally, it is true that the centralized locking algorithm has a bottleneck at the central node which the distributed algorithm does not have. This paper will illustrate when the bottleneck occurs and what problems it causes.

THE MODEL.

Figure 1 shows the model that was used to represent the distributed database system at each node. The model was designed to be as simple as possible while still displaying the principal characteristics of such a system. Requests for service arrive at a node from three sources: the users at the node, the network and the node itself. Each request for service can be of two types: a request for CPU time only and a request for IO service followed by CPU time. Both the IO and the CPU servers at each node serve at most one request at a time, so that first-in-first-out queues are provided for waiting requests. Once a request is served, it can generate further requests, either at the same node or at other nodes.

In our model, we assume that the arrival of updates from users is a Poisson process. We also assume that the number of items referenced by an update is exponentially distributed. (For the exact distribution, see (4).) An update that arrives from a user is first processed at the originating node. Then, depending on the update algorithm, messages are sent out to other nodes requesting service. The CPU and IO service times requested at a given node are the CPU and IO times needed by the algorithm at the particular step at that node.

In a real distributed database system, the IO operations for a request will be interleaved with the CPU computations for the same request. However, we believe that the effect of a collection of small IO operations and small CPU operations for a service request is equivalent to the effect of lumping all of the IO operations into one and of similarly lumping the CPU operations. Also, in a real system the requests will probably not be processed sequentially in a first-in-first-out fashion. Nevertheless, since most of the requests for service are small, the scheduling algorithm should not have a noticeable effect on the response times of the updates and the results obtained with this model should be comparable to the results obtained with a more complex model that allows concurrent IO or CPU operations at a single node. (In a multi-processing environment, the different operations in progress at a node can interfere with each other. Therefore, a local concurrency control is needed at each node for both algorithms. This extra overhead can be added to our model by varying some of the model parameters that will be described.)

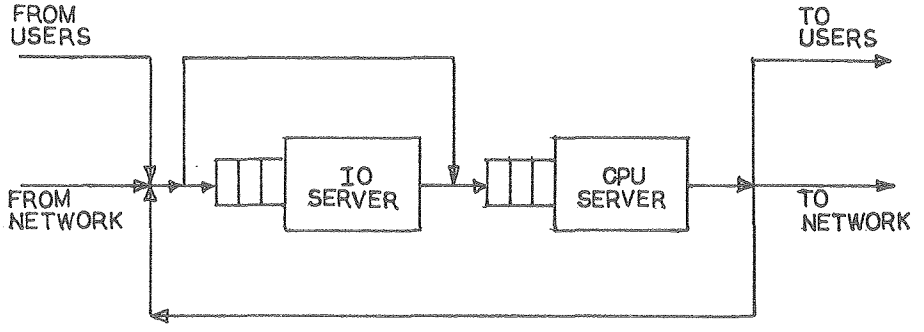


Figure 1.
The model at each node.

The model parameters.

The model parameters describe a particular instantiation of our model. In order to obtain useful results, it is important to select a small number of meaningful parameters. The parameters we selected for the model are the following:

1) Mean interarrival time of updates at each node, Ar. The interarrival time is exponentially distributed with mean Ar. (That is, it is a Poisson process.) This means that on the average 1/Ar updates per second arrive at each node.

2) Mean base set, Bs. The number of items referenced by an update transaction (the base set) is exponentially distributed with mean Bs. (Actually, its a truncated exponential distribution; see (4) for details.) This means that updates that reference a small number of items will occur more frequently than updates that reference many updates. Out of the referenced items, some will be read-only items while the rest (at least one) will be read and write. We assume that the fraction of the read and write items (items in the write set) in an update is uniformly distributed with mean 0.5. That is, on the average, half of the items will be read only.

3) The number of items, M. This parameter is the total number of items in each database.

4) The number of nodes, N. The number of nodes and databases in the system is N.

5) Network transmission time, T. We assume that the time it takes any message to go from any node to any other node is a constant T. This is an acceptable assumption if the communications network is lightly loaded or if the distributed database message traffic is only a small fraction of the total network traffic.

6) CPU time unit, Ct. The CPU computation time is the time it takes any CPU server to do a "small" computation. For example, in Ct seconds a processor can check and set a lock in memory, it can compare two timestamps in memory and maybe add something to a queue, or it can send a lock request to the central node.

7) CPU update time, Cu. If an update references x items, then the time to compute the actual update values is x times Cu. That is, once an update transaction has been accepted or all of its locks have been granted, it will need x times Cu CPU seconds before a message with the update values can be sent to all nodes.

8) IO time unit, It. This is the time it takes to read or write a lock or a timestamp from an IO device. For example, if a transaction at a node needs to read x timestamps, then it must get x times It seconds of service time from the IO server at the node.

9) IO item update time, Iu. The time needed to read or write one item value from or to the IO device is Iu.

10) Retry delay time, Rt. This last parameter is special since it only applies to the distributed voting algorithm. The retry time is the time a node must wait before retrying a rejected update transaction.

0 0 0 3 5 1 0 4 0 9 1

THE RESULTS.

Using the model described, two simulators were written to study the performance of the update algorithms. Updates were generated at each node according to the distributions specified in the model. (The items of the database they referenced were chosen at random.) Then the steps followed by each update were mimicked in a detailed fashion by the simulators. The statistics collected by the simulators included, among others, the average response time of updates (R), the number of messages sent per update, and the CPU and IO utilization. The response time of an update is defined as the difference between the finish time and the time when the update arrived at the originating node. The finish time is when the originating node has finished all work on the update. The messages to and from a user are considered internal messages to a node and are not counted here. A broadcast message is counted as N - 1 messages, where N is the number of nodes.

Very similar results were also obtained by performing an approximate queueing analysis of the model (5). The results were very close in all cases where the maximum utilization was less than 0.60 and the square of the average number of items in the base set was less than 5 percent of the total number of items. In the rest of the cases, the queueing analysis reported slightly smaller average response times than the simulators did. This difference is due to the approximate nature of the analysis. In any case, the analysis results provide a good "double check" of the simulation results.

Figures 2 through 10 present some selected results obtained with the simulators. The values selected for the parameters are what we feel would be typical values for a distributed database implemented with current technology.

Figure 2 shows the relationship between the interarrival time Ar, the number of nodes N and the mean response time R. In most cases, the centralized locking algorithm performs better than the distributed voting algorithm. For small number of nodes, the difference is not as dramatic, but as the number of nodes increases, the difference in performance increases. The so called "bottleneck" effect does appear in the centralized algorithm; there is a relatively sharp knee in the curve when the requests for locks swamp the central node (e.g., N = 6, Ar = 5). Since there is no bottleneck in the distributed algorithm, one expects this algorithm to do better under heavy loads. Indeed, the distributed algorithm does not have the sharp knee, but it turns out that the distributed algorithm does better only in a limited range and this range covers cases where both algorithms perform poorly. The explanation for this is that in the distributed algorithm all nodes get swamped when the update arrival rate increases too much. This can be seen in figure 3 which shows the IO utilization as a function of the load (for 6 nodes only). The CPU utilization shows similar behavior but the utilization is much less than the IO utilization, i.e., the IO resource is the critical one for our choice of parameters. Notice that the central node is the one that gets the heavy load in the centralized algorithm, but the total amount of IO time used in the distributed algorithm (sum of all nodes) is considerably greater than the total IO time needed in the central locking algorithm.

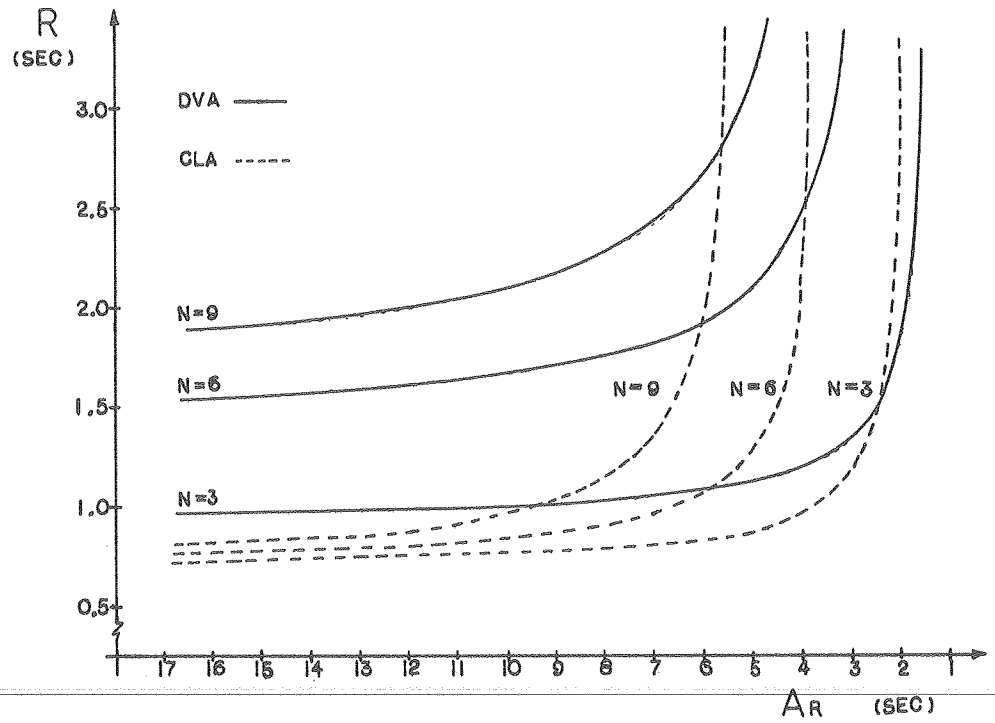


Figure 2.
The average response time of updates, R, versus the interarrival time Ar for several values of N, the number of nodes. (M = 1000, Bs = 5, It = Iu = 0.025 sec., T = .1 sec., Rt = 1 sec. Ct = 0.1 msec., Cu = 1 msec.)

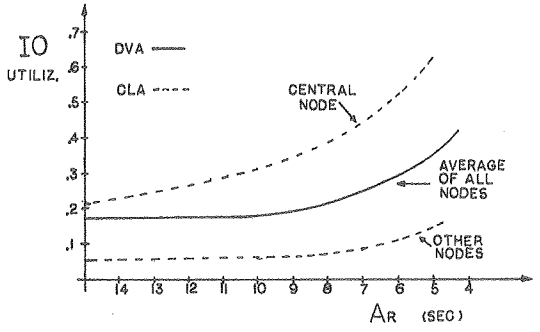


Figure 3.
The IO utilization versus the interarrival time Ar. (N = 6, M = 1000, Bs = 5, It = Iu = 0.025 sec., T = .1 sec., Rt = 1 sec., Ct = 0.1 msec., Cu = 1 msec.)

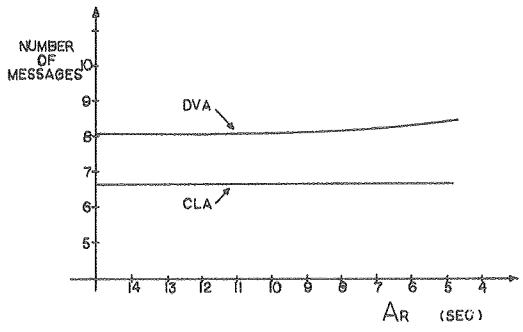


Figure 4.
The number of messages per update versus the interarrival time Ar. (N = 6, M = 1000, Bs = 5, It = Iu = 0.025 sec., T = .1 sec., Rt = 1 sec., Ct= .1 msec., Cu= 1 msec.)

0 0 - 0 5 1 3 4 0 9 2

Figure 4 shows the number of messages transmitted per update for the case of 6 nodes. In the central algorithm this number is independent of the load. Updates that originate at the central node only need 5 messages, while updates originating at other nodes require 5 + 2 messages. The average is 6.66 messages per update. In the distributed algorithm, the number of messages increases slightly as some updates are rejected. Without rejections, an update requires 3 messages to obtain a majority of votes, plus 5 messages for performing the update.

For 6 nodes, the difference in the number of messages transmitted is small. The difference accounts for about 0.23 seconds of the difference of response times (since T = .1 and since 5 of the messages in the centralized algorithm and 4 in the distributed algorithm do not add to the response time). This means that the big difference in response times comes primarily from the higher total IO utilization of the distributed voting algorithm. For a larger number of nodes, the difference in the number of messages transmitted is substantial and the difference in response times is even greater.

The fact that the centralized algorithm performs better than the distributed algorithm as the number of nodes increases is illustrated in figure 5. The jumps in the curve for the distributed algorithm reflect the number of nodes needed for a majority consensus.

Figure 6 shows the effect of M (the number of items) and Bs (the mean base set) on the system performance. As long as Bs is a very small fraction of M, the response time is independent of M. This is convenient since it is possible to extrapolate our results to very large databases. As the size of the database is reduced (with Bs held constant), the number of conflicts among transactions must increase, and therefore the response times grow as M decreases. Notice how the centralized algorithm handles the increased number of conflicts better than the distributed algorithm. Centralized control is a more efficient way of arbitrating many conflicts.

Figure 7 presents the effect of Bs (the mean base set) on the average response time R for constant M (number of items). For a small number of conflicts (in this case, Bs < 5), the curves are linear since the response time is proportional to the number of items in an update. As the number of conflicts increases (Bs > 4), the response time displays a nonlinear component due to the extra delays.

Figure 8 shows the effect of varying the IO time unit It. An IO time unit of 0 represents a system where all the locks or timestamps are kept in main memory. Once again, the distributed voting algorithm outperforms the centralized locking algorithm only in extreme circumstances.

The IO time unit is a critical parameter because response times grow rapidly as this parameter is increased. Therefore, a good distributed database implementation should try to reduce this value as much as possible. One obvious way to do this is by keeping the most commonly used locks or timestamps in main memory. But it turns out that this is simpler to do for locks than for timestamps. If the number of locked items is usually small, it is easy to keep a complete list of their names in memory. The absence of an item name in the list indicates that the item is free while the appearance of an item's name in the list means that the item is locked. Thus, the IO time unit can be reduced practically to 0 (as long as the number of locked items is small enough

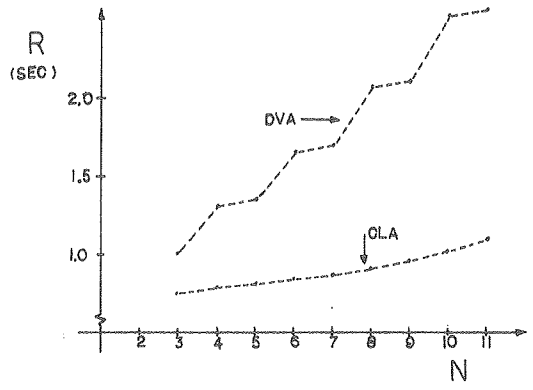


Figure 5.
The average response time of updates, R, versus the number of nodes N. (Ar = 10, M= 1000, Bs= 5, It = Iu = 0.025 sec., T = .1 sec, Rt = 1 sec., Ct = 0.1 msec., Cu = 1 mesc.)

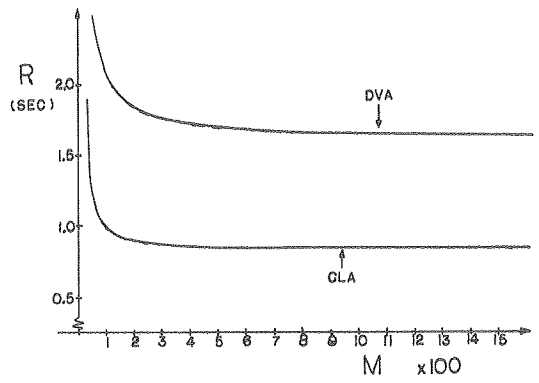


Figure 6.
The average response time of updates, R, versus the total number of items M. (N = 6, Ar= 10, Bs= 5, It = Iu = 0.025 sec., T = .1 sec, Rt = 1 sec., Ct = 0.1 msec., Cu = 1 mesc.)

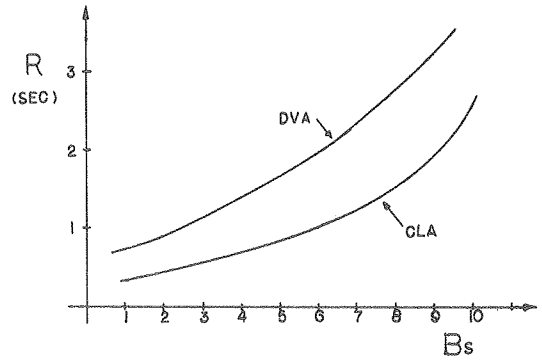


Figure 7.
The average response time of updates,R, versus the average number of items in the base set Bs.
(N = 6, Ar = 10, M = 1000, It = Iu = 0.025 sec., T = .1 sec., Rt = 1 sec., Ct = 0.1 msec., Cu= 1 mesc.)

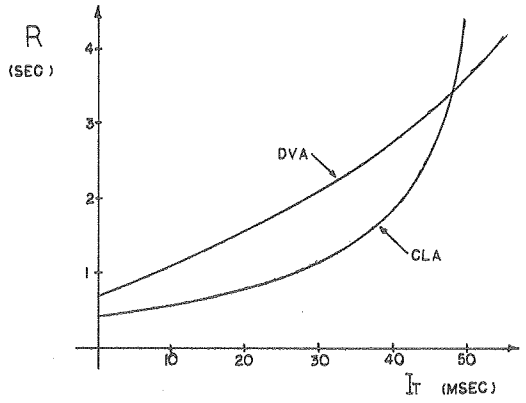


Figure 8.
The average response time of updates, R, versus the IO time unit It. (N = 6, Ar = 7, M = 1000, Bs = 5, Iu = 0.025 sec., T = .1 sec., Rt = 1 sec., Ct = 0.1 msec., Cu = 1 msec.)

0 0 0 0 5 1 0 4 3 9 3

so that their names fit in main memory). On the other hand, it is harder to apply the same idea to timestamps. Since it is usually not feasible to keep all timestamps in memory, only the most recently used timestamps are kept. If a timestamp is not found in memory, then it must be paged in from the IO device. If a timestamp is modified, the new value must be written out to the IO device. The savings in IO time will depend on the reference pattern of the updates. The average IO time per timestamp read can be reduced but it will probably be greater than 0. Thus, for a fair comparison of the algorithms, the parameter IO time unit for the distributed voting algorithm should have a greater than 0 value, while the same parameter should be 0 for the centralized locking algorithm. It is clear from figure 8 that this consideration only makes the distributed algorithm look even worse.

The effect of the network transmission time T on the average response time is shown in figure 9. One end of the range represents the slower networks like the ARPANET (9), while at the other end are the much faster ones like the ETHERNET (10). The centralized locking algorithm performs better and is less sensitive to this parameter.

Figure 10 shows a case where the distributed voting algorithm performs almost as well as the centralized locking algorithm. The parameters were chosen so as to favor the distributed algorithm, but as we can see, the centralized algorithm still performs slightly better. The IO utilization is the same for both algorithms (since It = 0). The distributed algorithm does slightly better with respect to the number of messages transmitted. The centralized algorithm needs 3.33 messages per update on the average, while the distributed algorithm only needs 3 (except at high loads). This can only happen in the special case of a 3 node network.

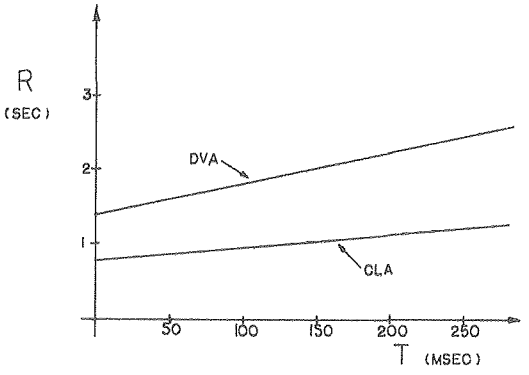


Figure 9.
The average response time of updates, R, versus the network transmission time T. (N = 6, Ar = 7, M= 1000, Bs = 5, It = Iu= 0.025 sec., Rt = 1 sec., Ct = 0.1 msec., Cu = 1 msec.)

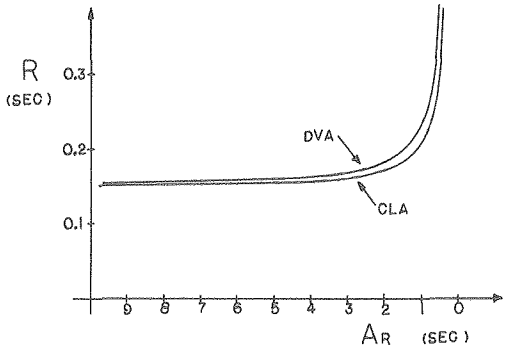


Figure 10.
The average response time of updates, R, versus the interarrival time Ar, for a special case. (N = 3, M = 1000, Bs = 3, It = 0 sec., Iu = 0.025 sec., T = 1.25 msec., Rt = 1 sec., Ct = 0.1 msec., Cu = 1 msec.)

CONCLUSIONS.

The centralized locking algorithm performs considerably better than the distributed voting algorithm except in cases of extreme IO utilization. In these high load cases, the average response time for updates in the distributed algorithm is smaller than the one for the centralized algorithm, although both response times are poor. In most cases, the redundant update problem seems to be solved more efficiently and naturally using centralized control strategies.

Of course, this does not mean that the centralized algorithm should always be chosen over the distributed one. There are many other factors that must be considered before choosing an algorithm. However, in the cases we have tested, it seems that the distributed voting algorithm is going to be a more expensive algorithm, both in terms of resources and response times.

Another advantage of the centralized locking algorithm is that it is much simpler than the distributed one. This fact can clearly be seen by comparing the simulators or the queueing analysis for both algorithms. Thus, the distributed voting algorithm will probably be harder to implement, and even if this algorithm is more resistant to failures, it will certainly be more prone to software bugs.

From the results obtained, it seems that a distributed database system is more sensitive to the following parameters: N, the number of nodes, Ar, the interarrival time, Bs, the mean base set size, It, the IO time unit, and T, the network transmission time. Under typical circumstances, these five parameters define the performance of the system and special care should be taken to reduce the magnitude of these parameters in a real distributed database.

Research is currently underway to study the performance of other update algorithms. The effects of relaxing some of the assumptions made in this paper will be studied. It is also possible to use the simulators to tune a particular algorithm. Some aspects that may be optimized are a lock granule size or the strategy for eliminating deadlocks in the centralized algorithm.

ACKNOWLEDGMENTS.

Several useful suggestions and ideas were provided by Clarence Ellis, Jim Gray, Maurice Maybury and Gio Wiederhold. This work was partially supported by a fellowship from IBM Corporation, by the SLAC Computation Research Group of the Stanford Linear Accelerator Center under Department of Energy contract EY-76-3C-03-0515, and by the Biotechnology Research Program of the National Institute of Health under grant NIH RR-00785.

REFERENCES.

1. R. H. Thomas, "A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control", BBN Report 3340, July 1976.
2. C. A. Ellis, "Consistency and Correctness of Duplicate Database Systems", Proc. 6th ACM Symposium on Operating Systems, Nov. 1977.

0 0 : 0 5 1 0 4 0 9 4

3. J. B. Rothnie, N. Goodman, "A Survey of Research and Development in Distributed Database Management", Proc. 3rd Intl. Conf. on Very Large Data Bases, Tokyo, Japan, October 1977.
4. H. Garcia-Molina, "Performance Comparison of Update Algorithms for Distributed Databases, Part 1", Report HPP-78-6, Computer Science Department, Stanford University, March 1978.
5. H. Garcia-Molina, "Performance Comparison of Update Algorithms for Distributed Databases, Parts 2 and 3", to be published, April 1978.
6. K. Eswaran, J. N. Gray, R. Lorie, and A. Traiger, "On the Notions of Consistency and Predicate Locks in a Data Base System", CACM November 1976.
7. P. Alsberg and J. Day, "A Principle for Resilient Sharing of Distributed Resources", 2nd Intl. Conf. on Software Engineering, San Francisco 1976.
8. P. Johnson and R. Thomas, "The Maintenance of Duplicate Databases", Network Working Group RFC 677, January 1975.
9. L. Kleinrock, "Queueing Systems", Vol. 2, John Wiley & Sons Publisher, 1976.
10. R. Metcalfe and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM, July 1976.

0 0 : 0 3 1 0 4 0 9 5

INTERACTION BETWEEN PROTOCOL LEVELS IN A PRIORITIZED CSMA BROADCAST NETWORK

James E. Donnelley and Jeffry W. Yeh

University of California, Lawrence Livermore Laboratory
Livermore, California 94550

Abstract

Discrete event simulation studies of the *Hyperchannel* product of the Network Systems Corporation* demonstrate surprising performance variations resulting from rather subtle protocol, timing, and topology changes. Early analytic and simulation studies showed the effectiveness of the lowest level Hyperchannel contention mechanisms. Later simulations, including the next level of microcoded error and flow control protocols, showed that great care must be taken to maintain high trunk utilization through both protocol levels. Examples of performance degradation arising from interaction between the two protocol levels illustrate the type of information that can be obtained from simulation analysis of communication systems too complex for current state of the art mathematical models.

INTRODUCTION

The major unifying theme of this paper is the characterization of interaction between protocol levels in a high-speed local network. The description of the network protocols, given as necessary background for the examples, may itself be of value to readers interested in this type of system. All of the performance results presented were obtained from discrete event simulations. Though a description of the simulations is beyond the scope of the present paper, the level of detail visible in the simulation results indicates the completeness that can be obtained with this type of analysis. The simulations were written in the ASPOL simulation language and were run on a Control Data Corporation 7600 computer.

At the time of writing, the Hyperchannel hardware available at the Lawrence Livermore Laboratory (LLL) was not sufficiently operational to allow detailed verification of the simulation results presented in this paper. Therefore, though great care was taken in making the simulations resemble the existing Hyperchannel hardware as closely as possible, the results presented only reflect the current level of the authors' understanding of the Hyperchannel operation. Note also that several important features of the Hyperchannel product (for example, the ability to support multiple trunks) that are not pertinent to the major theme of the paper are not included in the descriptions below.

---

*) 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.

THE HYPERCHANNEL NETWORK

The Network Systems Corporation (NSC) Hyperchannel product is a commercially available instance of a Carrier Sense Multiple Access (CSMA) broadcast local network [Hyper1]. This type of network (Fig. 1) broadcasts frames (Fig. 2) on coaxial cables to communicate units of information from one cable tap or *port* to another.

The term *carrier sense* means that each port on the cable or *trunk* can sense the presence of a passing frame. If a port has a frame to transmit but senses a frame already passing on the trunk, it can wait until the trunk is free before transmitting, thereby avoiding the destruction of both the passing frame and the frame to be transmitted.

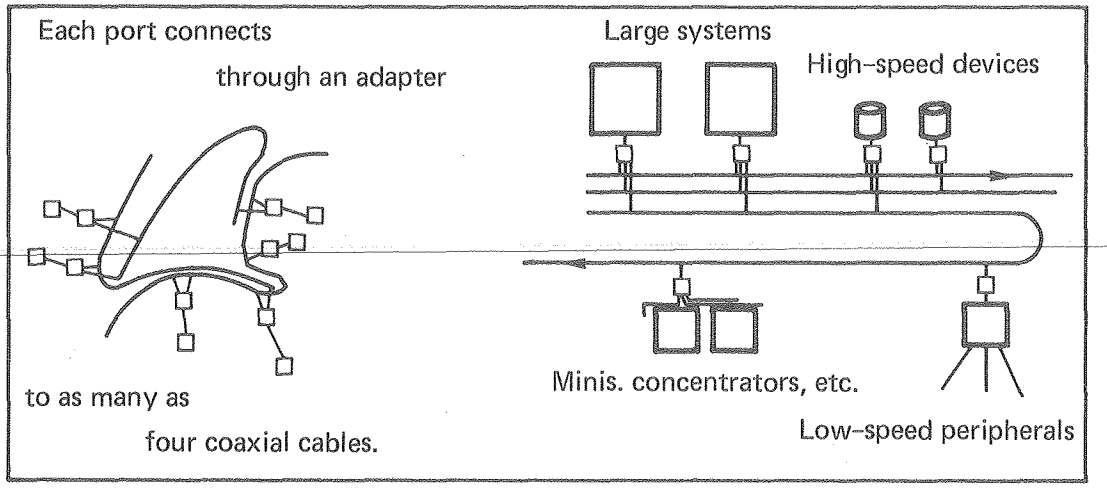


Fig. 1. Schematic of CSMA coaxial-cable ports.

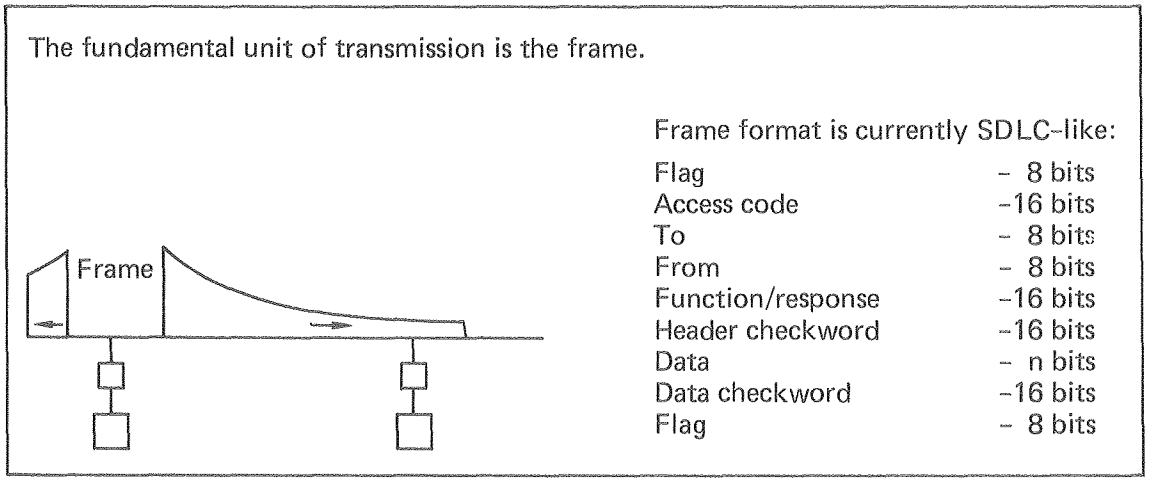


Fig. 2. The frame unit of the CSMA broadcast network.

0 0 1 0 3 1 0 4 0 9 6

The ability to sense carrier by itself is not sufficient to avoid all frame collisions. For example, two ports at the ends of a trunk might nearly simultaneously try to transmit frames to a third port located between them. When the middle port detects the frames going by, it will see the linear addition of the two signals which will generally be unintelligable (Fig. 3).

Methods of resolving contention for a broadcast medium have been described for [Aloha], [Ethernet], and [Hyper2]. Because of its importance in illustrating the interaction between protocol levels, the Hyperchannel contention-resolution mechanism is reviewed in the section on LEVEL-1 PROTOCOL.

The Hyperchannel trunk-level protocol itself supplies a rather primitive datagram transmission facility. To supply a more hospitable service to systems attached to a Hyperchannel, the ports supplied by NSC are packaged in an *adapter* that includes some high-speed buffer memory and a microprocessor that:

- Decouples the 50-Mbit/sec trunk speed from the possibly slower speeds of the interfaces to the attached systems.
- Handles duplicate detection on the trunk.
- Supplies a flow control mechanism.
- Recovers gracefully from various error conditions.
- Allows flexibility in communicating with attached systems.

The mechanisms that the microprocessor implements to satisfy these needs constitute a second level of protocol, which is described in the section on LEVEL-2 PROTOCOL.

LEVEL-1 PROTOCOL, TRUNK CONTENTION

Unlike the randomized retransmission schemes utilized in [Aloha] and [Ethernet] where all ports are equal, the Hyperchannel contention mechanism is built around a linear *priority* ordering of its ports. The intent is to give

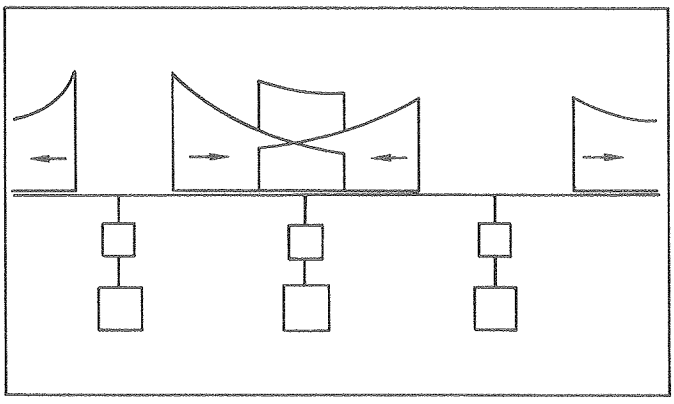


Fig. 3. "Passing in the night," a property of linear media.

the the highest priority port as much of the trunk bandwidth as it requests. The second priority port is given whatever it requests from what is left, and so on. As long as the total load on the trunk does not exceed the trunk capacity, each port should have all the transmission capacity that it needs. Any time the load on the trunk exceeds the trunk capacity, the high priority ports are allowed to keep their required bandwidth from the lower priority ports.

The Hyperchannel mechanism for implementing this priority scheme uses timers in each port. A port's timer is continuously being loaded with a time value called *Total Delay* (explained below) as long as carrier is sensed at the port. When carrier is no longer detected (e.g. a frame has passed), the timer begins counting down toward three events: Fixed Delay, then N-Delay, and finally Total Delay.

When the timer runs out and the Total Delay event is signaled, the timer is disabled. If at any time (even during the count-down sequence) carrier is detected on the trunk, the timer is reloaded with the Total Delay value and the sequence is restarted.

Signifigance of the Delay Signals

In this section we indicate how these delay values are set in the ports to implement the intent of the priority trunk allocation mechanism. The Fixed Delay signal is only used by a port that is waiting for a response frame (hardware-generated acknowledgement). If this timer level is reached, the port will never receive the awaited response (either the originally transmitted frame or its response having been lost). A port whose timer is enabled is not allowed to transmit a frame except when its N-Delay event is signaled or when its timer is disabled due to Total Delay expiration.

In the descriptions and examples below we do not include certain small delay components needed to allow for hardware response times (these times are included in the hardware and in the simulations).

1. Fixed Delay. Hardware in each Hyperchannel port generates a response frame immediately after receiving a transmission frame. Because of this, if a response frame is not received within: (a) the time it takes the trailing edge of the transmitted frame to reach the destination, plus (b) the time it takes the response to return, then a response will never be received. To simplify the detection of a lost transmission or response frame, a single Fixed Delay event is signaled in a transmitting port if the port's timer goes below a value that indicates a response cannot possibly be received from even the most distant receiving port.

Fixed Delay (K) = 2 * propagation time from K to the port most distant from K.

For the example topology shown in Fig. 4, we obtain:

Port	Priority	Fixed Delay
B	1st	2 X 500 nsec = 1.0 usec
A	2nd	2 X 1000 nsec = 2.0 usec
C	3rd	2 X 1000 nsec = 2.0 usec

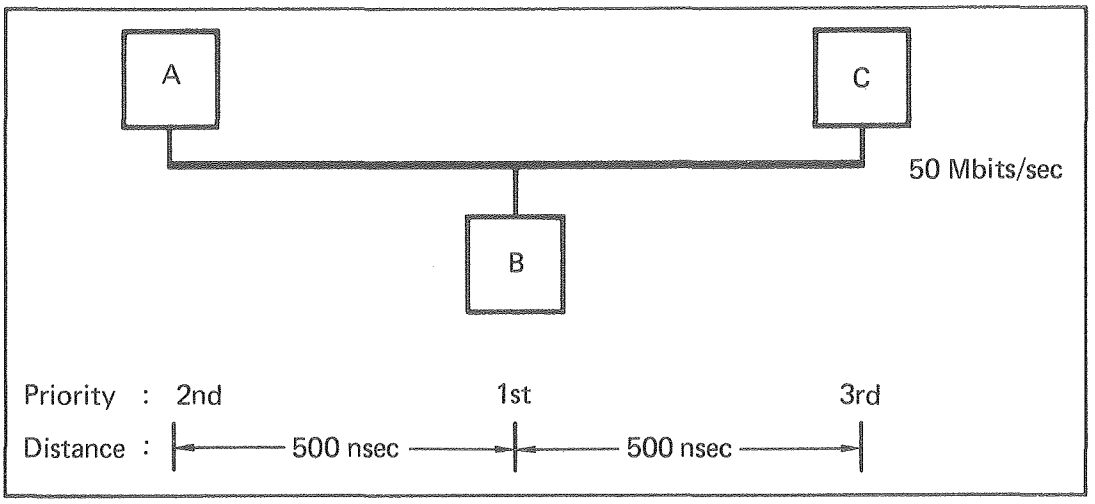


Fig. 4. Example of the three ports and one trunk network.

Although each port has its own Fixed Delay level, NSC currently sets the Fixed Delay levels in all of their ports to the same worst case value of the Fixed Delay (2.0 usec in the above example). The simulation results reported here also follow this convention.

2. N-Delay. To insure that the highest priority port does not interfere with responses to frames that have passed it, its N-Delay is set to the same level as its Fixed Delay. Each succeeding priority port, K, is required to wait long enough to insure that its transmission does not interfere with a properly slotted transmission by the next higher priority port, K-1. To do so, K must wait until its timer at least reaches: (a) the N-Delay of K-1, plus (b) one K to K-1 propagation time for the worst-case headstart that K's timer could have on K-1's, plus (c) one K-1 to K propagation time, to allow K to sense a proper transmission by K-1 and reset its timer.

N-Delay(K) = N-Delay(K-1) + 2 * propagation time from K to K-1.

For the same example topology of Fig. 4, we obtain:

Port	Priority	N-Delay
B	1st	Same as its Fixed Delay = 1.0 usec
A	2nd	1.0 usec + 2 X 500 nsec = 2.0 usec
C	3rd	2.0 usec + 2 X 1000 nsec = 4.0 usec

3. Total Delay. If a port is allowed to transmit a frame any time after its N-Delay event is signaled, a collision may occur with a frame transmitted in proper order by a lower priority port. To avoid this possibility, each port, K, that has a frame to transmit but has missed its N-Delay signal will not be allowed to transmit until it either gets reloaded and again reaches its N-Delay level or its timer runs out of its Total Delay value of: (a) the N-Delay of the lowest priority port, L, plus (b) one K to L propagation time, to allow for the worst-case headstart that K's timer could have on L's, plus (c) one L to K propagation time, to allow K to sense a properly slotted transmission by L and reset its timer.

Total Delay(K) = N-Delay(L) + 2 * propagation time from K to L.

Again, for our example topology of Fig. 4, we obtain:

Port	Priority	Total Delay
B	1st	4.0 usec + 2 X 500 nsec = 5.0 usec
A	2nd	4.0 usec + 2 X 1000 nsec = 6.0 usec
C	3rd	4.0 usec + 2 X 0 nsec = 4.0 usec

After this Total Delay event is signaled, a port's timer stops until it is reloaded by sensing carrier on the trunk. While a port's timer is stopped, it is in a free-for-all or contention mode. In contention mode, a port is free to transmit a frame at any time. Any frame it does transmit may be lost as a result of collisions with frames transmitted by other ports in contention mode. Frames transmitted at an N-Delay signal are guaranteed to be successfully transmitted (in the absence of hardware failures).

With Total Delay, as with Fixed Delay, NSC currently uses the worst-case delay for all ports (6.0 usec in the above example). The simulation results reported here also make this assumption.

A careful understanding of the level-1 protocol is important for appreciating the interaction examples given later. Following imaginary frames through chalk topologies has proven an indispensable learning tool for the authors.

LEVEL-1 PROTOCOL PERFORMANCE

We will usually consider only two network performance measures, throughput and delay. To analyse the performance of the trunk level contention mechanism, we simulated it by using a simple form of exponentially-generated traffic (Fig. 5). Each node waits an exponential time with mean gamma, submits a frame to the contention mechanism, waits until it is responded to or its Fixed Delay event is signaled, retransmits if necessary, and then waits for another exponential time. The average waiting time between transmissions (gamma) together with the number of ports (P) determines the load on the network: load = P * 1/gamma (Fig. 6).

The curves presented in Fig. 7 are for the 3-port-network example shown in Fig. 4. For these curves, the transmitted frames are fixed at 4K bits. This performance conforms to the intent of NSC. At low loads, the delays are

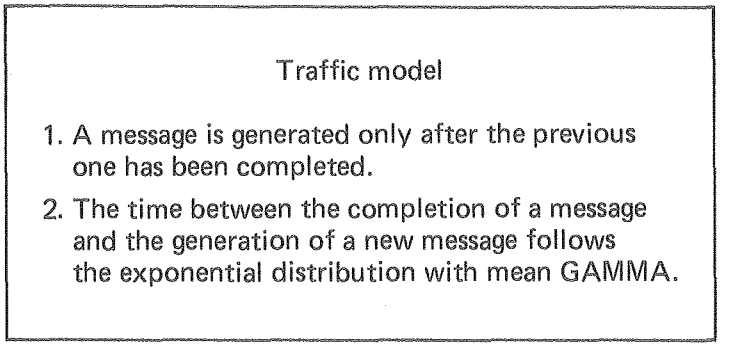


Fig. 5. Traffic model for our simulation.

GAMMA (sec)	Rate=1/GAMMA	Load=Adapters*Rate
50,000E-6	20	6E+1
10,000E-6	100	3E+2
2,000E-6	500	1.5E+3
400E-6	2,500	7.5E+3
80E-6	12,500	3.75E+4
16E-6	62,500	1,875E+5
3.2E-6	312,500	9.375E+5
0.64E-6	1,562,500	4.6875E+6

Fig. 6. Traffic load.

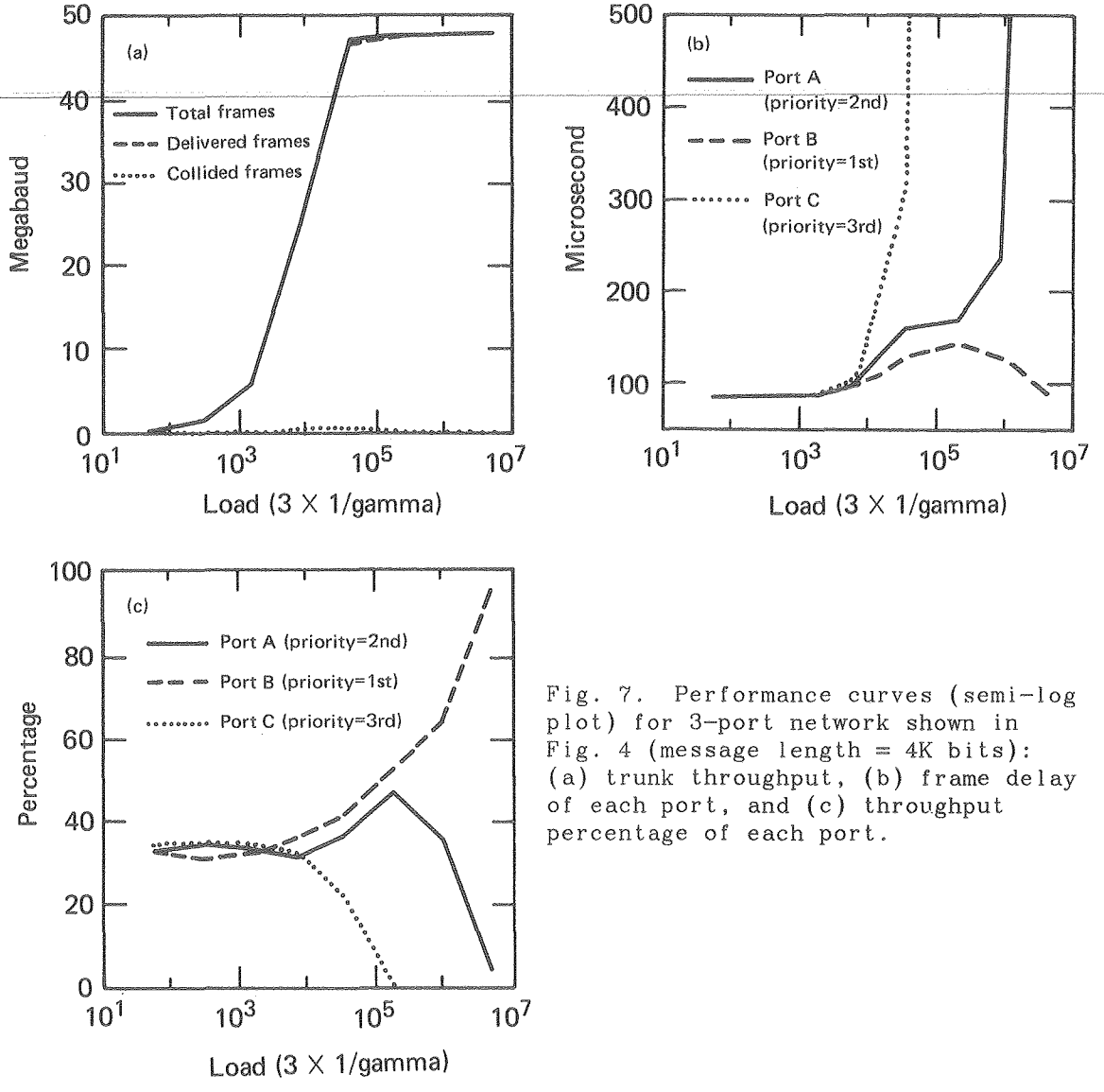


Fig. 7. Performance curves (semi-log plot) for 3-port network shown in Fig. 4 (message length = 4K bits): (a) trunk throughput, (b) frame delay of each port, and (c) throughput percentage of each port.

very nearly as small as possible (propagation delay + trunk bandwidth delay). As the load increases, occasional collisions that begin to occur increase delay and decrease throughput slightly. With high enough loads, the trunk gets into a slotted mode where the high priority ports gradually take over the trunk. As soon as the trunk becomes slotted, the total trunk throughput remains very close to the trunk bandwidth. Results similar to these have been previously derived by Franta from a simplified mathematical model of the Hyperchannel in [Franta].

LEVEL-2 PROTOCOL, ADAPTER CONTENTION

We will not describe the entire second-level protocol here. The flow charts (Figs. 8 and 9) of the transmitting and receiving portions of the protocol, with all error and contention conditions omitted, should suffice for understanding the examples of protocol interaction. These flow charts illustrate the network handling portion of the microprocessor firmware in the adapters.

There are three types of action boxes in the flow charts. If we start at the top of the transmitting flow chart in Fig. 8, all three can be seen in succession. The first, a diamond-shaped box, is a typical conditional branch. The second, a rectangular box, represents a state change internal to the adapter. The third, an arrow-shaped box, represents transmission of a frame by the adapter to another adapter with which it is conversing. Each of these frame transmissions must go through the level-1 timing mechanisms to contend for the trunk. In the current NSC adapters the microprocessors retransmit lost frames. Retransmission is further discussed in the first interaction example.

The reservation aspect of the level-2 protocol is its most important feature for understanding the interaction examples. An adapter with a message to send first reserves itself and then reserves the receiving adapter. This reservation mechanism is supported by special hardware in the NSC adapters. What the adapters do in the event of a reservation failure is discussed in the last two interaction examples in this paper.

The following notes on the major remaining features of the level-2 protocol illustrate the amount of detail included in the simulation and round out the description of the Hyperchannel protocol:

- Network messages are submitted by hosts for transmission in the form of a short *message proper*, optionally followed by an arbitrary amount of *associated data*. Both the transmitting and receiving adapters are reserved during the entire message transmission.

- The adapters signal one another by clearing bits in their partner's registers with special transmission frames devoted to this purpose. The flags are listed below:

  Wakeup - Notifies a microprocessor that it has received a message proper.

  EOM - The End-Of-Message flag is cleared by either partner in a conversation to terminate message transmission.

  RFB - The transmitter's Ready-For-Buffer flag is cleared by the receiver to indicate readiness for more data.

  EOB - The receiver's End-Of-Buffer flag is cleared by the transmitter after it has successfully transmitted a frame containing a data buffer.

0 0 1 3 5 1 0 4 0 9 9

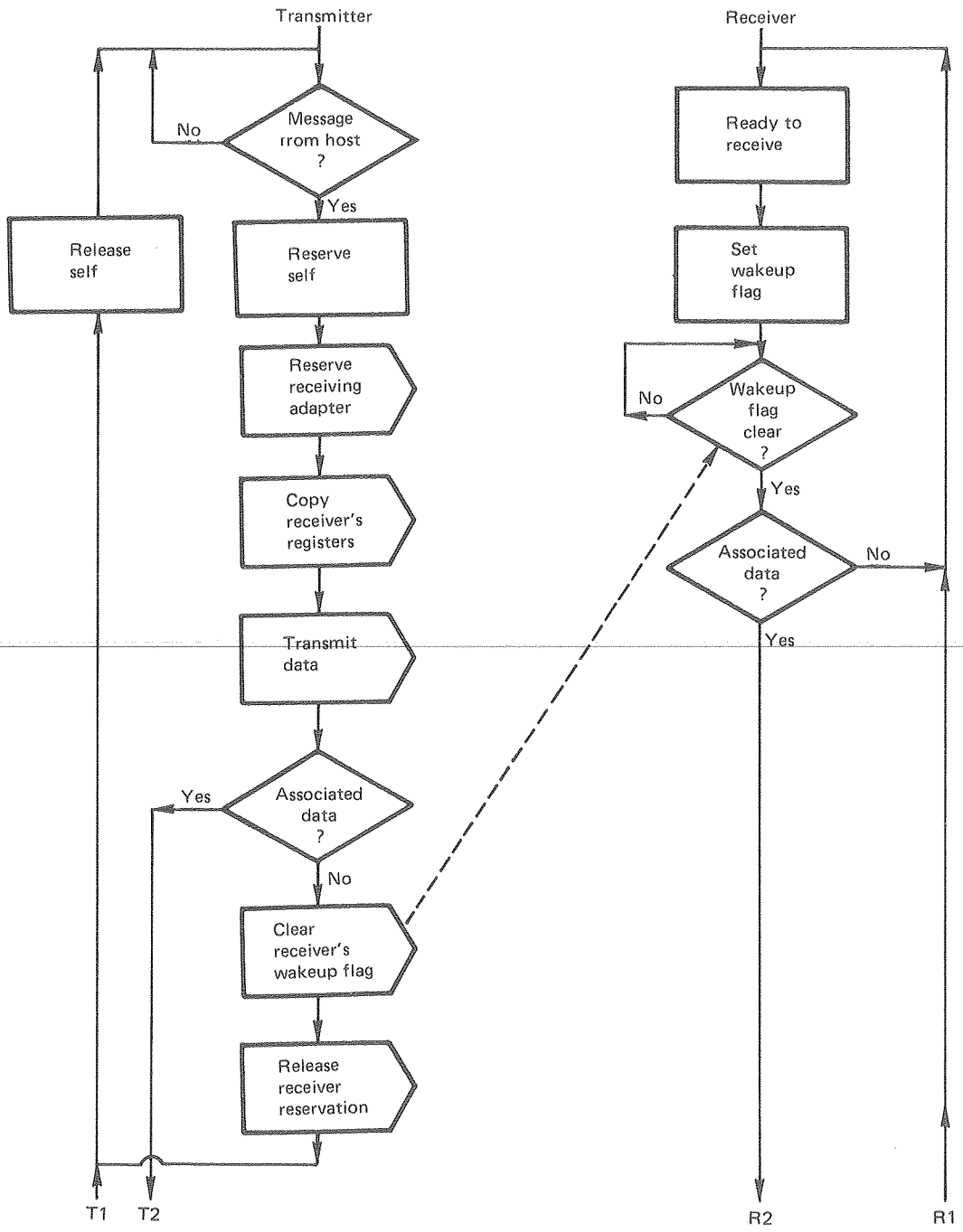


Fig. 8. Flow charts for transmitting and receiving protocol.

Before each data buffer transmission, the transmitting adapter copies the receiver's state with a special ***copy registers*** frame so that it knows how much data the receiver can accept. This state information is also used by the transmitter to restore the receiver's state in the event that a data retransmission is necessary.

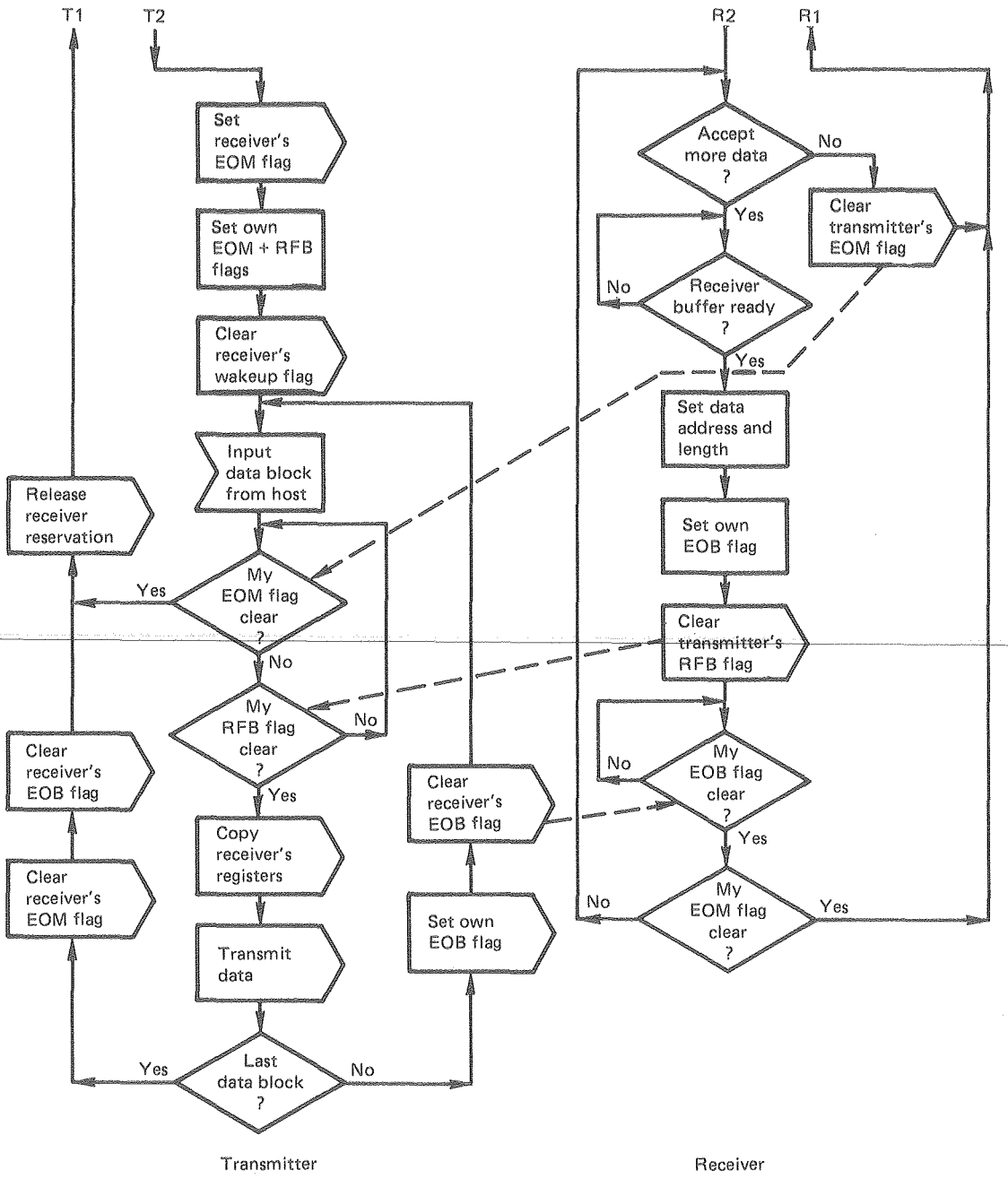


Fig. 9. Continuation of Fig. 8.

The part of the algorithm involved in the clearing of the RFB and EOB flags constitutes a flow control mechanism for associated data. The messages proper are stacked by the microprocessor and require no explicit flow control mechanism.

LEVEL-2 PROTOCOL PERFORMANCE

For a customer's host system to use a Hyperchannel network, it must submit network messages to its attached adapter. The adapter then uses both the second and first level protocols to transmit the message. Messages can be submitted to this second level of protocol in much the same way as frames were submitted to the first level protocol (Figs. 5 and 6). A natural question is: How does the performance of the Hyperchannel, as seen from a host (second level), compare with the view from an adapter (first level)?

Our host level analysis of Hyperchannel performance pointed up a number of difficulties that made level-2 performance very different from that for level-1. Part of the problem, some reservation deadlock issues, seems to be strongly tied to the reservation nature of the level-2 protocol. These problems would appear even if level-1 performance were "perfect" (i.e., limited only by the speed of light and the microprocessor memory bandwidth). Some of the level-2 performance problems that showed up in our simulations, however, involved interaction between the two protocol levels. Three examples are discussed here: retransmission failure, deadlock degradation, and priority reversal.

1. Retransmission failure

The problem. When we began validating our second level model, we found the rather surprising result shown in Fig. 10 (the plotted parameters are characterized in Fig. 11). For heavy loads, about two-thirds of the trunk frames appeared to be lost to collisions, suggesting that the trunk operated in contention mode under heavy loads. This seemed inconsistent with the results of our first level simulation. A detailed analysis revealed that this

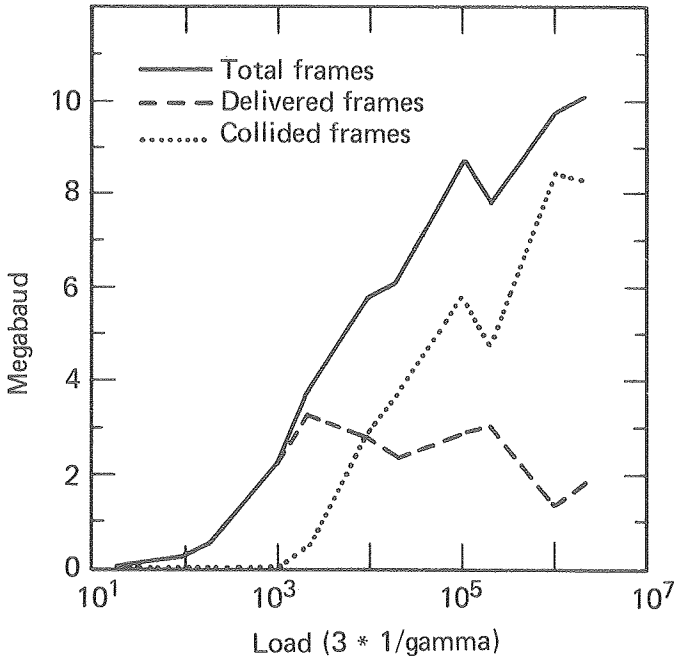


Fig. 10. Results of retransmission failure (semi-log plot). Trunk throughput with message length = 100 bits.

Network Throughput		
Total messages	=	$\frac{\text{total message bits}}{\text{total simulation time}}$
Accepted messages	=	$\frac{\text{total bits of the accepted messages}}{\text{total simulation time}}$
Rejected messages	=	$\frac{\text{total bits of the rejected messages}}{\text{total simulation time}}$
Trunk Throughput		
Total messages	=	$\frac{\text{total frame bits}}{\text{total simulation time}}$
Delivered frames	=	$\frac{\text{total bits of the delivered frames}}{\text{total simulation time}}$
Collided frames	=	$\frac{\text{total bits of the collided frames}}{\text{total simulation time}}$
Message Delay		
Effective delay	=	(time of a message successfully delivered) - (previous time of a message successfully delivered)
Processing time	=	time spent for processing a successfully delivered message
Waiting time	=	time spent waiting for the adapter reservation

Fig. 11. Characterization of plotted parameters of Figs. 10 to 20.

difficulty was due to the inability of the microprocessor to initiate retransmission of lost frames before it received its N-Delay signal. As illustrated in Fig. 12, by the time a port is ready to retransmit a lost frame, its N-Delay event has already been signaled. Unless another frame passes by, the Total Delay of the port will then run out, and the port will drop into contention mode. Fig. 12 also shows that if the retransmission intervals of two colliding frames are equal, subsequent retransmissions will continue to collide until a third frame is transmitted during the retransmission interval.

A solution. One NSC solution to this problem is to introduce new hardware that will retransmit an unacknowleged frame automatically without intervention of the microprocessor. For the existing hardware, NSC provides in the second level protocol a firmware solution called *Pseudo N-Delay*. As illustrated in Fig. 13, when a microprocessor gets a Fixed Delay time-out signal, indicating it won't receive confirmation for its last frame transmission, then instead of retransmitting immediately, it goes into a firmware loop waiting for a *Pseudo N-Delay* to expire. Assignment of distinct Pseudo N-Delays to each adapter eliminates the problem of synchronized retransmissions.

With the introduction of Pseudo N-Delay into the level-2 model, the shape of the trunk performance curves more closely match the shape obtained from the first level protocol simulation (compare Fig. 7a with Fig. 14). The absolute throughput values in Fig. 14 are smaller because of the small, 100-bit, message sizes needed to avoid the deadlock effects described below.

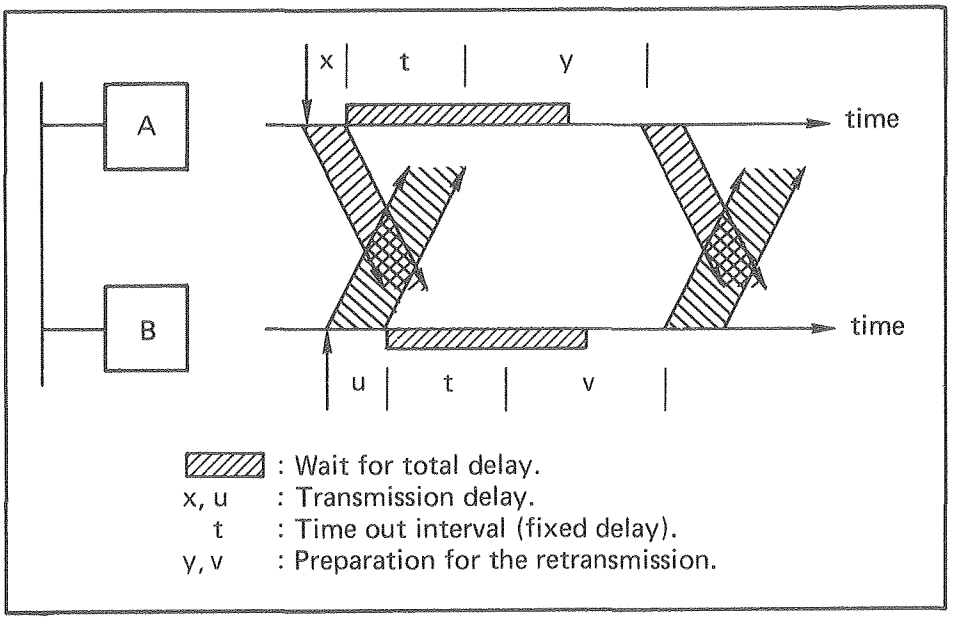


Fig. 12. Synchronized retransmission.

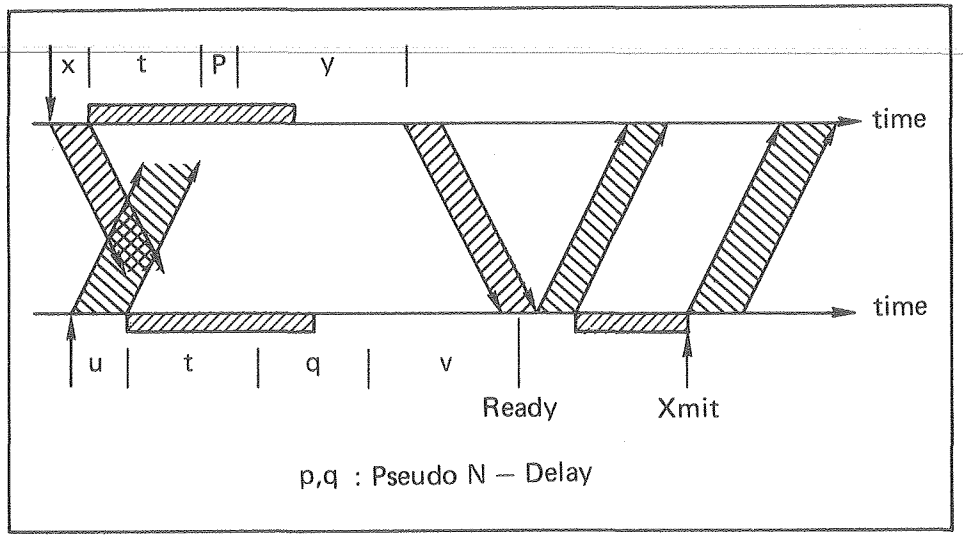


Fig. 13. Pseudo N-Delay.

Interaction. In this example, the interaction between levels involved timing characteristics of the two protocol implementations. The time for the microprocessor to retransmit was about 5 usec. If the N-Delay was less than 5 usec for a port, retransmission might fail. Speeding up the retransmission or slowing down the N-Delay made the protocols work together better.

2. Deadlock Degradation

The problem. In analyzing the level-2 protocol, we observed the existence of a kind of adapter reservation deadlock. That is, a collection of adapters could end up waiting indefinitely, each adapter holding its own reservation and waiting for another adapter to be released in order to proceed with message transmission.

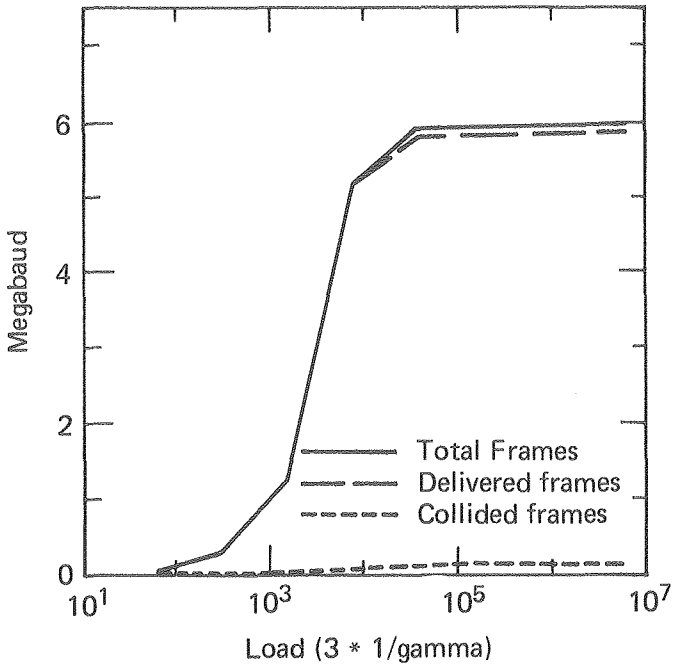


Fig. 14. Pseudo N-Delay performance (semi-log plot). Trunk throughput with message length = 100 bits.

The responses of the current second level protocol to this situation can be divided into two categories:

Two-way Deadlock. Two-way deadlock occurs when a pair of adapters try to communicate with one another at about the same time. They first reserve themselves and then try to reserve their partner. A "back off" scheme is employed by NSC to alleviate this situation. The scheme uses information contained in the reservation rejection frame that indicates to the sender which adapter the receiver is trying to reserve. If the receiving adapter is trying to reserve the sending adapter, the sender backs off and aborts its reservation attempt to allow the rejecting adapter an opportunity to complete its reservation.

Cyclical Deadlock. Cyclical deadlock involves a set of more than two adapters. Each adapter in the set reserves itself and then trys to reserve another adapter in the set. When the reservation requests form a loop (e.g., adapter A trys to reserve adapter B; adapter B trys to reserve adapter C; and adapter C trys to reserve adapter A), a cyclical deadlock exists. The current approach of NSC to the two-way deadlock problem will not correct the cyclical deadlock situation. An adapter has no way of knowing that a cyclical deadlock exists. With the current NSC firmware, each adapter will repeat the reservation request 256 times and then abort the reservation attempt. The current scheme also allows the host computer to intervene and abort the reservation attempt at any time during the retry cycle. In our simulation model, we assume that host computers take no part in solving this deadlock problem and that the adapters have sole responsibility for resolving deadlocks.

0 0 0 0 5 1 0 4 1 0 2

Our simulation of the second level protocol shows that the "back off" operation in the current adapters is often not fast enough and may cause the network to fall into what we have called a "deadlock loop" situation. This problem is illustrated in Fig. 15, where adapters A and B are trying to reserve one another. Adapter A has higher priority then adapter B. As shown in the figure, A sends a reservation request to B and then receives a reservation rejection from B. This reservation rejection tells A that B is trying to reserve A, so A backs off and aborts its reservation attempt. However, because of the slowness of the back off operation, the reservation request from B may arrive at A before A can back off. The resulting rejection frame tells B that A is trying to reserve B. Therefore, B also backs off and aborts its reservation attempt. In heavy load cases, almost every adapter has a message transmission pending for some other adapter. This implies that the adapters nearly always have themselves reserved and are trying to reserve another. In our example (Fig. 4) with three adapters and one trunk, this condition means that the network will always end up in the two-way deadlock or in the cyclical deadlock situation. In the case of two-way deadlock, because of the slowness of the back off, both adapters back off and both reservation attempts are aborted. In heavy load cases, these two adapters will get a new message to send almost immediately, and the network will get into one of the two deadlock cases again (Fig. 16). Whenever this "deadlock loop" occurs, the network throughput degrades dramatically. This performance degradation is shown in the three representative graphs in Fig. 17. When the load increases beyond 10**5, the network throughput drops below one-tenth of its highest value (Fig. 17a), and the message delay increases to more than ten times its minimum delay (Fig. 17c).

A solution. To solve the deadlock problems, Shoshani suggested a simple ordering scheme for deadlock prevention [Shoshani]. A detailed description of this scheme can be found in [Coffman71] and [Coffman73]. In brief, this scheme imposes a linear ordering on all adapters, so that adapters with higher

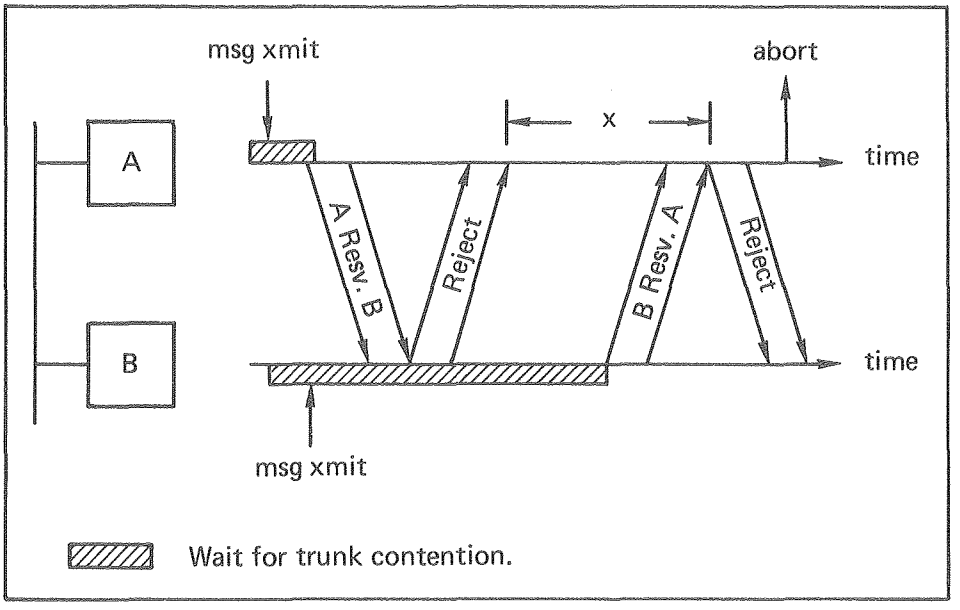


Fig. 15. Two-way deadlock.

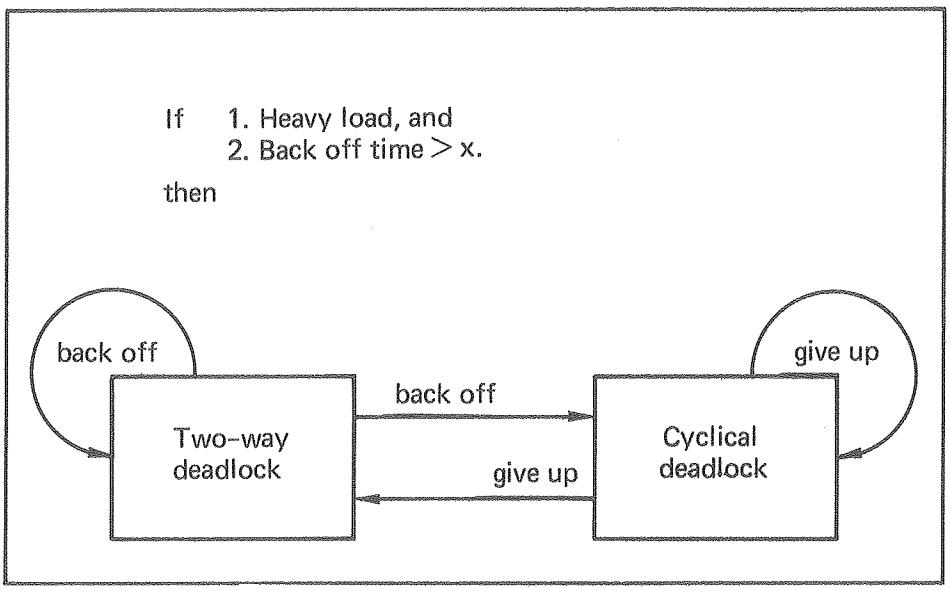


Fig. 16. Deadlock loop.

"order" are always reserved first, regardless of whether they are the sending or receiving adapters. Thus, if the sending adapter has higher order than the receiving adapter, it first reserves itself before attempting to reserve the other adapter. On the other hand, if the sending adapter has lower order than the receiving adapter, it will have to reserve the receiving adapter before trying to reserve itself. The performance improvements resulting from this deadlock prevention scheme can easily be identified by comparing Figs. 17 and 18.

Interaction. This deadlock problem is largely a result of the reservation nature of the second level protocol. Even with implementation of the suggested ordering scheme, reservation rejections will still occur. The handling of these rejections by the host systems places yet a third level of protocol in an important role. In the case of the two way backoff scheme, we again see the relative speeds of the protocol implementations causing throughput degradation.

3. Priority reversal

Among the most interesting performance anomalies that we observed in the protocol interactions was the priority reversal depicted in Fig. 19. Note that the current level-2 protocol defeats the intent of the adapter priorities by effectively allocating trunk usage to the low priority adapters under heavy load conditions. The "back off" scheme allows a low priority adapter to reserve a high priority adapter (the high priority adapter will back off first, because its reservation request and corresponding reservation rejection get through first), so the low priority adapter is given the trunk first. The "retry" scheme also favors low priority adapters, since the high priority adapters give up first.

A solution. One easy way to resolve this reversal problem is to simply reverse the meaning of the priorities. More interesting from the viewpoint of protocol interaction is to allocate priorities by using the ordering scheme introduced for deadlock prevention. Figure 20 demonstrates that this ordering dominates the level-1 priorities.

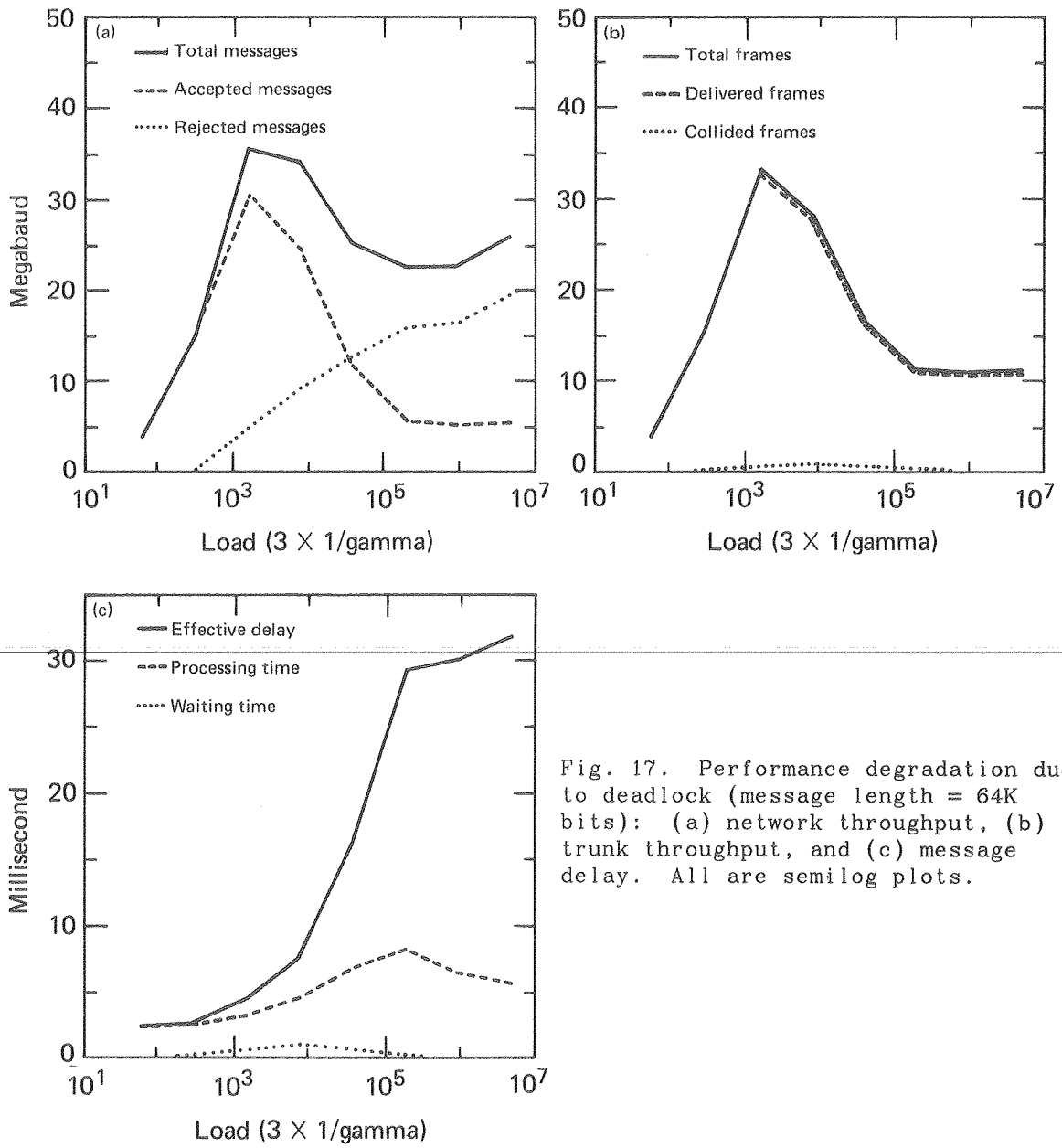


Fig. 17. Performance degradation due to deadlock (message length = 64K bits): (a) network throughput, (b) trunk throughput, and (c) message delay. All are semilog plots.

Interaction. The interaction most vividly illustrated in this example is the way in which "back off" and fixed "retry" schemes, if used in a relatively high level protocol, can reverse favored status. This fact and the dominance of *ordering* over *priority* seems to suggest a general principle of dominance by higher level protocols.

CONCLUSIONS AND DISCUSSIONS

The NSC Hyperchannel is pushing the state of the art for high-speed local networking. The timer-based protocol used for trunk contention appears well thought out and performs as intended.

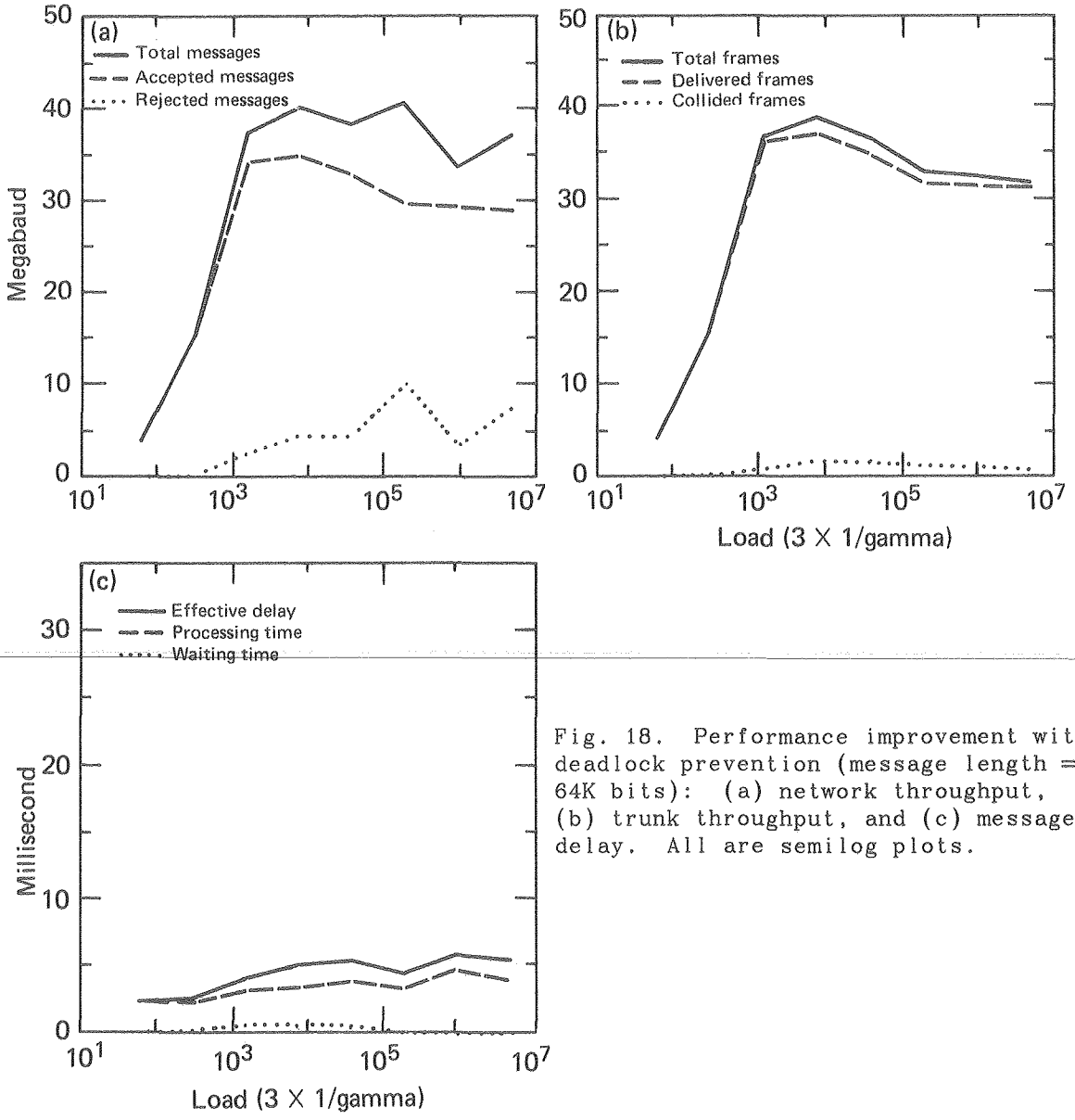


Fig. 18. Performance improvement with deadlock prevention (message length = 64K bits): (a) network throughput, (b) trunk throughput, and (c) message delay. All are semilog plots.

The adapter reservation protocol implemented in the current NSC adapters is less well understood. Though it seemed a viable second level protocol, simulation studies performed at LLL indicate that interaction between it and the trunk contention protocol can cause serious performance degradations. In some cases, the adapter reservation protocol even seems to entirely defeat the desirable characteristics of the trunk contention protocol.

Minor changes in the NSC hardware and firmware are currently being made to alleviate the difficulities posed by these protocol interactions. Changes such as those suggested here seem to solve the specific problems that we have presented. It appears, however, that the fundamental reservation nature of the current second level Hyperchannel protocol is the major source of difficulty. The possibility of introducing a nonreservation second level protocol for prioritized CSMA networks is currently under study.

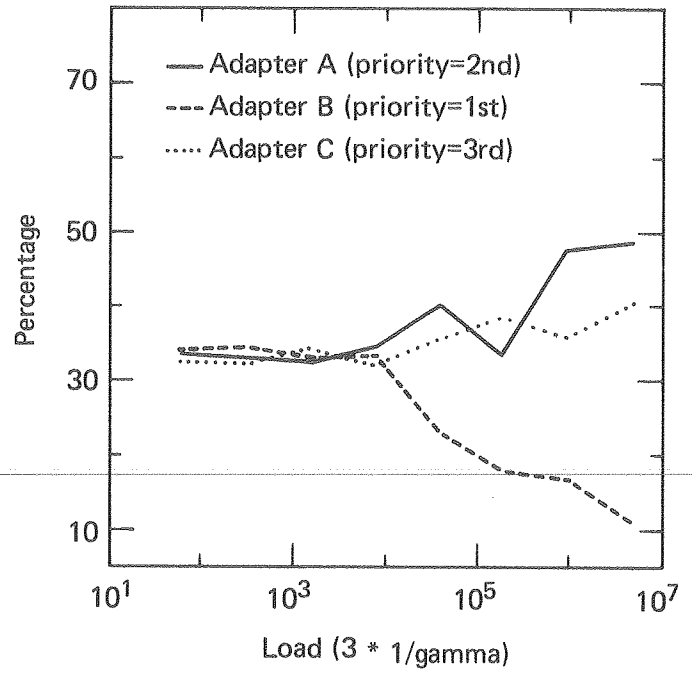


Fig. 19. Priority reversal (semi-log plot). Throughput percentage with message length = 100 bits.

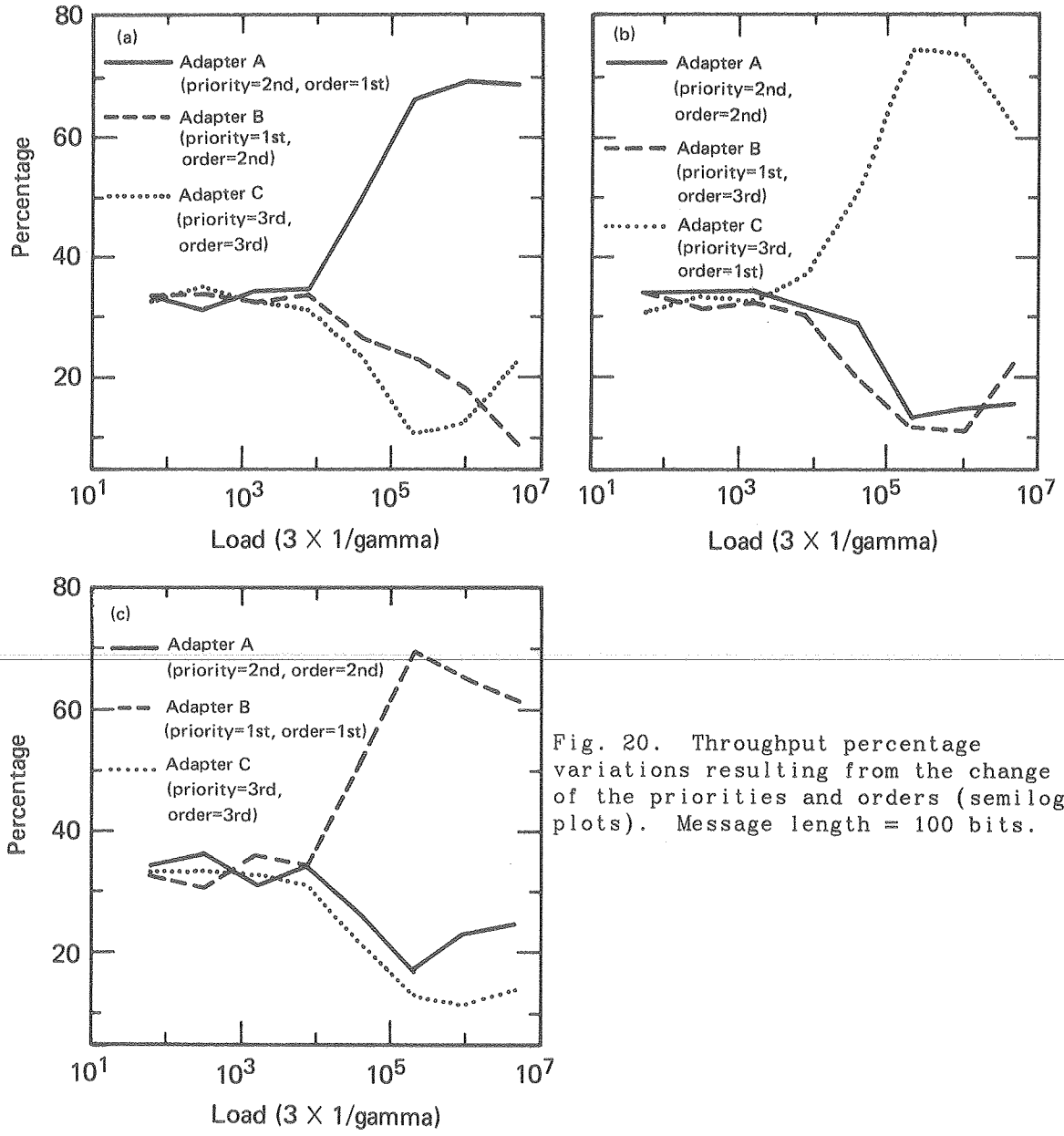


Fig. 20. Throughput percentage variations resulting from the change of the priorities and orders (semilog plots). Message length = 100 bits.

ACKNOWLEDGMENTS

The authors have been assisted by their colleagues Danny Nessett and Richard Watson of Lawrence Livermore Laboratory who contributed many helpful suggestions and criticisms. We also thank Gary Christensen and the staff of Network Systems Corporation for their assistance in understanding the detailed operation of the Hyperchannel. This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under contract No. W-7405-Eng-48.

0 0 0 3 1 0 4 1 0 5

REFERENCES

[ALOHA]
Abramson, N., *The ALOHA System - Another Alternative for Computer Communications*, 1970 FJCC, AFIPS Conf. Proc., Vol. 37, Montvale, N.J., AFIPS Press, 1970, pp. 281-285.

[Coffman71]
Coffman, E.G., Jr., Elphich, M.J., and Shoshani, A., *System Deadlocks*, Computing Surveys, 2(3), 67-78 (1971).

[Coffman73]
Coffman, E.G., and Denning, P.J., *Operating System Theory*, (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973), pp. 44-53.

[Ethernet]
Metcalfe, R.M., and Boggs, D.R., *Ethernet: Distributed Packet Switching for Local Computer Networks*, Comm. of the ACM, 19(7), 395-404 (1976).

[Franta]
Franta, W.R., and Bilodean, M.B., *Analysis of a Prioritized CSMA Protocol based on Staggered Delays*, TR-77-18, Dept. of Computer Science, Univ. of Minnesota, Sept. 1977.

[Hyper1]
Thorton, J.E., Christensen, G.S., and Jones, P.D., *A New Approach to Network Storage Management*, Computer Design, 14(11), 81-85 (1975).

[Hyper2]
Christensen, G.B., *Hyperchannel Data Trunk Contention*, Proc. of the Minneapolis Conference on Local Networks, Univ. of Minnesota, October, 1977.

[Shoshani]
Shoshani, A., private communication.

NOTICE

"This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or 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."

DB

THE PACIFIC NORTHWEST LABORATORY MINICOMPUTER NETWORK*

L. H. Gerhardstein
J. O. Schroeder
A. J. Boland

Battelle-Northwest
Richland, Washington 99352

Abstract

The Pacific Northwest Laboratory (PNL) of the Department of Energy (DOE) uses a multitude of minicomputers for real time data acquisition, data reduction and analysis, graphics, data base management, and time-sharing. Applications include a wide range of scientific and administrative disciplines. As in any institution where multiple computing facilities exist, there comes a time when a data communication network ceases to be a luxury and becomes a necessity. Computer systems engineers at PNL have been considering the networking problem. This paper describes the network being developed at PNL.

Battelle Memorial Institute operates the Pacific Northwest Laboratory under contract to the Department of Energy. The computer network being developed at PNL is called "BATNET," an acronym derived from "Battelle Network." BATNET employs broadcast communication over a shared communication channel (1,2) and carrier-sense-multiple-access (CSMA) protocol (3). CSMA protocol has been implemented in software. The computer interfaces perform necessary CSMA logical functions such as busy detection, collision detection, and timing for retransmission scheduling.

The present BATNET configuration has five stations, all PDP-11's. However, it can be implemented for virtually unlimited stations and on other manufacturer's mini/microcomputers. The BATNET protocol can be implemented on any minicomputer with 8 or 16 bit integer data modes. Low-level input/output handlers should be written in machine dependent assembly language or in higher level system languages when they are available. Most high level system routines will be written in system independent FORTRAN IV to maintain source program transportability.

The present plans call for multiple local networks tied together through dedicated relay stations with gateway processors for offsite data communication. BATNET uses electronic transceivers and 75 ohm

*Prepared for the U.S. Department of Energy, formerly the Energy Research and Development Administration, under Contract EY-76-C-06-1830.

0 3 : 0 5 1 0 4 1 0 5

coaxial transmission line. Future development will consider the use of fibre-optical devices, microwave, and VHF/UHF short range radio. The network will provide computer users with a powerful tool for automating the distribution of many data processing functions.

INTRODUCTION

Computer development engineers at Battelle-Northwest have been actively addressing local networking for the past two year period. The store-and-forward method was considered for the laboratory network but was abandoned in favor of broadcast data communication and CSMA protocol. Since most of the minicomputers at PNL operate in dedicated or semidedicated configurations, the advantages of broadcast message transmission prevailed in our decision.

The local minicomputer network (BATNET) is now becoming a reality at PNL. This has come about through development in three areas: broadcast data transceiver, data communication computer interface, and low-level protocol software. High level protocol and distributed applications are still in the early stages of development.

The present network employs broadcast data communication over coaxial transmission cable. The cable is approximately 2500 feet in length and passes through four of the buildings in the Battelle-Northwest complex at Richland, Washington. Eventually, several local networks will be interconnected by relay stations which will use store-and-forward message switching. The present plan calls for dedicated relay stations (mini/microprocessors) connecting the otherwise independent local networks. The present local network employs only coaxial cable; however, additional proposed links spanning distances from 3000 feet to five miles will require consideration of other methods such as fibre-optics, microwave and laser. The networking plan also includes data communication with mobile and portable computing and terminal equipment via VHF and UHF short range radio transceivers (4).

Potential applications of the network include: file transfer for source program and data exchange; sharing of resources such as data bases, specialized graphics devices, and other peripherals; and distribution of data processing. Several systems at PNL already employ some distribution of graphics, data base, and computing through dedicated point-to-point data links (5,6,7). The PNL network will greatly add to the capability of these minicomputer systems.

COAXIAL TRANSMISSION LINE AND TRANSCEIVERS

BATNET uses 75 ohm RG11/U coaxial transmission line with low loss foam polyethylene dielectric and single braided copper shield. Standard CATV pressure tap blocks and inserts (stingers) are connected to the coax at each transceiver site, providing a simple low cost connection. The inserts have been modified by replacing the internal isolation decoupling circuit with a wire, thus directly connecting the coax center conductor to the transceiver.

The transceivers perform three functions: interface to the coaxial transmission line, detection of collisions (simultaneous transmission by

0 0 0 3 5 1 0 4 1 0 7

two or more stations) and line busy, and interface to the computer. Coaxial line drivers and receivers (National 8T13/8T14) connect directly to the coax. The output of the receiver chip and input to the transmitter chip are passed to the collision detector which compares the two signals. Whenever these two signals differ, a collision is registered by the transceiver. A collision will always be detected when the transceiver is receiving a message from another station. Differential line drivers and receivers are used to connect the data input/output and collision status lines to the computer.

The station's computer supplies power for the transceiver through an isolating DC-to-DC converter. This scheme allows the transceiver circuit at each station to be directly coupled to the coax without introducing ground currents in the coaxial cable shield. The shield is then grounded at one end of the cable.

The transceivers and coaxial transmission line are capable of reliable data transmission at 3M baud data rates over approximately 3000 feet of cable. Total cost of a transceiver and associated hardware is under $100.

COMPUTER INTERFACE

The BATNET computer interfaces perform parallel-to-serial and serial-to-parallel bit conversions which allow the BATNET computers to transmit to and receive from the coax transceivers. The interfaces employ direct memory access in order to accommodate the BATNET data rates (up to 3 MHz). The interface also provides status information concerning the state of the communication media and generates delays useful in message rescheduling. A BATNET interface consists of four sections:

1. Computer bus interface - The bus interface handles all handshaking and bus protocol, DMA and interrupt requests, and other signals required to interface to a host computer's bus. This circuitry varies with individual computers.

2. Register bank - Six registers are utilized in the BATNET interfaces. The word count and bus address registers provide the DMA logic with the necessary information to control length and address of messages. A delay register provides an interrupt time delay for software protocol routines as they reschedule message transmissions. The data register is a temporary one word storage for words passing through the interface. The clock rate register controls the serial bit transmission rate of the interface; this feature has been used in applications where the interface connects to devices other than the high speed network coaxial cable. The status register provides two-way information about transmission status (line busy, collision, transmit-halt, and checksum error) and interface status (ready, interrupt enable, transmit mode, receive mode, and bus error).

3. Transmitter section - The transmitter consists of a parallel-to-serial converter, a bi-phase level phase encoder, and a CRC-16 checksum generator. The transmitter's transmit-halt logic prevents the interface from transmitting a message if the communication channel is found busy at the time transmission is initiated by the software. The transmitters send an eight-bit synchronizing bit pattern to identify start of message.

4. Receiver section - The receiver consists of a start-bit detector, a serial-to-parallel converter, a phase detector, and a CRC-16 checksum checker.

The bi-phase level phase encoders characterize the bit-serial data transmissions of "1" and "0" bits with mid-bit transitions from high-to-lower and low-to-high respectively. Since a bit will always have a mid-bit transition, a clock signal can be extracted from the incoming bit stream and be used to decode the incoming message. Tests with BATNET hardware have shown the interfaces to be tolerant of clock frequency deviations of over 10 percent at 3M baud.

The transmitted data bit stream consists of four parts. At the start of a message, a series of eight "0" bits are transmitted for the purpose of synchronizing the receiving station interfaces. The transmitter then sends a "1" bit start-bit to trigger the receivers into receive mode. The transmitter next sends the data words indicated by the bus address and word count registers. The transmitter terminates the message by sending the 16-bit CRC checksum word. The receivers detect end-of-message by sensing when the phase encoded bit pattern no longer appears on the input data lines.

The BATNET interface has been implemented for Digital Equipment Corporation PDP-11 family computers through the use of interface boards manufactured by MDB Systems, Inc. The interface consists of two PDP-11 quad modules in SPC slots on the UNIBUS backplane. The total cost of the boards, excluding labor, is less than $1300. An interface has also been constructed for the Intel MULTIBUS microcomputer backplane, with all the features of the PDP-11 interface except the delay generator. The ability of the microcomputer to interface to BATNET has yet to be fully exploited; however, it appears that microcomputer techniques will provide a low cost method for connecting a wide variety of devices to the network.

LOW-LEVEL SOFTWARE

The low-level software has been implemented on PDP-11 computers through highly modularized source programs. These low-level source modules contain conditional assembly switches to allow the assembly of low-level software for any of the PDP-11 configurations implemented so far. The PDP-11 operating system environments conditionalized on the BATNET software so far include: IAS, RSX11-D, RSX11-M, RT-11, and stand-alone.

0 0 : 3 5 1 0 4 1 6 3

RSX11-D AND IAS LOW-LEVEL SOFTWARE

The RSX11-D real time operating system runs on PDP-11 computers with memory management hardware. RSX11-D can run on PDP-11/40/45/55/60 systems with up to 124K words memory or on PDP-11/70 systems having up to 2048K words memory. IAS is an extension of RSX11-D that has all the RSX11-D real time features plus a timesharing executive. Input-output requests in RSX11-D systems are handled by input-output device handler tasks. The RSX11-D task which performs all BATNET packet input-output is the BL device handler [(B)ATNET (L)ow-level]. In the current BATNET version, BL occupies approximately 1700 words resident memory including the resident task header.

The BL task accepts queue-input-output (QIO) requests to either write or read logical blocks. BL logical blocks contain BATNET data packets, not RSX11-D input-output data records (record input-output will be handled by the BN handler task, yet to be developed). The BL handler task queues all input-output requests, performs BATNET hardware control, services BATNET interrupts, and interfaces with the RSX11-D executive and system common subroutines.

The physical BATNET input-output buffers are not located in the BL task or in the calling program's memory allocation but rather in a sharable-global area called BNPOOL. BNPOOL contains the BATNET input-output buffer node pool, the standard BATNET queue-list-heads, and buffer pool maintenance routines. Separately, BL and BNPOOL cannot perform useful work. However, when both are resident, BL and BNPOOL become an active BATNET station which can answer control messages from remote stations, or relay data packets between local networks. These basic protocol features will function on a RSX11-D or IAS station without any control from other tasks.

RT-11 LOW-LEVEL SOFTWARE

The RT-11 operating system runs on PDP-11 computers with minimal memory and a rotating mass storage device. Two RT-11 versions exist: the single job monitor version, and the foreground/background version. BATNET low-level software has been implemented as a set of input-output subroutines which must be invoked at link time. On RT-11 systems, object module BNRT11 contains all the BATNET low-level software, input-output buffers, and the buffer queues.

LOW-LEVEL MESSAGE PROTOCOL

The BATNET message protocol specifies the rules that participating stations must follow to ensure that messages sent by one station will be processed by the intended receiving stations. The message protocol defines the format of BATNET messages (packets), the use of each header word, and the content of special header words. Since all the stations on a local network "listen" to each others messages, they must be able to sort out which received messages should be processed and which should

be discarded. The rules which define the BATNET low-level message protocol are:

1. A local network consists of stations which can communicate over a single shared channel. A station can be in one of five possible states: transmitting, switching from transmit to receive, receiving, switching from receive to transmit, or switching from receive to receive. Each station on a local network has a unique-non-zero station number for identification. Station numbers are decimal integers between 1 and 255.

2. Local networks can be interconnected by relay stations, each local network is assigned a unique-non-zero network number. Local network numbers are decimal integers between 1 and 255. Message transmission on adjacent local networks progress independently until a relay station transfers a message across the local network boundary.

3. Each station in the composite network has a unique-non-zero station address word associated with it. The station address is a combination of the local net number and the station number:

```
          STN = 256*NET + STA
where:    NET = LOCAL NET NUMBER
          STA = STATION NUMBER
          STN = NET-STA ADDRESS
```

   The NET-STA address word can be represented by NET in the high byte and STA in the low byte of PDP-11 computers:

```
          KNET = 1              ; LOCAL NET NUMBER
          KSTA = 5              ; STATION NUMBER
  STN:    .BYTE  KSTA , KNET    ; NET-STA ADDRESS WORD
```

4. Every station in the composite network has a unique station name. Station names are ASCII strings of 1 to 8 characters in length. Only alpha and numeric characters are legal station name elements. Even though station addresses might change, high-level BATNET programs, in most cases, need only know the name of the remote system with which they wish to communicate. Low-level BATNET control functions allow software to associate a station name with that station's numeric address.

5. A BATNET message contains a header of unspecified length. One header word is the message destination address (a unique 16-bit word identifying the station to which the message is routed). Another is the message origination address (the sending station's 16-bit address). Destination address words can have wild-card elements. A zero net or station number in the destination address indicates a wild-card condition. A wild-card message must be processed by any station that receives it. Origination address

0 0 0 3 1 0 4 1 0 9

cannot be wild-card. The possible combinations for destination station address are:

N-S = Specific Net - Specific Station (a specific station)
N-* = Specific Net - Wild-Card Station (all stations on a specific local net)
- = Wild-Card Net - Wild-Card Station (all stations on all local nets in a composite network)

6. The first two header words (the first two words of the message) contain a BATNET software version number and letter respectively. The version format is:

WORD 1 = BATNET Version Number (Binary)
WORD 2 = BATNET Version Letter (ASCII)

Thus, Version 1-A is represented on PDP-11 systems by:

.WORD 1 , 101

New version identifiers will be assigned as significant changes occur in the BATNET protocol. The version number will be increased by one each time that protocol modifications would make the old and new versions noncompatible. Minor changes will be reflected by increasing the version letter. In order to ensure consistency of software version in a composite network, the version number of all stations must match. Any time that a receiving station receives a message with version number not equal to the station's current version number, the message will be discarded. Whenever future modifications to the low-level protocol occur, the version identifier must remain in the first two words of the message.

7. The remainder of the message header contains a flag word, a low-level packet sequence number, an optional software checksum, originator and destination process identifiers, and local packet routing keywords.

8. The message words which follow the header comprise the message text. The content and form of message text is a function of message type and particular application. Thus, this protocol does not place any restrictions on message text other than its starting address within the packet and its maximum length.

LOW-LEVEL COMMUNICATION PROTOCOL

The BATNET data communication protocol falls in the nonpersistent CSMA (NP-CSMA) category. NP-CSMA was chosen for the initial BATNET implementation because of its simplicity. At some time in the future, other CSMA protocols will likely be tested. The BATNET protocol allows

for multiple local networks interconnected by relay stations which perform message switching between the otherwise independent local networks. The basic protocol rules which govern all stations on BATNET are:

1. The default state of any active station is receive, i.e. when an active BATNET station has nothing to do, it must be primed to receive any message which might come along. Thus, when one station transmits a message, the remaining stations on the local network all receive that message.

2. The BATNET hardware performs CRC-16 checksum error checking. When messages arrive at a receiving station with an invalid checksum, the station must ignore the message. In a similar manner, other hardware or software errors pertaining to received messages should cause discarding of the received message.

3. When a station successfully receives a message (without error), the station must either process or discard the message based on the following decision process:

   A. If the message's destination net address is equal to the receiving station's address, the station must process the message.

   B. If the message's destination net address is equal to the receiving station's net number and the destination station number is zero (wild-card), the receiving station must process the message.

   C. If the message's destination address is zero (wild-card), the receiving station must process the message.

   D. If the receiving station is a relay point and the message's destination net number is zero (wild-card), the station must relay the message to it's other BATNET interfaces.

   E. If the receiving station is a relay point and the message's destination net number is contained in the station's packet relay matrix, then the station must relay the message to the BATNET interface indicated by the matrix.

   F. If all of the above tests fail, then the receiving station should discard the received message.

4. A station becomes a "ready-station" whenever it has a message to be transmitted. Ready-stations must remain in the receive state until they determine that they can enter the transmit state without causing a collision with another transmitting station. The BATNET interface hardware has features that help ready-stations enter the transmit state without causing collisions with other transmitters. The line-busy flag informs software if the

communication line is currently busy or idle. The transmit-half flag informs software if the transmission just initiated was aborted because the line was found to be busy. The decision process that governs ready-stations is:

A. If a ready-station finds that the communication channel is not busy, it can attempt to enter the transmit state.

B. If a ready-station finds that the communication channel is busy, the transmission must be rescheduled by a delay interval specified by the BATNET rescheduling algorithm. During the rescheduling interval, the station remains in the receive state. When the rescheduling interval has elapsed, the station becomes a ready-station and again attempts to transmit its message.

C. If the ready-station found the channel not busy and attempted transmission but then detected transmit-halt, transmission of the message must be rescheduled as in (B) above.

D. After a transmitting station successfully sends its message, it should then reenter the receive state.

5. BATNET stations connect to the communication channel through a data-communications interface in the central processor and a transmitter-receiver unit (transceiver) connected between the computer interface and the transmission media (coax, etc.). A station can have one such BATNET interface or more. When the station has more than one BATNET interface, each interfaces a different local network. These multi-interface stations are called relay stations since they are capable of relaying messages from one local network to another by the store-and-forward method.

   Relay station protocol is identical to single interface stations as long as messages on one local network route independently of other local networks. Relay stations must observe the following additional protocol:

   A. When a relay station receives a wild-card-net message on one of its channels, it should relay the message to its other channels and also to the local station's control message logic when required.

   B. When a relay station receives a specific-net/wild-card-station message on one of its channels, it should relay the message to another channel if the destination net number is contained in the station's packet relay matrix. If the destination net number is the same as the station's local net number, then the station should also route the message to its control message logic when required.

C. When a relay station receives a specific-destination message on one of its channels it must relay the message if the destination is not the local station address and the destination-net number is contained in the station's packet relay matrix.

D. When a relay station receives a specific-destination message on one of its channels and packet relay is not indicated by the relay matrix, then the station should process the message as in (3) above.

E. When a relay station transmits a message from its transmit queue, it must check the station packet relay matrix to find the appropriate interface to use. In the case of a specific-destination message, this will be a single interface unit.

F. When a relay station transmits a message from its transmit queue and the station number is wild-card, it must perform local message processing if the net number matches the local net number or if the destination net number is wild.

G. When a relay station routes a message to more than one queue or destination, it should copy the message to a new packet buffer, mark the new packet, and place it in the appropriate queue.

6. A key to CSMA protocol efficiency is the transmission rescheduling algorithm. A study of transmission rescheduling has not yet been made. For the present, retransmission delay is being derived from evenly distributed random numbers between a T-minimum and T-maximum. BATNET low-level software contains a uniform random number generator that generates pseudo-random numbers which are pseudo-independent from one sample to the next and also pseudo-independent from station-to-station. The output of the BATNET random number routine will be used to load values in the BATNET interface delay counter. The delay register value sets a delay equal to the number of word-times. The BATNET data rate is 3 MHz and the word-rate is 187.5 KHz. Thus, a value of one in the delay register represents a delay of 5.4 microseconds.

LOW-LEVEL CONTROL FUNCTION MESSAGES

BATNET protocol provides for low-level control functions. When a station receives a message from any station which has the low-level control function flag set, it (the remote station) must immediately process the control message and then queue the appropriate reply message for transmission to the station which sent the control message (requestor station). The primary purpose for low-level control messages/replies is to allow dynamic allocation of station resources at remote stations.

At present, the following control function messages have been implemented in the BATNET low-level software/protocol:

1. Send NET-STA address, station name, and station status buffer of the remote station to the requestor station.

2. Send NET-STA address, station name, and station status buffer of the remote station to the requestor station only if the remote station's name matches the name stored in the control message.

3. Send remote station description message(s) to the requestor station.

4. Receive the standard initializing measurements buffer from the requestor station. Store the received buffer in the remote station's measurements buffer.

5. Start or stop measurements on the remote station.

6. Transmit the remote station's measurements buffer to the requestor station.

7. Transmit the remote station's packet relay matrix to the requestor station.

8. Send a message to one of the remote station's terminals.

LOCAL MESSAGES

The low-level BATNET software allows messages to be transmitted to the local station (to itself). Such messages are called "local messages." Local message headers have both destination and origination address words equal to the address of the local station. When low-level BATNET software has a message to be transmitted to itself (a local message), it bypasses the normal transmitting process and places the message buffer in the station's receive queue. If the transmitted message was a control function message, the low-level software routes the message to the control-function processor and then the reply message shows up in the station's receive queue. If the transmitted message was wild-card, then the software must transmit the message and also process it as a local message. The low-level BATNET software creates local messages by copying the transmit message to a new buffer node, marking the new node as a local message, placing the transmit message in the transmit done queue, and placing the local message (or the control reply message) in the station's receive queue. Once the local message shows up in the station's receive queue, it is just as if the message had been received from a remote station on the network. Local messages are necessary to provide system-independent allocation of network resources.

INPUT-OUTPUT BUFFERS

To provide rapid processing of messages and rapid switching between modes, BATNET input-output buffers are arranged in linked-lists. On

RSX11-D and IAS operating systems, the BNPOOL shared global library contains the linked-list-heads or queue-list-heads (QLH) and the input-output buffer pool. On RT-11 systems, the queue-list-heads and the buffer pool are contained in module BNRT11. The format of a BATNET queue-list-head is:

```
WORD 1 = Forward pointer (address of first node)
WORD 2 = Backward pointer (address of last node)
WORD 3 = Number of nodes in the linked-list
```

When a queue is empty, it points to itself. Thus the empty condition of a QLH is:

```
HEAD:    .WORD    HEAD     ; FWD POINTER
         .WORD    HEAD     : BKG POINTER
         .WORD    0        : NUMBER OF NODES
```

The buffer queues which exist in the low-level BATNET software are:

```
POOL  = Linked-list of nonactive BATNET buffers
XMITQ = Linked-list of BATNET buffers to be transmitted by
        the BATNET interrupt service routine
XDUNQ = Linked-list of BATNET buffers transmitted by the BATNET
        interrupt service routine
CTRLQ = Linked-list of low-level control function replys to be
        transmitted by the BATNET interrupt service routine
RECVQ = Linked-list of received BATNET buffers waiting to be
        processed by intermediate-level and high-level BATNET
        application programs
```

BATNET buffer nodes can be allocated to processes other than queues. Examples of buffer nodes not contained in linked-lists are:

```
RECVB = Buffer node currently allocated as the receive buffer
XMITB = Buffer node currently allocated as the transmit buffer
```

ERROR HANDLING AND OTHER PROBLEMS

In broadcast networks, the problem of handling errors in received messages is somewhat different than at the store-and-forward nodes of a distributed packet switching network. Since the origination of erroneous messages can never be determined exactly, low-level negative acknowledgement should be avoided. In the case where a transmitted message fails to be received correctly at the receiving station, the sending station must timeout after some predetermined time period and then transmit the message again. By retransmitting the message, the sending stations assume that it's previous message was collided, was received erroneously at the destination station, the receiving station's transmit queue is congested, or that the receiving station has become inactive. Typically, the sender station must retransmit the message some specified maximum number of times and then abort.

0 3 , 2 5 1 3 4 1 2

Successive retransmission of messages by sending stations can cause some problems unless the low-level broadcast protocol has been equipped to handle them. Since a station remains in the receive mode by default and since stations queue all received messages, a station's receive queue can become congested and its buffer pool may be depleted during times of high system activity and slow input-output at the receiving station.

The BATNET low-level protocols take some steps to compensate for these problems. The packet header in BATNET messages contain a sequence number which must be initialized and maintained by the sending station. Any reply messages returned by the receiving station contain the same sequence number. Sequencing allows the two stations to make the necessary corrections to compensate for multiple transmission/reception of messages and to detect messages received out of order. A system of optional low-level acknowledgements is being developed. Thus, when slow response at the receiving station might otherwise cause receive queue congestion, a sending station will be informed when its message has been received at the destination. The sender process can then terminate retransmission and simply wait for the destination process' normal reply message. Another feature being developed in the low-level BATNET protocols is a cleanup procedure of the station's receive queue. Congestion of a receive queue caused by retransmission will be eliminated through a cleanup process. Through a comparison of originator station address, originator station process number, and sequence number, the receiving station can determine if a message is already in its receive queue. Thus, the receiving station will be able to identify and ignore multiple receptions of the same message.

Through the development of the BATNET low-level software and a number of network applications, there is some evidence that undetected hardware errors can occur. Hardware CRC-16 verifies the message content between the interfaces but does not compensate for errors in transfer of the messages between computer memory and the interfaces. Therefore, low-level software checksums will be implemented as an optional feature in the low-level protocol. One or more words in the message header will be set aside for checksums and, when required, checksum calculation will be performed.

A major problem in the implementation of the low-level network has been in transporting the many source program modules from the development machine to other computers. Since protocol dictates identical performance at all stations, the protocol software was maintained separately at two test stations. Through the development of a network file transfer utility program, this problem has now been overcome. The source program modules are now maintained only at one station. Whenever the network software must be updated, the network is utilized to make partial or complete copies of the network source program modules on the participating network stations. As an example, a complete update of a station utilizing a PDP-11/35 with a RK05 cartridge disk and the RT-11 operating system can be performed from the development machine in approximately 12-15 minutes. The development machine is a PDP-11/70

with RP06 disk and the IAS V2.0 operating system. The source program storage requires 2400 blocks (1.2 million bytes) on the PDP-11/35's RK05 disk.

During the initial phases of software development, emulators and simulators were used to checkout and debug large sections of the low-level code. A network interface simulator proved very useful in the development and debugging of the interrupt service routines and the buffer pool maintenance routines. A remote station emulator helped in the development of some of the intermediate-level protocols and the low-level control messages.

REFERENCES

1. V. A. Demarines and L. W. Hill, "The Cable Bus in Data Communications," Datamation, August 1976.

2. R. M. Metcalfe and D. R. Boggs, "ETHERNET: Distributed Packet Switching for Local Computer Networks," Xerox Palo Alto Research Center Report CSL 75-7, November 1975.

3. Leonard Kleinrock, Queueing Systems Volume 2: Computer Applications, John Wiley & Sons, Inc., N.Y., pp. 393-407, 1976.

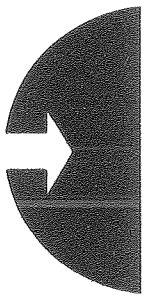
4. J. J. Thomas, et al., "Distributed Data Analysis in a Mobile Real Time and Minicomputer Network Environment," Proceedings of Third Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, August 29-31, 1978.

5. R. E. Mahan and M. D. Erickson, "Directions of Computing in Support of Laboratory Research," Proceedings of Conference on Computing Directions, Chico, CA, October 11, 1975.

6. D. R. Friedrichs, "Comprehensive Information Retrieval and Model Input Sequence (CIRMIS)," Battelle-Northwest Report BNWL-2235, April 1977.

7. D. R. Friedrichs, "Comprehensive Information Retrieval and Model Input Sequence (CIRMIS),--Users Manual," Battelle-Northwest Report BNWL-2236, April 1977.

COMMUNICATION TECHNIQUES



0 0 : 0 5 1 0 4 1 1 4

Inter-Network Fragmentation and the TCP

John F. Shoch

Xerox Palo Alto Research Center
Palo Alto, California 94304

1. Introduction

As efforts continue to inter-connect heterogeneous computer networks, we continue to struggle with the problem of determining the appropriate upper limit on the size of packets in an inter-network environment. The problem is simple: some networks are prepared to generate packets which may be larger than the maximum packet size of a subsequent network.

The general approach to a solution is obvious: we must either 1) find a way to carve up large packets into smaller fragments, or 2) legislate them out of existence; there are strong differences of opinion on this subject:

> "A universal maximum length must be adopted if datagrams are to be used for internetwork data transmission." [Bochmann and Goyer 1977]
> "...a standard minimum maximum packet size for all nets ... appears ill-advised for several reasons...." [Sunshine 1977]

The fragmentation alternatives affect the source and destination processes, and particularly impact the inter-network gateways which may lie between differing networks. Implementation of a fragmentation mechanism must reflect both the *tactical* choices available at individual gateways, and the *strategic* choices available as the system is designed.

In any case, fragmentation need not be made part of a higher level protocol, but should be incorporated into a separate inter-network level of protocol.

2. Tactics: The Fragmentation Alternatives Available to a Gateway

The critical decision about fragmenting a large packet would usually take place at an inter-network gateway: it might be necessary to forward the packet to the next gateway, through a network which could not directly handle a packet of that size. There are three possible alternatives available to the designer of a gateway process:

1. Discard the packet

The gateway could find itself completely unable to forward the packet, and merely

discard it. In our model of gateway processes it is acceptable -- but certainly not desirable -- for a gateway to take such an action. One would hope that this information might propagate back through the inter-network, so that subsequent packets would be routed away from this black hole.

2. *Intra-network fragmentation* (or *network-specific fragmentation*)

The *entrance gateway* can choose to do some sort of network-specific fragmentation, breaking up the oversize packet into smaller pieces, and allowing the *exit gateway* at the other end of this network to do the reassembly. (This might be the Arpanet host-to-host protocol, or the Packet Radio Network's SPP protocol.)

Advantages:

---This form of fragmentation is essentially performed by the network-specific driver, and is invisible to any higher level process in the gateway.
---The network can select a locally efficient protocol for transporting the fragments, and need not, for example, replicate the inter-network header on every fragment.
---If there is only one network in the path which necessitates fragmentation, all of the small pieces are brought together at the exit from that net, and the many pieces need not individually cascade through the rest of the path.
---If the destination host was within this "small packet" network, that host could also use the network-specific protocol to reassemble an over-size inter-network packet.

Disadvantages:

---This is, of course, more complicated than throwing the packet on the floor: there is some processing to be done, and the exit gateway must try to accumulate the fragments and perform reassembly.
---One could not take advantage of alternate gateways out of the intermediate network, since all of the fragments must reach the same exit gateway for reassembly.
---If there are many subsequent nets in the path which also require fragmentation, then there may be a fair amount of duplication as the same large packet is repeatedly fragmented and reassembled within each network.

It is important to note that this kind of fragmentation is always an acceptable decision by the gateway, since it remains transparent to other inter-network processes. We should already be familiar with circumstances where it might take place:

---Oversize "messages" are fragmented into smaller "packets" by the Arpanet IMPs.
---In a low-bandwidth telephone link between two gateways, it may be desirable to fragment a large data packet in order to 1) minimize the probability of an error which would corrupt the whole unit, and 2) allow

0 0 - 0 3 1 0 4 1 1 5

smaller packets (characters) to be interleaved with the fragments, and not be stuck for several seconds behind a lengthy data block.

3. Inter-network fragmentation

In this case the gateway may instead divide the packet into smaller pieces which are themselves inter-network packets, or fragments. These fragments remain addressed to the ultimate destination, and must be reassembled there. The gateway uses a suitable means for identifying the related pieces: sequence numbers, hierarchical subdivision, etc.

Advantages:

---There is no necessity for the next gateway to do reassembly.
---If an entrance gateway fragments into inter-network fragments, they need not all depart the net through the same exit gateway. (Provided, of course, that the inter-network routing process in the gateway has enough information to sensibly decide how to apportion the outgoing pieces.)

Disadvantages:

---The gateway must be able to suitably divide up the original inter-network packet into pieces that the destination -- any destination -- can properly reassemble; it cannot merely use a network-specific technique, and may need to know some intimate details about the internals of the inter-network packet.
---The inter-network header must be replicated in all of the outgoing fragments.
---The destination now has to perform the reassembly task.

3. Strategy: The Selection of a System-wide Technique

With these tactical alternatives, what kinds of strategies can one build for handling over-size packets?

1. Discard packets

The simplest strategy is to just use the simplest tactic: if one tries to route a packet out over a network for which it is too large, just drop it. Certainly not an acceptable approach.

2. Avoid small networks

If a source process can specify details of the inter-network route, and can obtain good information about the topology and size constraints in specific nets, then the source routing mechanism could merely dispatch "large" packets in such a way that they did not encounter any networks where they would not fit.

This strategy means that the gateways need only implement the simplest of the tactics: discard a packet if it wanders into the wrong gateway. But the preconditions are severe: if we are using some form of dynamic routing, we must be able to propagate the size parameter back through intervening networks to each source -- a non-trivial task. (If, on the other hand, we are limited to fixed source routing where the size limitations can also be known in advance, then this may become an attractive strategy. [Shoch 1978b])

3. Use intra-network fragmentation

Here we permit gateways to use either of the first two tactics: discard a packet, or fragment it in a network-specific fashion, but don't try to create inter-network fragments.

Such a strategy is usually accompanied with the specification of a maximum size for inter-network packets, implying that any network which wishes to join the inter-net must have a driver that is able (by some means) to forward a packet of this size to the next gateway [Zimmermann and Elie 1974].

As a corollary, a process can then be assured that any packet up to this size can be transmitted through any inter-network gateway. Furthermore, the destination process need not provide any kind of reassembly mechanism.

4. Use inter-network fragmentation

Finally, one could endorse the use of all three tactics by the gateways. Note that the gateway process would still have the choice of using some form of intra-network fragmentation, if there were some information available which made that the preferred choice.

Now, however, the destination process would be required to perform reassembly.

It is worth noting that this approach still specifies a packet size which all networks must be able to handle: an inter-network header plus (presumably) one unit of data.

4. Choosing a Strategy

We have very little empirical evidence upon which to base the choice of any particular strategy -- there is no analytical work, for example, describing the error probabilities or efficiency associated with fragmenting a packet into ten small ones which cascade through subsequent nets.

Most of our decisions have been based on instinct, some experience, and a desire to minimize the "complexity coefficient". While the inter-net fragmentation strategy may provide the greatest generality, endorsing only intra-network

fragmentation seems perfectly adequate; it also obviates the need for simple destination processes to perform reassembly.

This latter point is important: we should be able to layer our protocols in such a way that a destination process is able to specifiy if it is willing to do reassembly of inter-network packets, in return for possible benefits (simpler code).

In any case, however, the selection of a fragmentation strategy should be an inter-net issue, and not part of a higher level end-to-end protocol.

5. More on Inter-network Fragmentation

If one chooses to implement a full inter-network fragmentation scheme, the fragmentation and identification of fragments can be done in several ways:

---Have a sequence number for each byte, allowing individual fragments to be identified by the sequence number of the first data byte in that fragment [Cerf and Kahn 1974]. An "End of Segment" control bit is set on the last fragment of the original inter-network segment. (In TCP-3, this evolved into a pair of bits: "Beginning of Segment" and "End of Segment" [Cerf and Postel, 1978].)
---Fragment on the basis of some larger unit, with each fragment indicating which set of these blocks (one or more) it currently contains [McKenzie 1974] [ANSI 1977].
---Split fragments in some hierarchical manner, perhaps identifying each of two descendents with a binary encoding of their place in the family tree of this fragmented packet [LeMoli 1975].

A general inter-net strategy, perhaps using one of these techniques for identifying fragments, should still allow a resource-limited process to decline the use of the more complex inter-net fragmentation; thus, the strategy may be embellished with these capabilities:

1. We include in the inter-network header a bit which indicates if the destination process is willing to reassemble inter-network fragments.

2. When a gateway receives a packet which must then be fragmented, this bit is checked. If it is off, then the gateway is not permitted to generate inter-net fragments, and must either use a network-specific method to reach the next gateway, or discard the packet. If the bit is on, the gateway has the option of doing inter-network fragmentation.

Thus, a minimal gateway need not even implement any inter-network fragmentation scheme. Alternatively, if a gateway has no intra-net method available, but the destination refuses to reassemble inter-net fragments, then the packet can only be discarded (although it would be nice if some sort of error

message were returned to the source). (If this turned out to be a common situation, it might be a good argument for adaptive source routing.)

This strategy would even support the coresident use of different inter-network fragmentation systems: an optional field in the internet header could indicate the type of fragmentation to be done on the way to this destination (based on byte sequence numbers, larger blocks, hierarchical, etc.).

In the process of establishing a connection, a destination could indicate to the source of packets what kind of reassembly it could perform, thus providing the information needed to appropriately mark packets which will then flow to that destination. The packets exchanged while setting up the connection would have to indicate that they themselves could not be fragmented.

By splitting out the fragmentation strategy, as outlined here, it would now be possible for different higher-level protocols to share this mechanism, or decline to use it if it were deemed unnecessary.

6. Fragmentation and the TCP

The initial designs of the Internetwork Transmission Control Program (TCP) attacked this problem by integrating a full inter-network fragmentation scheme into the higher-level protocol ([Cerf and Kahn 1974], TCP version 2 [Cerf 1977], and TCP version 3 [Cerf and Postel 1978]).

That proposal was the subject of comment, and criticism, from various researchers: it depended upon TCP-specific functions (the byte sequence numbers), required the gateway process to look not only at the inter-net header, but also to have knowledge of the TCP header (FIN, SYN, etc.), did not provide a clear layering of protocols, and more. Version 3 warned that "...fragmentation...may in the future be removed from TCP's concern...."

In early 1978, further discussions of these issues took place among researchers examining TCP and inter-networking (based in part on an earlier version of this paper [Shoch 1978a]); those collective efforts have led to proposals for major change to the TCP fragmentation strategy:

---The entire fragmentation facility will be removed from the TCP, and a distinct inter-net level of protocol will be responsible for this function. Thus, end-to-end protocols other than TCP will be able to co-exist, taking advantage of the fragmentation facility provided by this "lower" level in the protocol hierarchy [Postel 1978]. Fields will be reserved in the inter-net header for use by fragmentation processes in gateways.

---The fragmentation units will not be based upon the TCP-specific byte sequence numbers, but instead will use a larger block of data as the fragmentation unit; a field in the inter-net header is provided to identify the fragment, with a "fragment offset" and a "not-end-of-segment-flag".

0 0 0 3 5 1 0 4 1 1 7

Selection of a large block size reduces the number of bits required for this field, but a large size can also lead to inefficient use of communications facilities; proposals have been put forward for both 512 bits and 64 bits per unit [Postel 1978] [Cerf 1978a and 1978b].

---A "Don't Fragment" bit is proposed for the inter-net header, indicating that no fragmentation should take place, and the destination will not perform reassembly of inter-network fragments (although, of course, network-specific fragmentation may still occur).

This revised TCP fragmentation proposal is now very similar to the scheme included with the proposal to add datagram service to the X.25 international standard [ANSI 1977] [Sevcik 1978]. That proposal uses a "more data" flag, fragmentation units of 128 bits, and identification using a "Number of First Part in the Fragment" field combined with a "Number of Parts in the Fragment" count.

These proposed changes should represent significant improvements in the TCP fragmentation strategy.

7. Acknowledgements

The material in this paper has evolved from many conversations with David Boggs, Ed Taft, Bob Metcalfe, and Yogen Dalal. A discussion with Vint Cerf several months ago led us to the notion of specifying the type of fragmentation as an option within TCP; subsequent discussions with the TCP Working Group and the Internet Planning Group provided the impetus for recording these ideas. As always, Jon Postel's very careful and thoughtful work has been invaluable in following the development of the TCP.

8. Bibliography

[ANSI 1977]
ANSI submission to CCITT, X3S37-77-115, *Proposal for the DATAGRAM Interface*, December 1977.

[Bochmann and Goyer 1977]
Gregor V. Bochmann and Pierre Goyer, *Datagrams as a public packet-switched data transmission service*, Internet Experiment Note #17, March 1977.

[Cerf 1977]
Vinton G. Cerf, *Specification of Internetwork Transmission Control Program -- TCP (Version 2)*, March 1977.

[Cerf 1978a]
Vinton G. Cerf, *A proposed new internet header format*, Internet Experiment Note no. YY, February 14, 1978.

[Cerf 1978b]
Vinton G. Cerf, *A proposal for TCP version 3.1 header format*, Internet

Experiment Note no. XX, February 14, 1978.

[Cerf and Kahn 1974]
Vinton G. Cerf and Robert E. Kahn, *A protocol for packet network intercommunication*, IEEE Trans. Communications, COM-22, 1974.

[Cerf and Postel 1978]
Vinton G. Cerf and Jonathan B. Postel, *Specification of Internetwork Transmission Control Program: TCP Version 3*, January 1978, ISI.

[LeMoli 1975]
G. LeMoli, *A Proposal for Fragmenting Packets in Internetworking*, INWG Protocol Note #21, April 1975.

[McKenzie 1974]
A. M. McKenzie, *Internetwork host-to-host protocol*, INWG General Note #74, December 1974.

[Postel 1978]
Jonathan B. Postel, *Draft Internetwork Protocol Specification, Version 2*, ISI, February 1978.

[Sevcik 1978]
Peter J. Sevcik, *Why the datagram is needed -- and how it will operate*, Data Communications, March 1978.

[Shoch 1978a]
John F. Shoch, *Inter-network Fragmentation and the TCP*, Internet Experiment Note #20, January 1978.

[Shoch 1978b]
John F. Shoch, *Inter-network Naming, Addressing, and Routing*, Internet Experiment Note #19, January 1978; revised version submitted to IEEE Computer Society COMPCON, Fall 1978.

[Sunshine 1977]
Carl A. Sunshine, *Interconnection of Computer Networks*, Computer Networks, 1977.

[Zimmermann and Elie 1974]
Hubert Zimmermann and Michele Elie, *Proposed Standard Host-Host Protocol for Heterogeneous Computer Networks: Transport Protocol*, INWG General Note #43, December 1973. (Also IRIA Project Cyclades report SCM 519.)

0 0 0 3 5 1 0 4 1 1 8

DYNAMIC MANAGEMENT OF PACKET RADIO SLOTS

Glenn Ricart — The National Institutes of Health and the University of Maryland

Ashok K. Agrawala — The University of Maryland

Abstract

Different protocols may offer better performance under different loading conditions in a packet radio network. The concept of an adaptation algorithm is presented which allows the network to shift from one transmission technique to another dynamically as the load changes in order to keep performance high. The resulting single control algorithm is a hybrid of the individual protocols which has overall characteristics superior to either of its component parts. It adjusts to load changes by making changes in slot allocations and policies to be used in those slots.

The adaptation algorithm which moves the scheme from one policy to another can be executed independently and identically as a distributed algorithm by stations in the network without the overhead of control messages.

An example slot management scheme is presented which behaves like a non-deterministic low delay scheme at low offered load but switches smoothly to a high utilization TDMA at high loading levels. This combination permits both low delay times at low loading levels and full utilization at high loading.

In another example, the dynamic nature of the adaptation algorithm produces an average delay time less than either of the two techniques being blended in the cross-over region.

INTRODUCTION

The ALOHA system[1] developed at the University of Hawaii helped prove the feasibility of using packet-switching technology in digital radio communications. Several protocols for multiple access to a single high bandwidth radio channel have been proposed.

By dividing the available time in a fixed way among stations of the communications network a Time Division Multiple Access (TDMA) scheme is created[5]. TDMA works best with loads balanced evenly across the stations. At low loading levels, TDMA offers comparatively high delay times. At high balanced loads, TDMA offers full utilization of the high speed channel.

The bursty nature of many types of communications have focused attention on contention schemes [4] for allocation of the radio channel. They are characterized by low delay times at low load levels and a maximum channel utilization less than one. This is because bandwidth is lost due to slots left empty by the contention scheme and the fact that collisions occur when multiple stations attempt simultaneous transmission.

Reservation techniques[4] are also possible in which a fraction of the bandwidth is allocated to control messages. Usually a master station is required to confirm reservations. In any case, a fraction of the total bandwith is unusable for data traffic.

The thesis of this paper is that information gained from observing the radio channel can be used to apply the correct techniques for channel management at the current dynamic loading level.

No specific control information messages or time slots are necessary and hence full utilization of the channel by data is possible if a non-contention protocol is in use at the full loading level.

SYNTHESIS

To minimize packet transmission delay at low loading levels a non-deterministic contention algorithm such as slotted non-persistent Carrier-Sense Multiple Access (CSMA)[4] is desirable. Load is measured by S, the input packet rate per packet transmission time. For low loads (S<.4) CSMA is typically better (lower delay times) than other techniques such as TDMA, Slotted ALOHA[4], polling[5], Carrier-Sense Split-Channel Reservation Multiple Access (CS SRMA)[4], and Mini-Slotted Alternating Priorities (MSAP)[3]. Depending on parameters, CSMA has a maximum usable channel capacity of about .8.

The deterministic TDMA technique does not have very good delay characteristics at low loads but will handle balanced loads up to S=1.0 since it can use all of the channel bandwidth for data.

This paper suggests a hybrid combination of two such techniques. A single control scheme behaves like a non-deterministic policy (like slotted non-persistent CSMA) at low loads and shifts smoothly to a deterministic TDMA at high loads.

At low loads the non-deterministic policy gives low delay times and at high loads the deterministic policy gives high channel utilization.

An adaptation algorithm is used to vary the parameters which control the way in which the system shifts toward one policy or the other. Each station receiving all transmissions can independently run the adaptation algorithm so that no central control station is necessary.

In this way, control information is derived from the succesful and unsuccesful transmissions of all components of the network.

Unlike reservation schemes, no overhead messages need to be transmitted. Unlike polling schemes, the round-trip transmission time is not overly important. In contrast to schemes such as MSAP and Coordinated Access Broadcast[2], the number of stations (especially inactive stations) does not have a major impact and there is little dependence on the ratio of propagation time to packet size.

SLOT AND FRAME ALLOCATION

All time on the transmission medium is divided into slots of time equal to the transmission time for a fixed-length packet. (For CSMA purposes, the slots are divided into mini-slots, each mini-slot being one propagation time. The parameter a describes the ratio of a mini-slot to a regular slot.) The slots are grouped into k equivalence classes or subchannels. Each interval of m slots (m>=k) is called a frame and contains at least one slot for each equivalence class.

Assume that a packet radio network of n stations is being considered. Each of the n stations is at any one time assigned to one of the k equivalence classes. (Further refinements are possible, such as transmissions from station i to station j are in class x but transmissions from i to 1 are in class y; these refinements will be ignored.)

All stations in a given class contend for slots belonging to that class by using a non-deterministic algorithm such as slotted non-persistent CSMA.

The adaptation algorithm varies the size of the frame, and the assignment of slots within the frame to classes of stations.

At low loads, the adaptation algorithm may put k=m=1 and all stations are in class 1. Therefore the algorithm performs as a pure non-deterministic low-delay algorithm.

At crushing loads from each station, it is possible to put k=m=n and all stations are in their own class. Therefore the algorithm performs as a pure TDMA high-utilization algorithm. (Operating CSMA in an uncontested slot results in full-time transmission as if TDMA were the protocol in use.)

At intermediate loads, k, m and station assignment can be varied to give low delay times.

The k parameter is increased when necessary to limit the number of collisions causing wasted bandwidth. The station assignment is varied to help balance the load in the subchannels. The m parameter allows smooth changeover by giving some classes more capacity than others. For

example, 3 slots may be reserved for individual TDMA-like users while 10 slots of the 13 slot frame remain available for all others in a CSMA mode.

ADAPTATION

The adaptation algorithm is responsible for changing frame sizes and frame scheduling to make a smooth transition from one policy to another based upon changing offered load while keeping delay times as small as the load and applicable techniques will allow.

If each station sees the same transmissions, collisions, and empty packets, each station can simultaneously and identically make the same decisions on the way in which k, m, and station assignment should be changed by executing an identical control algorithm. Therefore the adaptation algorithm requires no central control or specific overhead tranmissions.

SHIFTING BETWEEN POLICIES

The contention algorithms which provide low delays at low loads do so by incurring the possibility of collision with another packet. In order to avoid repeated collisions, the contention algorithms leave some slots empty. Therefore, even under heavy loads there are some empty slots as well as collisions. These empty slots and collision slots delay successful transmission (increasing delay time) as well as decreasing channel capacity which causes queueing delays.

Deterministic non-collision algorithms such as TDMA may waste slots due to unbalanced loads and also incur greater latency delays.

It is the job of the adaptation algorithm to gauge the impact of the present policy under the current load and decide whether a change toward the other policy is warranted.

The adaptation algorithm operates using only globally available information: the number and position of empty slots and/or mini-slots, the number and position of collisions, and the number and position of successful transmissions. It cannot, for example, use the current queue length because that is node-specific information which is not available to the other nodes which must execute the adaptation algorithm identically.

In the case of CSMA at high loads, we find that faltering performance is accompanied by a rising collision rate and a decreasing rate of empty mini-slots. Therefore a decision to shift out of CSMA may be made on the basis of a low average number of empty mini-slots, a high number of collisions, or a comparison between these variables.

TDMA overhead can be measured by finding empty slots. As soon as

0 0 1 3 5 1 0 4 1 2 0

one mini-slot of a TDMA slot goes empty, the remainder of the TDMA slot can be cancelled and the adaptation algorithm may wish to cancel future TDMA slots assigned to the station(s) which did not have any traffic.

AN ADAPTATION ALGORITHM

As an example of an adaptation algorithm, one is presented here which uses CSMA under low loads and TDMA under high loads.

Initially k=m=1 and all stations operate under pure CSMA. A weighted exponential average with time constant 12 slots is kept of the number of empty mini-slots. In the case at hand, the ratio of mini-slots to slots is 20:1 (or a=0.05) for a propagation distance of 10 miles using 100 bit packets (interactive traffic) at 100Kbaud modulation rate.

Under such conditions CSMA has very poor performance at loads above .7. With a retransmission delay spread out uniformly over an interval intended to provide maximum throughput the wasted mini-slots will be evenly divided between collisions and empty slots. Therefore the percentage of expected empty slots at full CSMA capacity is about .15. With this fore-knowledge, the algorithm was designed to shift into TDMA when exponentially averaged empty mini-slots fell below .2.

All stations are initially placed into the TDMA mode when the threshold is passed. If a station fails to transmit during the first mini-slot of its TDMA slot, its current and all future TDMA slots are cancelled and it is returned to CSMA operation. Others stations may continue in TDMA mode.

The algorithm always leaves at least one slot in CSMA mode and assigns all non-TDMA stations to the CSMA slot. (Therefore this particular algorithm cannot actually achieve a capacity of one because it is not intelligent enough to release the CSMA slot if all stations are in TDMA mode.)

The performance of this algorithm under simulation[i] for n=8 is shown in Figure 1. Each station has a message generator whose inter-message times are drawn from an exponential distribution. At loads below .6, the delay is identical to CSMA because the adaptation algorithm has all stations assigned to a one-slot CSMA frame. At loads above .6, TDMA starts to be used. Over the region of load above .6, the hybrid algorithm does better than either pure CSMA or pure TDMA because of its TDMA slot-cancellation policy. The performance curve in this

---

[i] The simulation model was constructed in SIMULA for the DECsystem-10. The simulations proceeded for 4,000 message times after giving the model a chance to reach equilibrium. The delay times for each message from its placement in a station's queue to successful reception were averaged to compute the delay figures reported.

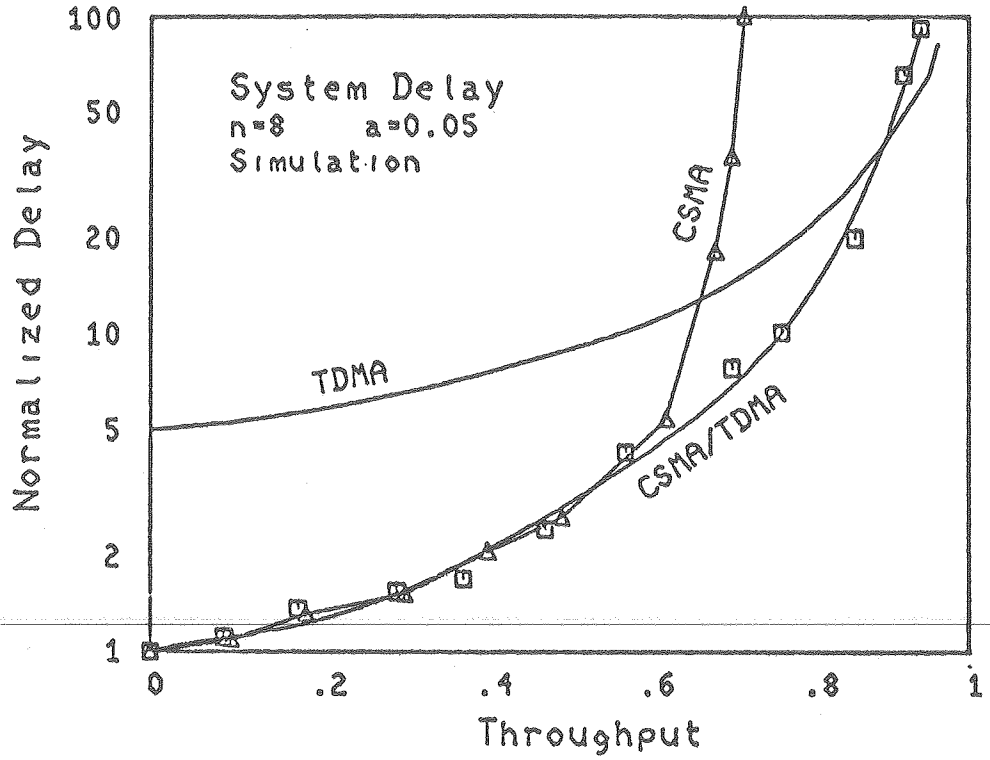


Figure 1

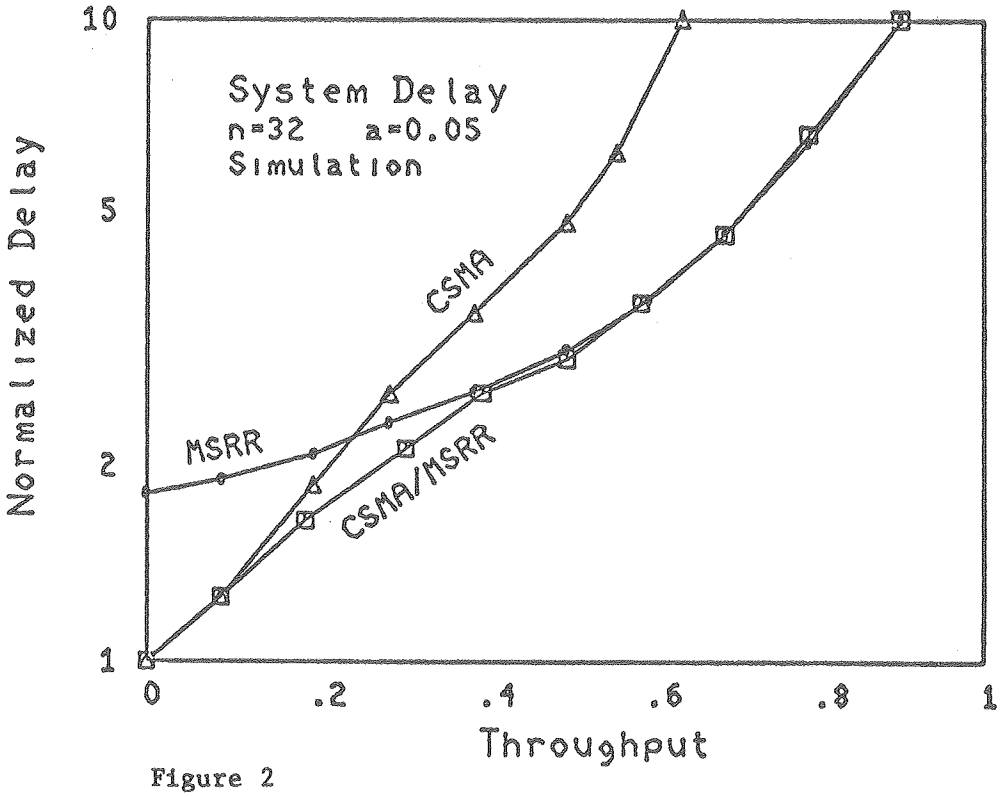


Figure 2

region approximates a TDMA curve for a lower effective number of stations than the total because not all stations have backlog. At full capacity is approached, the new algorithm rises rapidly toward the pure TDMA curve as nearly all stations have backlogs.

ANOTHER ADAPTATION ALGORITHM

In this case, CSMA is again used as the low-delay contention algorithm and MSRR (Mini-slotted Round Robin) is used as the deterministic high-load algorithm. MSRR is very similar to MSAP (Mini-slotted Alternating Priorities) except that it offers fair service to all stations regardless of past load history.

The other assumptions are the same as in the previous example. The parameters are a=0.05 and n=32. The adaptation algorithm uses the exponential average number of empty mini-slots to make a change at empty=.65 in the direction of higher load and empty=.7 in the direction of lower load.

The results are shown in Figure 2. The algorithm follows CSMA at low loads and MSRR at high loads. Figure 3 shows the proportion of time each algorithm is in effect as the load varies.

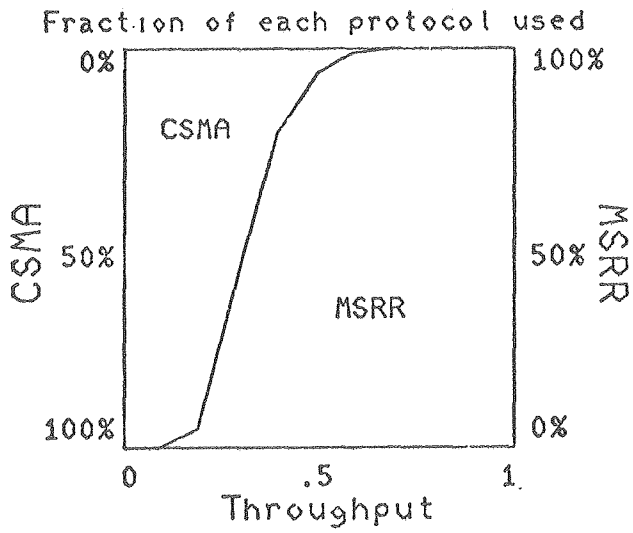


Figure 3

In the cross-over region, the average delay in the hybrid algorithm is less than either of the two component algorithms. This result is due to the fact that the Markovian load fluctuates during the simulation and the adaptation algorithm can use the low-delay algorithm during relatively light loading portions while limiting the delays under relatively high load by switching to MSRR.

By using dynamic switching between the two algorithms under the fluctuating load in the cross-over area the average delay time was reduced.

DERIVING INFORMATION

The adaptation algorithms derive information only from examination of the high speed channel yet are able to make load-dependent decisions since the load is reflected in channel use.

At low loads it is easy to get several parameters from empty mini-slots, collisions, and successful transmissions. At a channel utilization of 1.0 there is still sufficient information: the very fact that data is present in each and every mini-slot is information that the current algorithm is the correct one.

If 5 nodes have very heavy loads and the remaining 27 have no load, the adaptation algorithm would not be remiss in assigning the entire frame to 5 TDMA slots. If any of the other 27 nodes have something to send later, they can cause a collision which will alert the adapatation algorithm that adjustments are needed.

In loads that are perfectly steady, repeated, or predictable, the adaptation algorithm can be complex enough to do perfect scheduling of the channel resulting in full channel use. Empty slots and collision slots are only needed as the traffic becomes less deterministic.

COPING WITH MISTAKES

The assumption has been made that each station sees the same activities on the high speed radio channel. Ordinarily this is a pretty good assumption if all stations are within range of each other. If a station should stray too far away, its adaptation algorithm may make different allocation decisions than those of the other stations. In some contention algorithms this will not be noticed. But if some station starts transmitting in a TDMA slot not assigned to it, clearly it had different decision-making information from the rest.

At that point the adaptation algorithms can reset themselves so that they erase past knowledge which may bring the network back into synchronism. Alternatively the adaptation algorithm may assign the channel in such a way that the adaptation algorithm itself does not vary parameters; for example, slotted ALOHA could be instituted.

CONCLUSION

If the load on a high-speed packet radio channel varies, different protocols may produce better delay results over sections of the load curve. By using a distributed adaptation algorithm based upon information globally available on the channel, the network can move from one

00-0510122

technique to another to dynamically adapt to the current loading conditions. This allows a network to produce superior overall delay characteristics by combining protocols and using each over its region of best performance.

For example, it is possible to have minimum delay at low loads by using a non-deterministic contention algorithm such as slotted non-persistent CSMA. At high loads, a deterministic algorithm such as TDMA allows full channel utilization by data traffic. None of the transmission bandwidth needs to be assigned to overhead or scheduling information.

By moving dynamically between various protocols, the adaptation algorithm may produce lower average delays in the cross-over region than is possible with either of the component protocols alone.

REFERENCES

1. Abramson, N., "The ALOHA System--Another Alternative for Computer Communications," Fall Joint Computer Conference 1970, Vol. 37, pp. 281-285.

2. Agrawala, A.K, and Larsen, R.L., Efficient Communication for Local Computer Networks - Coordinated Access Broadcast, Univ. of Md. Tech. Rpt. TR-639, February 1978.

3. Kleinrock, L. and Scholl, M., "Packet Switching in Radio Channels: New Conflict-Free Multiple Access Schemes for a Small Number of Data Users", IEEE International Conference on Communication, 12-15 June 1977, Pt. II, p. 105-111.

4. Kleinrock, L., and Tobagi, Fouad, "Random Acess Techniques for Data Transmission over Packet-Switched Radio Channels", National Computer Conference, AFIPS Conf. Proc., Vol. 44, Montvale, N.J., pp. 187-201, 1975.

5. Martin, J., Teleprocessing Network Organization, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

FAILSAFE DISTRIBUTED ALGORITHMS FOR ROUTING IN COMMUNICATION NETWORKS

Adrian Segall

Dept. of Electrical Engineering
Technion - Israel Institute of Technology
Haifa, Israel.

Abstract

We present several algorithms for routing of information in communication networks. All protocols are characterized by distributed computation, loop-freedom in the routing tables for each destination in the network, adaptability to changes in flow requirements and immunity in face of arbitrary number, location and timing of topological changes. Two algorithms converge to the minimum delay routing in message and line-switched networks respectively. A third, somewhat simpler algorithm, maintains at any given time only one route between each pair of nodes in the network and converges to the shortest paths between each source and each destination in terms of arbitrary weights on the links.

1. INTRODUCTION

Reliability and the ability to recover from topological changes are properties of major importance in the design of communication networks. In today's networks it happens occasionally because of technical malfunctions or, in military networks because of war actions, that nodes and links fail and then possibly recover; also new nodes or links become operational and have to be smoothly added to an already operating network. The reliability of a communication network depends on its ability to cope with these changes, meaning that no breakdown of large portions of it will be caused by such changes and that in finite - and hopefully short - time after their occurrence, the remaining portions of the network will be able to operate normally.

The present paper gives an overview of several algorithms for routing in communication networks. In order to fix ideas, we shall call them ALGORITHM A, B and C, respectively. All algorithms have the following features and properties:

a) Distributed computation.
b) Loop-freedom for each destination at all times.
c) Adaptability to load changes.
d) Recoverability from arbitrary number, timing and location of topological changes.

This work was supported by the Advanced Research Projects Agency of the U.S. Department of Defense, under Contract N00014-75-C-1183.

0 0 : 0 5 1 0 n 1 2 3

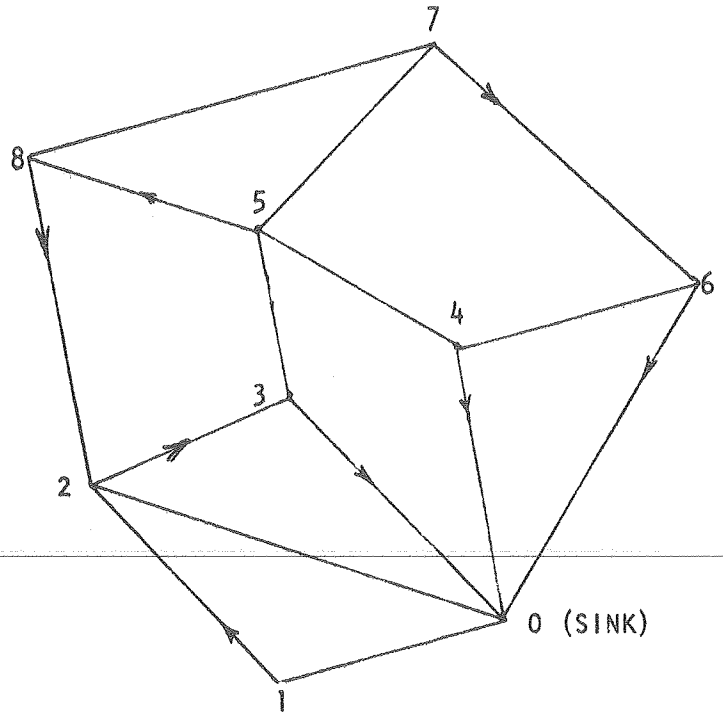
In addition, each algorithm brings the network in steady-state to some type of optimal condition:

e) If topological changes stop after a certain instant for long enough time and during this time the requirements are stationary, ALGORITHMS A and B bring the network to minimal average delay routing for message and (virtual) line-switched disciplines, respectively. If the links are assigned arbitrary weights that are time-invariant for long enough time, and during this time topological changes do not occur, then the third algorithm (ALGORITHM C) provides in steady-state the shortest route between all node pairs in the network.

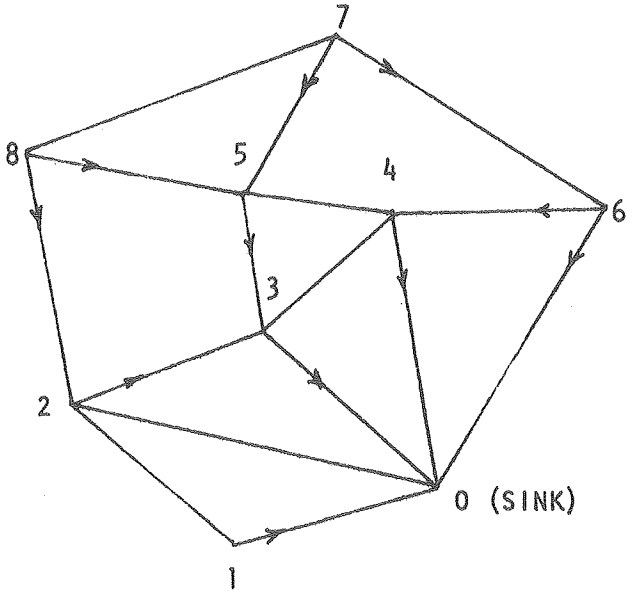
In the following sections we shall describe the algorithms and discuss in some detail their features. Algorithms A and B for networks with no topological changes were introduced in (1),(2), respectively; in the same references, proofs of their properties b), c), e) are given. The extention of these algorithms to take into account topological changes and complete proofs of recoverability appear in (3),(4). Proofs of the properties of Algorithm C are given in (5),(6). In Sections 2 and 3 we shall describe the algorithms for networks without topological changes and in Section 4 we introduce their extension to cover the possibility of such changes.

2. UPDATING AND REROUTING

Loop-freedom of routing is a desirable property in communication networks, because it saves network resources and message delays. As said in the Introduction, our algorithms indeed maintain loop-free routing tables at all times for each destination, but in our case, this property has an additional important purpose. Loop-freedom defines a per-destination partial ordering of the nodes in the network, which is used by the protocols for the propagation of updating information and of rerouting commands. To understand this propagation, it will be useful to look at a specific network example and at several possible partial orderings for a given destination (Figs. 1a,b). Also, since the loop-freedom property and the corresponding partial ordering hold on a per destination basis, and since the algorithms work independently from destination to destination, it will be useful to look from now on at the algorithm for a given destination, which we denote by SINK. Observe that in Fig. 1a each node has one and only one path to the destination, and therefore the partial ordering defines a tree, whereas in Fig. 1b nodes may have several paths to the SINK. This is the main feature distinguishing ALGORITHM C from ALGORITHMS A and B, and will be discussed in more detail later. We may mention here that ALGORITHM C, which maintains only one route to the destination, does not provide the policy for actual splitting of the traffic. For example, in a line-switched network, this route can be used to establish new connections, while the already established connections are not rerouted. On the other hand, ALGORITHMS A and B provide policies for actual routing of information, so that minimum average delay is reached in steady state. As such the latter give better performance, but



(a)



(b)

Fig. 1 - Examples of networks and of routing partial orderings.

0 0 : 0 5 1 0 4 1 2 4

ALGORITHM C is much simpler.

As a matter of definition, if node k is on a path from node i to the SINK, we say that k is downstream from i and that node i is upstream from node k. Since the routing is loop-free, no node can be both upstream and downstream from any other node (for a given destination). Also, if node k is the next node after i on a path from i to the SINK, we say that k is a son of i and i is a father of k. For example, in Fig. 1b nodes 5,6 are sons of 7, and nodes 7,8 are fathers of 5.

As said before, the partial ordering defined by the loop-free routing sustains the propagation of updating control messages and of the timing of rerouting. An update cycle is started by the SINK by sending control messages to all its neighbors. A cycle consists of, first, propagation of update control messages over the partial ordering from the SINK upstream and second, propagation of rerouting commands downstream towards the SINK. The interesting fact here is that this procedure allows the SINK to be informed of the termination of a cycle, so that no new cycle will be started before the previous one has been completed. In addition, if the link delays for transmitting control messages are finite and no topological changes occur, then each cycle will be completed in finite time (for proofs, see (2),(6)).

Next, we describe the node algorithms allowing the two waves of a cycle to propagate. As said before, a cycle is started by the SINK when it sends control messages to all its neighbors and during a cycle exactly one control message is sent over each link in each direction (for each destination).

A node can be in one of two states: normally it is in state S1, whenever it receives control messages from all its sons, it goes to state S2, and by the time it receives control messages from all neighbors the node returns to state S1. In addition to the list of its sons, each node i also maintains an estimated distance d_i to the destination where this distance is measured in terms of some link weights that are possibly time-varying, but always strictly positive. In the transition from state S1 to S2, a node, i say, takes the following actions: decide upon the new d_i using the control messages corresponding to the current cycle received up to now and send a control message of format (SINK, d_i) to all neighbors except the sons. The exact policy for calculating the new d_i is different from algorithm to algorithm and is not very important at this stage. One can see that the transition from S1 to S2 is exactly the updating part of a cycle and propagates from the SINK upstream. Next consider the second part of a cycle, namely the propagation downstream from the network towards the SINK. A node i will perform the transition from S2 back to S1 only after receiving control messages from all its neighbors. During this transition, node i takes the actions of first sending a control message (of format (SINK,d_i)) to all its sons and then rerouting, while possibly choosing new sons and cancelling old ones. Again, the exact strategy of rerouting is not important at this stage, except that one needs to be careful not to close loops when choosing new sons. Strategies to insure this appear in Section 3.

Returning to the propagation of signals in the network, one can easily see that with the above algorithm no node can perform the transition from S2 to S1 before all its fathers do. Therefore, the first to perform this transition will be nodes without fathers, and the return to S1 will propagate downstream. The last node to return to S1 will be the SINK, and this provides signal to the SINK that the cycle has been completed, namely all nodes have returned to S1. The SINK can start a new cycle at any time afterwards.

3. LOOP-FREEDOM

Observe that nodes can form loops only when choosing a new son. It is at this stage that care must be used to prevent such a loop from forming.

Consider first ALGORITHM C, where only one route is maintained from each node to the destination. One of the main theorems regarding this algorithm (see (6)) states:

Theorem 1

At all times the pattern defined by the son-father relationship is a tree rooted at SINK with the following properties:

a) if one defines S2 > S1, then the states are nondecreasing when moving downstream.
b) for all nodes in S1, the estimated distance d_i to the SINK, is strictly increasing.

The full proof of the Theorem appears in (6), but roughly speaking, it proceeds by induction as follows: Suppose the properties hold up to time t^-. Since a node, i say, can change his son only during the transition from S2 to S1, it is easy to see that it will not close a loop if we insure that the newly chosen son, k say, has distance d_k strictly less than d_i. This is because all nodes upstream from i are already in S1 (from part a) of the Theorem) and they all have distances strictly larger than d_i, which implies that k cannot be upstream from i. The policy for calculating d_i and choosing the new son assures that $d_k < d_i$. Whenever node i received a control message (SINK,d) from neighbor ℓ, it calculates $d + d_{i\ell}$, where $d_{i\ell}$ is the weight of the link (i,ℓ) and stores this in memory as $D_i(\ell)$. The estimated distance d_i is calculated at the time of transition from S1 to S2 as the minimum of all $D_i(\ell)$ over all neighbors ℓ from which i has already received control messages during the current cycle. On the other hand, the new son k is chosen by i at the time of transition from S2 to S1 as the neighbor with minimum $D_{i\ell}$, where the minimum is taken over all neighbors. With this policy, it is clear that $d_k < d_i$.

The procedure to prevent loops in ALGORITHMS A and B is somewhat more complicated. This is because these algorithms maintain more than one route to the destination, and more than that, because reaching optimum delay requires a certain step-size for the amount of rerouting allowed during each cycle. As such, it is possible in these algorithms that a node, i say, will have a son, k say, with estimated distance

0 0 0 3 5 1 0 4 1 2 5

$d_k > d_i$. In this situation loops are prevented as follows: if a node i has a son k with $d_k > d_i$ and node i is not sure that it can cancel all traffic through k during the present cycle, then node i declares itself blocked for the present cycle at the time of transition from S1 to S2. This blocking information is sent as an extra-flag together with all control messages of the present cycle. Any node hearing that a son of his is blocked, declares itself blocked too and sends an extra-flag attached to all control messages. In this way, node i and all nodes upstream from it become blocked for the present cycle. The protocol then requires that no blocked node can become a new son of any node during the present cycle. The full proof that blocking prevents loops for ALGORITHMS A and B appears in (1),(2).

4. TOPOLOGICAL CHANGES

For networks with topological changes it is necessary to use counter numbers m for the updating-rerouting cycles. It is possible that when topological changes occur, a previously started cycle cannot be completed, and if the SINK decides to start a new cycle before completion of the previous one, it will increase the cycle number m. If a cycle has been completed, then the SINK may use the same cycle counter number for the next cycle. Exact procedures to trigger the start of a new cycle by the SINK even if the previous one has not been completed (because of a topological change) will be described presently. Nodes will also have counter numbers n_i corresponding to the cycle counter number, and all control messages will now have the format (SINK,m,d) where - if node i is the sender - then $m = n_i$ and $d = d_i$. The counter number n_{SINK} is exactly the counter number of the last cycle started by the SINK. Also, a node i receiving a message (SINK,m,d) from a neighbor k, stores m in a cell $N_i(k)$ in addition to storing $d + d_{ik}$ in $D_i(k)$.

For ALGORITHM C, a failure in the network may happen on a tree-link or on links that are not on the tree (failures of nodes are considered as failure of all links adjacent to the node, so that they need no special attention). In either case, the SINK must be informed (by a distributed protocol to be presently indicated) not to wait for the completion of the present cycle, because this completion may never occur, and to start a new cycle with a higher counter number. In addition, if the failure occurs on a tree-link, all nodes upstream of the failure must be informed that their route to the destination has been disconnected and therefore not to wait for updating control messages over the tree. For example in Fig. 1a, if link (2,3) fails, then nodes 1,5,8 lose their route to the destination.

We next describe the propagation of failure information upstream and the actions taken by the nodes receiving such information. A node whose link to its son fails, enters a state S3 and sends a message (SINK,m,$d = \infty$) to all its remaining neighbors. A node receiving such a message from its son, also enters state S3 and sends a similar message to all neighbors except its son. In this way, all nodes upstream from a tree-link failure eventually enter state S3. A node i entering S3 looks at its tables $N_i(k)$ for messages with counter numbers higher

than its own node counter number, and if none are present, waits for such messages to come. In the former case, it picks the neighbor with smallest $D_i(k)$ as the new son, in the latter, it picks the neighbor from which it first receives such a message as the new son. At the time this happens, the node transits to state S2, chooses the corresponding $D_i(k)$ as its new estimated distance d_i to the SINK, sends control message (SINK,m,d) to all neighbors except the new son, and from now on proceeds as usual.

Bringing a link up is done by a similar protocol. If a link between two nodes i and k becomes operational, the two nodes compare their counter numbers via a local protocol and choose the maximum of those as the number to use for bringing the link up. Node i actually brings link (i,k) up whenever it receives a message with counter number strictly higher than this maximum, either from its best link or from k. A similar algorithm holds for k.

Finally, we describe the (distributed) protocol for informing the SINK that because of a topological change, the present cycle might not be completed, so that the SINK will know to immediately start a new cycle with a higher counter number. A node, i say, discovering a failure or a link (i,k) becoming operational sends a special control message REQ(m) to its son (if i is not in S3), where $m = n_i$ in the case of failure, or m equals $\max(n_i,n_k)$ for links becoming operational. A node ℓ receiving a message REQ(m), forwards it to its son if ℓ is not in S3 or destroys it if ℓ is in S3. Clearly, a REQ-message may encounter another failure or a node in S3, namely with no routing path to the SINK, in which case it gets destroyed. However, one of the main theorems of (6) states that if a message REQ(m) is generated, then some message REQ(m) will positively arrive at the SINK in finite time. The protocol requires that if the SINK receives REQ(m), then it will immediately start a new cycle with counter number (m+1), provided such a cycle was not started before (because of a REQ(m) message previously received). It is proved in (6) that with this protocol the network will recover in finite time, in the sense that all nodes physically connected to the SINK will have a routing path to it.

The extension of ALGORITHMS A and B to take into account topological changes is similar in principle to the one for ALGORITHM C, except that one has to separately consider situations when a node has only one or more than one son. The full procedure is described in (4).

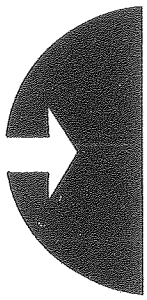
0 0 0 3 5 1 0 4 1 2 6

5. REFERENCES

1. R.G. Gallager, A minimum delay routing algorithm using distributed computation, IEEE Trans. on Comm., COM-25, 73-85 (1977.
2. A. Segall, Optimal distributed routing for line-switched data networks, submitted to IEEE Trans. on Comm.
3. M. Sidi, M.Sc. Thesis, Dept. of Electr. Engng., Technion, Haifa, Israel, Sept. 1978.
4. A. Segall and M. Sidi, Optimal failsafe distributed routing in data-communication networks, in preparation.
5. A. Segall, P.M. Merlin and R.G. Gallager, A recoverable protocol for loop-free distributed routing, Proc. of Intern. Conf. on Comm., Toronto, Canada, 1978.
6. P.M. Merlin and A. Segall, A failsafe distributed routing protocol, submitted to IEEE Trans. on Comm.

0 0 - 0 5 1 0 4 1 2 7

CONCURRENCY CONTROL I



0 0 1 0 5 1 0 4 1 2 8

A FORMAL MODEL OF CONCURRENCY CONTROL MECHANISMS FOR DATABASE SYSTEMS†

Philip A. Bernstein*

David W. Shipman

Computer Corporation of America
575 Technology Square
Cambridge, MA 02139

Abstract

In a database system, an arbitrary interleaving of transactions can lead to an inconsistent database state. This spurious behavior can be prevented by a variety of proposed synchronization mechanisms. To gain insight into these mechanisms, we analyze them in a simple centralized system which permits one read operation and one write operation per transaction. We show why locking mechanisms lead to correct operation and show that two proposed mechanisms are actually special cases of locking. We also examine conflict graph analysis, the method used in the SDD-1 distributed database system, we prove its correctness, and we show that it is not a locking mechanism.

1. INTRODUCTION

If several update requests are processed concurrently in a database system, an inconsistent database state can result. This spurious behavior can be prevented by synchronizing transactions, to avoid the arbitrary interleaving of operations. A number of concurrent update mechanisms have been proposed for synchronizing transactions, most of them built on the notion of *locking* (e.g., [Eswaran et al., 76], [Gray et al., 75]). Indeed, virtually all commercial systems use some form of locking.

---

†This work was supported in part by the National Science Foundation under grant number MCS77-05314, and in part by the Defense Advanced Research Projects Agency of the Department of Defense, monitored by the Naval Electronic Systems Command under Contract No. N00039-77-C-0074. 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 U.S. Government.

*Author's Address: Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138

Recently, several proposals have emerged for alternative synchronization mechanisms (e.g., [Rothnie et al., 77], [Stearns et al., 76], [Thomas, 75]). These mechanisms were developed primarily for distributed database systems. However, each of them could be applied to a centralized setting as well.

To gain insight into the operation of these new mechanisms and to compare them to locking and to each other, we will analyze these mechanisms in a very simple centralized system. In this system, a single transaction can execute only one read operation and one write operation. Of course, for many applications such a system would have little practical value. However, we have chosen this model more for its didactic qualities than for its practical ones. By examining the mechanisms in this simple environment, we will be able to show the essential reasons why they work.

We begin in Section 2 with an informal description of our concurrency model. A mathematical formalism for this model is presented in Section 3, with some basic theorems about concurrency correctness. Then, in the following four sections we analyze four different concurrency control mechanisms: Section 4 analyzes locking; Section 5 analyzes Thomas' algorithm [Thomas, 75]; Section 6 analyzes the Stearns, Lewis and Rosencrantz algorithm [Stearns et al., 76]; and Section 7 analyzes conflict graph analysis as used in the SDD-1 distributed database system [Rothnie et al., 77]. We conclude with a discussion of the relative merits of the methods.

2. A SIMPLE MODEL OF CONCURRENCY

We assume that the basic unit of user computation is the *transaction*. A transaction executes in three steps: first it reads a portion of the database into a local workspace; then it performs some local computation on its workspace; and finally it writes some values into the database. A subset of the possible database states are designated (through some mechanism external to our model) to be *consistent*. We assume that each transaction halts for all inputs and preserves database consistency. If a transaction executes alone on a database state that is initially consistent, then it halts within finite time and leaves the database in a consistent state.

The database system is a storage facility that can process read and write requests on behalf of transactions. Read and write operations are executed *atomically* by the database system. (Of course, the system can physically interleave the execution of two operations as long as the interleaving is not visible to the transactions.)

A set of transactions executes *serially* if each transaction executes its write operation before the next transaction executes its read operation. That is, the transactions are in no way interleaved. A serial execution of transactions preserves database consistency because

0 0 : 0 5 1 0 4 1 2 9

each individual transaction preserves database consistency. If an interleaved execution of transactions produces the same effect as a serial execution of those same transactions, then the execution is called *serializable*. Since a serial execution preserves consistency, a serializable execution also preserves consistency. We take serializability as our fundamental notion of concurrent correctness.

Serializability is an appealing correctness criterion, because it can be guaranteed without having the database system know the precise computation performed by each transaction. It need only know the portions of the database that each transaction reads and writes.

Serializability has come under some attack for being too strong a correctness criterion. Gray et al. [1975] argue that a less than serializable output from some transactions is logically acceptable and computationally cheaper than serializability. Lamport [1976] shows situations where results can be obtained that are correct in the program verification sense but are not serializable. Still, for most applications serializability is apparently adequate, given its nearly universal adoption as the correctness criterion for commercial systems. A thorough defense of serializability is a topic in itself and is beyond the scope of this paper.

3. THE FORMAL MODEL

3.1 Terminology

A database, D, is a set of data items $\{x_1,\ldots,x_m\}$. For our purposes, the granularity of the data items is unimportant. The domain of data item x_i is $dom(x_i)$, $i=1,\ldots,m$; so a database state is an element of $dom(D) = dom(x_1) \times \cdots \times dom(x_m)$.

From the database system's viewpoint, a transaction is completely defined by the subset of D that it reads and writes. Each transaction, T_i, can execute one read operation, R_i, and one write operation, W_i. We define two functions, $readset:\{T_i \mid i=1,2,\ldots\} \rightarrow 2^D$ and $writeset:\{T_i \mid i=1,2,\ldots\} \rightarrow 2^D$, which specify, for each transaction, the portion of the database read and written. Note that we do *not* require that $writeset(T_i) \subseteq readset(T_i)$ for each i. For convenience, we will denote $readset(T_i)$ by "$readset_i$" and $writeset(T_i)$ by "$writeset_i$."

The actual computation performed by a transaction can be quite complex and difficult to analyze. Therefore, it is common practice to ignore the exact nature of the computation and treat the computation as

a collection of function symbols open to arbitrary interpretations (in the program schema theoretic sense [Manna, 74]). Formally, we associate a function symbol f_i with each transaction T_i, where f_i:dom(readset$_i$) $\rightarrow$ dom(writeset$_i$). All of the results that follow will be designed to hold over all possible interpretations of the function symbols.

We will be working with sets of transactions defined on a single database. We formalize this as a *transaction system*, TAU = <D,T, readset,writeset> where D is a database, T is a set of transactions, readset:$\{T_i \mid T_i \in T\} \rightarrow 2^D$, and writeset:$\{T_i \mid T_i \in T\} \rightarrow 2^D$.

For a given transaction system, TAU = <D,T,readset,writeset>, a *log* $L = s_1, s_2, \ldots, s_n$ is a string of symbols from the alphabet $\Sigma = \{R_i, W_i \mid T_i \in T\}$ such that:

1. every symbols from Σ appears in L exactly once; and
2. for each i, R_i precedes W_i in L.

A log models the interleaved execution of a set of transactions. It corresponds to a "schedule" in [Eswaran et al., 76] and a "history" in [Papadimitriou et al., 77].

A *serial log* is a log of the form $L = R_{i_1} W_{i_1} R_{i_2} W_{i_2} \ldots R_{i_n} W_{i_n}$. A serial log represents a noninterleaved execution of transactions and, by virtue of the discussion in Section 2, preserves database consistency. A log is serializable if it is "equivalent" to a serial log. To examine serializability, we need a characterization of "equivalence of logs."

3.2 Equivalence of Logs

Intuitively, two logs are equivalent if they have the same effect on the database for all interpretations of transactions and all initial database states. (To account for I/O, we assume that each input and output operation is treated as a unique data item.) The following characterization of equivalence is from [Papadimitriou et al., 77].

Let L be a log and let R_i and W_j be symbols in L. We say that R_i *reads* x_k *from* W_j *in* L iff

1. W_j precedes R_i in L;
2. $x_k \in (\text{readset}_i \cap \text{writeset}_j)$; and
3. for all $T_p \in T$ such that $x_k \in \text{writeset}_p$, either W_p precedes W_j or W_p follows R_i in L.

Not every read and write operation in a log has an effect on the final database state produced by the log. We term those transactions in log L that *do* have an effect as *live* and define them as follows:

1. We add a special serial transaction, T_{out}, to the end of L with $\text{readset}_{out} = D$ and $\text{writeset}_{out} = \emptyset$. We define T_{out} to be live (the purpose of T_{out} is to read the final state produced by L);
2. if T_i is live, and R_i reads some x_k from W_j in L, then T_j is live; and
3. a transaction is live iff it so follows from (1)-(3).

A transaction that is not live is called *dead*. (Since virtually every transaction will record the fact that it ran by writing on an output device, few if any transactions will ever be dead.)

Two logs, L_1 and L_2, are *equivalent* iff for every consistent database state $S \in \text{dom}(D)$ and for all interpretations of the transactions, L_1 and L_2 map S into the same final database state. The following lemma characterizes log equivalence:

LEMMA E [Papdimitriou et al.,77]. *Two logs,* L_1 *and* L_2, *are equivalent if and only if they have the same set of live transactions and, for each live* T_i *and each* $x_k \in D$ *and* $T_j \in T$, R_i *reads data item* x_k *from* W_j *in* L_1 *if and only if* R_i *reads* x_k *from* W_j *in* L_2. □

3.3 Serializability

As discussed in Section 2, we are primarily interested in logs that are serializable (abbr. SR). However, not every log is SR. For example, $L = R_1R_2W_1W_2$ where $\{x_j\} = \text{readset}_1 = \text{writeset}_2$ and $\{x_k\} = \text{readset}_2 = \text{writeset}_1$ is not SR. The precise conditions under which a log is SR can be stated in terms of a simple graph model. The graph is essentially a model for the 'reads-from' relation defined in Section 3.2.

Given a log, L, on transaction system TAU = <D,T,readset,writeset> augmented by T_{out} (as defined above), we define the set of serialization graphs for L, denoted SG(L), to be the set of all node-labelled directed graphs $SG_n(L) = \langle V,E \rangle$ where

$$V = \{T_i \mid \text{transaction } T_i \text{ appears in } L\} \cup \{T_{out}\};$$

$$E = E_{\text{reads-from}} \cup E_{\text{interferes}};$$

$$E_{\text{reads-from}} = \{\langle T_i,T_j \rangle \mid T_j \text{ is live in } L \text{ and } R_j \text{ reads some } x \text{ from } W_i \text{ in } L\};$$

$$E_{\text{interferes}} = \{\text{either } \langle T_k,T_i \rangle \text{ or } \langle T_j,T_k \rangle \mid T_j \neq T_{out} \text{ and } \exists\, x \in D \text{ such that } R_j \text{ reads } x \text{ from } W_i \text{ in } L \text{ and } x \in \text{writeset}_k\} \cup \{\langle T_k,T_i \rangle \mid \exists x \in D \text{ such that } R_{out} \text{ reads } x \text{ from } W_i \text{ in } L \text{ and } x \in \text{writeset}_k\}.$$

The edges in a serialization graph reflect the notion of "happened before." The edges in $E_{interferes}$ guarantee that if R_i reads x from W_j in L, then in the serialization of L no T_k that writes into x appears in between T_i and T_j. Note that there are many serialization graphs for a given log, depending on which edges are chosen for $E_{interferes}$. When we use a subscript on SG, as in "$SG_n(L)$", we mean a particular member graph of SG(L).

THEOREM SR_1 [Papadimitriou et al., 77]. *A log L is SR iff at least one of its serialization graphs in SG(L) is acyclic.* □

Checking whether L has an acyclic serialization graph is an inherently difficult problem (see [Aho et al., 75] for a description of "NP-Completeness").

THEOREM SR_2 [Papadimitriou et al., 77]. *The question "does L have an acyclic serialization graph in SG(L)?" is NP-complete.* □

We can simplify the problem, computationally speaking, by requiring that $\text{writeset}_i \subseteq \text{readset}_i$ for all i. This is done in [Stearns et al., 76], in which their "augmented ancestor graph" (essentially a special

serialization graph, with the restriction that read-sets include write-sets) is acyclic iff the given log is serializable.

THEOREM SR_3. *If* $\text{writeset}_i \subseteq \text{readset}_i$ *for all* T_i *in* T, *then the question "does* L *have an acyclic serialization graph in* SG(L)*" can be answered in polynomial time.* □

The algorithm for Theorem SR_3 is essentially just to construct the augmented ancestor graph and to check for acyclicity.

In [Eswaran et al., 76], a binary relation (denoted "<" in their paper) is defined which is acyclic iff the given log is SR. This can be translated into a fast serializability test. However, their model differs from ours in several respects: They do not distinguish between read and write operations; an operation can only manipulate one data item; and all transactions are live. It is these restrictions that allow their serializability test to run fast. (Note that Theorem SR_2 holds even if all transactions in L are live.) As we will see in Section 4, these assumptions do not affect their results on locking, which hold for our model as well.

A transaction system TAU is said to be *safe* if all logs over TAU are SR. A concurrency control mechanism places restrictions on the logs generated by TAU so that safety can be guaranteed. The first concurrency control mechanism we will examine is locking.

4. LOCKING

The standard method of ensuring the correctness of concurrently executing transactions is locking. A transaction, T, locks those data items that it reads or writes, thereby preventing other transactions from modifying these data items while T is working on them. We wish to show formally that if all transactions in a log are locked, then the log is SR. Using a different formalism, this theorem was proved in [Eswaran et al., 76]. We restate the result here in our formalism so that we will later be able to compare locking with other approaches.

Let L be a log, let T_i be some transaction in L, and let L_{T_i} be the sublog that separates R_i from W_i (L_{T_i} is empty if T_i is serial). Partition L_{T_i} into two disjoint parts, L_{R_i} and L_{W_i}, such that L_{T_i} equals the concatenation of L_{R_i} with L_{W_i} (written, $L_{T_i} = L_{R_i} L_{W_i}$). Possibly, L_{R_i} and/or L_{W_i} can be empty. The point in the log between L_{R_i} and L_{W_i} is called a *locked-point* for T_i and is denoted lp_i. Notice that a choice of lp_i between R_i and W_i completely determines L_{R_i} and L_{W_i}. Finally, let $L^+_{R_i} = R_i L_{R_i}$ and let $L^+_{W_i} = L_{W_i} W_i$.

We define a log, L, to be *two-phase locked* if for each T_k in L there is a locked-point lp_k such that for each $j \neq i$ with T_i, T_j in L: 1. If $L^+_{R_i}$ overlaps $L^+_{W_j}$ then $\text{readset}_i \cap \text{writeset}_j = \emptyset$; 2. If $L^+_{W_i}$ overlaps $L^+_{W_j}$ then $\text{writeset}_i \cap \text{writeset}_j = \emptyset$; and 3. If $lp_{i_1} = lp_{i_2} = \ldots = lp_{i_n}$ then for each m, $1 \leq m \leq n$ either

$\text{readset}_{i_m} \cap \text{writeset}_{i_p} = \emptyset$ for all $1 \leq p \leq n$, $p \neq m$, or

$\text{writeset}_{i_m} \cap \text{readset}_{i_p} = \emptyset$ for all $1 \leq p \leq n$, $p \neq m$.

(Two sublogs *overlap* if they share some elements in common.)

The notion of two-phaseness originated in [Eswaran et al., 76], and corresponds to the fact that by the time a transaction, T_i, releases its read-locks (at the end of $L^+_{R_i}$) it must have obtained its write-locks (which it must hold during $L^+_{W_i}$). Part (3) of the definition prevents the situation where several transactions simultaneously own read-locks on each other's writesets and request write-locks on their writesets, thereby causing a deadlock. A weaker formulation of (3) may be possible.

To prove that two-phase locked logs are SR, we begin with an important lemma. Let L be a log, let L' be a sublog of L, and let $x \in D$. We say that L' *is read-free in* x if there is no read operation, R_i, in L' with $x \in \text{readset}_i$. L' *is write-free in* x if there is no write operation W_i in L' with $x \in \text{writeset}_i$.

LEMMA L_1. *If* L *is two-phase locked then for each* T_i *in* L:

1. L_{R_i} *is write-free in* readset_i;
2. L_{W_i} *is read-free in* writeset_i; *and*
3. L_{W_i} *is write-free in* writeset_i. □

0 0 0 0 5 1 3 4 1 3 2

THEOREM L_2. *If a log* L *is two-phase locked, then it is* SR.

Proof (Sketch). By Lemma L_1, each transaction in L can be serialized at its locked-point without changing L's computation. It can be shown that after a transaction is serialized at its locked-point the resulting log is still two-phase locked. So, by induction on the number of transactions in L, L can be shown to be SR. □

Theorem L_2 can be proved in exactly the same way if multiple reads and multiple writes are allowed for each transaction. In this case, there is still one locked-point for each transaction. For each R_i, define L_{R_i} to be the sublog between lp_i and R_i. Then proceed as before, quantifying the definition of two-phase locking over all R_i-W_j and W_i-W_j pairs where $i \neq j$. Notice that the locking conditions hold symmetrically, whether R_i's and W_i's are to the right or left of lp_i. The proof of serializability follows as per Theorem L_2, where each transaction can be serialized at its locked point without violating the two-phase locking condition.

The converse of Theorem L is not true in general. That is, there are SR logs in which not every transaction is two-phase locked. An example is the log $R_1R_2W_2R_3W_3W_1$ where $readset_2 = writeset_3 = \emptyset$, $readset_1 = writeset_2 = \{x_1\}$, and $readset_3 = writeset_1 = \{x_2\}$. This log is equivalent to the serial log $R_3W_3R_1W_1R_2W_2$ and is therefore SR, although it is not two-phase locked. Notice that T_2 and T_3 must appear in reverse order in the serialization. It turns out that this kind of reversal of two non-interleaved transactions need never occur when serializing a two-phase locked log.

THEOREM L_3. *If log* L *is two-phase locked, then there is a serialization of* L, L_S, *such that every pair of noninterleaved transactions appear in same order in* L_S *as in* L. □

(Transactions T_i and T_j are *noninterleaved* in L if W_j precedes R_i or W_i precedes R_j in L.)

5. THOMAS' ALGORITHM

Thomas [75] describes an algorithm for distributed concurrency control over multiple-copy databases. The following is an adaptation of his approach to the simple non-distributed model we have been using.

His concurrency control works as follows. Before performing a write operation for a transaction, the system checks that the values read by that transaction's read operation are still current (i.e., have not been over-written by another transaction). If any of the values that were read have subsequently been rendered obsolete, then the transaction is aborted; otherwise the write operation takes place.

This algorithm is similar to locking in the following sense. When a transaction performs a read operation, it is as if the transaction

had set an imaginary lock on its read-set. If this imaginary lock is ever violated, then the transaction which set the lock is invalidated and consequently aborted. This approach assumes that, in a distributed environment, it can be more efficient to set such an imaginary lock in the hope that it will not be violated than it is to set a real lock globally throughout the system.

This algorithm provides less concurrency than two-phase locking as described in the previous section. In a log produced by this algorithm, the sublog separating a read operation from its corresponding write operation is write-free in its read-set. This means that the transaction is two-phase locked, with its locked-point appearing just before its write operation in the log. So every log generated by this algorithm is two-phase locked (although the converse is not true, in general). Therefore, the serializability of these logs follows directly from Theorem L_2.

The main concern of Thomas' algorithm in a distributed system concerns the procedure for arriving at a concensus decision to validate a write operation and involves a technique called "majority voting". The algorithm also uses a time-stamping procedure to ensure consistent ordering of writes throughout the system.

6. THE STEARNS, LEWIS AND ROSENCRANTZ ALGORITHM

Stearns et al. [76] describe a number of mechanisms for concurrency control in both centralized and distributed systems. The technique described here is an adaptation of their approach to our simple model.

Before issuing its read operation, a transaction must lock its read-set and write-set against write operations from other transactions. A write operation from another transaction that attempts to violate this lock is either restarted (because of deadlock considerations) or forced to wait until the lock is no longer in effect. Similarly, sometime before executing its write operation, a transaction must lock its write-set against reading by other transactions. Again, an attempt by another transaction to read a locked data item is either restarted or forced to wait. The locks are all released when the write operation for the transaction is completed.

Let L be a log produced by this algorithm and let T_i be a transaction in L. Without loss of generality we can assume that the lock against write operations on the read-set and write-set is obtained immediately before the read operation is processed. We can represent L by a log of the form $L_x\ R_i\ L_{Ri}\ L_{Wi}\ W_i\ L_y$, where the lock against read operations on the write-set is obtained at the point between L_{Ri} and L_{Wi}. Now, the sublogs L_{Ri} and L_{Wj} have the following constraint: L_{Ri} is write-free in readset_i *and* $\textit{writeset}_i$; and L_{Wi} is read-free and write-free in writeset_i *and is write-free in* $\textit{readset}_i$. We see that this mechanism is strictly stronger than two-phase locking (cf. Lemma L_1); the constraints in italics are above and beyond those required by two-phase locking.

0 0 : 0 5 1 0 4 1 3 3

The primary thrust of [Stearns et al., 76], however, lies in the study of deadlock and perpetual restart situations and in the generalization of their algorithms to distributed databases.

7. CONFLICT GRAPH ANALYSIS

Conflict graph analysis is used for concurrency control in SDD-1, a database system which handles distributed redundant copies of data [Rothnie et al., 77], [Bernstein et al., 77a]. The analysis here is a simplified variant of the general approach used in that system.

The technique involves the construction of a *conflict graph*, CG(TAU), over a transaction system TAU. This a node-labelled undirected graph, <V,E>, where

$$V = \{r_i | T_i \in T\} \cup \{w_i | T_i \in T\}$$

$$E = \{<r_i, w_i> | T_i \in T\}$$

$$\cup \{<r_i, w_j> | T_i, T_j \in T,\ T_i \neq T_j,\ \text{and readset}_i \cap \text{writeset}_j \neq \emptyset\}$$

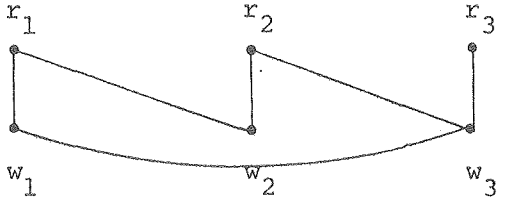
$$\cup \{<w_i, w_j> | T_i, T_j \in T,\ T_i \neq T_j,\ \text{and writeset}_i \cap \text{writeset}_j \neq \emptyset\} .$$

The nodes in the graph correspond to elements in a log. The edges in the graph indicate either that the endpoints are from the same transaction or that two transactions have a read-write or write-write intersection.

For example, a transaction system TAU_1, with three transactions with read-sets and write-sets as follows:

transaction	readset	writeset
T_1	x_1	x_2
T_2	x_3	x_1
T_3	x_4	x_3, x_2

leads to the following conflict graph.



Two log elements from different transactions that have a connecting edge in the CG are said to be in *conflict*. A log which contains the same elements as the given log and in which conflicting pairs or elements occur in the same order as in the given log will necessarily be equivalent to the given log.

LEMMA CG_1. *Two logs are equivalent if they contain the same elements and if conflicting elements occur in the same order in both logs.*□

In serializing a given log, we will only consider those serial logs in which conflicting elements appear in the same order both in the given log and in the serial log. By conforming to this condition, we guarantee that the serial logs under consideration are equivalent to the original log.

For example, consider the following logs defined on TAU_1:

$$L_1: \quad R_1 \; R_2 \; W_2 \; W_1 \; R_3 \; W_3$$

$$L_2: \quad R_1 \; W_1 \; R_2 \; W_2 \; R_3 \; W_3$$

$$L_3: \quad R_1 \; R_2 \; W_2 \; R_3 \; W_3 \; W_1$$

In L_1 and L_2, all conflicting pairs of elements appear in the same order. By Lemma CG_1, L_1 and L_2 are equivalent. Since L_2 is serial, this demonstrates that L_1 is serializable. Log L_3 is identical to L_1, except that W_3 and W_1 are reversed. Since $\langle w_1, w_3 \rangle$ is in $CG(TAU_1)$, W_3 and W_1 conflict. So, L_1 and L_3 do not satisfy the condition of Lemma CG_1.

The problem now is to constrain the set of logs that the concurrency control mechanism can produce so that a log will necessarily have an equivalent serial log in which conflicting elements remain unswitched. The argument below shows that as long as CG(TAU) is acyclic, all logs over TAU will have such an equivalent serial log.

Whenever two transactions are connected by an edge in CG(TAU), their order in a serialization of a log, L, over TAU is fully determined by L. For an edge of the form $\langle r_i, w_j \rangle$, the order of T_i and T_j in the serialization must be the same as the order of R_i and W_j in the log L. For an edge of the form $\langle w_i, w_j \rangle$, the order of T_i and T_j in the serialization must be the same as the order of W_i and W_j in L. We define the relation "$\twoheadrightarrow$" to capture this idea, as follows:

$$T_i \twoheadrightarrow T_j \quad \text{iff} \quad T_i \neq T_j \quad \text{and}$$

1. $\langle r_i, w_j \rangle$ is an edge in CG(TAU) and R_i precedes W_j in L; or
2. $\langle w_i, w_j \rangle$ is an edge in CG(TAU) and W_i precedes W_j in L; or

3. $\langle w_i, r_j \rangle$ is an edge in CG(TAU) and W_i precedes R_j in L.

The $\twoheadrightarrow$ relation on log L_1 over TAU_1 includes:

$$T_1 \twoheadrightarrow T_2; \quad T_2 \twoheadrightarrow T_3; \quad T_1 \twoheadrightarrow T_3 .$$

Similarly, a path of edges through the conflict graph may fully determine the order of the transactions at the ends of the path. This notion is reflected in the relation $\overset{*}{\twoheadrightarrow}$ which is the transitive closure of $\twoheadrightarrow$.

The relation $\overset{*}{\twoheadrightarrow}$ provides a characterization of serializable logs. In general, a log over TAU is not serializable if there are two transactions which are forced to appear in one order as a result of considering one path through CG(TAU), yet are also forced to appear in the opposite order as a result of a different path through CG(TAU). This can only occur if CG(TAU) has a cycle.

LEMMA CG_2. *If* $\overset{*}{\twoheadrightarrow}$ *defines a partial order (i.e., it is transitive, anti-reflexive, and anti-symmetric) then the log over which it is defined is serializable.* □

So, a transaction can be serialized as long as it does not lie on a cycle in CG(TAU). To deal with transactions which do lie on cycles, it is necessary to prevent non-serializable logs from being produced. Preventing such logs is the function of the concurrency control mechanism.

An adequate concurrency control mechanism is one that constrains logs as follows. Assign a total ordering to all the transactions. Whenever two log elements from different transactions have a connecting edge in CG(TAU) which forms part of a cycle, those two elements must appear in the log in the same order as the prespecified total order of their corresponding transactions. This constraint guarantees that "$\overset{*}{\twoheadrightarrow}$" defines a partial order, and (by Lemma CG_2) the log is serializable.

One convenient total ordering of transactions is the order in which the write operations appear in the log. Using this ordering, we can express a suitable log constraint as follows:

Log Constraint LC: *For all* $T_i \neq T_j$ *in* T, *if the edge* $\langle r_i, w_j \rangle$ *lies on a cycle in* CG(TAU), *then* R_i *precedes* W_j *in log* L *only if* W_i *precedes* W_j *in log* L.

And, we can prove the following theorem:

THEOREM CG. *If a log obeys the log constraint* LC *then it is serializable.* □

In TAU_1, log L_1 satisfies log constraint LC. And, as we previously showed, L_1 is serializable. Log L_3, on the other hand, does not satisfy log constraint LC; in L_3, R_1 precedes W_2 but W_1 follows

W_2, in spite of the existence of edge $\langle r_1, w_2 \rangle$ in a cycle in $CG(TAU_1)$. As it turns out, this violation of log constraint LC is fatal, since it makes L_3 nonserializable.

A concurrency control mechanism that implements the log constraint LC is not hard to construct. If an $\langle r_i, w_j \rangle$ edge lies on a cycle in CG(TAU), and R_i and R_j have been processed but W_1 and W_j have not yet been processed, then W_i must be processed before W_j. If W_j asks to be processed first, then it must wait. Notice that if $\langle r_j, w_i \rangle$ also lies on the cycle, then this implies W_j must be processed before W_i. Of course, both constraints cannot be satisfied once R_i and R_j have been processed. So, the concurrency control must sometimes delay a read operation, to make sure constraint LC is satisfied. In this case, if R_i is processed, then W_i must be processed before R_j, delaying R_j if it is submitted before W_i.

(The situation described here is somewhat unrealistic in it requires advance knowledge of all the transactions to be submitted to the system so that cycles in CG(TAU) can be determined. In the actual SDD-1 system the problem is handled by predefining classes of transactions based on their read-sets and write-sets. The conflict graph is then preanalyzed on the classes in order to predict all cycles in CG(TAU) before the transactions are processed. See [Bernstein et al., 77a, 77b, 78].)

It is instructive to relate this approach to the concept of serialization graphs discussed earlier. By considering only those serial logs in which conflicting elements remain unswitched, we have chosen to consider only a particular $SG_n(L)$. Paths through that $SG_n(L)$ correspond to "$\overset{*}{\rightarrow}$", and both reflect the notion of "happens before". The log constraint LC guarantees that this $SG_n(L)$ is acyclic, and thus (by Lemma SR_1) a serialization of L exists. Yet as with the other techniques, some logs are prohibited by constraint LC which in fact could be serialized.

The concurrency control for conflict graph analysis described in this section is neither strictly stronger nor strictly weaker than locking. The following log, for example, is two-phase locked, but does not obey the log constraint LC:

log: $R_i \quad R_j \quad W_j \quad W_i \quad R_k \quad W_k$

$\uparrow$

locked-point_i

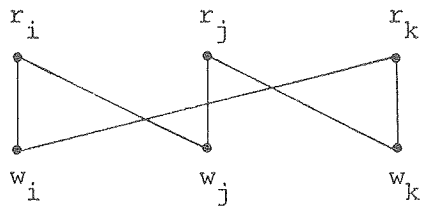
where

$\text{readset}_i = \text{writeset}_j = \{x_1\}$

$\text{readset}_j = \text{writeset}_k = \{x_2\}$

$\text{readset}_k = \text{writeset}_i = \{x_3\}$

Conflict graph



At locked-point$_i$, transaction T_i has released its read-set lock on x_1, thereby allowing T_j to write. This log is prevented by log constraint LC; since the $\langle r_i, w_j \rangle$ edge lies on a cycle in the conflict graph, it is illegal that R_i precedes W_j but W_j precedes W_i. The concurrency control implementing constraint LC must force W_j to wait until after W_i is processed (if it does not, R_jW_j may occur between W_j and W_i, thereby making the log unserializable).

The log below, however, is allowed by log constraint LC (the associated conflict graph is acyclic) but is not two-phase locked.

log: $R_i \quad R_j \quad W_j \quad R_k \quad W_k \quad W_i$

where

$\text{readset}_i = \text{writeset}_j = \{x_1\}$

$\text{readset}_j = \text{writeset}_k = \emptyset$

$\text{readset}_k = \text{writeset}_i = \{x_2\}$

Finally, there are SR logs that neither are two-phase locked nor satisfy log constraint LC. For example,

log: $R_i \quad R_j \quad W_j \quad R_k \quad W_k \quad W_i \quad R_\ell \quad W_\ell$

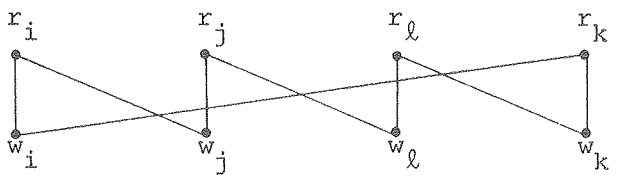
where

$\text{readset}_i = \text{writeset}_j = \{x_1\}$

$\text{readset}_k = \text{writeset}_i = \{x_2\}$

$\text{readset}_j = \text{writeset}_\ell = \{x_3\}$

$\text{readset}_\ell = \text{writeset}_k = \{x_4\}$



Log constraint LC requires that since R_i precedes W_j, W_i must precede W_j; the log violates this constraint. The log is not two-phase locked, as in the previous example. However, the log is SR since $R_k\ W_k\ R_i\ W_i\ R_j\ W_j\ R_\ell\ W_\ell$ is a correct serialization of it.

We wish to reiterate that our description of conflict graph analysis is an extreme simplification of the concurrency control mechanism used in the distributed SDD-1 system. In that system, transaction classes are specified in advance and the conflict graph analysis is performed *a priori* over the class conflict graph. Several log constraints are defined, determined by graph topology and transaction frequency, with alternate protocols available for implementing these various constraints. Timestamps have been adopted from Thomas' algorithm to provide the needed global total ordering of transactions.

8. CONCLUSION

By analyzing concurrency control mechanisms using a simple model of transaction execution, we have been able to determine the essential reasons why the mechanisms guarantee serializability. We have found that two proposed mechanisms, those of [Thomas 75] and [Stearns et al., 76] are special cases of locking. A third mechanism, conflict graph analysis, was shown to use a different approach. In fact, conflict graph analysis is neither strictly stronger nor strictly weaker than locking in the amount of concurrency it permits.

Given this analysis, it would be interesting to compare the amount of concurrency permitted by locking to that permitted by conflict graph analysis given certain stochastic assumptions. One approach could be to select a "random" transaction system, TAU, and count the number of logs that are permitted by each mechanism. Finding a realistic definition of "random", though, is difficult. We leave this as an open question.

There is an algorithmic dimension to the concurrency control problem that we did not treat. Given a mechanism that guarantees serializability, problems such as deadlock, indefinite postponement, and distributing control still require solutions (see, e.g., [Chamberlin et al., 74], [Stearns et al., 76], [Thomas, 75], and [Rothnie et al., 77]). However, our analysis here shows that such problems can apparently be ignored when examining serializability and can be treated subsequently as a separate question.

ACKNOWLEDGEMENTS

We would like to thank Oded Shmueli for finding an error in our first formulation of two-phase locking and Nathan Goodman for helping us to correct the error. We also thank Renate D'Arcangelo for her expert assistance in the preparation of the manuscript.

0 0 - 0 3 1 0 - 1 3 6

REFERENCES

1. Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, Mass., 1975).

2. Bernstein, P. A., N. Goodman, J.B. Rothnie, and C.H. Papadimitriou, "Analysis of Serialization in SDD-1: A System for Distributed Databases," *Proc. of 1977 IEEE COMPSAC Conf.*, Nov. 1977.

3. Bernstein, P.A., D.W. Shipman, J.B. Rothnie, and N. Goodman, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases," Technical Report TR-77-09, Computer Corp. of America, Dec. 1977.

4. Bernstein, P.A., J.B. Rothnie, N. Goodman, and C.H. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Trans. on Software Eng.*, May 1978.

5. Chamberlin, D.D., R.F. Boyce, and I.L. Traiger, "A Deadlock-Free Scheme for Resource Allocation in a Data-Base Environment," *Info. Proc. 74*, North-Holland Pub. Co., Amsterdam, 1974, 340-343.

6. Eswaran, K.P., J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM*, Vol. *19*, No. 11 (Nov. 1976), pp. 624-633.

7. Gray, J.N., R.A. Lorie, G.R. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," IBM Research Report RJ1654, Sept. 1975.

8. Lamport, L., "Towards a Theory of Correctness for Multi-User Data Base Systems," Mass. Computer Assoc. Inc., Report CA-7610-0712, Oct. 1976.

9. Manna, Z., *Mathematical Theory of Computation* (McGraw-Hill, New York, 1974).

10. Papadimitriou, C.H., P.A. Bernstein, and J.B. Rothnie, "Some Computational Problems Related to Database Concurrency Control," *Proc. of Conf. on Theoretical Computer Science*, Waterloo, Aug. 1977.

11. Rothnie, J.B., N. Goodman, and P.A. Bernstein, "The Redundant Update Methodology of SDD-1: A System for Distributed Data," *Proc. 1977 IEEE COMPSAC Conf.*, Nov. 1977.

12. Stearns, R.C., P.M. Lewis, and D.J. Rosencrantz, "Concurrency Control for Database Systems," *Proc. of Conf. on the Foundations of Computer Science*, ACM, N.Y., 1976, pp. 19-32.

13. Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Databases which Uses Distributed Control," BBN Rep. No. 3340, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., July 1975.

Maximally Concurrent Transaction Processing

Toshimi Minoura

Digital Systems Laboratory
Stanford University
Stanford, California

ABSTRACT

The problem of concurrent processing of transactions in a database system is discussed. A necessary and sufficient condition for the concurrent processing of transactions without violating consistency is shown, and a maximally concurrent and deadlock-free synchronization scheme which makes use of the above condition is given. The limitation of the more common locking mechanism is studied, and the necessity of two-phase locking for consistency is proved when locking is the only mechanism used. NP-completeness of the shortest-time scheduling problem is also discussed.

INTRODUCTION

The problem of concurrent processing of transactions in a database system has been studied by many people (2,3,4,5,6,12). Even though transactions are processed concurrently, their effects to the system and the users must be as if they were processed in sequence. The concurrent execution of logically indivisible operations, called atomic actions in (10), is also desirable for other systems, e.g., an operating system working in a multiprocessor environment. These problems are of the same nature, and we discuss them in terms of transaction processing.

We assume that a transaction consists of a linear sequence of actions, and that the synchronization mechanism is informed of the structure of the transactions when the transactions are initiated. This assumption excludes the case discussed in (3), where some locking depends on data values. This assumption is not always impractical; it is valid for most operating system primitives and for some real-time systems which handle many short transactions. Also, in some environments transactions are predefined. At first glance, the problem looks very similar to the deadlock avoidance problem treated in (7), which is NP-complete (11), and seems to be intractable (1). However, this problem has a simpler structure, and the same technique as used for the analysis of concurrently executable instructions in (9) can be applied. The synchronization scheme thus obtained is maximally concurrent in the sense of (8) and deadlock-free.

Share and exclusive locks have been commonly used as a means of enforcing consistency. It is shown that simple locking cannot provide maximally concurrent operation. The fact that two-phase locking is sufficient for system

0 0 0 3 5 1 0 4 1 3 7

consistency is proved in (4). The proof that it is a necessary condition, when locking is the only mechanism used to preserve system consistency, is given here.

Although the maximally concurrent transaction scheduling algorithm can be realized in polynomial time, a minimum-time transaction scheduling problem is shown to be NP-complete as are many similar scheduling problems treated elsewhere (13).

DATABASE MODEL

We assume that a database consists of a set of components, each of which can be assigned a value independently. A transaction is a sequence: $T=t_1 T_1 t_2 T_2 \ldots t_n T_n t_{n+1}$ of n actions and n+1 synchronization primitives, where T is the transaction name, t_i is the synchronization primitive to initiate action T_i, and t_{n+1} is the synchronization primitive terminating the transaction. The database components are accessed inside an action. There may be more than one access, possibly to different components, in an action. The actions are not atomic, i.e., multiple actions may be executed concurrently. The synchronization primitives are atomic and indivisible. There is an underlying control state q used by the synchronization primitives. A synchronization primitive s defines a predicate Ps(q) and a function Fs(q). When the control reaches s, if Ps(q) does not hold then the transaction is suspended, and if Ps(q) holds then q is updated to the new value Fs(q) and the action following s is initiated unless s is the last synchronization primitive in the transaction.

The execution sequence of transactions $S=s_1 S_1 \ldots s_m S_m s_{m+1}$, $T=t_1 T_1 \ldots t_n T_n t_{n+1}, \ldots$ is a sequence of synchronization primitives: $a_1 a_2 \ldots a_\ell$, where $a_1, \ldots, a_\ell$ is a merge of the synchronization primitives in S, T, ... The binary relation $<<$ on the set of synchronization primitives in the execution sequence is defined by: $a_i << a_j$ if and only if $1 \le i < j \le l$. Thus $a_i << a_j$ has the intuitive meaning: "a_i occurs before a_j." Note that the execution sequence completely describes the possible concurrent operations of the actions.

Let A be an action. We define D(A), the domain of A, to be the set of components read by A, and R(A), the range of A, to be the set of components written by A. The binary relation conflict on the set of actions is defined by:

conflict={<A,B>| A and B belong to different transactions and either
$D(A) \cap R(B) \neq \phi$ or
$R(A) \cap D(B) \neq \phi$ or
$R(A) \cap R(B) \neq \phi$}.

Given the execution sequence of the transactions, another binary relation $<<<$ on the set of transactions is defined by: for any two transactions S of m steps and T of n steps, $S<<<T$ if and only if

[conflict(S_i, T_j) and $s_i << t_j$ for some $1 \le i \le m$, $1 \le j \le n$].

Thus S<<<T has the intuitive meaning: S must logically precede T, i.e., S must precede T in the equivalent sequential execution sequence.

A consistent execution sequence of the transactions is one in which the system and the users see the database state as if the transactions were processed sequentially.

A scheduling mechanism is maximally concurrent (see (8)) if and only if any consistent execution sequence can occur.

MAXIMALLY CONCURRENT SCHEDULING

In this section, we will show the condition for consistent transaction processing and a maximally concurrent transaction scheduling algorithm.

Any execution sequence which avoids concurrent operations of conflicting actions, and for which the relation <<< is acyclic, is consistent, and any which violates this constraint is not consistent. One important consequence of the constraint is: if R and S have any actions that conflict then either R<<<S and $r_{i+1} <<< s_j$, or S<<<R and $s_{j+1} <<< r_i$, for all i and j such that R_i and S_j conflict.

To be strict, the term "transaction" used here stands for a "transaction schema" which represents a class of transactions accessing the same database components in the same sequence, but whose internal processing may differ. The execution sequence of transaction schemas is consistent if all of the execution sequences of actual transactions represented are consistent, otherwise it is not consistent.

Any execution sequence which satisfies the above constraint defines a partial order on the set of the transactions. Any sequential execution of the transactions according to some topological order of the partial relation has the same effect as when the transactions are executed as defined by the execution sequence. Assume that R S T ... is the topological order of the transactions obtained from some execution sequence which observes the above constraint. The actions in R can be moved up to the beginning of the execution sequence without changing the effect, because the actions in R need only be moved over actions with which they do not conflict. Then, we can move all of the actions in S to the positions just following those of R, and so on.

Next, consider an execution sequence that violates the above constraint. If two conflicting actions are executed concurrently, inconsistency can arise. If the relation <<< contains a cycle, there can be an interference among conflicting actions. They can give rise to inconsistency in the effect of the transactions.

Therefore a scheduling algorithm which realizes the above constraint, and no more, is maximally concurrent.

The implementation of the algorithm is straightforward. A transaction is represented as a linear graph. Each node stands for an action. Two adjacent nodes in a transaction are connected by a directed edge whose direction shows the

0 0 1 0 5 1 9 4 1 5 8

execution order of these actions. The synchronization mechanism maintains a graph structure K to represent an underlying control state q. The nodes in K stand for actions. Some edges in K are directed, and the direction shows the required execution order. Other edges in K are undirected and show the conflict relation among actions. Those nodes whose corresponding actions are complete are marked as done. When the first synchronization primitive of a transaction is issued, the linear graph corresponding to the transaction is added to K. Also, directed edges, when the nodes in K are either being executed or done, or undirected edges, when otherwise, are added to K due to the conflict relation between the nodes in K and those in the newly added transaction. An action may be initiated when all its incoming directed edges emanate from done nodes. When the initiated action has undirected edges, they are turned into directed edges emanating from the action. When one undirected edge between the actions of two transactions is assigned a direction, all other edges between the nodes of these two transactions are assigned the same direction. When a new $<<<$ is defined between two transactions, its transitive closure must be calculated, and a direction must be assigned to each undirected edge between the actions according to the closure $<<<$ to keep the relation $<<<$ acyclic.

The scheme is deadlock-free because the induced precedence relation is cycle-free.

The maximally concurrent scheme may cause an indefinite delay of an individual transaction. This can be avoided by assigning the direction to all of the conflict edges from the nodes in K to the nodes in the newly added transaction. The resultant scheme is not maximally concurrent. Maximal concurrency necessarily allows individual starvation.

LOCKS

Locking is the most common consistency control mechanism. Its mechanism is well-known, so we do not repeat how it works. Two types of locks, namely share and exclusive locks, are assumed in the following discussion.

When locking is used, there is a possibility of deadlocks. A locking is partially consistent if and only if all of the execution sequences which lead to the completion of all transactions are consistent.

We show that locking is not powerful enough to realize all possible concurrent operations, i.e., not maximally concurrent.

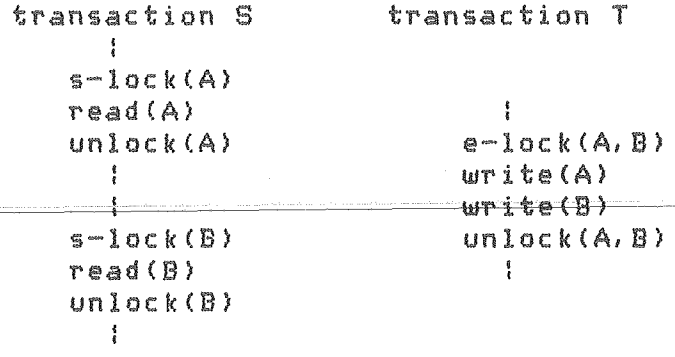
Consider the following two transactions $S=s_1 S_1 s_2 S_2 s_3 S_3 s_4$ and $T=t_1 T_1 t_2 T_2 t_3$, where conflict=$\{<S_1, T_1>, <S_3, T_2>\}$. Assume maximal concurrency was realized using locks. $s_1 s_2 t_1$ is the prefix of a consistent execution sequence, but $s_1 s_2 t_1 t_2$ and $s_1 s_2 t_1 s_3 t_2$ are not. To prevent these inconsistent execution sequences, there must be a lock l which is active from s_2 through s_4 in S and from t_2 through t_3 in T. However, lock

1 does not allow the consistent execution sequence $t_1\ t_2\ s_1\ s_2\ t_3\ s_3\ s_4$. This is a contradiction.

In (4), two-phase locking is proved to be a sufficient condition for consistency. Here, we prove that it is a necessary condition, if a transaction is to run concurrently with unknown transactions.

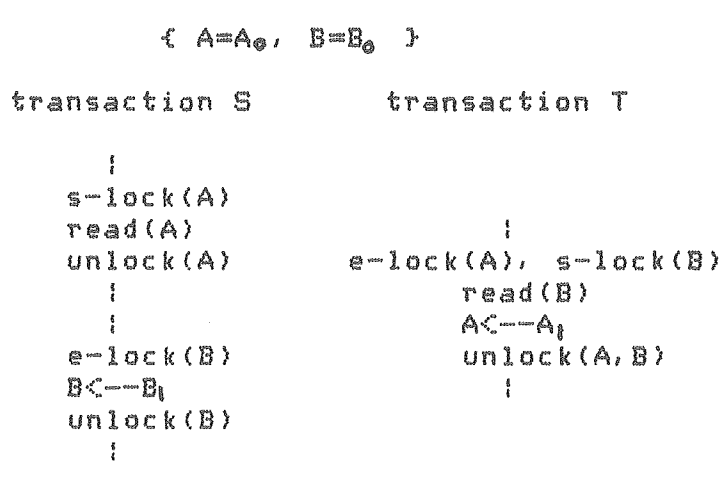
First, we show that any two locks in a transaction must overlap or instantaneously follow one another. We show this by enumerating all cases, and show that inconsistency arises wherever locks in S are neither overlapping nor instantaneously following and transaction T intervenes between these locks. s-lock stands for a share lock request, and e-lock for an exclusive lock request.

read-read



Transaction S could see a state that has been partially updated by T.

read-write



Assume that $A=A_0$ and $B=B_0$ before S and T begin. Transaction S sees $A=A_0$ and transaction T sees $B=B_0$. This is inconsistent, since if transaction S sees $A=A_0$ transaction T should see $B=B_1$.

0 0 1 0 5 1 0 4 1 3 9

write-read

$\{ A=A_0,\ B=B_0 \}$

transaction S	transaction T
¦	
e-lock(A)	
$A \leftarrow A_1$	¦
unlock(A)	s-lock(A), e-lock(B)
¦	read(A)
¦	$B \leftarrow B_1$
s-lock(B)	unlock(A,B)
read(B)	¦
unlock(B)	
¦	

Transaction S sees $B=B_1$ and transaction T sees $A=A_1$.

write-write

transaction S	transaction T
¦	
e-lock(A)	
write(A)	¦
unlock(A)	s-lock(A,B)
¦	read(A)
¦	read(B)
e-lock(B)	unlock(A,B)
write(B)	¦
unlock(B)	
¦	

T could see a state that has been partially updated by S.

Hence, any two locks in a transaction must overlap or instantaneously follow one another. This means that there should be a point in a transaction where all locks are active. Before that point only locking is done, and after that point only unlocking is done. This structure is called <u>two-phase</u> in (4).

Note that a share lock is not required while an exclusive lock is active when read and write actions to the same component exist in a transaction. In this case the interpretation of an initiation or termination point of a share lock may have to be extended to the initiation or termination point of the exclusive lock to satisfy the two-phase rule.

Sufficiency of two-phase locking for the partial consistency can be simply stated: we can assume that all accesses to the components are done at the point when all locks in a transaction are active, and that the transaction is processed at that moment. All locks of two interfering transactions can never be active at the same time. There is no ambiguity about the order of the moments when all locks

in a transaction are active.

MINIMUM-TIME SCHEDULING

From a practical standpoint, achieving minimum-time scheduling of a set of given transactions might be a more meanigful goal than maximal concurrency. However, the minimum-time transaction scheduling problem belongs to a class of NP-complete problems and seems to be intractable.

Many similar problems have been shown to be in that class, so only a brief outline of the proof is given here.

The problem of deciding whether a set of transactions can be completed within a time limit k, is NP-complete.

We assume that time is spent only within actions and not within synchronization primitives. The fact that the scheduling problem belongs to the class NP defined in (1), can be easily verified by giving the nondeterministic algorithm which solves the problem in polynomial time.

The NP-completeness of the problem can be shown by transforming the graph k-coloring problem, which is known to be NP-complete, to the scheduling problem.

Let the graph in the graph k-coloring problem be G=<N, E>, where N is the set of nodes and E the set of edges. Consider the problem of scheduling transactions which each contain a single action of unit length, whose set of transactions is N, and two transactions T_1 and T_2 conflict if and only if T_1 and T_2 are connected by an edge in G. Here, a transaction is identified with its action.

The graph is k-colorable if and only if the scheduling of the transactions in time limit k is possible.

If the graph is k-colorable then the scheduling of the transactions in time limit k is possible because transactions of the same color do not interfere and can be processed concurrently.

If the scheduling of the transactions in time limit k is possible then the graph is k-colorable. We can assign the same color to those nodes which are processed concurrently. Note that two nodes of the same color are not connected by an edge since they do not conflict.

The proof that the minimization problem is NP-complete when the above kind of decision problem is NP-complete, can be found in (1).

SUMMARY

We have studied the condition for the concurrent processing of transactions without violating consistency, and have given an algorithm which realizes the maximally concurrent processing. It is interesting as a theoretical bound for concurrent transaction processing. Its implementation incurs some overhead, but has a nice property, i.e., deadlock-freeness. The fact that locking cannot realize maximal concurrency was shown as well as the

0 0 - 0 5 1 0 4 1 4 0

proof that two-phase locking is necessary for consistency. In spite of the existence of polynomial time maximally concurrent scheduling algorithm, a minimum-time scheduling algorithm is NP-complete.

ACKNOWLEDGMENT

The author wishes to thank Prof. Susan Owicki for providing comments on the manuscript and for her supervision of the research. This work was partially supported by the Joint Services Electronics Program under contract N-00014-75-C-0601.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, Mass., 1974).

2. R. Bayer, On the Integrity of Data Bases and Resource Locking. In Data Base Systems: Lecture Notes in Computer Science 39 (Springer-Verlag, Berlin, 1976).

3. D. D. Chamberlin, R. F. Boyce, and I. L. Traiger, A Deadlock-Free Scheme for Resource Locking in a Data Base Environment. Proc. IFIP Congress 1974.

4. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, The Notions of Consistency and Predicate Locks in a Database System, CACM 19, 11 (Nov. 76).

5. J. N. Gray, R. A. Lorie, and G. R. Putzolu, Granularity of Locks in a Large Shared Data Base. Proc. International Conf. on Very Large Data Bases (Sept. 1975).

6. D. A. Hawley, J. S. Knowles, and E. E. Tozer, Database Consistency and the CODASYL DBTG Proposals. Comp. J. 18, 3 (Aug. 1975).

7. P. G. Hebalkar, Coordinated Sharing of Resources in Asynchronous Systems, Project MAC Conference Record (1970).

8. R. M. Keller, Parallel Program Schemata and Maximal Parallelism, J. ACM 20, 3 (July 1973); and J. ACM 20, 4 (Oct. 1973).

9. R. M. Keller, Look-Ahead Processors, Computing Surveys, 7, 4 (Dec. 1975).

10. D. B. Lomet, Process Structuring, Synchronization, and Recovery Using Atomic Actions. Sigplan Notices, 12, 3 (1977).

11. Y. Sugiyama, T. Araki, J. Okui, and T. Kasami, Complexity of the Deadlock Avoidance Problem. Tr. of Inst. of Electronics and Comm. Engrs. of Japan, J60-D, 4 (April 1977), in Japanese.

12. R. E. Stearns, P. M. Lewis II, and D. J. Rosenkrantz, Concurrency Control for Database Systems. IEEE Symp. on Foundations of Comp. Sci. (1976).

13. E. G. Coffman, Jr., editor, Computer and Job-shop Scheduling Theory (John Wiley & Sons, 1976).

0 0 : 0 5 1 0 4 1 4 1

LOCKING AND DEADLOCK DETECTION IN DISTRIBUTED DATABASES *

Daniel A. Menasce and Richard R. Muntz

Computer Science Department
University of California,
Los Angeles, California 90024

Abstract

This paper describes two protocols for the detection of deadlocks in distributed databases - a hierarchically organized and a distributed one. A graph model which depicts the state of execution of all transactions in the system is used by both protocols. A cycle in this graph is a necessary and sufficient condition for a deadlock to exist. Nevertheless, neither protocol requires that the global graph be built and maintained in order for deadlocks to be detected. In the case of the hierarchical protocol, the communications cost can be optimized if the topology of the hierarchy is appropriately chosen.

INTRODUCTION

This work is concerned with issues of locking and deadlock detection mechanisms in distributed databases. The problem of system deadlocks in multiprogramming and multiprocessing systems has received considerable attention in the literature and is well understood (1) (2) (3). There are three approaches to the treatment of deadlocks: deadlock prevention, deadlock avoidance and deadlock detection and resolution.

Deadlock prevention requires that all the resources be acquired at once by a transaction. This requirement cannot always be satisfied in a database environment since the resource needs of a transaction may be data dependent and not precisely known at the start of the transaction. Therefore it would be necessary for a transaction to acquire all possible resources required thereby decreasing system concurrency.

---

* This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211.

Deadlock avoidance requires some advance knowledge of the resource usage of transactions in order to determine at each point in time whether there is a valid sequence of actions of the already initiated but not yet completed transactions such that all of them can be run to completion. Again, this approach is not practical in distributed databases since the necessary advance information to avoid deadlocks is either absent or is distributed enough to render inefficient any attempt to avoid deadlocks.

Deadlock detection can be done by searching for cycles in a "state graph" (3). A method for the detection and resolution of deadlocks in a centralized database system was presented in (4). In distributed databases, however, it is not efficient to maintain a global state graph for the whole system. Two methods for detecting deadlocks in distributed databases, which do not require that a global graph be built and maintained, are presented in this paper - one hierarchical and one distributed. An outline of the proof that both protocols will detect all existing deadlocks is given here. For the case of the hierarchical protocol the problem of establishing the hierarchy in a way such that minimizes the cost of using the protocol is introduced.

FORMAL MODEL OF TRANSACTION PROCESSING

This section introduces the necessary notation and formalism upon which we base the locking protocols and deadlock detection mechanisms presented in this paper. The database is considered to be distributed among n sites, S1, S2, ..., Sn, of a computer network. Users interact with the database via transactions. A transaction is a sequence of actions which can be either read, write, lock or unlock operations. If these actions involve data at a single site the transaction is called local as opposed to a distributed transaction which involves resources at several sites. We assume that distributed transactions are implemented as a collection of processes which act on behalf of the transaction. Those processes are called transaction incarnations. There may be one or more incarnations of the same transaction at each participating site. A transaction incarnation is responsible among other things for:

a. acquiring, using and releasing resources local to the site at which it is executing as needed by the transaction.

b. exchanging messages with remote incarnations of the same transaction for purposes of cooperating with incarnations located at foreign sites.

0 0 1 0 5 1 0 4 1 4 2

Note that this model of a transaction execution is general enough to accommodate other models.

A transaction can be in two different states, namely active and blocked. A transaction is blocked if its execution cannot proceed because a needed resource is being held by another transaction, and the transaction is active otherwise. We introduce now a graphic model which depicts the state of execution of all transactions in the system. This model is in the form of a graph called the transaction wait for graph or TWF graph. The nodes of this graph are associated with transaction incarnations and are labeled by the pair (transaction_name, site_name). Note that labeling transaction incarnations with the pair (transaction name, site name) provides unique global names for these nodes if there is at most one transaction incarnation per site per transaction. If more than one incarnation is to be allowed then distinct local names should be assigned for them. Since this distinction is irrelevant for the forthcoming discussion we will assume that there is only one incarnation of any transaction per site.

- there is a directed arc from node (Ti,S) to node (Tj,S) if the incarnation of transaction Ti at site S is blocked and waiting for the incarnation of transaction Tj to release a resource needed by Ti. (Ti,S) is said to be in "resource wait" for (Tj,S).

- there is a directed arc from node (T,Si) to node (T,Sj) if the incarnation of transaction T at site Si is blocked and waiting for a message from the incarnation of T at site Sj. (T,Si) is said to be in "message wait" for (T,Sj).

It can be easily seen that the existence of a cycle in the transaction_wait_for graph is a necessary and sufficient condition for a deadlock to occur.

Figure 1 shows an example of a transaction_wait_for graph for a network with two sites S1 and S2. This graph shows two deadlock cycles. One of them is a local deadlock since it involves only incarnations of transactions at site S2 and therefore only resources local to S2. The other deadlock cycle spans both sites and is an example of what we call a global deadlock.

DEADLOCK DETECTION APPROACHES

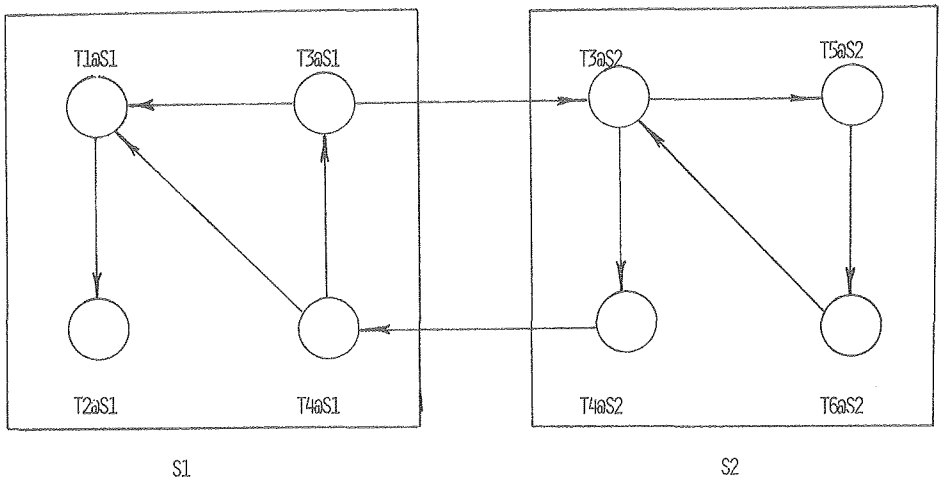


Figure 1 - A transaction wait for graph for a network with two sites S1 and S2.

Deadlock detection involves building and maintaining the transaction wait for graph and searching for the existence of cycles in the graph. The graph has to be updated every time that a transaction changes state, either from active to blocked or vice versa. It should be noted however that new cycles can only potentially arise when a transaction is blocked. Deadlock resolution involves the selection of one or more transactions to be preempted in order for the cycle to be broken. The criteria used in this selection should try to minimize the penalty of preemption by any suitable metric. Examples of deadlock resolution methods are: preempt the transaction which owns less resources, preempt the transaction with the smallest roll back cost, preempt the transaction with the longest expected time to complete, preempt any transaction in the cycle. The examination of such criteria is beyond the scope of this paper.

A centralized approach for deadlock detection in distributed databases was suggested by Gray in (5). In this approach there is a centralized deadlock detector which is responsible for constructing a global graph equivalent to the transaction wait for graph considered here. The central deadlock detector builds the graph out of information received from all the participating sites in the network.

While the centralized method may be practical and efficient for local networks it may impose fairly large communications cost in geographically distributed systems. This observation stems from the fact that the central deadlock detector may be located very "far" from some of the sites in

0 0 : 0 5 1 0 4 1 4 3

the network. As an example, we certainly do not want deadlocks which only involve resources located at some host computers located in Southern California to be detected at the East Coast. The hierarchical approach to deadlock detection presented in the next section lends itself to an optimization by which deadlocks can be detected by a site which is located as "close" as possible to the sites involved in the cycle.

A HIERARCHICALLY ORGANIZED LOCKING AND DEADLOCK DETECTION PROTOCOL

Let the database, DB, be partitioned into a set of subdatabases DBi's such that DB is the union of all the DBi's and DBi and DBj are disjoint for $i \neq j$. The locking and deadlock detection mechanism presented here has as its core a hierarchy of lock controllers which interact in a way to be explained in this section. First, we are going to distinguish between the controllers which are at the bottommost level of the hierarchy, called leaf-controllers or LKs, and the non-leaf controllers or NLKs.

A leaf controller, LKi, is assigned to each subdatabase DBi. In the example shown in figure 2 we have three leaf controllers LK1, LK2 and LK3 and two non-leaf controllers NLK0 and NLK1.

Each leaf controller, LKi, maintains a transaction wait for graph, TWF(LKi). This graph contains all the nodes of the global TWF associated with transaction incarnations local to LKi. In addition, two special types of nodes called output port nodes and input port nodes are introduced in the TWF of a leaf controller. These nodes are associated with the arcs of the global TWF which join incarnations in two distinct controllers and are defined as follows.

a. a node in the TWF graph of LKi is called an output port and denoted O(LKi,T) if the global TWF contains an outgoing arc from an incarnation of transaction T local to LKi into a non local incarnation of T.

b. a node in the transaction wait for graph of LKi is called an input port and denoted I(LKi,T) in TWF(LKi) if in the global TWF there is an incoming arc into an incarnation of transaction T local to LKi from a non local incarnation of T.

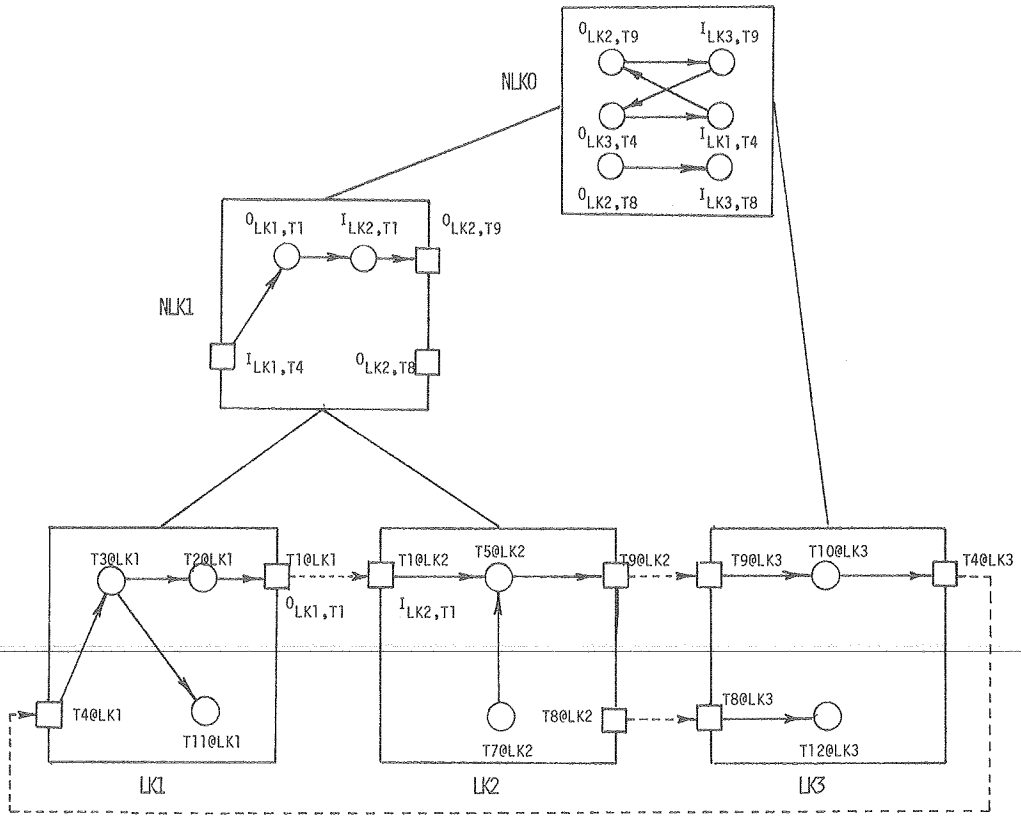


Figure 2 - A Hierarchy of Lock Controllers

Note that labels assigned to input and output ports are unique since there is only one transaction incarnation per transaction per site. In the example of figure 2, the output port O(LK1,T1) and the input port I(LK2,T1) correspond to an arc in the global TWF from T1@LK1 (the incarnation of T1 at LK1) to T1@LK2. The dashed lines indicate arcs in the global TWF. These arcs are represented explicitly at the upper levels of the hierarchy as will be explained below.

Non-leaf controllers maintain a graph called input-output-ports (IOP) graph. Nodes of an IOP are associated with input and output ports of leaf-controllers. We will refer to them as i-nodes and o-nodes respectively. Some of the i-nodes may be themselves input ports for the IOP and some of the o-nodes may be output ports for the IOP. The IOP for controller NLKi, denoted IOP(NLKi), is defined by the following rules:

R1 - Arcs from i-nodes can go only to o-nodes and vice-

0 0 : 0 5 1 0 4 1 4 4

versa.

R2 - There is an arc from o-node Oa to i-node Ib if Oa is an output port of a leaf controller in the subtree rooted at NLKi and Ib is a corresponding input port of another leaf controller in the same subtree. In the example of figure 2, there is an arc from O(LK1,T1) to I(LK2,T1) in the IOP of NLK1 since O(LK1,T1) is an output port of LK1 and I(LK2,T1) is its corresponding input port in LK2. LK1 and LK2 are in the subtree rooted by NLK1.

R3 - There is an arc from the i-node Ia to o-node Ob in IOP(NLKi) if there is a path from an input port Ia to an output port Ob of a son of NLKi. In the example of figure 2 there is an arc from I(LK1,T4) to O(LK2,T9) in NLK0 since in NLK1 there is a path between I(LK1,T4) and O(LK2,T9).

R4 - An input (output) port of IOP(NLKi) is also an input (output) port of a leaf controller in the subtree rooted by NLKi. In the example of figure 2 the input port of IOP(NLK1) is also an input port of LK1.

Hierarchical Protocol - A Description

Before we describe the protocol let us define the lowest common ancestor between controllers K1, K2, ..., Kn, denoted lca(K1,K2,...,Kn), as the common ancestor between them at the lowest level in the hierarchy (the root is at the highest level). Rules R1 through R3 below describe the hierarchical protocol.

RULE 1: (transaction incarnation T requests a local resource): The requested resource R is in the same subdatabase as the transaction incarnation T. Let LKi be the controller for resource R.

R1.1: [the resource cannot be granted] Let {T1, T2, ...,Tk} be the set of transactions which currently hold resource R. Add an arc from (T,LKi) to (Tj,LKi) for j=1 to k. Check the transaction_wait_for graph at LKi for the existence of cycles.

R1.1.1: If cycles were formed then one or more local deadlocks have been detected and an appropriate action is required for deadlock resolution.

R1.1.2: The addition of the arcs mentioned in Rule R1.1 may have created one or more paths between input and output ports of LKi. For each such path send the (input port, output port) pair which delimits the path to the father of LKi.

RULE 2: (transaction T requests a non-local resource): The requested resource R is in a different subdatabase from the previously requested resource. Therefore it has to be acquired by an incarnation of T local to R. Let LKi be the controller for the previously requested resource and let LKj be the controller for resource R. The incarnation of T at LKi becomes blocked and waiting for a message from the incarnation of T at LKj. The node (T,LKi) is now an outport port of the transaction_wait_for graph at LKi and the node (T,LKj) is an input port of the transaction_wait_for graph at LKj.

R2.1: An arc from O(LKi,T) to I(LKj,T) is created in the IOP graph of the lowest common ancestor between LKi and LKj.

R2.2: An o-node labeled O(LKi,T) is added to the IOP graph of each controller in the path between LKi and lca(LKi,LKj). Each such o-node is also an output port of the corresponding IOP graph.

R2.3: An i-node labeled I(LKj,T) is added to the IOP graph of each controller in the path between LKj and lca(LKi,LKj). Each such i-node is also an input port of the corresponding IOP graph.

The protocol followed by a non-leaf controller, NLKi, is described by rule 3 below.

RULE 3: an arc is added to IOP(NLKi).

R3.1: If a cycle is generated by the addition of the new arc then a global deadlock has been detected and an appropriate action is required to resolve it.

R3.2: If no cycle was generated check whether any input output port connection has been generated in IOP(NLKi) and report the endpoints of any such connections to the father of NLKi.

0 0 0 3 5 1 0 4 1 4 5

The method described above for deadlock detection requires that non-leaf controllers be kept up to date continuously. Other variations can be used when appropriate. For example, the information concerning connections between input ports and output ports can be sent periodically. For a sufficiently long period this would reduce the amount of traffic generated but may result in a deadlock existing for too long a period of time. Another method which is intermediate to continuous and periodic deadlock detection is to report connections between input and output ports after they have persisted longer than some threshold. Since, if a deadlock occurs the cycle persists until it is detected and relieved, this method will detect a deadlock after some delay. It appears that judicious choice for the threshold can result in less traffic being generated than with continuous checking and less delay in detecting a deadlock than with periodic checking.

Hierarchical Protocol - Plausibility Argument

The hierarchical protocol has the following properties:

a. deadlocks which involve resources of a single subdatabase, DBi, are detected by the formation of a cycle in the TWF of the leaf controller associated to DBi.

b. deadlocks which involve resources controlled by the leaf controllers LK1, LK2, ...,LKi are detected by the formation of a cycle in the IOP graph of the non-leaf controller which is the lowest common ancestor between the LKi's.

Property a follows directly from Rule 1 of the protocol. The validity of property b is not so straightforward and will be shown to hold by the following theorems.

Let us introduce some notation first. Let the arcs which connect an i-node to an o-node in an input-output-ports graph be called i->o arcs and let o->i arcs be the ones which connect o-nodes to i-nodes. Let tree(K) be the subtree of the hierarchy rooted at controller K. Let us now extend the notation TWF(K) to indicate the subgraph of the global TWF obtained by considering only the resources controlled by all the LKs in tree(K).

Let us show that if a cycle is detected in the IOP graph of a non-leaf controller there is a deadlock.

THEOREM 1: If there is a cycle in the input-output-ports

graph of a non-leaf controller NLKi then there is a deadlock.

Proof: In order to prove this theorem we need the following lemma.

LEMMA 1: For each path P in the IOP graph of a non-leaf controller NLKi there is a path P' in TWF(NLKi) between the transactions associated with the endpoints of P.

Proof: Let us show how to construct the path P'. Let P be a path in IOP(NLKi). For the purpose of this proof let us label each i->o arc (Ij,Oj) in P with the label Kj if Kj is an immediate son of NLKi such that the creation of an input output port connection in Kj caused the arc (Ij,Oj) to be created in IOP(NLKi) - see rules R1.1.2 and R3.2 of the protocol. Arcs of the type o->i in IOP(NLKi) will not be labeled. Using the notation (a,b,c) to indicate an arc labeled c from node a to node b let the path P be P = ($\overline{I}$1,O1,K1), $\overline{O}$1,I2,-), $\overline{I}$2,O2,K2), ..., (In,On,Kn).

The repeated application of the operation 1 below will transform the path P into a path in which all the labels indicate leaf controllers.

Operation 1: If Kj in (Ij,Oj,Kj) is a non-leaf controller, replace (Ij, Oj,Kj) by a path connecting Ij to Oj in IOP(Kj).

After this transformation, there is a path in the TWF of an LK in tree(NLKi) associated with each i->o arc P. There is also an arc between incarnations of transactions in the TWFs of distinct LKs associated with each o->i arc in P. More precisely, for each i->o arc, (Ii,Oi,Ki), in P there is a path in TWF(Ki) between the input port Ii and the output port Oi of TWF(Ki). Now, by rule R2.1 of the protocol each o->i arc, (Oi,Ii+1,-), in P connects two incarnations of the same transaction. These incarnations are local to leaf controllers in tree(NLKi). Finally, the sum of all the paths and arcs thus obtained is the path P' in TWF(NLKi). This concludes the proof of this lemma.

A cycle is a closed path. Therefore, applying Lemma 1 we have that if there is a cycle in IOP(NLKi) then there is a cycle in TWF(NLKi). This indicates the presence of a deadlock.

We want to show now, that given a cycle in the global TWF there is a corresponding cycle in one of the controllers in the hierarchy. In order to state this property more precisely some definitions are in order. Let the graph G' be

0 0 0 3 5 1 0 4 1 4 5

called an arc-condensation or simply condensation of the graph G if the node set of G' is a subset of the node set of G and if (v1,vn) is an arc in G' then there is a path from node v1 to node vn in the graph G.

THEOREM 2: Given a cycle C in the global TWF there is an arc condensation of C in one and only one controller in the hierarchy. This controller is the lowest common ancestor of all the controllers which manage the resources in the cycle C.

Proof: Let C be a cycle in the global TWF which involves resources controlled by leaf controllers LK1, LK2, ..., LKn. Each of the n leaf controllers only know the portion of the cycle which contains resources local to the controller. Let us consider an initial condensation of the cycle C obtained by substituting every path in the TWF from an input port to an output port of a leaf controller by a single arc connecting these ports. For the purpose of this proof let us introduce a representation for a cycle which clearly illustrates how the knowledge about portions of the cycle is distributed throughout several controllers in the hierarchy. So, let a cycle be represented by a sequence of labeled arcs (I0,O0,K0), (I1,O1,K1), ..., (Ij,Oj,Kj), ..., (In-1,On-1,Kn-1) where (Ij,Oj,Kj) indicates that there is a connection between the input port Ij and the output port Oj in the graph of the controller Kj. Also, the output port Oj is connected to the input port Ij+1(mod n). Let father(K) be the father of controller K in the hierarchy. The repeated application of operations 1 and 2 below gives us the desired condensed cycle C.

> Operation 1: If all the labels in C are the same then stop. Substitute every maximal path P in C composed solely of arcs labeled with sons of a common controller K by a single arc having as endpoints the endpoints of P and having as label the common father controller K. This operation is illustrated in figure 3 and can be thought as a short circuit between consecutive brothers . Repeat operation 1 until all the labels in C are different and then do operation 2.
>
> Operation 2: Find all the arcs in C which have as label a controller whose level in the hierarchy is the lowest of all the controllers which appear as labels in C. Substitute the label Ki in each of these arcs by father(Ki). Do operation 1. Operation 2 is illustrated in figure 4.

In order to show the validity of operation 1 let us refer to figure 3 and consider the following observations:

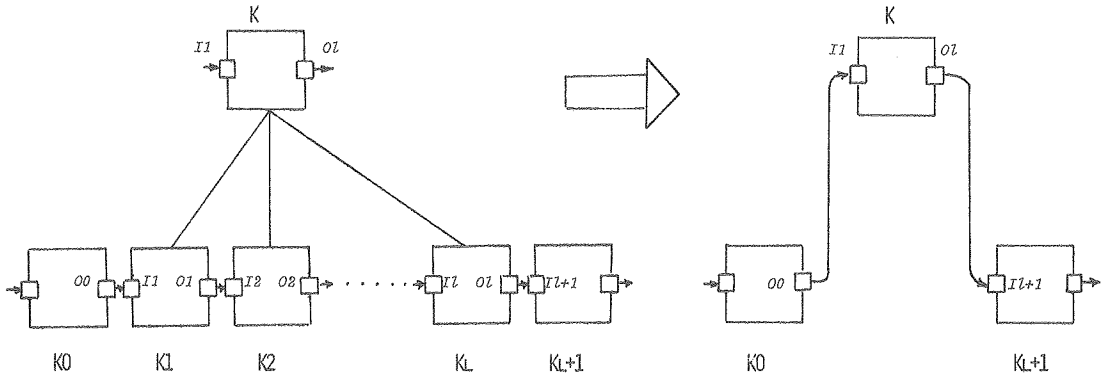


Figure 3 - Operation 1 in Theorem 2

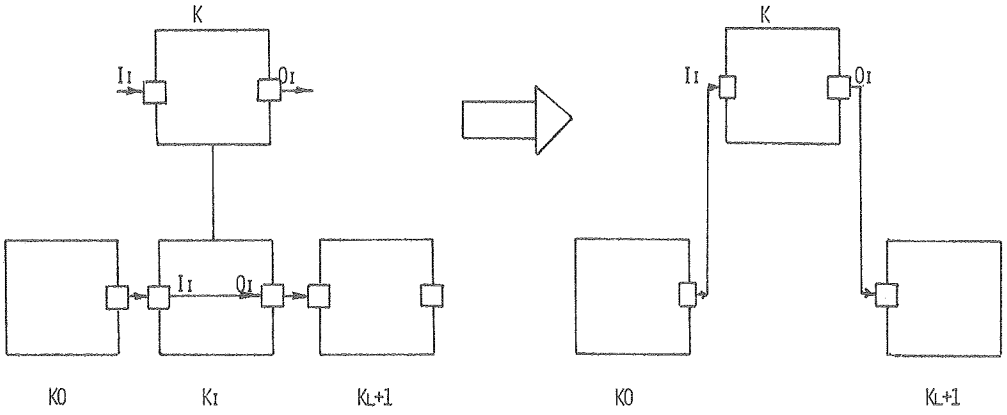


Figure 4 - Operation 2 in Theorem 2

a. There is an arc from Oi to Ii+1 for i=1 to (l-1) in IOP(K) since the controller K is the lowest common ancestor between K1, K2, ..., Kl. See rule R2.1 of the protocol.

b. There is an arc from Ii to Oi, for i = 1 to l in IOP(K) since every time a connection between an

0 0 0 0 5 1 0 4 1 4 7

input and an output port is created it is propagated to its father. See rules R1.1.2 and R3.2 of the protocol.

c. Since the controller K0 is not a son of controller K the input port I1 is also an input port of K which is in the path between K0 and lca(K1,K0). See rule R2.3 of the protocol. The same observation applies to controller K1+1 and the output port O1 (see rule R2.2 of the protocol).

The reader should notice that operation 1 preserves the lowest common ancestor between all the controllers involved in the cycle.

The validity of operation 2 follows from observations b and c above. Notice that the choice of a controller at the lowest possible level to substitute during operation 2 preserves the lowest common ancestor between the controllers involved in the cycle. The reader can easily convince himself that this must be the case.

Therefore when all the labels are the same we have a representation of a condensed version of the original cycle. The common label is the name of the controller where the cycle is completely represented. Since operations 1 and 2 preserve the lowest common ancestor between the controllers involved in the cycle the original cycle will be represented in a condensed way in lca(LK1, LK2, ..., LKn).

Theorems 1 and 2 together give us a necessary and sufficient condition for a deadlock to be detected in a distributed database involving resources at different sites. This result will be stated as the following theorem.

THEOREM 3: A necessary and sufficient condition for a deadlock involving resources in controllers LK1, LK2, ..., LKn to exist is that there is a cycle in the IOP graph of lca(LK1, LK2, ..., LKn). This cycle is a condensation of the corresponding cycle in the global TWF.

Hierarchy Establishment

So far we have assumed the existence of the hierarchy without considering how it is established in the first place. The performance of the hierarchical protocol, in terms of the overhead message traffic incurred by it, can be minimized if the hierarchy is appropriately chosen. This choice should consider the pattern of the DB traffic with respect to the locality of access to a controller or group of controllers. For instance, assume that one is able to

identify groups or clusters of leaf controllers such that a high percentage of the DB traffic involves controllers in the same cluster while very little traffic is of the inter cluster type. One possibility here would be to assign non-leaf controllers to each cluster and try to further cluster the non-leaf controllers or put all of them together under the same root.

The problem in general can be stated as follows. Given a set of leaf controllers, assigned to the nodes of a computer network, given the DB traffic pattern and given the cost of sending messages between every pair of nodes in the network, find a hierarchy which minimizes the total cost incurred in using the protocol.

There are clearly some heuristic rules which if applied to a given hierarchy result in another hierarchy of less cost. The general optimization problem, however, is the subject of current research effort.

A DISTRIBUTED LOCKING AND DEADLOCK DETECTION PROTOCOL

A locking protocol which uses distributed control and a distributed deadlock detection mechanism is presented here. Before we describe the protocol, some definitions are in order. Each database site controls a set of resources. Transactions request resources by sending their requests to the controller of the resource. Each controller is responsible for:

1. processing lock and lock release requests for local resources. Requests may originate from any node in the network.

2. building a simplified version of the transaction_wait_for graph and detecting deadlocks. The TWF maintained by a controller is a subgraph of the global TWF.

For the purpose of stating this algorithm we will use a simplified version of the transaction_wait_for graph. The reader should be aware that this version is a redefinition of the graph used in the section on the hierarchical protocol although the same name for the graph is retained. In this version there is no notion of transaction incarnations. Nodes are associated with transactions and there is a directed arc from transaction T' to transaction T'' if T' is blocked and must wait for T'' to release a resource (not necessarily a resource needed by T') before T' is able to proceed.

0 0 3 0 5 1 0 4 1 4 8

Some definitions are in order. A non-blocked transaction is a node in the transaction wait for graph with no outgoing arcs or a sink node. Let us now define blocking set(T) as the set of all non-blocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction T. This is the set of transactions which are ultimately blocking transaction T. The pair (T,T') is said to be a blocking pair of T if T' is in blocking_set(T).

The execution of a transaction can be described as follows. A transaction has a site of origin which is the site where the transaction entered the system. The transaction starts running at this site, performing local operations until operations on non-local data are necessary. Then, a lock request in the appropriate mode is built and sent to the controller for the requested resource. This controller will either accept or reject the lock, sending the reply to the site of origin of the transaction. If there are multiple copies of data, lock requests have to be sent to all controllers which keep a copy of the data.

Distributed Protocol - A Description

Let Sorig(T) be the site of origin of transaction T and let TWF(K) be the TWF graph at site Sk. The protocol is described by rules 1 and 2 given below as carried out by controller Sk. Let T be a transaction requesting resource R.

RULE 1: The resource R cannot be granted to T because it is being held by transactions T1, T2, ..., Tk.

Add an arc from transaction T to each of the transactions in the set {T1, T2, ..., Tk}. If the addition of these arcs caused a cycle to be formed in TWF(K) then a deadlock was detected and an appropriate action is required for its resolution. For each transaction T' in blocking_set(T) send the blocking pair (T,T') to Sorig(T) and to Sorig(T') if Sorig(T) ≠ Sorig(T').

RULE 2: A blocking pair (T,T') is received.

Add an arc from T to T' in TWF(K). If a cycle was formed then a deadlock exists and it must be resolved by an appropriate action. If T' is blocked and Sorig(T) ≠ Sk then for each transaction T'' in the blocking_set(T) send the blocking pair (T,T'') to Sorig(T'') if Sorig(T'') ≠ Sk.

Some comments are in order.

a. the arcs of the transaction_wait_for graph considered here may represent one of two types of relationships between transactions, namely a direct wait and an indirect wait. Transaction T1 is said to be waiting directly on T2 if the resource needed by T1 for its continued execution is being held by T2. A transaction T1 is said to be waiting indirectly on Tk if there is a set of transactions T2, T3, ..., Tk-1 such that Ti is waiting directly on Ti+1 for i = 1 to k-1.

b. from the previous observation it can be seen that cycles in a TWF graph may be a condensation (in the sense defined in the section on the hierarchical protocol) of the cycle that would exist in the global transaction_wait_for graph for the whole system.

Distributed Protocol - Plausibility Argument

That the protocol described in the previous section is able to detect all deadlocks is shown in the following theorem.

THEOREM 4: The above described protocol detects all possible deadlocks.

Proof: In order to show this result we will consider a global deadlock cycle, as shown in figure 5, and we will show that this cycle will appear in the TWF of the site of origin of at least one of the transaction in the cycle.

There are many orderings of resource requests that can lead to the same deadlock cycle. Each time a request is rejected the newly formed blocking pairs will increase the knowledge that some controllers have about the global graph. Therefore, the bigger the advance knowledge obtained when a request is rejected, more rapidly the deadlock will be detected. The ordering of resource requests used in this proof is such that controllers will get the minimum possible knowledge of the rest of the graph when a request is rejected. The reader should not have any problem in convincing himself that if the theorem holds for this ordering then it holds for any other. The chosen ordering is the following. Initially, transactions T1 through Tk-1 are blocked and each of the controllers at the site of origin of these transactions have the knowledge of a single transaction ahead in the cycle. At this point the transaction_wait_for

0 0 3 0 5 1 0 4 1 4 9

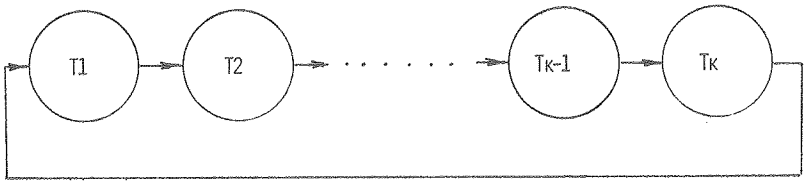


Figure 5 - Deadlock Cycle Involving Transactions T1, T2, ..., Tk.

graph at the site of origin of each transaction is shown

```
Sorig(T1)     : T1 -> T2
Sorig(T2)     : T2 -> T3
   .
   .
   .
Sorig(Tk-1)   :  Tk-1 -> Tk
```

below.

Now, when Tk makes a request and is blocked by T1, the blocking pair (Tk,T2) will be sent to Sorig(T2) where a new blocking pair (Tk,T3) is formed and sent to Sorig(T3). There the blocking pair (Tk,T4) is formed and sent to Sorig(T4) and so on until the blocking pair (Tk,Tk-1) reaches Sorig(Tk-1). This causes the arc from Tk to Tk-1 to be added to the TWF graph at that site. Since this graph already contained an arc from Tk-1 to Tk a cycle is formed and the deadlock is detected.

CONCLUSION

This paper presents two solutions to the problem of deadlock detection in distributed databases. The first solution consists of a hierarchy of lock controllers and is

0 0 : 0 5 1 0 4 1 5 0

CONCURRENCY CONTROL AND CONSISTENCY OF MULTIPLE COPIES OF DATA IN DISTRIBUTED INGRES

by

Michael Stonebraker

Electronics Research Laboratory
University of California, Berkeley

ABSTRACT

This paper contains algorithms for ensuring the consistency of a distributed relational data base subject to multiple concurrent updates. Also included are mechanisms to correctly update multiple copies of objects and to continue operation when less than all machines in the network are operational. Together with [EPST78] and [RIES78], this paper constitutes the significant portions of the design for a distributed data base version of INGRES.

I INTRODUCTION

This paper contains a collection of algorithms to handle concurrency control, crash recovery and the update of multiple copies of relations in a distributed data base system. Together with [EPST78] and [RIES78] this suggests the heart of the design for a distributed version of the relational data base system, INGRES. Earlier thoughts on the same subject are presented in [STON77, WONG77].

The algorithms presented herein are based on what might be called the "primary site" model [ALSB76, STON77] for copies. As such, these algorithms borrow notions from the work of [ALSB76]. Hence, each object possesses a known

---

Research sponsored by Naval Electronics Systems Command Grant #N00039-78-C-0013 and National Science Foundation Grant MCS75-03839-A01.

primary site to which all updates in the network for that object are first directed. It should be clearly noted that different objects may have different primary sites.

In our environment an object is a subset of the rows of a relation (a so called fragment [ROTH77]). Consequently, a relation is partitioned into fragments, each with a primary site and some number of redundant copies. The syntax by which fragments, primary sites and copies are specified is given in [RIES78].

Concurrency control is, of course, a problem to contend with. Unlike [MENA78], who proposes a centralized "GOD" called the lock controller to maintain lock tables and detect and resolve deadlock, we propose to handle locking in a distributed fashion. Deadlock detection and resolution can also be distributed, but we propose allocating these tasks to one machine called the SNOOP. It will be seen that extremely low lock overhead can be achieved using our scheme.

Lastly, crash recovery entails two tasks:

1) handling the failure of a single node in the network and
2) dealing with a failure that partitions the network.

It will be seen that both situations can be handled by very simple algorithms.

In the next section we point out some fundamental assumptions we have made and explain the environment in which distributed data management takes place. Then, in Sections III and IV we present the algorithms for concurrency control and consistency of multiple copies respectively. Lastly, in Section V, we examine our proposal in light of those of [BERN77, MENA78, ROSE77, THOM75].

Throughout this paper we only sketch our algorithms and make no attempt at formal proofs. Arguments along the lines of [MENA78, LAMP76] can be applied to our situation without difficulty, but for the sake of brevity, they are omitted.

II THE ASSUMED ENVIRONMENT

Our work is based on the four following basic assumptions. (The reader can think of these as global assumptions

0 0 0 0 5 1 0 4 1 5 1

about the "state of the world". For environments where these are not true, our proposals may not be appropriate.)

1) Locality

At least 95 percent of the traffic to be processed by a distributed data base system is LOCAL. By this we mean that all data to successfully process an interaction is resident at the site where the interaction originates. Moreover, this statement holds even if there are no redundant copies of objects.

We make this assumption based on our belief that other architectural choices (namely centralization or tightly coupled networks with a notion of "GOD") become cost effective when traffic is not predominantly local.

2) Reliability

Computer hardware (and perhaps software) is becoming more reliable. Hence, when possible, mechanisms should make normal processing faster at the expense of lengthened crash recovery time.

3) Closeness

The main performance consideration when processing an interaction is whether it is LOCAL or not. Non local interactions are not expected to be scaled on "closeness", and the notion of a "nearby" site is not assumed to be meaningful. This is a reasonable approximation of most real networks that we are familiar with.

4) Types of networks

The algorithms should work well for both "broadcast" networks and "point-to-point" networks. Hence, they should be able to utilize the capabilities of a network where the cost to send a message to all sites is almost equal to the cost to communicate it to one site. Likewise, the algorithms should also perform well if only point-to-point communication is possible. To a first approximation, we think of the ETHERNET [METC76] and the ARPANET [ROBE70] as generic examples of these two cases.

The algorithms we propose must exist within the context of the query processing done by a distributed INGRES in handling transactions. We now give necessary information on distributed query processing.

A transaction, T, in our environment consists of ONE QUEL command which is either a RETRIEVE or an UPDATE (APPEND, DELETE, REPLACE). There is no extra algorithmic complexity in handling the more general case where a transaction consists of an arbitrary collection of such commands. The only problem is that crash recovery will be decomposed into local crash recovery followed by some extra processing. Current INGRES one-site crash recovery software can only handle the smaller transactions, and we do not plan to rewrite it to be more general.

In the discussion to follow, the reader should note that while we present the restricted context, our algorithms will work for the general case if one pretends that the local notions of "commit transaction" and "recover transaction" handle the general case for a one-site environment.

Such a transaction originates from a user process (application program or interactive terminal monitor) at some site, i, in the network. A "master" or coordinating collection of INGRES processes is invoked at site i. This master creates "slave" INGRES processes at other sites and ensures that each slave knows the identity of all other slaves. Although slaves need be created only at sites where processing will take place, it may be convenient to pretend that they exist at all sites. Hence, we make no further note of creating or destroying slaves. As discussed in [EPST78], a master can send two commands to slaves:

1) run the (local) interaction, I, at a subset of the sites.
2) send a copy of fragment $R sub j$ to a subset of the sites in the network.

The slaves eventually return a "done". By a sequence of such commands, as shown above, the master ensures that the user interaction is eventually processed. The details on how such a sequence is generated appear in [EPST78].

If T is a RETRIEVE command then it can be equally thought of as a sequence $I sub 1$,...,$I sub n$ of

0 0 : 0 5 1 0 4 1 5 2

interactions where each $I sub j$ is either a local RETRIEVE command or a SEND command to be executed on a subcollection of the sites.

If T is an update, the master first generates a RETRIEVE command to figure out what to do. This command is processed as above and (eventually) generates a deferred update list of the changes to be made. As noted in [EPST78], these lists of updates may not always be at the site where the tuples to be updated reside. Consequently, the lists may have to be appropriately shuffled so that each slave has a complete deferred update list for its site. Since each slave knows the identity of all other slaves, it can simply wait for a message from all other sites containing the appropriate tuples. Alternatively, the master site knows the collection of sites where deferred update is possible. It can send this information to the slaves with some interaction, I. Each slave, in this case, need only expect a message from this subset of sites.

When all is appropriately arranged, a "commit transaction" message must be sent by the master to all slaves. Hence, an update transaction is a sequence of local interactions, $I sub 1$,...,$I sub n$, followed by some shuffling and a "commit transaction". The detailed protocols to be followed on updates will be explained presently.

It should be noted that $I sub 1$,...,$I sub n$ for RETRIEVES and updates is always a collection of local RETRIEVES and SENDS. For updates however, the local RETRIEVES that isolate changes to be made sometimes contain a "tuple identifier" as part of the requested data. In this way, the deferred update list will have an indication of exactly which tuples must be modified. The "tuple identifier" also indicates that the current RETRIEVE is really isolating tuples which will later be changed. Hence, for some local locking algorithms (such as [GRAY76]), these must be considered writes rather than reads.

III CONCURRENCY CONTROL ALGORITHM

If T is a RETRIEVE command, proceed according to step one or step two.

STEP ONE: (high performance but has data integrity problems)

The master directs each interaction, I, to the local copy of each object. If one does not exist, he directs it to the primary copy. Section 4 indicates how the identity of the primary copy is determined.

STEP TWO: (lower performance but no "surprises")

The master directs each I to the primary copy of each object. (We will discuss in Section 4 what the "surprises" are and when step one can be safely chosen.)

Alternatively, if T is an update, the master must direct each I to the primary copy of the object without exception.

Note that the choice of step one or step two will affect whether a particular interaction is local or not. Hence, it may affect the sequence of interactions which is chosen by the master during distributed decomposition.

A local concurrency controller (cc) runs at each site. This cc sees a transaction, T, as consisting of a collection of interactions, $I sub 1$,...,$I sub n$, EACH LOCAL TO THAT SITE which it receives one by one from the master. It is assumed that level 3 consistency [GRAY76] for such a transaction is satisfied. How cc performs this task is of no concern here. All that matters is that the response to each interaction, I, which a master sends to a slave, is a "done" (indicating that it is complete) or no response (indicating that it is either still processing or is waiting for permission from the cc to proceed further).

Since the sequence of interactions may not be known in advance, deadlock may occur. If it happens within one machine, the cc picks a victim and performs local backout. The appropriate slave sends a "reset" message to his master which in turn resets all other slaves. This accomplishes backing out T.

However, deadlock may happen in such a way that it involves multiple machines. We suggest the following mechanism to deal with this problem. Whenever an interaction is run at a node it will either run to completion or wait, due to a lock conflict. Say transaction Tj holds a lock for

0 0 1 0 5 1 0 4 1 5 3

which transaction Tk waits. When cc detects such a condition, it sends identifiers for Tj and Tk to an agreed on machine (The SNOOP). The SNOOP then detects deadlock by ordinary analysis of the global "wait for" graph, which it can generate from all such messages. When a master completes a transaction, it sends a "done" to the SNOOP, who can appropriately update his "wait for" graph.

In summary, each site has a cc that handles concurrency control for local transactions (using some technique). The lock tables each cc creates and uses are LOCAL to its site and are present nowhere else. Only the SNOOP needs to be informed of "wait for" conditions.

In the absence of crashes, it is clear that such a scheme can work correctly. We now turn to control of multiple copies and crash recovery.

IV COPIES AND CRASH RECOVERY

There are two algorithms to handle copies and crash recovery. One is called the "performance" algorithm. It processes updates with the minimum possible delay (i.e. fastest possible response time). However, it will be seen to have a data consistency problem since transactions can be lost and relations can become inconsistent if certain sorts of crashes happen. In the absence of crashes, however, there are no consistency problems.

This algorithm will require the notion of a "copy INGRES". This is an invocation of INGRES that can receive a complete deferred update list from a slave INGRES for some object, and performs the update on a copy of that object. A copy INGRES must exist for every copy of the object which is not the primary copy.

The other algorithm is called the "reliability" algorithm. It ensures that there is never a data consistency problem. However, there may be a substantial response time penalty in order to guarantee this.

In this section we make the following assumptions:

1) Crashes of a single node can happen. Service is restored after some delay. Local crash recovery procedures will

automatically be run and will guarantee that local transactions, which were "committed" before the crash but whose effects were not completely merged into the local data base, are correctly committed.

2) Communication failures can happen. They will either be transparent or will partition the network into subsets that cannot communicate with each other.

3) Every message sent is reliably received if the recipient is "up" and the sender does not crash inopportunely. If the recipient is down, the sender can queue the message for later delivery. Presumably, it is queued at the sender's site so that subsequent delivery can only happen if the sender is "up".

In this section we will indicate several algorithms that must be run by the various actors under various circumstances. The data structures used in the algorithms are the following.

4.1 Data Structures

Each machine in the network maintains the following information:

1) a state flag

This is either "normal", indicating normal operation, or "restructure", which means the network is dynamically adjusting to a crash or restoration of service.

2) an up-list

This is a list of sites that the given node thinks are in operation. These sites form a "logical partition" [MENA78] of the network that may lag in time behind the actual physical state of the network.

3) the identity of the current SNOOP

This is an identifier for the machine that performs deadlock detection. If the up-list changes in such a way that the current SNOOP is not in the up-list, there is a globally agreed upon procedure for choosing a new SNOOP.

Moreover, each master INGRES must have:

4) a commit flag

This flag is set by the master when it thinks that a transaction is committed. It is reset otherwise.

Each slave INGRES must have:

5) a ready flag

This flag is set if the slave has the proper deferred update list and is ready to commit a transaction. It is reset otherwise.

6) a commit flag

This is set when the slave believes that his transaction is committed and is reset otherwise.

Lastly, the location of all copies for each object is known (by an examination of some system catalogs whose composition will not concern us here). In addition, there is a known linear ordering of all these copies. As a result, any site can always access its up-list and find the particular copy which is lowest in the ordering among those at sites in the up-list. This is by definition the PRIMARY COPY of the object.

Now, if the network is currently partitioned, nodes in different partitions will disagree on the identity of the primary copy. Obviously, chaos would result if more than one partition was allowed to independently update a given object, since there would be no possible way to make the objects consistent when the network was repaired. The easiest way to avoid this situation is to allow a primary copy to exist ONLY if a majority of all copies are at sites in the up-list. Otherwise, the object is inaccessible.

However, it should be clearly noted that a problem arises in the case where there are exactly two copies of an object. In this (presumably very common) situation, any network partition will make both copies inaccessible. To handle this, one must be able to tell the difference between:

a) a crash of a single site and

b) a network partition that leaves each copy in a different partition.

In the former case we wish the backup copy to become the primary, while in the latter case we do not. If these two situations cannot be differentiated, the backup copy is worthless.

There are six algorithms to be presented. In order to assist the reader we first outline the general flow of control before proceeding with the details.

ALGORITHM MASTER, SLAVE and COPY are executed respectively by master, slave and copy INGRESes. The key point is that a master has a "commit point" when it sets its commit flag. Before that point, it will back out a transaction if a failure occurs. After that point it will ensure that the transaction is correctly committed. The tactics used amount to a "two phase" commit protocol [GRAY77] and are similar to the proposal of [LAMP76].

Each slave responds to directives from the master. It uses the local commit flag to indicate whether it thinks that the transaction is committed. On crashes, local recovery examines the flag to decide whether to backout or complete the local transaction. When a slave commits a transaction, it in turn directs copy INGRESes to correctly update the auxiliary copies.

The final 3 algorithms are run in the context of failures. ALGORITHM LOCAL RECOVERY performs "local cleanup" and is run when a site wishes to resume service. ALGORITHM RECONFIGURE is used to adjust the up-list after a failure or a service restoration. It allows the network to continue with less than all sites operational or even in a partitioned state. Lastly, ALGORITHM SLAVE PROMOTE is run when a master crashes. Its purpose is to allow the slaves to correctly finish or backout the transaction in question.

4.2 The Performance Algorithms

The reader is again cautioned that data integrity problems exist in the algorithms which are now presented.

ALGORITHM MASTER

0 0 1 0 3 6 1 0 4 1 5 5

This is the algorithm to be used by a master INGRES when it receives a an update transaction, T, from a user process.

m1) refuse T unless state = "normal"

m2) Examine the up-list and calculate the primary site for each object involved in T. Refuse T if the primary site of any object is inaccessible.

m3) Supervise distributed decomposition for T and coordinate getting a correct local deferred update list (du) to each slave.

m4) Wait for a "I am ready" message from each slave. If not all respond, go to m8).

m5) Set commit flag. Send a "commit" message to all slaves. Send a "done" to the user process.

m6) Wait for a "done" from all sites. If not all respond, go to m9).

m7) Send a "done" to the application program (if it is more convenient to wait until master processing is complete). Send a "done" to the SNOOP. Go to m1).

m8) Send "reset" to all slaves, the user process, and the SNOOP. Queue this message for later delivery to the non responding machines. Send a "reconfigure" message to each site. Go to m1).

m9) Queue a "commit" message and a "potential trouble" message for the down machine(s). Send a "reconfigure" message to each site. Go to m1).

ALGORITHM SLAVE

This algorithm is to be run by each slave which assists a master in processing a transaction, T.

s1) Execute interactions at the request of the master and

assemble a deferred update list (du) for the local site. When it is correct, set the ready flag.

s2) Send "I am ready" to the master.

s3) Wait for "commit" or "reset"

s4) If a "commit" is received, then set local commit flag. Commit the transaction. Respond "done" to the master when completed. Spawn a process to execute steps s6) to s8). Go to s1).

s5) If a "reset" is received, run ALGORITHM LOCAL RECOVERY. When complete, go to s1).

s6) Send (du) to each copy INGRES for the object being updated that is operational. Queue (du) for later delivery to non responding sites.

s7) Wait for a "done" from each copy INGRES. If not all respond, queue a (du) message for the non respondants and send a "reconfigure" message to all sites.

s8) Terminate.

ALGORITHM COPY

This algorithm is run by each copy INGRES that is assisting in processing a transaction, T.

c1) Wait for (du).

c2) Perform update and respond "done"

ALGORITHM LOCAL RECOVERY

This algorithm must be run when a site restores service or when a transaction wishes to "reset".

r1) Read all outstanding (i.e. queued) messages and perform appropriate local actions.

0 0 - 0 5 1 0 4 1 5 6

r2) If a local commit flag is set (either for the transaction in question or all transactions), commit the update. If not, correctly back out the transaction. Techniques for doing r2) for "soft" crashes, (i.e. ones in which data is not lost from secondary storage), are well known [STON76, ASTR76, LAMP76] and easily implementable. A more elaborate (and tedious) restoration must be done for hard crashes where data may be lost.

r3) If this is a service restoration, then proceed, otherwise, send a "done" to master and terminate.

r4) Send a "reconfigure" message to all sites.

r5) Execute ALGORITHM RECONFIGURE.

ALGORITHM RECONFIGURE

This algorithm must be executed whenever a "reconfigure" message is received by a site. Its purpose is to dynamically alter the up-list.

f1) set state = "reconfigure". Instruct each master INGRES at the local site to backout or complete the current transaction. Go to f2) when the site is quiescent.

f2) Send "I am up" to all sites.

f3) Wait for a predetermined interval for replies and set the up-list equal to the set of all respondants.

f4) If current machine is the lowest in a predetermined ordering of all machines in the up-list, go to f5); otherwise, go to f8).

f5) Send up-list to all sites in the up-list.

f6) Wait for "I agree" from everybody. If any do not respond (or respond with "disagree"), send a "reconfigure" message to each site and go to f1).

f7) Set state = "normal". Send "normal" to all sites on the

up-list and go to f11).

f8) Wait for up-list from some other site.

f9) Compare with local up-list and respond with "agree" or "disagree".

f10) Wait for "normal". When received, set state = "normal".

f11) If current SNOOP is in the up-list, resume normal operation. Otherwise, calculate the new SNOOP by the prearranged algorithm and send local conflict graph to that site. Resume normal operation.

It should be noted that step f1) will take a varied amount of time and an operational site may not send an "I am up" message within the "predetermined interval" of step f3). This problem can be handled by a straight forward (but tedious) extension of the algorithm.

It should also be noted that various multiple crash conditions can leave all up sites at step f8) in the algorithm. This can be detected and handled by a timeout and algorithm restart mechanism.

ALGORITHM SLAVE PROMOTE

This algorithm is executed by any slave that cannot communicate with its master.

p1) invoke execution of ALGORITHM RECONFIGURE

p2) Wait for state = "normal".

p3) Examine up-list to see whether all other slaves are up. (If not, there may be a data consistency problem which is discussed in the sequel).

p4) If the slave is the lowest one in a predetermined ordering of the slaves in the up-list, go to p5); otherwise, go to p9).

0 0 : 0 5 1 0 4 1 5 7

p5) Send "is there a commit flag on" to all other up slaves.

p6) Wait for all replies.

p7) If there are no commits, send a "reset" to all slaves, the SNOOP and the user process. Queue this message to down sites.

p8) If there is one or more commits, send a "commit" to all slaves and go to m6) of the ALGORITHM MASTER.

p9) Wait for "is there a commit" message.

p10) Reply "yes" or "no" and go to step s3) of the ALGORITHM SLAVE.

4.3 Data Integrity Problems with the Performance Algorithms

This collection of algorithms has the following three "features".

1) Transactions may be lost

This situation will normally be detected by a received "potential trouble" message. If a slave crashes after it has set its local commit flag but before it has reliably communicated (du) to any copy INGRES, the following situation arises. No copy of the object in question has received the update. However, ALGORITHM LOCAL RECOVERY will ensure that a service restoration at the crashed site will commit the update. Hence, when service is restored and the network is reconfigured, one copy will not agree with the others because it will have had an extra update performed. This "phantom" is lost until service restoration, at which point it generates a consistency problem.

Of course, the slave can make this possibility very remote. However, there is another serious case to consider. Suppose a slave crashes after the master has set its commit flag. In this case, other slaves associated with the same master will proceed with the update, and the crashed site will not. Moreover, no copy of the object at the crashed site will ever be updated. This generates the following feature.

2) The data base may be inconsistent

According to the above scenario, a user process could be giving a ten percent raise to all system programmers. If system programmers appear at fifteen sites, then fifteen slaves will be processing. A crash of one of them at an inopportune time will leave the update committed at fourteen sites and undone at the other. Hence, some system programmers got the raise and some did not.

3) Messages can get lost

The algorithms all assume that a down machine recovers by (among other things) reading its outstanding messages. However, some outstanding messages may be on machines that are currently down. This may mean, for example, that (du) messages that were queued are not delivered before a machine resumes normal operation. Depending on the composition of the lost messages, the result can be benign or catastrophic.

4.4 Reasons for Tolerating Data Inconsistency

Obviously, these are serious drawbacks. Moreover, it is not clear what a user process can do to deal with these problems. However, there appear to be cases where a user might be willing to tolerate such "surprises". Three examples are suggested.

1) Update transactions always affect only one fragment of a relation.

In this case, problem 2 cannot arise and problem 1 can be easily recovered from. If machines are reasonably reliable, problem 3 may not be an issue. (It only arises if at least two machines are simultaneously out of service.)

2) Almost all updates are local.

With the above algorithms, a "done" can be given to the user process for a local transaction after ONLY local processing. Hence, the response time can be (more or less) the same as for a centralized system. To alleviate any of the above problems would require at least one other machine be signalled before a "done" could be sent to the user process. This may degrade response time substantially (by perhaps an

0 0 0 0 5 1 0 4 1 5 8

order of magnitude, depending on the speed of the network). There may be situations where such a degraded response time is unacceptable.

3) Almost all updates can be decomposed into a sequence of interactions, each of which is local to some machine. Again, response time may be substantially degraded by any mechanism dealing with the above problems.

As a result, a data base administrator can decide if better response time is worth the above headaches. For administrators who answer "no", we offer a collection of much more reliable (and expensive) algorithms.

4.5 The Reliable Algorithms

The previous algorithms may be made more reliable by making the following changes:

1) Treat each copy INGRES identically as the slave for which it is updating a back-up copy. Hence, each copy INGRES will receive a correct (du) before a transaction is committed. All messages which are sent to slaves are also sent to copy INGRESes.

2) ALGORITHM COPY is no longer necessary.

3) Steps s6) to s8) of ALGORITHM SLAVE are no longer necessary.

4) At every step where the instruction to queue a message for a process at a non responding site occurs, change it to queue the message at K sites, K>1.

With these modifications it is easy to show that all three problems mentioned above are avoided if less than K machines are down at any one time and if there are at least K copies of each object.

The cost of this scheme is, of course, that a "done" cannot be signalled to the user process until a correct (du) is received reliably at ALL sites where a copy resides. Clearly, substantial network traffic (and delay) will be required to perform this.

It is obvious that there is no possible scheme which can guarantee that there will be no "surprises" if K-1 machines crash, unless that scheme ensures that any update is reliably received at K sites before commiting the update. Whether users are willing to pay the cost in response time for such a guarantee is unknown.

4.6 Accesses to the Local Copy

We are now in a position to discuss when a RETRIEVE can be reliably directed to the local copy (step one from Section 3). Clearly a RETRIEVE can be directed to the local copy of an object (step one) if it accesses only one fragment of any relation. If a RETRIEVE accesses multiple fragments of the same relation, then the possibility exists that the local copies of each fragment are not consistent. This will happen if there is an update which spans multiple fragments (such as the previously discussed ten percent update to all system programmers) and some local fragments have received the update and the others have not (since copies are updated asynchronously). This will be termed the "multiple fragment read problem".

Clearly, a multiple fragment RETRIEVE may yield an erroneous answer and there is no way to guard against this possibility (except by synchronizing the updating of copies of fragments). However, since it is a performance winner to access local fragments if possible, we plan to allow the user to set a flag indicating that he wishes to "take a chance" and use the local copy for RETRIEVE commands.

V COMPARISON WITH OTHER SCHEMES

We now examine our scheme in comparison with the updating schemes proposed by [BERN77, MENA78, ROSE77, THOM75].

In [MENA78] a centralized concurrency control scheme is proposed. Moreover, to successfully handle arbitrary crashes, the lock table for this centralized controller must be fully redundant (i.e. present at each of N sites). Therefore, each interaction, I, in our model - using their scheme, requires 3*N-1 messages to correctly alert all N copies of the lock table. Our scheme only requires 1 message (the interaction I) to be sent to a SUBCOLLECTION of

0 0 - 0 5 1 0 4 1 5 9

the sites. Moreover, this message must be sent anyway. Only if a "wait for" condition is generated must a second message be sent to the SNOOP. Moreover, equal robustness on crashes is achieved by our scheme with far less messages. In addition, our recovery algorithms are much simpler because the lock table is not redundant (and hence cannot be inconsistent with copies of itself).

[ROSE77] proposes a concurrency control scheme that avoids using the SNOOP by having timestamps on transactions to resolve deadlock in a distributed fashion. However, that scheme may abort a transaction without a deadlock existing. Our scheme, on the other hand, backs out a transaction only if a deadlock actually occurs. Whichever scheme has less overhead is application dependent, and a simulation model (perhaps along the lines of an extension to [RIES77]) would have to resolve the issue.

In [THOM75] a scheme is presented for updating multiple copies of objects based on majority voting. In this proposal, a "done" cannot be given to a user until n/2 copies are serially updated. Only when this number is achieved can the transaction be sure that it will not be backed out due to conflict. Clearly, our performance algorithms generate a "done" much sooner. However, our reliability algorithms (in essence) require all copies to be updated before a commitment is signalled. On the other hand, we deal robustly with arbitrary crashes. [THOM75] does not contain a crash recovery proposal (and would have to be extended to do so). Without adding crash recovery procedures to the algorithms of [THOM75], there is nothing more that we can say.

We now consider the proposal of [BERN77]. In environments where traffic is not predominantly local, it appears to be a desirable (but very complex) solution. Our scheme, on the other hand, is directed toward environments where there is high locality (assumption 1 from Section II).

Without giving the details of [BERN77], it should just be noted that it proposes doing conflict resolution (if possible) at data base design time. If transactions do not conflict, each can update the copies of an object in any order. The preferred order is to update the local copy first and give a "done" to the user process when this is complete. Hence, excellent response time can be guaranteed.

Transactions that cannot be shown to be non conflicting do not have this freedom. For such transactions more extensive coordination must be done. In general this will require communicating with all sites where copies reside, although in fortuitous circumstances a "luck out" may happen and a site will send the right sort of a message without being requested to.

Comparing our scheme with that of [BERN77] we first presume that an equal number of copies of objects exist at the same sites for either situation. Also, we note that the algorithms in [BERN77] are not robust on machine crashes. Providing such a feature will require additional mechanisms. Hence, it appears appropriate to compare [BERN77] with our performance algorithms, which also fail to make robustness guarantees. Lastly, [BERN77] avoids the "multiple fragment read problem" mentioned earlier by requiring that a transaction only send ONE write message to each site (p 13). This will ensure that all fragments of objects at a given site are updated atomically. This requirement either substantially reduces the potential parallelism with which a transaction can be executed or requires extra synchronization effort. In any case, this condition can be equally well guaranteed by our proposal. It should be noted that assuming this notion allows us to direct RETRIEVE transactions to the local copy of every object.

The comparison to follow is based on response time only. The actual overhead of the two schemes depends on what is counted, is stochastic in nature, and would require additional study to quantify. The comparison is based on the following cases:

1) The transaction is a RETRIEVE and there is a local copy of each referenced object.

Here both proposals can access the local object and response time is the same either way (assuming that both proposals synchronize multiple fragment updates as discussed above).

2) The transaction is a RETRIEVE and 1) is not satisfied.

In either case, a non-local copy is accessed. By our assumptions concerning closeness (assumption 3), it does not

matter which copy is used, and the delay is the same.

3) The transaction is an update and the primary copy of all objects is local, and the transaction is one which can be preanalyzed by [BERN77] as a "non conflict" one.

Both algorithms will generate a "done" after local processing only.

4) The transaction is an update and its primary copy of all objects is local and it is not in the "non conflict" category.

Here, [BERN77] requires a message be sent to each site and a response received (unless there is a "luck out"). Our scheme requires only local processing.

5) The transaction is an update and is "non conflicting" but is not local.

Here, [BERN77] requires no messages (but only if there is a local copy of all objects). Our scheme requires communicating with the primary site of each non local object.

6) other cases

This is the same as 4) for [BERN77] and 5) for our proposal.

Suppose X percent of the update transactions are local for our algorithm and Y percent are "non conflicting" for [BERN77]. Figure 1 indicates the number of sites which must be communicated with for each scheme in cases 3)-6). Here, c represents the number of copies of an object which are present in the network.

CASE	PROBABILITY	NUMBER OF SITES TO BE COMMUNICATED WITH PER OBJECT ([BERN77])	NUMBER OF SITES TO BE COMMUNICATED WITH PER OBJECT (OUR PROPOSAL)
3	XY	0	0
4	X(1-Y)	c	0
5	Y(1-X)	0	1
6	(1-X)(1-Y)	c	1

A Comparison of Our Scheme With [BERN77]
Figure 1

CLearly, [BERN77] will win only if the following inequality holds:

$$Y > 1 - \{ (1-X)/c \}$$

By assumption 1 concerning locality, $1-X < 0.05$. Hence, if $c = 3$ then $Y > 0.983$ in order for our proposal to lose. Of course, the actual value for Y is application dependent. To support the plausibility of a high Y, an example is presented in [ROTH77a] where Y=0.9925. However, for the identical example an interested reader can ascertain that $X = 0.9925$ also. A cursory examination of the above equation indicates that our proposal wins for the case Y=X.

REFERENCES

[ALSB76] Alsberg, P. A. and Day, J. D., "A Principle for Resilient Sharing of Distributed Resources," Center for Advanced Computation, University of Illinois, Urbana, Iii, 1976.

[ASTR76] Astrahan, M. M. et. al., "System R: A Relational Approach to Database Management," TODS 2, 2, June

0 0 3 5 1 9 4 1 6 1

1976.

[BERN77] Bernstein, P. A. et. al., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Data Bases," Computer Corp of America, Cambridge, Mass., December 1977.

[EPST78] Epstein, R. S., et. al., "Distributed Query Processing in a Relational Data Base System," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May, 1978.

[GRAY76] Gray J. et. al., "Granularity of Locks and Degrees of Consistency in a Shared Data Base," IBM Research, San Jose, Ca., RJ 1849, July, 1976.

[GRAY77] Gray, J., "Notes on Data Base Operating Systems," unpublished course notes, July 1977.

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed System," Xerox Palo Alto Research Center, 1976.

[MENA78] Menasce, D. A. et. al., "A Locking Protocol for Resource Coordination in Distributed Systems," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May 1978.

[METC76] Metcalfe, R. M. and Bogs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM 19, 7, July 1976.

[ROBE70] Roberts, L. and Wessler, B., "Computer Network Development to Achieve Resource Sharing," Proc. 1970 Spring Joint Computer Conference, 1970.

[RIES77] Ries, D. and Stonebraker, M., "A Study of the Effect of Locking Granularity in a Relational Data Base System," TODS 3, 3, September 1977.

[RIES78] Ries, D. and Epstein, R., "Specification of the Distribution Criteria for a Distributed Data Base

System," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo #M78-25.

[ROSE77] Rosenkrantz, D. J. et. al., "A System Level Concurrency Control for Distributed Database Systems," Proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.

[ROTH77] Rothnie, J. B. and Goodman, N., "An Overview of the Preliminary Design of SDD-1: A System for Distributed Data Bases," Proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.

[ROTH77a] Rothnie, J. B. et. al., "The Redundant Update Methodology of SDD-1: A System for Distributed Databases," Computer Corporation of America, Cambridge Mass., June 1977.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON77] Stonebraker, M. and Neuhold, E., "A Distributed Data Base Version of INGRES," Proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1976.

[THOM75] Thomas, R. H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Use Distributed Control," BBN Report #3340, July 1975.

[WONG77] Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Data Bases," proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.

0 0 - 0 5 1 0 4 1 6 2

ALGORITHMS FOR DISTRIBUTED DATA-SHARING SYSTEMS WHICH USE TICKETS

Gérard LE LANN

IRIA, Projet SIRIUS - BP.105
78150 LE CHESNAY, France

Abstract

Data coherence and resiliency are natural requirements for a distributed data-sharing system (DDSS). Achieving data coherence implies that constraints of data internal integrity and mutual consistency for multiple copy data are met.

How this can be done depends on the organization of the system under consideration. First, a description of two DDSS architectures is given. Then a solution to the problem of maintaining data coherence in a DDSS is presented.

This solution is based on the utilization of a ticketing scheme which achieves a distributed and sequential identification of requests entering the system. This scheme is shown to survive failures.

Four different algorithms are derived from this scheme, each of which allowing for a different degree of parallelism within the system.

INTRODUCTION

A distributed data-sharing system (DDSS) is assumed to include four ingredients :

- a database being dispersed over several storage devices
- a set of processors, called storage processors, responsible for physical data handling on these devices
- a set of processors, called controllers, in charge of running tasks submitted by users which intend to access the database
- a transmission medium.

The database is viewed as a collection of data units which can be accessed individually. We do not make any assumption regarding the size of these units.

Analysis of user tasks results in the identification of user requests being expressed in terms of database subsets and predicates. Controllers are allowed to read the database at any time, so that they can infer which data units should actually be given to a user. On behalf of users, controllers issue allocation requests, one per data unit. We assume that controllers are provided with a mapping mechanism whereby they can decide which storage processor handles a given data unit.

Storage processors receive allocation requests and store them in waiting queues. There is one waiting queue per data unit.

The transmission medium serves the purpose of carrying messages. Within every processor, communications are handled by a specific piece of software which is specifically tailored to the transmission medium being used. For instance, if a packet switching network is used, that piece of software would be the ARPA NCP or TCP or the CYCLADES TS.

DDSS ARCHITECTURES AND COHERENT DATA-SHARING

We will consider two different architectures (figures 1 and 2). In one architecture, any storage processor can access the whole database ; this is called an integrated architecture or integrated DDSS. In another architecture, the database is split into several partitions and a storage processor can only access one partition. This is referred to as a partitioned architecture or partitioned DDSS.

Specific benefits can be expected from systems including several processors, for example a better potential availability and an improved response time because of a de facto parallelism.

For this to be true, it is necessary to provide all processors with equal rights and equal responsibilities. This is the hypothesis we make for controllers : all controllers are equal.

This means that, at any time, users may interrogate several different controllers and controllers may initiate simultaneously several data updates. We say that control is distributed over controllers.

Coherent data-sharing

Internal integrity. Internal integrity is a concept which can be defined in many ways.

We assume that user tasks can be split into processing units called processes which are such that, if run until completion, they meet the database integrity criterion, to be defined according to semantic considerations. We then limit ourselves to the adoption of a system-related notion of integrity which can be explained as follows : considering a collection of data units to be accessed and modified by a process, we say that internal integrity is achieved for that collection if only one of the following two situations can occur : none of these data units is modified, all data units are modified.

Two different notions of integrity conditions can be derived from this.

When considering various applications, it can be observed that in some cases, it is not necessary to provide a process with the most recently updated version of the data it is using. Indeed, if an "older" version is given to a process Q being run at local time t, this is very similar to a situation where process Q is initiated at local time $t'<t$, t' being such that the latest update has not been performed yet. If Q can ignore that such an update is imminent, all is well. The only requirement is that data units to be updated by another process and to be accessed by Q should reflect either all updates or none of them.

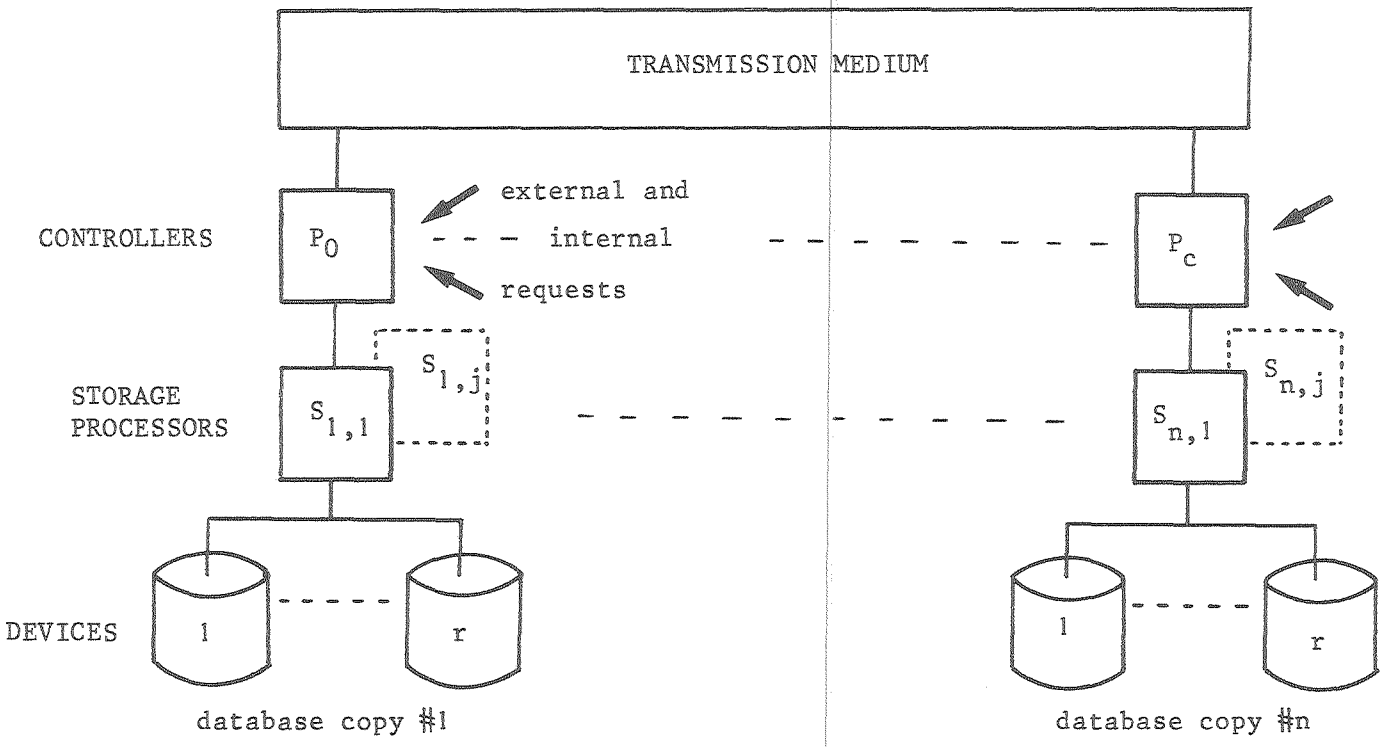


Figure 1. - An integrated DDSS

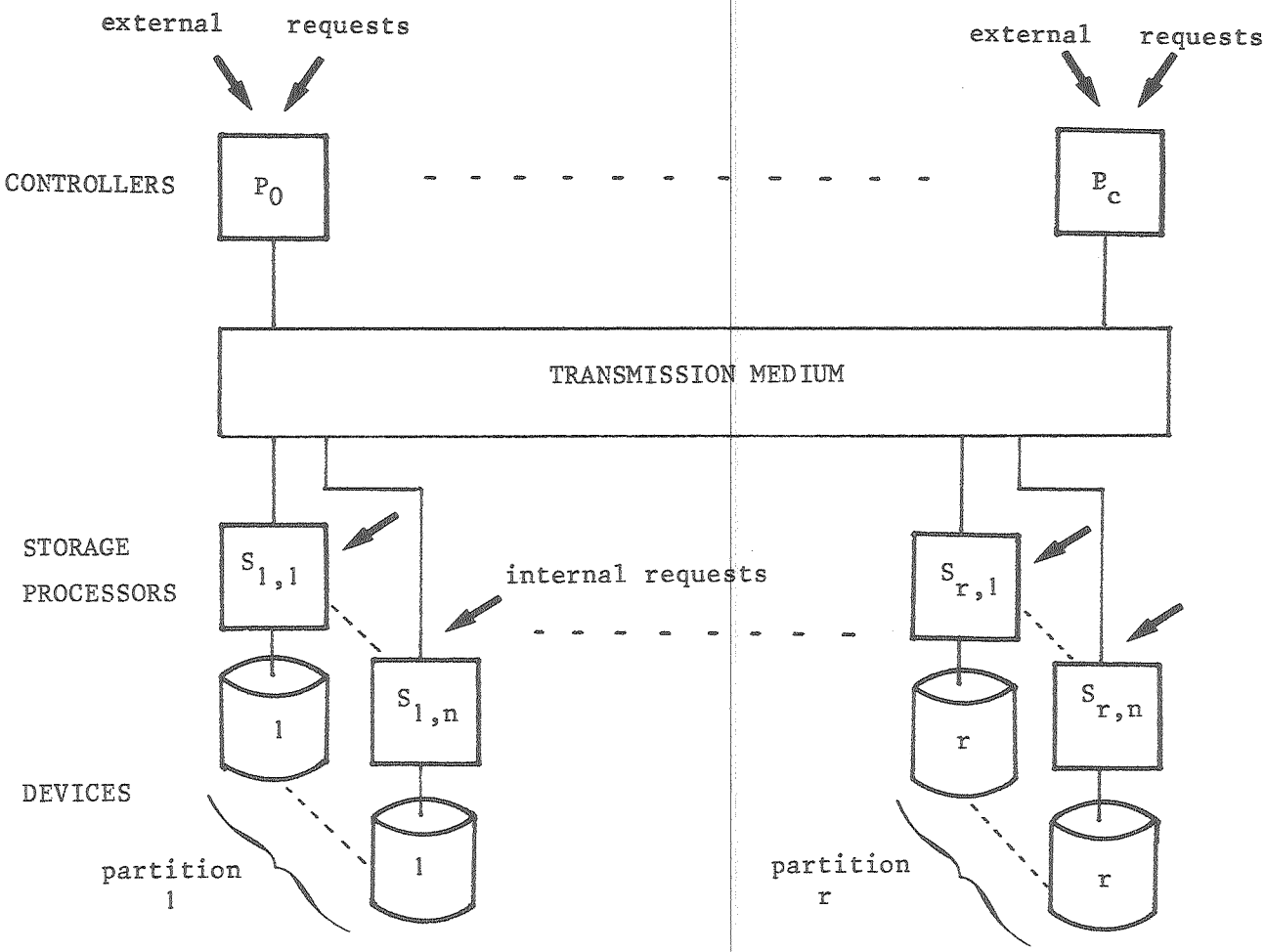


Figure 2 . - A partitioned DDSS

0 0 : 0 5 1 0 4 1 6 4

Such applications will be said to present only weak integrity constraints.

On the other hand, it may be necessary for some applications to make sure that only the most recent version of data they are using is given to processes. Particular measures are needed to provide the system with such a property. These applications will be said to present strong integrity constraints.

Clearly, uncontrolled interferences between processes running concurrently may threaten the database integrity. This should be either avoided or detected. The database integrity may also be threatened if processes do not run until completion. This may happen in the case of failures.

Then, some precautionary measures are needed to protect the database against a process failure. The simplest measure one can think of is to require processes to perform updates only on copies of data units until they reach completion. At that time, updates are displayed by storage processors.

Identical requirements are described in (3). If we can assume also that the system can recover from storage processor failures - this is a problem not to be addressed in this paper - then controllers are always provided with a correct view of the database (integrity is achieved).

Mutual consistency. This is a problem which arises because of the existence of several copies of the database or copies of partitions. An update can be initiated from any controller ; this update must be effected on all copies. Furthermore, it is required for update and retrieval request sequences to be strictly identical for all copies (strong consistency).

This problem bears some relation with the problem of internal integrity to the extent that mutual consistency may be threatened if a given sequence of requests is not properly ordered for at least one copy. Decisions for processing a request or delaying it must be coherent system-wide.

Coherent data-sharing is said to be achieved if and only if internal integrity and mutual consistency requirements are met.

These problems have received much attention in the past few years. Authors have adopted different approaches which can be classified (may be arbitrarily) as follows :

- explicit utilization of physical time (7)
- voting and utilization of timestamps, i.e. logical time (11), in conjunction with physical time control for the purpose of efficiency (2)
- explicit and resilient utilization of a general "controller" (1)
- voting and utilization of eventcounts (4).

As explained below, data units can be considered as resources to be allocated to processes. This leads to another approach where data must sometimes be locked at specific points in time.

Data considered as resources

We assume that any operation is allowed on data units. Then, it is necessary to provide processes with the means of locking a specific data unit. For example, this is needed when a process intends to read and does not want another process to modify the data it is reading. Actually, locking of a data unit results in preventing any other process to use this data unit.

A detailed analysis of locks needed in a database environment can be found in (5). Data units may be accessed by a process under three usage modes, two of them being locking modes :

- no lock ; the process only intends to read and concurrent accesses are allowed
- lock-R ; the process only intends to read but concurrent accesses for updating are forbidden
- lock-U ; the process intends to update and concurrent locking accesses are forbidden.

Compatibility between these options is summarized in table 1.

It is the responsability of controllers to set appropriate locks on those data units which are needed by their local processes. When incompatibility occurs, a process will be denied access to a data unit. This is very similar to a situation where a process has to wait because a needed resource is missing.

Therefore, we will consider data units as resources.

REQUESTED USAGE MODE \ CURRENT USAGE MODE	NO LOCK	LOCK-U	LOCK-R
LOCK-R	YES		YES
LOCK-U	YES		
NO LOCK	YES		

YES : simultaneous access allowed

Table 1. - Compatibility table

0 0 : 0 3 1 0 4 1 5 5

As mentionned earlier, incompatible requests are queued up by storage processors which process them on a FIFO basis as data resources become free. But deadlocking is possible as no particular ordering of simultaneous requests on several resources can be assumed to hold within waiting queues. Actually, nature of problems is dependent on which architecture is chosen.

We notice that for an integrated DDSS, deadlock avoidance or detection is to be performed locally by every controller. Conventional (centralized) techniques can be used for that purpose. This is not the case for a partitioned DDSS where deadlock avoidance or detection requires utilization of a distributed algorithm.

On the other hand, whatever the architecture, mutual consistency should be achieved by the means of a distributed algorithm.

There exists a solution which is based on the utilization of a mutual exclusion scheme specifically tailored to distributed systems where failures can occur. This scheme can be briefly described as follows : controllers are linked together on a virtual ring and a unique specific message, the control token, circulates on the ring. Only the controller owning this token is allowed to start a specific activity. Upon completion, the token is given to the successor on the ring.

Problems related to failures and ring reconfiguration are analyzed in (8). Two algorithms intended for automatic control token regeneration are also given. One of these algorithms is presented in this paper. Clearly, if only one controller at a time is issuing allocation requests, there is no potential for deadlocks *. The choice which has been made is to avoid deadlocks rather than detecting them when they occur. In addition to this scheme, a protocol is needed whereby controllers and storage processors can cooperate. Such a protocol can be found in (9). Very briefly, this protocol consists of granting access to the waiting queues according to a dynamic time division multiplexing between controllers.

It results in a mutual exclusion for accessing the entire database but only locking of a subset of the database is done (those data resources which are needed).

In some cases, this solution may prove to be acceptable, e.g. :

- systems with large transit delays but a limited number of processors (data base managers for instance)
- systems not requiring short response times
- localized distributed computer systems
- systems such that storage processors service time is large compared to requests interarrival delays (system-wide).

When storage processors service time is not a limit per se and when response time constraints on the system are severe, a solution is needed whereby full parallelism for accessing the database is possible. Such a solution is described in the remaining of the paper.

* We assume that every controller takes care of deadlocks possibly existing between its local processes.

A RESILIENT CONTROL STRUCTURE FOR A DDSS

We begin by describing how control can be performed within a DDSS. For every controller, a successor and a predecessor are defined such that controllers are located on a virtual ring. Controllers are given permanent identities ranging from 0 to C, and they are sequentially arranged along the ring (hypothesis H1).

Each controller owns a timer. However, timer values are not necessarily identical (hypothesis H2). The sequence of control messages sent on the ring is unchanged i.e., messages received by a controller are retransmitted FIFO.

Let us assume that a particular message, called a control token, circulates on the ring, whereby controllers are entitled to initiate some specific activity.

As we assume that failures can occur, it is necessary to show that this virtual ring + token structure is resilient.

Mutual suspicion

The virtual ring should be able to reconfigurate itself whenever a controller goes down or gets up again. In this paper, we only describe briefly how a failure is noticed and recovered. Regularly, every controller is required to check its successor on the ring. This is done by having "life messages" being sent to the successor. These messages must be acknowledged. Acknowledgment is possible only if some internal checking is successful. This successful internal checking is also a condition for having a controller issuing "life messages". Thus, checking transitivity is achieved along the virtual ring. A good example of such an approach is the checking protocol being used between IMP's in ARPANET (6). If several successive "life messages" are not acknowledged, then a reconfiguration is undertaken by a controller which sends a specific message to the potential successor of its previous successor (H1). Attempts are made until a positive response is received. The ultimate situation is that of a virtual ring including only one controller.

The resiliency property

Transmission of a token between adjacent controllers is monitored through a positive acknowledgment + retransmission protocol. The token carries with it an integer value, called the cycle number, which is incremented for every complete revolution on the ring. This incrementation is performed by controller x such that x > successor (x).

Because of H1, this processor is unique. Also, the numbering cycle to be used should be chosen so that duplicate detection can be performed safely. This is possible if maximum "hardware" transit delays are known.

Timer values being used by controllers correspond to the expected round-trip time with the successor on the ring. A timer is reset when the token has been acknowledged by the successor. Because of hypothesis

0 0 3 0 5 1 0 4 1 6 6

H_2, each controller keeps a recording of the value (N) carried within the token during its last visit. Next real token to be received (not duplicates) must carry value N + 1 (+ stands for addition modulo C+1). When the sender's timer awakes, transmission is tried again, up to a maximum number of attempts. Should this limit be reached, a ring reconfiguration is undertaken. The token is not lost.

If failure of a controller is noticed through the mutual suspicion protocol, then it may be the case that the token was held by this processor which failed. Detection of such a situation and regeneration of the token can be performed as follows.

Let h be the identity of the predecessor of that controller which has failed and i the identity of the successor. Controller h undertakes a ring reconfiguration. The reconfiguration message carries with it value N(h), last token value known in h. Upon reception of this message, processor i runs the following algorithm :

```
if (i > h and N(h) ≠ N(i)) or
   (i < h and N(h) = N(i)) then
create token N(i) := N(i) + 1 ;
```

With such an algorithm, it is possible to assert that a token is never lost on the virtual ring.

As mentionned above, this control structure can be used to achieve mutual exclusion between controllers. In this paper, a different utilization is suggested. The principle is simple : a sequence number is carried within the control token which serves the purpose of ticketing allocation requests. Every request must be ticketed. The sequence number is incremented each time it is used. A description of mechanisms and primitives intended for an identification of events occuring within a distributed system can be found in (10). Here, we suggest a solution for achieving a purely sequential identification of requests within systems using distributed control.

It has been shown how the virtual ring + token structure can survive failures. But ticket allocation must also be resilient. To this end, one may require that a controller should be either selecting tickets or using them but not both. What this means is that those tickets which are selected by a controller should not be used until the token has been acknowledged by the successor. As a consequence, should a failure occur in the midst of ticket selection, the correct sequence number (next ticket value to be used) can be regenerated with the token exactly like this is done for the cycle number (see above). Another issue is that of failures interrupting processing at random. In particular, what should be done regarding those requests which have been issued by a controller which failed later on ? Another problem is what to do with tickets not being used because they were held by a controller which died.

Actually, the whole issue would require a rather lengthy discussion which is out of the scope of this paper.

TICKETING, DATA COHERENCE AND PARALLELISM

Internal integrity

Clearly, a sequential ticketing of requests to be issued by controllers makes it easy for storage processors to decide whether an incoming request should be delayed or processed immediately.

For a partitioned DDSS, internal integrity is achieved by having r ticket values being maintained within the control token, one per partition. It is also possible to provide for r control tokens, each of them carrying only one ticket value corresponding to a specific partition.

Mutual consistency

For an integrated DDSS, only one token with one ticket value is needed to identify external allocation requests (new requests), thus achieving potentially mutual consistency. Every request is then broadcast as an internal request to all other controllers. Three broadcasting techniques can be used : parallel broadcasting (n-1 messages are sent), linear broadcasting, as described in (1), and circular broadcasting (one message on a virtual ring).

In a partitioned DDSS, mutual consistency has to be performed by storage processors for every partition. Again, any technique can be used to broadcast internal requests.

Ticketing algorithms

Let p be a controller, $p \in [0,C]$. At token revolution i, let $C_i(p)$ be the exact number of requests which are pending when the control token is received, $T_i(p)$ be the current ticket value upon reception and $T^*_i(p)$ be the new ticket value when the token is sent on the ring.

A stop-and-wait ticketing algorithm. Each time p receives the token, it is allowed to acquire $C_i(p)$ consecutive tickets, starting from $T_i(p)$. In some cases, the necessity for a request to wait up to R (round-trip time of the token on the ring) before being ticketed may be judged unacceptable.

We can think of two different approaches to solve this problem :
(1) anticipate future requests
(2) provide for several tokens

A one-stage anticipation ticketing algorithm. At token revolution i, let $f_i(p)$ be the predicted number of future requests to be received by p between token visit #i and visit #i+1.

Each time p receives the token, it is allowed to acquire $C_i(p) + f_i(p)$ consecutive tickets, starting from $T_i(p)$. Ideally, during token revolution #i+1, p needs exactly $f_i(p)$ tickets. Clearly, predictions are only predictions. Furthermore, the token circulating speed is variable. Hence, it is necessary to consider two possible situations :

0 0 : 0 5 1 0 4 1 6 7

(i) p runs short of tickets : it has to wait for reception of the token.
(ii) some tickets are not used when the token comes back : let $u_i(p)$ be the number of unused tickets. Because of the mutual independance principle, these tickets should be used up immediately. For that purpose, we provide for the utilization of a No-operation code. Exactly $u_i(p)$ "fake" requests carrying a NOP code will be issued by p.

When needed, and as long as tickets are available, new requests are processed. The main advantage of this solution is to avoid future requests (those being received after the token has been sent) to be waiting for a ticket in order to be processed.

Probably, this will achieve a higher parallelism between controllers for issuing requests but it is not clear whether or not this will result in a better average response time for requests, as compared to the previous algorithm. Response time for a given processor is dependent on how fast the predecessor uses up its tickets.

Should such an interference be judged unacceptable, another algorithm can be proposed.

A two-stage anticipation ticketing algorithm. What we would like to build is a mechanism whereby current pending requests and future requests are distinguable, so that current pending requests in any controller receive tickets "smaller" than those given to future requests.

Let us make it clear that we do not attempt to build a perfect chronological ordering of incoming requests. We only try to achieve some system-wide statistical FIFO service so that the average response time for every request can be kept below a reasonable value.

The way this can be done is rather simple. It is only needed to maintain two ticket values, T and θ, in the token.

T, as before, is to be used for ticketing current pending requests and θ for ticketing future requests. By the time the token is back in p, only one of the three following conditions can hold :

* $u_i(p) = C_i(p) = 0$ (ideal case)
* $C_i(p)$ requests are waiting because p is lacking tickets, $u_i(p) = 0$, $C_i(p) > 0$ (under-estimation)
* $u_i(p)$ tickets are still available, $u_i(p) > 0$, $C_i(p) = 0$ (over-estimation.

A requirement regarding the ticketing function is that the two sets of numbers being used to assign a value to T and θ should not be overlapping.

Two numbering cycles N(T) and N(θ) should be chosen so that no confusion can occur because of possible race conditions and duplicates.

As T-ticketed requests and θ-ticketed requests will be received interleaved by other processors, it is necessary to provide for some mean whereby those processors are able to decide when to stop processing T-ticketed requests and start processing θ-ticketed requests as well as the reverse.

Such a "switching" should correspond to a complete revolution of the token on the virtual ring. We need a processor to flag the corresponding T and θ ticket values.

That processor could be x such that successor (x) <x. Due to the properties of the virtual ring, this processor is unique and always exists.

The algorithm to be followed by p upon reception of the token is described below (+ and - operations are modulo N(T) or N(θ)).

BEGIN
IF suc (p) <p AND $C_i(p) = 0$ THEN
BEGIN $C_i(p):=1$; create Fake Request END ;
IF $C_i(p)>0$ THEN $T^{\star}_{i}(p):=T_i(p) + C_i(p)$
(acquisition of tickets $\# T_i(p),\ldots, \# T_i(p) + C_i(p)-1$)
ELSE IF $u_i(p)>0$ THEN
send $u_i(p)$ Fake Requests (ticketed with the $u_i(p)$ highest θ-tickets obtained during the last token visit) ;
Assign a value to $f_i(p)$;
IF suc (p)<p AND $f_i(p)=0$ THEN
BEGIN $f_i(p):=1$; create Fake Request END ;
$\theta^{\star}_{i}(p):=\theta_i(p) + f_i(p)$ (acquisition of tickets $\#\theta_i(p),\ldots,$
$\# \theta_i(p) + f_i(p) - 1$) ;
IF suc (p)< p THEN Flag requests carrying tickets $\# T_i(p) + C_i(p)-1$
and $\# \theta_i(p) + f_i(p)-1$;
END

The algorithm to be followed by a processor (controller or storage processor) upon reception of an internal request is given below.

Notations :
X stands for either state T ("current") or state θ ("future") ;
$\bar{X}$ = T if (X=θ),
= θ if (X=T) ;
t(X) is a local variable containing the ticket value of the last processed request i.e. t(T) or t(θ).

WHEN IN STATE X DO
LOOP : Scan for, or wait for reception of request X-ticketed t(X)+1 ;
CASE1 (X-ticket t' > t(X)+1 (mod) is received) :
Record request ;
CASE2 ($\bar{X}$-ticket is received) :
Record request ;
CASE3 (X-ticket t(X)+1 is present or received) :
BEGIN Initiate processing ;
IF request t(X)+1 is flagged THEN switch to state $\bar{X}$ ELSE
t(X):=t(X)+1
END
CASE4 (timeout) :
Marks itself out of synchronization and initiate a recovery procedure ;

A simple way to provide for two separate numbering schemes of equal length is to use one bit to distinguish between T-tickets and θ-tickets. However, one should mention that, if predictions are not too inaccurate, θ-tickets are to be used up more rapidly than T-tickets.

Then, an equal share of the ticket number space may not be the best solution.

We will discuss only briefly the issue of fairness in estimating $f_i(p)$. We consider two cases :

- controllers are processors (maxis, minis, micros) cooperating within a DDSS to be viewed as a unique system from users. Algorithms to be followed by controllers are designed by system builders who are responsible for choosing convenient values for $f_i(p)$.
- controllers are computers connected on a computer network. Over-estimation is costly to controllers because (i) processing wasted in handling NOP requests cannot be used to process useful requests (throughput is lower), (ii) a controller is billed for messages carrying NOP requests and for the corresponding processing in the distant computer.

A multi-token ticketing algorithm. With this scheme, several tokens (#1,..., #i,..., #v) are circulating on the ring. Ticket acquisition is done according to the stop-and-wait algorithm, with the difference that $C_i(p)$ is the exact number of requests received between visit of token #i and visit of token #i+1.

Again, processor x such that successor (x)<x is in charge of flagging requests corresponding to the last selected ticket in every token.

The algorithm for internal request processing is very similar to the previous one. Any desired degree of parallelism can be obtained by selecting an appropriate value v.

AS A CONCLUSION

Not all issues have been addressed in this paper. In particular, mechanisms needed to recover from storage processor crashes have not been described. However, a solution has been given to the problem of achieving data coherence in a DDSS (partitioned or integrated) by the means of a sequential identification of user requests.

This identification technique has been shown to be resilient. Four algorithms have also been given, each of them achieving a different degree of parallelism in request ticketing and processing.

When considering an implementation on a real system, these algorithms should be compared with others so as to trade features like response time, imposed programming discipline (static or dynamic data requesting) and rejection avoidance against system overhead i.e. message transmission requirements, algorithm complexity and additional information accompanying user data.

It is our feeling that experimental implementation and performance measurements would be the next big step toward a better assessment of these algorithms.

REFERENCES

1. P.A. Alsberg, J.D. Day, A principle for resilient sharing of distributed resources, IFIP WG 6.1 General Note 127 (Aug. 1976), 20 p.

2. P.A. Bernstein et al., The concurrency control mechanism of SDD-1 : a system for distributed databases, Computer Corporation of America, Technical Report 77-09 (Dec. 1977), 172 p.

3. D.D. Chamberlin, R.F. Boyce, I.L. Traiger, A deadlock-free scheme for resource locking in a data-base environment, IFIP Congress, (North-Holland Pub. Co., 1974), 340-343.

4. C.A. Ellis, A robust algorithm for updating duplicate databases, Proc. of the 2nd Berkeley Workshop on Dist. Data Management and Computer Networks (May 1977), 146-158.

5. G.C. Everest, Concurrent update control and data-base integrity, in Data Base Management (North-Holland Pub. Co., 1974), 241-268.

6. F.E. Heart et al., The interface message processor for the ARPA computer network, SJCC, (AFIPS, 1970), vol. 36, 551-576.

7. L. Lamport, Time, clocks and the ordering of events in a distributed system, Mass. Computer Associates, Report CA-7603-2911 (March 1976), 23 p.

8. G. Le Lann, Introduction à l'analyse des systèmes multiréférentiels, Thèse de Doctorat d'Etat, University of Rennes, France (May 1977), 202 p. (French).

9. G. Le Lann, Pseudo-dynamic resource allocation in distributed databases, to appear in ICCC'78 Proceedings, Kyoto, Japan (Sept. 1978).

10. D.P. Reed, R.K. Kanodia, Synchronization with enventcounts and sequencers, MIT Report (1976).

11. R.H. Thomas, A solution to the update problem for multiple copy data bases which uses distributed control, BBN Report 3340 (July 1976), 50 p.

0 0 0 0 5 1 0 4 1 6 9

A PROPOSAL FOR DISTRIBUTED CONCURRENCY CONTROL FOR PARTIALLY REDUNDANT DISTRIBUTED DATA BASE SYSTEMS

D. Z. Badal and G. J. Popek

Computer Science Department
UCLA, Los Angeles, Ca. 90024

Abstract

This paper describes two distributed concurrency control methods for partially redundant distributed database systems. The proposed methods are shown to maintain internal and external consistency of database, to avoid deadlocks and to be robust in case of network partition(s), and node(s) or communication link(s) failures. The protocols require low synchronization overhead and they support an automatic recovery to a consistent state in case of network partition(s).

INTRODUCTION

This paper outlines two efficient distributed concurrency control methods for those distributed systems in which the identity of the items to be locked can be determined before actually running the transactions. Properties of the protocols include:

1) The protocols are efficient in that they require approximately 2(n - 1) messages per transaction for synchronization in partially replicated databases, and introduce a maximum delay of 4Tmax (protocol P1) or 3Tmax (protocol P2), which is largely independent of the number of sites in the net. Tmax is the maximum average delay between the sender and several destinations. A high degree of concurrency is permitted by the protocols.
2) Automatic recovery to a consistent state in case of network partitioning is provided.
3) The protocols preserve mutual consistency of multiple copies of data, do not allow temporary internal inconsistencies of data, and are deadlock free.
4) No assumptions are required about the order of arrival of messages from different sites.
5) The protocols tolerate multiple site failures, network partitions, and lost messages.
6) The protocols are specialized in the sense that, while they can support data dependent locking, such transactions must be executed serially to preserve the consistency of the database. By accepting this limitation, overall performance is otherwise considerably improved over other proposed protocols.

The synchronization protocols are based on associating time stamps with transactions (not data items) and assuring that all interfering

actions of transactions at any given network site are executed in order of their time stamps. Other recent distributed synchronization protocol work either supports increased generality at a cost in greater message traffic and delays, or requires high storage overhead, or does not preserve certain data consistency properties.

PROTOCOL P1: OVERVIEW (NORMAL CONDITIONS)

The synchronization protocol P1 can be informally outlined as follows. When a transaction arrives at a particular site, it is analyzed and the data items which it will read and write are identified as the transaction's Read and Write sets respectively. It is then assumed that this "initiating site" contains a database directory and suitable decision algorithms to decide which sites in the network will be involved in this transaction. We label those sites Read sites and Write sites as appropriate. The initiating site creates and sends a Setup message (SUM) for each Read and Write site of the transaction. The Setup message contains the definition of the transaction, including the list of sites involved, objects to be accessed, and time stamp. A preferred Read site then creates a Request (REQ) message and sends it to each site in the network.

The intent of the Request message is to demand all recent update operations and SUM messages generated elsewhere but not yet received by the sites involved in read operations in this transaction.

It is expected that all sites which have originated any update activity or SUM messages relevant to the Read sites for this transaction (not yet sent to their destination sites) will forward these updates and SUM messages at this time to the requesting Read site in REQ-ACK messages. All other sites will send a null REQ-ACK message. In normal operation, when a response has been received from all sites in the net, the requesting Read site forwards any updates received and broadcasts them in Read Commands to each of the other Read sites for this transaction. The SUM messages will be used for recovery, as discussed later.

Each Read site performs the updates in the order indicated by their time stamps and then performs the appropriate reads for the transaction of interest, together with any associated local processing. Some Read sites may send messages to other ones in the course of operation. Each Read site eventually prepares Write messages for each of the relevant Write sites, carrying the time stamp of the original transaction, but need not send them immediately. The actual transmission of data for the Write sites will occur in response to subsequently received Request messages. Of course, if there is excess communication capacity at any point, the data can be sent ahead of the actual request for it. The write stages of the transaction are thus normally "piggybacked" onto the early stages of subsequent transactions, reducing communication traffic.

The underlying reason that the protocol works is that no read operation takes place until all relevant write operations for the site of the read have been carried out. The transmission of the updates is demanded by the reads. Thus we have a "read driven" protocol. It of course possesses many opportunities for optimization, in a manner logically analogous to prepaging possibilities in a demand paging system, although the value here of transmitting the updates ahead of demand may

0 0 3 3 5 1 0 4 1 7 0

be greater.

The protocol allows the concurrent execution of non-interfering transactions. If the transactions interfere, for example by requiring the same resource, the execution of such transactions is in the order of their time stamps. However, since each transaction is modelled as a set of possibly independent read-execute and write actions, the interfering transactions can still execute concurrently except for those transaction actions which interfere, i.e. which mutually overlap. Let's consider two transactions T1 and T2, which arrived at site 1 and site 2 close together in time. Assume that T1 and T2 consist of two independent segments each having its Read and Write sets T1RS1, T1RS2, T1WS1, T1WS2, T2RS1, T2RS2, T2WS1 and T2WS2. Suppose that segment 1 of T1 reads at site 1 (i.e. T1RS1 = {1}), and writes into site 2 (i.e. T1WS1 = {2}). Further assume the following Read and Write sets:

T1RS2 = {3}	T1WS2 = {4}
T2RS1 = {4}	T1WS1 = {3}
T2RS2 = {5}	T2WS2 = {6}

Suppose that the time stamp of T1 is less recent, i.e. TST1 < TST2. The protocol will allow concurrent execution of segment 1 of T1 and segment 2 of T2. However, segment 1 of T2 will be executed only after segment 2 of T2 because their Read and Write sets overlap. Since all interfering actions of transactions are executed in their time stamp order, the protocol provides deadlock prevention.

One can see that this protocol's approach can prevent internal inconsistencies of transactions simply by considering the Read set of each transaction to be a union of the set of objects to be read by the transaction and the set of objects which are the arguments of any integrity assertions for each transaction. Then the problem with a potential internal inconsistency of the transaction T1 in the above example is that some T1 integrity assertions whose arguments range over T2WS2 might evaluate to different values depending whether they were evaluated before or after the segment 2 of T2 executed its write on T2WS2. However, the enlarged Read sets of transaction would prevent any internal inconsistencies at the price of potentially lower degree of concurrency.

The protocol avoids the use of time stamps to label database data as suggested in other protocols [THO78, ROT77, BER77]. The labeling of data by time stamps does simplify the protocol and speeds up transaction execution. The reason is that any read action has to wait only for the latest preceding write action instead for all preceding write actions. The write actions are then treated as commutative operations and thus executed in any order. However, there are two problems associated with such an approach. First, one has to label the smallest updatable items, which will considerably increase the database size. Second, the execution of write actions in an arbitrary order can lead to temporary internal inconsistencies which may not be acceptable for some applications. Similar observations have been made by Thomas [THO78, p.60].

Protocol P1: Algorithm

Setup messages play a central role in the actual details of this protocol. The SUM message is the same for all Read and Write sites of

transaction. We will refer to those SUM messages placed at Read sites of a transaction as SUM Read events and to those SUM messages placed at Write sites of a transaction as SUM Write events. An actual Write message, from Read site to Write site, makes a previously arrived SUM Write event ready for execution. Once the SUM messages have been placed at their proper sites, the following two rules are followed:

(1) Each SUM Read event can be executed only if all preceding SUM Write events addressed to the given Read site from all sites have been received and all preceding SUM Write events interfering with the most recent SUM Read event have been executed in order of their time stamps.
(2) Any SUM Write event can be executed only after Write messages from its Read sites have been received and its execution is triggered either by:
    a) the SUM Read event of some other transaction; or by
    b) the SUM Read event of the same transaction (occurs if the Read set and the Write set of the transaction are the same and are at the same site).

At the preferred Read site which has received the SUM message the following algorithm is to be executed:

(1) Broadcast REQUEST (REQ) messages to n - 1 sites (assuming n nodes in the network). The REQ message to any site i has the following functions:
    a) Acknowledge all received messages sent from site i to the originator of REQ message since the previous REQ message from the originator.
    b) Request all Write messages whose destination is any Read site of the transaction.
    c) Request all unacknowledged SUM messages sent to any Read site of the transaction.
    d) Request the list of all rejected transactions from each site i*.
    e) Carry the time stamp of the SUM Read event which caused the broadcast of REQ message together with a list of transactions rejected by the originator of the REQ.

Besides these functions, the REQ message to any Read or Write site of the transaction carries the copy of SUM message. All functions of the REQ message except b) are related to the robustness of the protocol.

(2) Wait for the acknowledgement of the REQ messages from n - 1 sites for a certain time (Tmax); perhaps repeating REQ to nonresponding sites. All REQ-ACK messages are to be generated immediately on the arrival of REQ message and sent to the originator of the REQ message (i.e. preferred Read site) only. The REQ-ACK messages may contain the following:
    a) All Write messages addressed to any Read site of the transaction.
    b) All unacknowledged SUM messages which any of the n - 1 sites has previously sent to any Read site of the transaction.

*How transactions are rejected is described in section Rejection of Transactions and Deletion of SUM Messages.

c) Special message - "Write message still being generated" (used to indicate delay without triggering rebroadcasting).
d) List of transactions rejected by site i.
e) Acknowledgement that the Read and Write sites of the current transaction received a copy of the SUM which caused the REQ message broadcast.
f) Acknowledgement of REJECT messages. (The meaning and the function of REJECT messages is explained in the section dealing with robustness of the protocol P1.)

(3) If all n - 1 sites responded, i.e. if the preferred Read site received n - 1 REQ-ACK messages, then the preferred Read site combines the REQ-ACK messages into one READ COMMAND message and broadcasts it to the nonpreferred Read sites of the transaction. Then all Read sites execute all SUM Write events which interfere with the SUM Read events of the transaction being synchronized, in the order of their time stamps. The noninterfering SUM Write events can be executed at any time as follows: if they interfere with each other, then they must be executed in the order of their time stamps, otherwise they can be executed in any order. If some of the n - 1 sites did not respond after repeated REQ broadcasts, they are considered to be a part of the network partition and the SUM Read event can be executed as outlined in the section Robustness of the Protocol.

(4) Execute all SUM Read events of the transaction and generate Write messages if required. Such Write messages can either be sent immediately to the Write sites of the transaction or later as part of a REQ-ACK message.

The SUM Write events are executed only after another SUM Read event made sure that all preceding Write messages were placed at its site. The principal reason for not allowing the SUM Write event execution on the arrival of their Write messages is that we require all interfering SUM Write events preceding a SUM Read event to be executed in the order of their time stamps, but they are not known until replies to Setup message are received. Otherwise, one would have to check, for each SUM Write event, whether there was any missing SUM Write event interfering with SUM Write event and with earlier time stamp ready for execution. To do so would mean that each SUM Write event would have to broadcast a REQ message to n - 1 sites and could execute only after receiving REQ-ACK messages from n - 1 sites. We avoid such a high communications overhead simply by relegating this responsibility to the SUM Read events only.

Notice that even in the case of immediate broadcast of Write messages to the Write sites of the transaction the SUM Write events will be executed only if there is a subsequent SUM Read event at the Write site. Thus, the protocol is Read event driven, i.e. only Read events can enable the execution of earlier stamped Write events (that is, SUM Write events) and other Read events (that is, SUM Read events).

The preceding protocol assumed that all Read and Write sets of the transaction were known before its execution, i.e. the transaction did not require data dependent locking. In such cases the protocol allows a high degree of concurrent execution. However, if some transaction requires data dependent locking, one could allow the generation of additional SUM

messages during transaction execution to block any other transaction execution. The data dependent transaction can prevent all other transactions from executing by acknowledging any REQ message from any other transaction by some sort of delay flag, indicating the site is up, but preventing the requesting site from going forward.

The Efficiency of Protocol P1 Under Normal Conditions

We characterize the efficiency of protocol in terms of two parameters - the number of messages needed for the synchronization of one transaction and the time delay due to synchronization.

Each transaction requires two steps to synchronize it. First, each site which receives a transaction must broadcast the SUM message to the (r + w) sites where the Read and Write sets of the transaction are located. Since each such site acknowledges each SUM message we have 2(r + w) messages.

The second set of the synchronization messages is generated during the next step of synchronization involving the preferred Read site of the transaction. It sends out n - 1 REQ messages. Each site which received the REQ messages sends the REQ-ACK message only to the preferred Read site. After that site received n - 1 REQ-ACK messages, it is responsible for broadcasting READ COMMAND messages to the remaining r - 1 Read sites. The total number of messages introduced by this protocol is therefore 2(r + w) + 2(n - 1) + (r - 1). The total delay needed to synchronize one transaction is Tmax (to broadcast SUM messages) + 2Tmax (REQ and REQ-ACK messages) + Tmax (READ COMMAND message), i.e. 4Tmax*. (Tmax is maximum average delay between a sender and several destinations.)

PROTOCOL P1: OVERVIEW (ABNORMAL CONDITIONS)

The preceding paragraphs outlined the P1 protocol's operation under normal conditions. There are many ways in which difficulties can arise however: sites crashing, messages lost, messages received out of order, etc. The most important of these, which gives rise to a second phase of the protocol, is the lack of response to a Request message. When the preferred Read site does not receive a reply from a given site, that Read site decides that the silent site is partitioned from the rest of the net. The transaction at the initiating site might be permitted to go forward however, if there exists among the still accessible sites at least one copy of each data item needed for reading. (The Write sites need not be currently accessible at all.) However, suitable log records will be kept so that transactions can be rerun if necessary after the network partitions are reconnected.

In particular, suppose a transaction has been presented for which the Read sites have not received Write messages with time stamp equal to

*The total delay 4Tmax can be decreased to 3Tmax if each site which received a REQ message would send REQ-ACK to each Read site instead to the preferred Read site only. The total number of messages is then 2(r + w) + (r + 1)(n - 1) + (r - 1).

0 0 0 0 5 1 0 4 1 7 2

or later than the given transaction from all sites. Then all Write messages resulting from the transaction carry a tag indicating potential inconsistency. Each site which receives a tagged message is responsible for keeping a local log so that its portion of the database can be backed up if necessary. Further, any site which receives a tagged Write message must notify all sites which will subsequently read at that site that the database is potentially inconsistent. Any site which receives such a notification must also keep a local log so that backup can be done there too. When the partitions are reconnected the local logs in each partition are compared and if inconsistency due to missing Write or Read events which could not cross partitions is detected, then some transactions must be rerun.

Robustness of the P1 Protocol

By the robustness of protocol we mean two things - the protocol's ability to tolerate lost messages and the protocol's ability to recover to a consistent state when partition of the underlying network occurs. Since the protocol handles any nonresponding site as a partition we cover site(s) crashes and communication link(s) in the subsection Partition Recovery.

Lost Messages. The protocol is able to correctly synchronize transactions in spite of any number of lost SUM synchronization messages under normal operation. Any messages that are lost due to site or the communication link crashes are not lost messages; such messages demonstrate themselves as messages which cannot be acknowledged because their destination site is not responding. Therefore in this section we deal with the messages which are lost due to any other reasons except communication links and sites crashes.

The protocol uses two types of synchronization messages: SUM, SUM-ACK, REQ, and REQ-ACK. (READ COMMAND and WRITE messages are part of REQ-ACK messages.) We will now consider the P1 protocol behaviour when any of the above messages are lost. To do so, we first outline the basic mechanism of message handling by sending and receiving sites. Any sending site S keeps a copy of any message it sent to any other site R until it receives the acknowledgement that such message has been received at the receiving site R. The acknowledgement of some messages may be either by the receiving site R explicitly acknowledging or by appending the acknowledgement to another message sent from the receiving site R to the site S. For example, the SUM message sent by the transaction initiating site can be acknowledged either by SUM-ACK or by the REQ message from the receiving site.

It is essential for synchronization purposes that the SUM messages are always received by the transaction Read and Write set sites before any access by another transaction with more recent time stamp is performed at these sites.

Lemma: Under normal operation of the system (i.e. when no crashes occurred) all SUM messages will be always received at SUM destination sites before any access is performed there.

Proof outline: Suppose that the SUM1 message to site 1 of transaction T1 was lost (and therefore not acknowledged). Also suppose that later site 1 becomes the preferred Read site for transaction T2. If site 1 is the preferred Read site for T1 whose SUM1 message was lost, then the REQ generated by the SUM2 message will fetch the SUM1 message either from any nonpreferred Read site of T1 (so long as all SUM1 messages were not lost) or from the initiating site for T1. If site 1 is the nonpreferred Read site for T1 or a Write site for T1, then it will receive its SUM1 message via REQ message from the preferred Read site of T1 or via REQ message from the preferred Read site of T2 (i.e. site 1). If the REQ or REQ-ACK is lost, then site 1 repeats the broadcasting of REQ messages to all sites which did not respond with REQ-ACK message. Since we assume that no crashes occurred, each site will eventually respond to REQ message. □

If some nonresponding sites crashed, then such sites are timed out after some m unsuccessful broadcasts of REQ message and they are treated as a partition.

Partition Recovery. An interesting characteristic of the protocol P1 is the automatic recovery of the database state into a consistent state at partition reconnection time. The protocol can support three types of automatic recovery through basically the same set of procedures. In Type I, recovery is into the database state which would have been reached if no transactions were rejected at all. In Type II, recovery is into the database state which would have been reached if the only rejected transactions were those which did not have at least one copy of the Read sets exclusively available in one partition. The third type of recovery is into the database state which would have been reached if the only rejected transactions were those which have had at least one copy of Read and Write sets available in one partition.

These three types of recovery are distinguished by the following considerations. Suppose that transaction T wants to execute in one partition. If T attempts to read some data items, the following situations can occur:

(1) There might be some preceding reads and writes from another partition which T is unaware of (in Type I recovery the protocol does not reject transactions which may not have all Read sets available).
(2) There might be some preceding writes (updates) from another partition which T is unaware of but no preceding reads (in Type II recovery the protocol will allow the execution of transactions even if they do not have all Write sets available).
(3) There cannot be any missing preceding reads and any preceding writes on unreplicated data items or on replicated data items when all multiple copies are in the same partition as T (the Type III case). However, there can be some missing preceding writes on replicated data items if not all copies of such data items are in the same partition as T.

It is intuitively obvious that the first type of recovery is the most complex and the most expensive, and third type of recovery is

0 0 : 0 5 1 3 : 1 7 3

least so*. The cost of each type of recovery depends on the size of data in each partition which can become potentially inconsistent and on the number of transactions that must be undone and redone at partition reconnection time.

We now describe the recovery strategy which is common to all three types of recovery. When a partition is detected by any site of the transaction (i.e. initiating, Read or Write sites) during the first or the second step of synchronization the protocol rejectes or accepts the transaction depending on the type of recovery being supported. If the transaction is accepted, then it must execute according to specific rules. Such rules specify what kind of information has to be kept in order to support recovery, i.e. the rules specify when to start keeping the local logs and what the local logs should contain.

Whether or not a transaction running under protocol P1 causes the system to start keeping local logs depends on the recovery type. Before the transaction can execute its Read event at the preferred Read site, it will know from missing REQ-ACK messages whether the transaction should be accepted or rejected, i.e. whether for example, all of its Read sites received the SUM messages and are available or whether all of its Read and Write sites received the SUM messages and are available. At the same time the transaction can determine whether any of its Read or Write events would be executed using potentially inconsistent data due to missing Read or Write events from nonresponding sites.

If the transaction T is accepted, and Type III recovery is being supported, T will execute on potentially inconsistent data only if the transaction reads one copy of a multiple copy data item M and not all copies of M are available in the partition in which T executes. When this condition occurs, the transaction T will start keeping local logs at its Read and Write sites. Such local logs consist of copies of T's SUM messages and assorted Write messages.

Further, any subsequent transaction executing on potentially inconsistent data (including the data modified by Write events of the transaction T) must be notified to keep local logs at its Read and Write sites. Any such notifications are by tagged REQ and REQ-ACK messages.

The recovery strategy for the second type of recovery is the same as for the third type of recovery except the time when the local logs should be kept, i.e. when the potential inconsistency can occur. Any accepted transaction T running under protocol P1 supporting the second type of recovery will execute on potentially inconsistent data if at least one site is not responding to repeated REQ messages from the preferred Read site of the transaction T.

The recovery strategy for Type I recovery is the same as for Type II except that one must additionally keep in local logs all transactions which would have been rejected if P1 supported only Type II recovery.

---

*Type I is surely impractical for most applications.

Rejection of Transactions and Deletion of SUM Messages. Since any transaction initiating site broadcasts SUM messages to the Read and Write sites regardless whether they are reachable or unreachable, the protocol must delete such SUM messages when the transaction is rejected.

The transaction can be rejected as follows. We define a cover condition which specifies which Read and Write sites of T must be responding to SUM and REQ messages. If the cover condition is satisfied, then transaction T is rejected, otherwise it is accepted. We have three types of cover conditions depending on the recovery the protocol P1 is supporting.

Type I: T initiating site is not responding.
Type II: Not all Read sites are responding.
Type III: All Read sites are responding but not all Write sites are responding.

The initiating site of transaction T will not reject T only on the basis of missing SUM-ACK messages because some of them might have gotten lost. Suppose that some SUM messages were delivered and the preferred Read site of T received a SUM message. The preferred Read site immediately broadcasts n - 1 REQ messages. One purpose of such messages is to find whether the Read and Write sites are still up and whether they received their SUM messages. Now if after repeated broadcast of REQ messages some sites fail to respond with a REQ-ACK, the preferred Read site now can decide whether the appropriate cover condition is satisfied to reject T if necessary. (Notice that if some Read or Write site of T has not received its SUM message, it is supplied by the REQ message.) Also the preferred Read site of T has to inform the initiating site for T of its decision to reject or to accept T. Such a decision is indicated by a REJECT message which becomes a part of the next REQ from T's preferred site. However, if T's preferred Read site is down, the nonpreferred Read sites of T will have not received READ COMMAND message from the preferred Read site.

To select a new preferred Read site, the nonpreferred Read sites then broadcast one REQ message to all Read sites of T. If the preferred Read site does not respond with READ COMMAND, then it is considered by all nonpreferred sites to be in another partition. In such a case the nonpreferred responding Read site with the highest number becomes the preferred Read site in each partition. (All Read sites of the transaction T are numbered.) The new preferred Read site in each partition repeats the process of broadcasting REQ messages and identifying nonresponding sites. The intention here is that the preferred Read site has the highest priority to reject T. If no preferred site can do so, i.e. all of them are crashed, the initiating site can make such a decision only after it has not received REQ messages from any T Read site within $2(m + 1)$ Tmax (where $2(m + 1)$ Tmax is maximum delay it takes to reject or accept the transaction if at least one Read site of T is up).

After T has been rejected there remains the problem of discarding SUM messages from the Read and Write sites of T. There are two possibilities depending on whether T was rejected by the preferred Read site or by the initiating site. If T is rejected by the preferred Read site, that site uses READ COMMAND message to inform the remaining Read sites

0 0 : 0 3 1 0 4 1 7 4

that T has been rejected. Each Read site deletes the SUM messages of T and prepares tagged Write messages for all appropriate Write sites of T, i.e. for all Write sites in which T was to write. Thus the deletion of SUM messages at Write sites will occur when tagged Write messages eventually are received.

If T is rejected by the initiating site, then SUM messages at T's Write sites must be discarded. Such SUM messages will be deleted at the time when any Write site of the rejected transaction becomes a Read site for some subsequent transaction. Such a Read site will broadcast REQ messages whose additional function is to collect from all sites the identity of all rejected T's.

In general, the problem of dangling or forgotten SUM messages at any site can be resolved as long as each site keeps the list of rejected T's for some period of time.

Efficiency of the Protocol P1 Under Abnormal Conditions

The efficiency of the protocol P1 decreases under abnormal conditions, i.e. when there are lost messages and partitions occur. We can provide upper bounds for the number of messages and total delay. The lost SUM messages do not require any additional synchronization messages but they might introduce additional delays. The worst case is when no SUM message is delivered to the Read and Write sites of the transaction. In such a case, if there is no REQ message within $2(m + 1)$ Tmax from any Read or Write sites of transaction T, T should be rejected. Thus the maximum delay is $2(m + 1)$ Tmax. If any REQ or REQ-ACK is lost, then the REQ is rebroadcasted m times to nonresponding sites only. If REQ-ACKs are still not received, the transaction is either rejected or executed on potentially inconsistent data with appropriate logging. Thus we have $2m * K$ additional messages involving the K nonresponding sites, where $k \leq n - 1$, and additional delay of $2m * Tmax$.

Once the T has been accepted for execution in the presence of the partitioning, there is no additional delay or messages until partition reconnection. Overhead is introduced by the generation of local logs. The reconnection of partitions is detected simply by any transaction which receives the REQ-ACK carrying SUM or WRITE messages with time stamps less recent than the transaction to be executed but more recent than the first SUM message in the local log. Once reconnection is detected by any transaction, then each Read site of the transaction compares its local logs with the local log at every other site. Each site decides how far the database state should be rolled-back in order to restore consistency at that site. After global roll-back has been performed the transactions in local logs are rerun. There is no estimate we can provide at this time on recovery overhead because it very much depends on the recovery policy and on the type of recovery the protocol is intended to support.

AN OPTIMIZATION - PROTOCOL P2 OUTLINE (NORMAL AND ABNORMAL CONDITIONS)

The synchronization protocol P2 differs from protocol P1 as follows: the transaction initiating site, instead of broadcasting SUM messages to the transactions's Read and Write sites, only broadcasts a combined SUM and REQ (SUMREQ) message to n - 1 sites. All n - 1 sites respond with SUMREQ-ACK message to the preferred Read site of the transaction. Any Read or Write site of the transaction also sends SUMREQ-ACK message to the initiating site of the transaction. The preferred Read site then forwards the combined SUMREQ-ACK messages as READ COMMAND to the nonpreferred Read sites of the transaction.

The protocol P2 is very similar to P1. However, there are slight differences in the functions of SUMREQ and SUMREQ-ACK messages and in the flow of synchronization messages. Also P2 protocol combines two synchronization steps of P1 protocol into one step. The protocol P2 is more sensitive to lost SUMREQ messages because each lost SUMREQ message is treated by the protocol P2 in the same way as the protocol P1 treats a lost REQ message, i.e. by repeated broadcast to sites which did not respond with SUMREQ-ACK. Notice that protocol P1 is never concerned with lost SUM messages because they will eventually be supplied to all appropriate sites by REQ or REQ-ACK messages of the transaction which is being synchronized or by subsequent transactions.

The behaviour of the P2 in the partitioned network is very similar to that of P1. The main advantage of P2 is its improved performance. The total number of messages to synchronize one transaction under normal conditions is $2(n - 1) + (r + w) + (r - 1)$ and the total delay under normal conditions is $3Tmax^*$. The performance of the protocol P2 and P1 under abnormal conditions is also very similar.

RELATED WORK

The synchronization control problem in distributed database systems has been addressed by numerous authors [ROT77]. However, relatively few deal with distributed concurrency control [ELL77, BER77, ROS77, THO78, LAM76], and only one, [THO78] is concerned with reliability aspects of concurrency control. We are aware of only one proposal [MEN78] which is able to support automatic recovery of partitions in a distributed database. However, the protocol described in [MEN78] depends on centralized concurrency control.

For comparison purposes, the distributed concurrency control design by Ellis [ELL77] requires 2n messages per transaction and the protocol by Thomas [THO78] requires between 1.5n and 2n messages per transaction. However, both synchronization mechanisms introduce significant

*The total delay introduced by the protocol P2 can be decreased to 2Tmax, at the cost of considerably increased communications traffic, if each site which received SUMREQ message would send SUMREQ-ACK to each Read site of the transaction instead only to the preferred Read site. Then the total number of messages is $(r + 1)(n - 1) + (r + w)$.

0 0 0 3 5 1 0 4 1 7 5

communication delays due to their sequential communication characteristics. In both protocols the delays are directly proportional to n, where n is the number of sites in the distributed database.

The proposal outlined in this paper compares favourably with the work of others for non fully replicated databases under the constraints mentioned earlier as the protocols, besides being robust, introduce $2(n - 1) + \epsilon$ messages per transaction and the maximum delay 4Tmax or 3Tmax is largely independent of the number of sites in the network.

CONCLUSION

Here we have outlined synchronization protocols for a somewhat restricted class of database transactions, although one can expect that many systems will be covered by that class. The protocols which result are more efficient than more general proposals, while still retaining the robustness properties outlined in the introduction.

REFERENCES

BER77 Bernstein, P. A. et al. "The concurrency control mechanism of SDD-1: A system for distributed databases (the general case)," Tech. Report CCA-77-09.

GRA76 Grapa, E. "Characterization of a distributed database system," Tech. Report UIUCDCS-R-76-831, Dept. of Computer Science, University of Illinois, Urbana, October 1976.

ELL77 Ellis, C. A. "A robust algorithm for updating duplicate databases," Proc. of 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977.

LAM76 Lamport, L. "Time, clocks and the ordering of events in a distributed system," Massachusetts Computer Associates Report, CA-7603-2911, March 1976.

MEN78 Menasce, D., Popek, G. and Muntz, R. "A locking protocol for resource coordination in distributed databases," 1978 SIGMOD Int. Conference on Management of Data (to appear in ACM TODS).

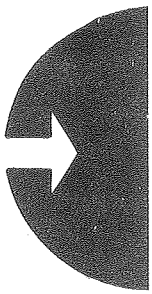
ROS77 Rosenkrantz, D. J. et al. "A system level concurrency control for distributed database systems," ibid ELL77.

ROT77 Rothnie, J. B. and Goodman, N. "A survey of research and development in distributed database management," Proceedings of 3rd International Conference on Very Large Databases, Tokyo, October 1977, pp. 48-62.

THO78 Thomas, R. H. "A solution to the concurrency control problem for multiple copy data bases," COMPCON 78, San Francisco, February 1978, pp. 56-62.

0 0 / 0 5 1 3 4 1 7 6

NETWORK MODELS AND PERFORMANCE



0 0 0 5 1 0 4 1 7 7

ON THE DESIGN OF ELEMENTARY DISTRIBUTED SYSTEMS

Randolph T. Yeh* and K. Mani Chandy

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

Distributed systems range from the elementary to the very complex. This paper is concerned with the design of elementary systems in which there are no multiple copies of files, transactions exhibit a high degree of geographic locality, and the network topology is a tree. Issues considered include the selection of processors at different sites in the network, line topology and bandwidths, and allocation of application functions to processors. A case study is presented.

INTRODUCTION

Overview

This paper deals with the design of special-purpose distributed data base systems with tree-structured topologies. The design consists of selecting processors for geographically distributed sites of a system, choosing a network topology, selecting communication line bandwidths, and allocating application functions and data bases to sites. In most cases, the problem is to enhance the capability of an existing centralized data base system. The focus of this paper is on distribution of processing power and application functions and not on the overall problem of data base design.

We shall report on an ongoing effort to develop programs that will aid in designing distributing systems for the Texas Department of Mental Health and Mental Retardation (TDMHMR), a government agency of the State of Texas. The problem encountered by TDMHMR is also encountered by other medium-sized organizations. We shall also discuss similar problems encountered by other organizations. The problems discussed here are much simpler than those faced in designing general distributed data base systems such as SDD-1 (55). However, our problem is encountered often enough to merit consideration on its own terms. There are now vendors who provide hardware and software for the kinds of distributed problems that we are concerned with here. The problem discussed here is not a hypothetical one that may arise in the future; it is a pressing problem that designers are faced with now. The frequency with which this problem arises leads us to believe that computer-aided design (CAD) tools are warranted for this problem.

---

* Now employed at Bell Laboratories, Naperville, Illinois.

We shall first discuss the specific design problem faced by TDMHMR, which has its headquarters in Austin. The department serves schools, hospitals, and mental health centers throughout the state and carries out several data processing functions. Examples of current and planned functions include:

- The Behavioral Characteristics Progression (BCP) System, which helps teachers and psychologists in schools keep track of the behavioral characteristics of their students.
- The Time and Attendance System (TAS), which aids in posting personnel leave requests and auditing individual attendance records.
- The Pharmacy System (PS), which helps keep track of medications administered to patients.

Not all application functions are required at all institutions; for instance, BCP is used at schools but not at hospitals. The functions can be partitioned into independent function groups so that a transaction accesses the data bases of one and only one function group. For instance, a BCP transaction will not access a TAS data base. Similarly, a TAS function will not access a BCP data base. Queries that require information from the data bases of two or more function groups (such as BCP and TAS) are so infrequent that they may be ignored for the purposes of initial design.

The major interactive load at this time is from schools using BCP. All computing is carried out at headquarters on an IBM 370/145. Schools are connected to headquarters by leased multidropped telephone lines. There are 13 schools (Fig. 1) with over 13,000 students in the school system. The size of the BCP data base for all schools together is over 280 megabytes. The department may be called upon to add more institutions to the network such as hospitals, MHMR Centers, and halfway houses. Thus, the network may grow from 14 sites (currently) to about 30 sites (when hospitals are included) to over 100 sites later. The department is continually developing new applications to increase the quality of care delivered by its institutions. TDMHMR expects the load on the system to grow as new applications are made available and as more institutions are served by the network. Measurements and analytic studies have shown that the 370/145 is already heavily loaded. Hence, TDMHMR initiated studies to enhance its data processing capabilities in a cost-effective manner to handle expected increases in the load.

System capabilities can be enhanced by (a) keeping the system centralized and improving the power of the central computer or (b) purchasing computers or intelligent terminals for some of the component institutions, thereby constructing a distributed system.

The number of possible designs (computer types, bandwidths, topologies) is very large, even for a 14-site system. Of course, the number of alternatives in both the centralized and distributed cases increases exponentially with the number of sites. It is impossible to consider all design alternatives. Hence, TDMHMR, in cooperation with a small group of faculty and students in the Computer Science Department of the University of Texas at Austin, started a project to develop

aids (programs) to help in the design process. These programs were to be used in designing the current system and in planning the growth of the system from its current configuration to a system with (potentially) over 100 sites. The emphasis in this paper is on a study of the trade-offs involved in designing systems of this magnitude (10 to 100 sites).

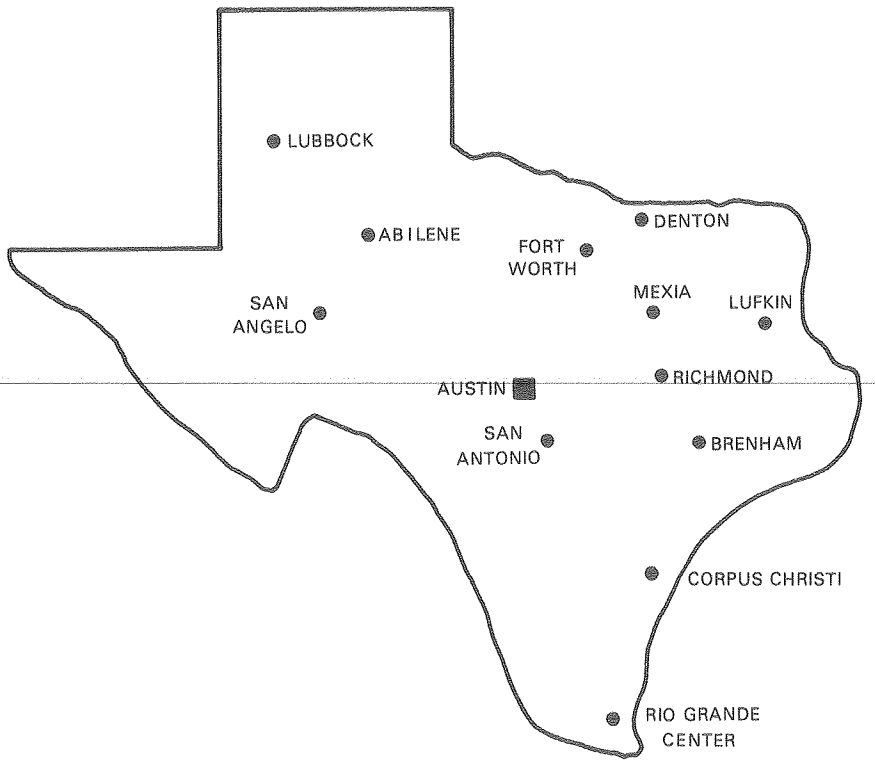


Fig. 1. Sites in a Distributed Network

Distributed computing is not a viable option in some organizations, given the current state of the technology. However, the data processing functions carried out by TDMHMR are such that consideration of a distributed system is justified. This is discussed next. Variants of this problem encountered by other organizations are discussed later.

Why Consider Distributed Computing?

Autonomous Institutions. The component institutions of TDMHMR generally carry out their data processing independently. Each school is responsible for recording, for example, the progression of behavioral characteristics of its students. Although staff members at a school may frequently access the BCP data base pertaining to their school, they will rarely access data bases dealing with other institutions. Thus, the data bases for the different institutions are independent in

the sense that queries and updates are directed towards one and only one of the institutional data bases. Note that the data bases for application function groups are also independent of one another. If it were not for the use of institutional data bases by headquarters staff or the rare transaction that requires access to data bases of more than one institution, there would be no need for a network: the system could consist of totally independent computers at each institution. Furthermore, there could be independent computers for each function group at each institution!

For practical purposes, we may assume that an institutional data base is used only by the institution and by headquarters. An institutional data base may be stored either at headquarters or at the institution itself. If the data base is stored at headquarters, then all transactions generated either by the institution or by headquarters are applied to the data base at headquarters. Another option is to store the master copy of the data base at the institution and a "slave" copy at headquarters. All updates are immediately applied to the master copy, but updates to the slave copy are applied after hours (8:00 a.m. to 5:00 p.m.) in a batched mode. Headquarters queries the institutional data bases primarily for statistical analysis and report generation and can use the slightly out-of-date slave copy without materially affecting the value of the reports generated. Thus, traffic from headquarters to the remote master copy is reduced to updates and urgent queries.

Based on the preceding discussion, the TDMHMR data processing organizational structure may be thought of as semiautonomous institutions coupled to headquarters but independent of other institutions. The locality in data processing functions makes TDMHMR an ideal candidate for distributed processing. Other candidates for distributed processing do not exhibit the same degree of functional and geographical autonomy; and example is provided later.

Independent Function Groups. Transactions generated at an institution in a given application function group can be processed remotely or at headquarters independent of where other application function groups are processed. This fact allows for a simple method of load-sharing between remote and central computers. Functions may be assigned to remote and central computers to reduce congestion and utilize all systems to the best advantage. Furthermore, the allocation of functions to different sites does not require complex directories. All that is needed is a table specifying the processing site of each function group for each institution. (There may be restrictions placed on the locations at which transactions in certain function groups are processed; for instance, if software for running a certain function on a remote computer is not available, then that function must be processed at the center.)

Predictable Special-Purpose Workload. The institutions carry out only a small number of well-defined functions. The application programs prompt the user by providing appropriate mask (panel) information and guide the user towards the appropriate query and data entry formats. Since the institutions' computing requirements are special-

0 0 1 0 5 1 0 4 1 7 9

purpose rather than general-purpose and since only a small number of application programs are used, the behavior of jobs in the system is relatively predictable (compared with research environments). Therefore, computing systems for the institutions can be designed to handle expected job (transaction) behavior; they do not need to be overdesigned to handle the rare large job.

Even though the kinds of transactions are relatively predictable, future transaction rates are very difficult to predict. Uncertainty about future transaction rates creates the need for the system to grow in a modular fashion.

Minicomputer Costs. The decreasing cost of minicomputer systems makes multiple computer systems an alternative to simple computer systems. Furthermore, several main frame manufacturers now provide competitive systems for remote data entry and inquiry.

Smooth Expansion of System Capabilities. The processing power of the system can be expanded smoothly by distributing minicomputers to the institutions, one at a time, as they are needed to handle the increasing load. This is an important consideration in organizations in which the load is increasing continually.

Disadvantages of Distribution. Managing a single centralized data processing group is simpler than managing widely dispersed staff. The increased complexities in managing computer professionals may result in significant costs, since the cost of professionals is a major part of the cost of a computing department. Personnel management may be a key issue in deciding between distributed and centralized systems.

We assume that computers at all remote sites (institutions) are software compatible with each other; the remote computers will probably be provided by a single vendor. Ideally, software will be transportable between central and remote sites; otherwise the cost of modifying the software to make it transportable must be considered in the design process.

Other Examples of Distributed Processing

Other examples of design problems similar to those faced by TDMHMR will be discussed before formally defining the scope of this paper. The high degree of autonomy in data processing functions in TDMHMR is probably not typical of most organizations. However, organizations with greater centralized control may also be candidates for distributed processing (and hence may be able to use the results presented in this paper).

Consider a hypothetical distributed inventory control system in which local data bases at branch warehouses are updated when items are removed from the branch warehouses, but where all inventory reordering is a headquarters function. A slave copy of all branch warehouse data bases is stored at headquarters. The headquarters copy is updated in a batched mode after hours by the branch warehouses. The reordering computation is carried out at headquarters, and the updated stock

reordering information is sent to the branch warehouses after hours; the branch warehouse data bases are updated in time for the next day's work. During peak load periods there is very little traffic between headquarters and the branches. During off-load periods there is traffic in both directions; however, this traffic is not a crucial performance consideration.

Even though this hypothetical system is more strongly coupled than the system being considered by TDMHMR, it is a candidate for distributed processing. Organizations that can be decoupled by allowing some sites to use slightly out-of-date information are candidates for the simple forms of distributed processing discussed here. Even if headquarters continuously updates branch data bases, the system may be a candidate for distributed processing if the fraction of transactions directed towards a branch data base is small compared to the transactions generated at the branch itself. This paper restricts its attention to organizations in which data is used in a relatively restricted geographical area during peak load periods.

Design Issues

Narrow and Broad Design Issues. Different vendors have slightly different constraints on the use of their equipment; for instance, some vendors do not permit certain terminals to be more than a fixed distance (say 2000 feet) from a processor, others only permit certain combinations of CPUs and disks, and so on. Furthermore, the range of options and constraints continually varies as new products are offered. Though the narrow design issues vary over time and from vendor to vendor, the broad issues are the same. This study is concerned with a quantitative analysis of the broad issues; nevertheless, this study should be helpful in configuring systems offered by specific vendors at specific times. Vendor-specific algorithms, which will have to be developed based upon the general approaches developed in this paper, are not discussed here.

Nomenclature. We will initially restrict our attention to organizations with a single headquarters and several branches, all of which are assumed to report directly to headquarters. We will begin by considering the simplest topology (star) (Fig. 2) and then will add complex features.

Processor Distribution. Which branches should have local computing facilities and what should the processing power (bandwidths) of these computers be? If the processing power of a branch computer is large, then a greater proportion of transactions can be processed remotely, leading to less congestion at the central site and better response times; this improvement must be traded off against cost.

Function Distribution for Load-Sharing Between Computers. What functions should be carried out at each branch computer and what functions should be carried out at the central computer? For instance, in the TDMHMR environment, the BCP data base for a school can be

stored locally and all BCP transactions can be processed locally, whereas all accounting data may be stored at the central computer and all accounting transactions may be processed at the center. In this case, the branch computer acts as a "concentrator" for accounting transactions, since it stores and forwards accounting transactions between terminals and the headquarters computer. This process of storing and forwarding data places an additional load on the branch computers. The goal is to share the load to optimize response times.

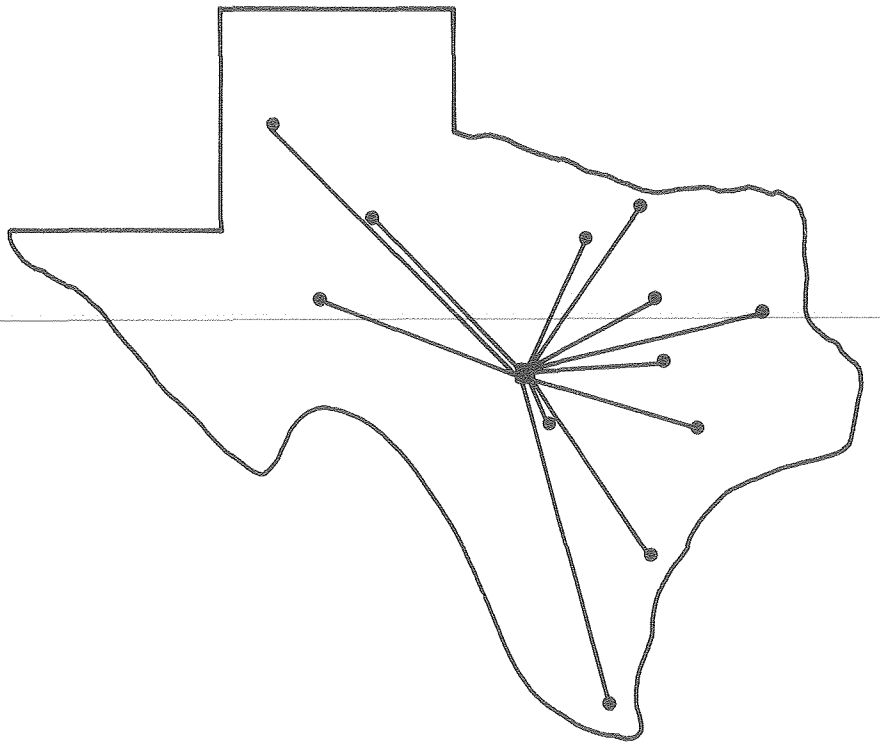


Fig. 2. A Star Topology

Line Bandwidths. What should be bandwidths of the lines between headquarters and the branches be? This depends upon modem costs as well as the bandwidths required by some vendors. Typical options in the TDMHMR environments are 1200, 2400, 4800, and 9600 band lines. Once again, higher service rates must be traded off against cost.

Regional Computers. We will now consider a star-star topology (Fig. 3).

Savings may be achieved by pooling the data bases and processing the requirements of many branches situated near one another. For instance, in the TDMHMR environment we could have a regional computer at San Antonio (Fig. 3) serving Corpus Christi, Rio Grande Center, and Richmond as well as San Antonio itself. Regional centers may be designated for reasons other than processing cost; for instance, a

district headquarters may be an appropriate site for a regional computer. The selection of regional centers is a key issue.

Intelligent Terminals. A significant portion of the traffic from a computer to a terminal may be screen formats (masks or panels) to prompt users. An intelligent terminal has the capability of storing screen formats and possibly also the capability to carry out some editing functions. Intelligence in a terminal reduces traffic (caused by prompting information and editing) between the terminal and the computer, and it reduces the processing requirements at the computer; thus, intelligent terminals serve much the same purpose as branch computers. We make the following distinction between intelligent terminals and computers: a data base must be stored at a computer; a data base cannot be stored at a terminal. Hence, a transaction must be shipped to a computer for processing. Nevertheless, there are similarities between the problems of selecting intelligent terminals and branch computers.

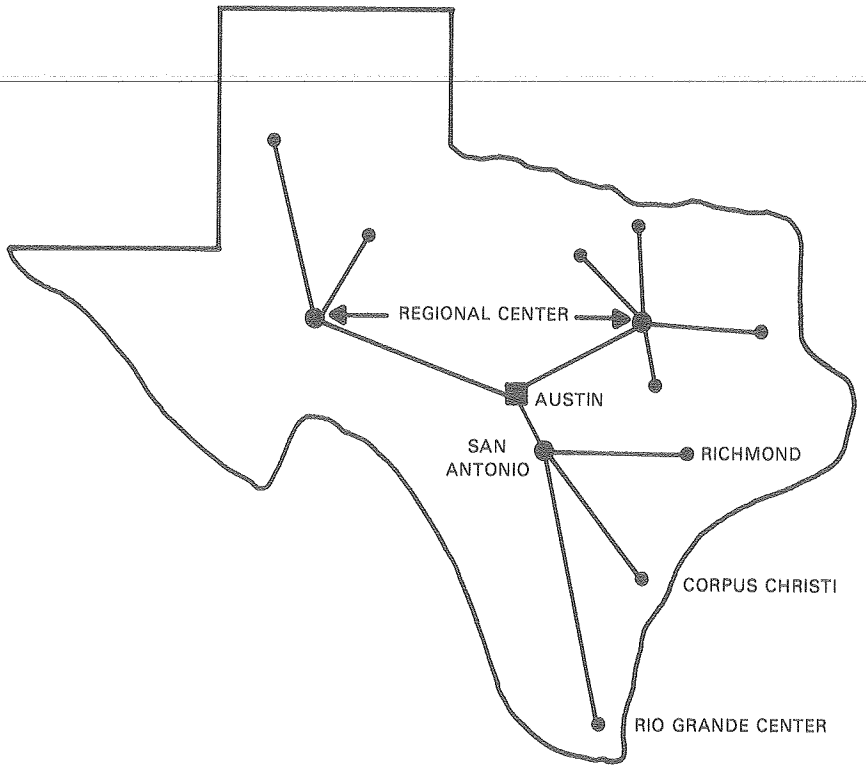


Figure 3. A Topology With Regional Centers

Multidropped Lines. It may be cost effective to multidrop several computers on a single leased line. In this case the lines connecting the central computer to the regional computers form a tree, and the lines connecting the terminals to each regional computer may also form a tree (Fig. 4). The polling mechanism involved in shipping messages along the multidropped line must be taken into account in

0 0 1 0 5 1 0 4 1 8 1

computing delays. Determining the best network topology is a design problem.

Similarity to the Concentrator Location Problem. Concentrators are devices that store data coming in from several low-speed channels and ship the data to the central computer on a single high-speed channel. Line costs may be decreased by the judicious placement of concentrators. Though our problem is similar to the concentrator location problem (61), there are some important differences. Issues of data base sizes and disk capacities must be considered in the distributed processing problem. Furthermore, decisions regarding the allocation of functions to computers do not arise in the concentrator location problem. Despite the differences, many of the ideas dealing with concentrator location may be applied to our problem. In particular, a branch computer may act as a concentrator for certain classes of transactions, since the only function of the branch computer (for these transaction classes) may be to route transactions to the headquarters computer.

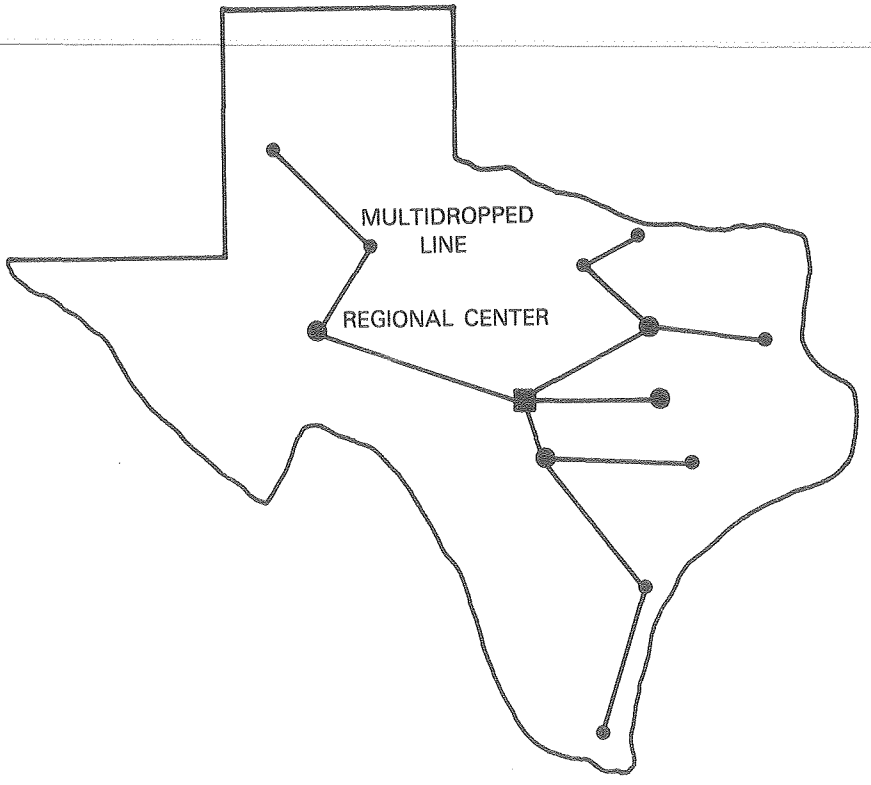


Fig. 4. A Topology With Regional Center and Multidropped Lines

Earlier Work

A great deal of work has been carried out on distributed data bases. General approaches are described by Chang (16), Mahmoud and Riordan (44), and Maruyama and Tang (49). Our problem is a subset of

the problems considered by the general approaches, since we restrict the network topology to a tree structure. However, the very generality of earlier approaches does not facilitate a detailed study of the small set of design variables considered in this paper. Furthermore, the complexity of the general problem precludes the development of reasonably quick optimal algorithms. The simpler problem described in this paper occurs often enough to merit the development of techniques specifically for this problem. We see our approach being used in parallel with the general approaches: the general approach is used for designing general distributed data bases such as SDD-1, and our approach is used for the restricted problem.

A significant amount of work has been carried out on the allocation of files to nodes in a distributed data base system (7,14,18,19,42). The general problem is known to be NP-complete (62), which implies that the computation time for all optimal algorithms will (in the worst case) increase exponentially with the number of sites in the distributed system, given the current state of the art in combinatorics. The cited work on file allocation has not simultaneously considered all the design issues: processor allocation, topology, line bandwidth, and queueing. In this paper, we attempt to consider all these issues together. The earlier work on file allocation has been concerned with distributing files after the network hardware is in place, whereas we are concerned with selecting the network hardware too. However, the references cited have been concerned with a very general distributed data base environment, whereas we are concerned with a limited problem. Fig. 5 depicts our work in relation to earlier work on distributed data base design.

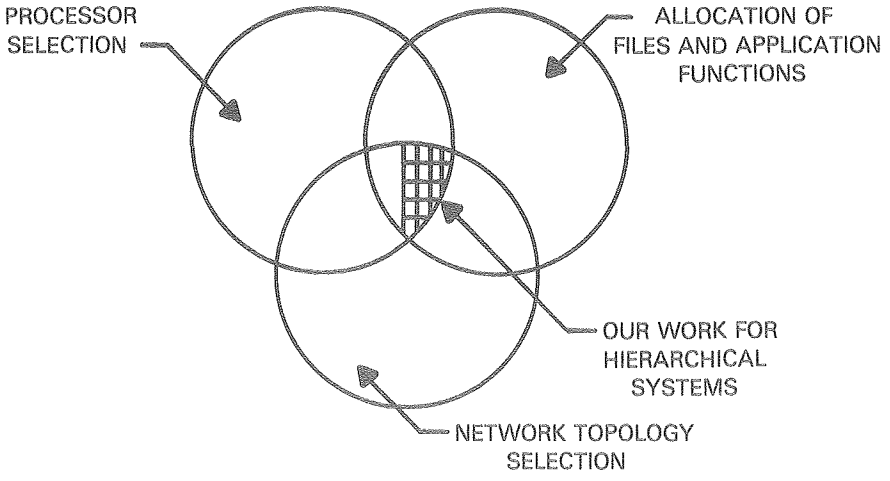


Fig. 5. Our Work in Relation to Previous Work

The work on concentrator location (see the many references in Tang and Woo [61] for instance) is certainly relevant to this work. For a general discussion of distributed system design, see Schwartz (60).

0 3 : 0 5 1 0 4 1 8 2

The Design Process

The number of alternative designs is too large in even a medium-sized problem (14 sites) to allow for a brute force search of all possible alternatives. If there are 5 possible computer configurations for each site and 4 alternative line bandwidths, then a system with 13 remote sites has $(5 \times 4)^{13} \approx 10^{17}$ alternatives. The goal of our study is to develop design aids that select a small number of designs, from the vast number of alternatives, for intensive analysis. Our algorithm does not attempt to produce a single "optimum" design. It should be emphasized that the final design may be influenced by factors other than cost and performance; indeed, these other factors may be paramount. Our programs are CAD tools that aid (but certainly do not supplant) the designer. The process by which these tools should be used are discussed next.

Specification of Alternatives. The design team considers one set of compatible remote computing equipment at a time. Normally, all remote equipment will be provided by a single vendor. The design team produces alternative designs for each remote computing family and, finally, compares the designs for different families. Examples of families of remote computing equipment are the IBM 3790, TI DXS, and FOUR PHASE IV/40 and IV/70 (all of which support 3270 terminals); note that this list is not exhaustive by any means, and we shall not compare the different families in this paper.

The set of alternative line bandwidths and remote computer configurations are specified by the design team. Lines are specified in terms of bandwidth and cost. Computers are specified in terms of cost, secondary memory (usually disk) size in megabytes, and processing power (in transactions processed per unit time). If there are several kinds of transactions, processing rates must be specified for each type. It is difficult to estimate transaction processing rates; these issues are discussed in the section on "Practical Problems in Systems Design."

The design team will normally consider a range of alternate specifications, since performance parameters cannot be estimated precisely.

Specification of Load. The design team specifies the load offered to the network at all the branches in the system (in units of transactions per unit time). If there are different types of transactions, then it is necessary to specify the load for each type. The design team will normally consider several types of load estimates; average daily peak load and peak load two years hence are examples of such estimates.

Design Selection. For each set of system configurations and load estimates, a cost-response time curve (Fig. 6) is computed using the algorithms presented here. The design team will select a range of points from the curve for further intensive study.

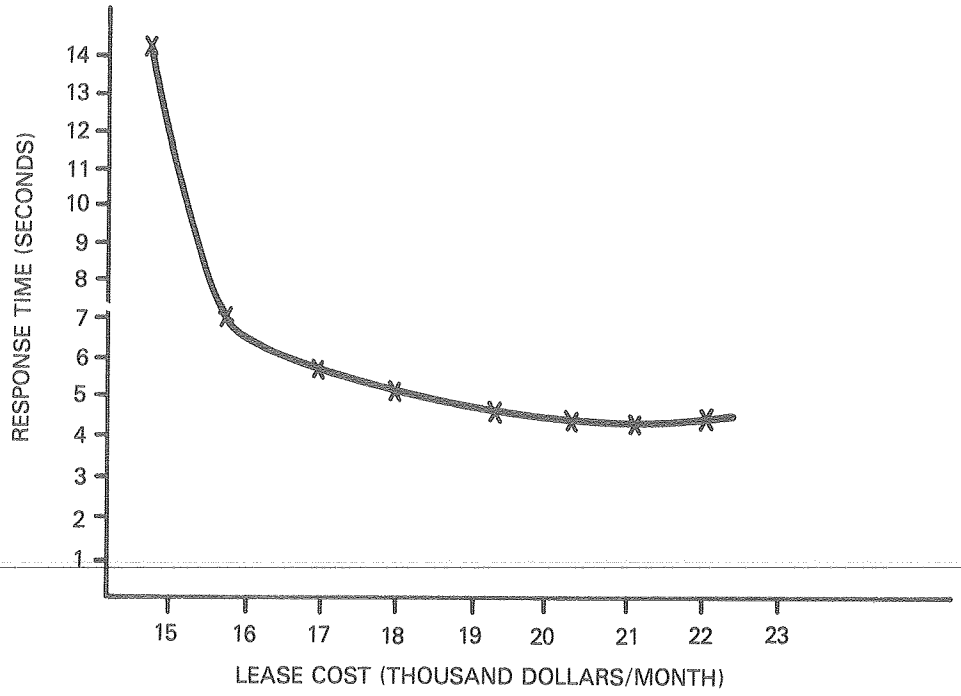


Fig. 6. Cost-Response Time Curve

Robustness. The small set of designs selected by the algorithms will be evaluated by simulation and other methods to study the sensitivity of performance measures to fluctuations in the load, reliability of computer systems and lines, and other parameters of interest. The orderly growth of the network as new application functions are offered and as new branches are added is a vital concern during this phase of analysis. Finally, a specific design is selected for implementation by the design team. It must be emphasized that although the design team may choose to use the algorithms in designing networks, the team will also bring the bear their knowledge of factors that the algorithms do not take into account.

THE ALGORITHMS

The algorithms are outlined in this section. Details are found in Yeh (65).

Star Topology - Exact Solutions

Consider the simplest case of the star topology. We assume initially (for simplicity of exposition) that each branch may either have a branch computer or a passive terminal (but not an intelligent terminal). We also assume that, for each branch, any application function can be processed at that branch or at the central computer

independent of other branches. Problem variations are considered later. Each site (branch or headquarters) is assumed to use F application function groups (such as BCP, TAS, and PS in the TDMHMR system). A data base is associated with each function (we use function rather than function group for brevity) and each branch; each data base may be stored either at the branch or at headquarters independent of the location of the other data bases. Thus, there are 2^F possible ways of storing the F data bases belonging to a branch. If there are E alternative local computer configurations and H alternative line bandwidths, then there are a total of $T = HE2^F$ alternatives for each branch and its associated communication line. A selection from among these T alternatives will be called a branch design. With F = 3, H = 4, and E = 5, we have T = 160 possible branch designs. If each branch could be designed independently, then the overall design problem would be tractable, since the computation time for the total system would only increase linearly with the number of branches. Indeed, our goal is to develop an algorithm that computes cost-response time curves in time linearly proportional to the number of branches. Unfortunately, one branch design impacts other branch designs, so we need to consider from among T^B alternate network designs, where B is the number of branches. The key to the solution process is to decompose the overall system design into branch designs that are relatively independent of each other. One branch design impacts another in only two ways:

- Cost - The greater the expenditure on one branch, the less the amount available for others.
- Central load - The delay that a transaction experiences at the central computer increases with the total load at the central computer. If, in one branch design, many functions are processed at headquarters, then the central load imposed by that branch will be high, and hence the central delay experienced by transactions from other branches will be relatively large. The rate at which transactions arrive at the central computer from a branch will be called the central load requirement of that branch.

The algorithm hinges on determining the best branch design for each possible branch cost and central load requirement, thus decoupling branch designs, as will be explained later.

Let Q be the mean number of transactions in the system (i.e., those either queueing up for processing or being processed by some resource in the system). Let D be the total rate at which the transactions are generated throughout the system (by branches and by headquarters). Then by Little's law (see Kleinrock [34] for example), the average response time is Q/D. Since D is a constant, minimizing the mean response time is equivalent to minimizing Q. Q is the sum of the mean queue lengths at (a) each branch computer, (b) the channel from each branch computer to the central computer (assuming full-duplex lines), (c) the channel from each central computer to the branch computer, and (d) the central computer (Fig. 7).

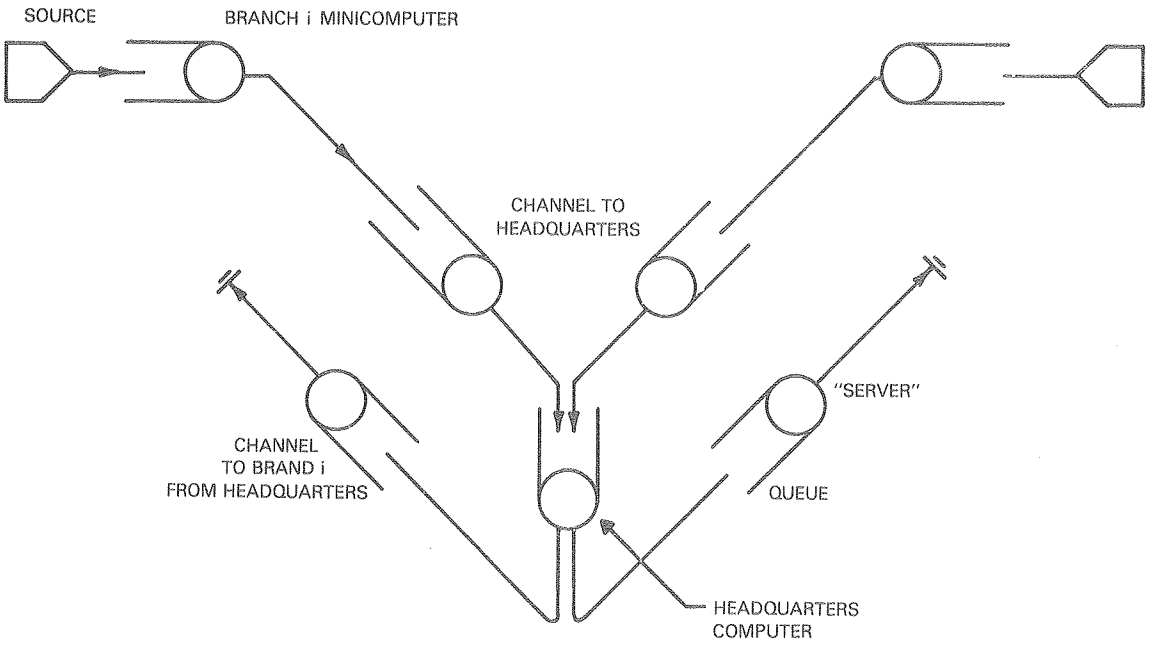


Fig. 7. The Queues in the System

The mean queue length for a given branch is defined to be the sum of the first three (of the preceding four) components for that branch (Fig. 8).

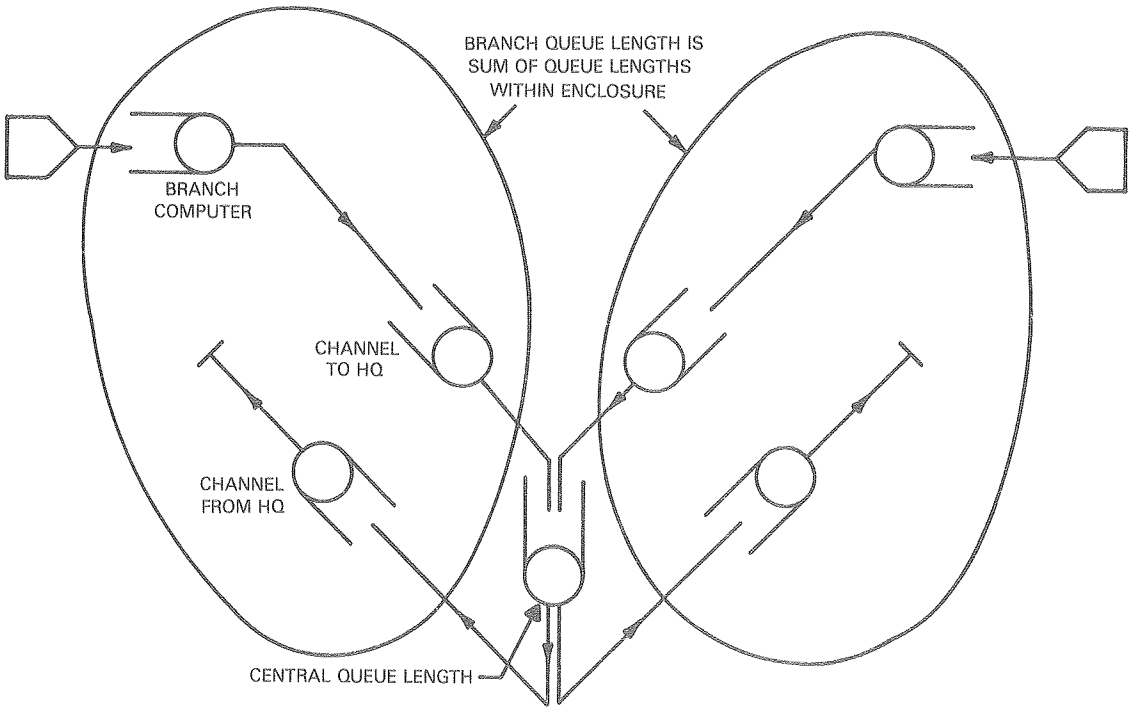


Figure 8. Branch Queue Lengths and Central Queue by Length

0 0 : 0 5 1 0 - 1 8 4

Thus we wish to minimize the sum of the mean queue lengths at headquarters and all branches.

Similarly, the cost of a given branch is defined to be the sum of the costs of the local computer at that branch (if any) and the full duplex line and associated modems required to connect that branch to the central computer. Thus, the total system cost is the sum of the costs of the branches and headquarters.

We generate each of the T possible branch designs for a given branch, and for each design we compute the branch queue length, the central load requirement, and the branch cost (Fig. 9). Methods for carrying out this computation are discussed in the section on "Practical Problems in System Design."

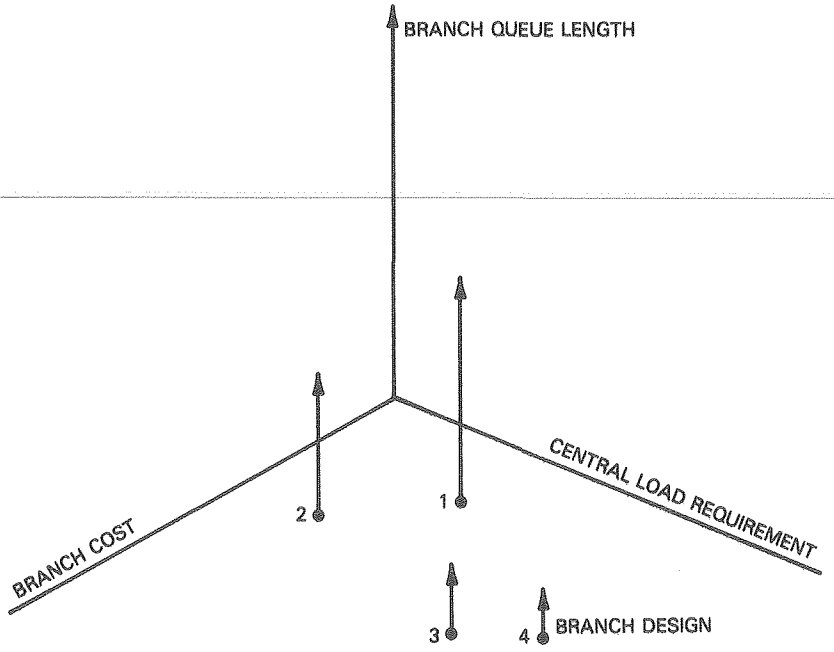


Fig. 9. Selection of Branch Configuration

This process is repeated for each of the B branches, and thus a total of BT branch designs are evaluated.

For a given branch cost c_b and central load requirement ℓ_b, let us search all feasible designs for branch b (in which the branch cost does not exceed c_b and the central load requirement does not exceed ℓ_b) and pick the best design (i.e., the one with the smallest branch queue length); $g_b(\ell_b, c_b)$ is defined to be the branch queue length for that best design (Fig. 10).

An error is introduced by carrying out computations only at grid points; the coarser the grid the greater the error but the less the computation time and memory requirements. Algorithms for computing upper bounds on the amount of error are discussed in Yeh (65).

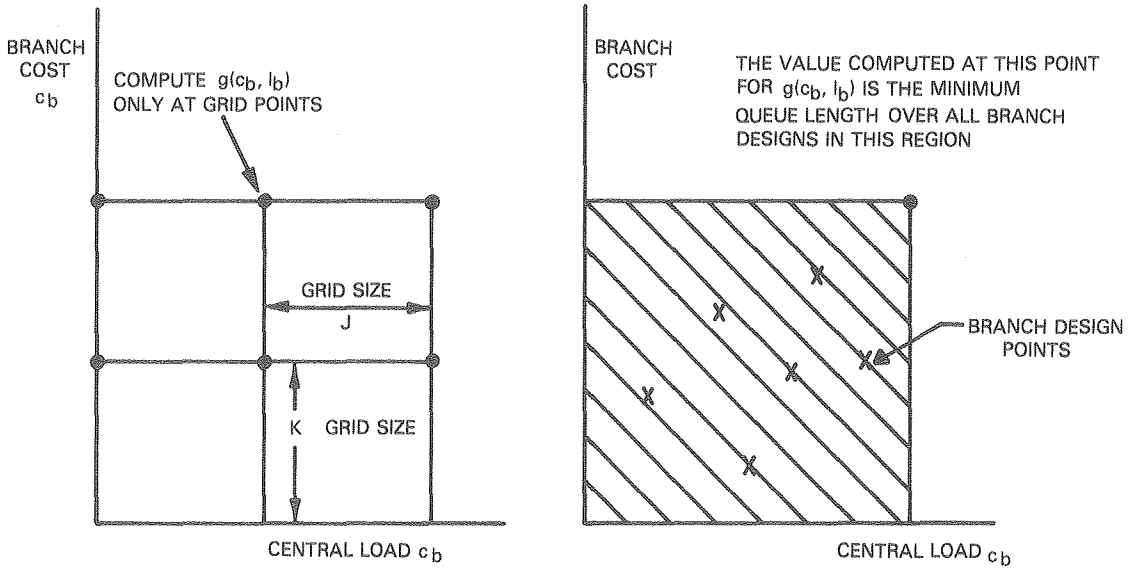


Fig. 10. Computation of $g_b(\ell_b, c_b)$ at Grid Points

Assume that $g_b(\ell_b, c_b)$ has been computed for all branches b, central load ℓ_b, and cost c_b. Let there be I alternative central computer configurations under consideration, and let the cost of the ith configuration be A_i, i=1,..,I. Let L be the total load on the central computer (transactions per unit time). L is the sum of the central load requirements over all the branches and $L_{CENTRAL}$, which is the load generated at headquarters for the central computer. The design team specifies $L_{CENTRAL}$; it is not a design variable. Let $q_i(L)$ be the mean queue length at the central computer, given that the total central load is L, and given that the ith central computer configuration is used. Let Q(C) be the optimum mean number of transactions in the entire system, given an expenditure of C units for the system. Our goal is to compute Q(C) for a range of values of C (Fig.6). The design problem is:

Minimize Q(C)

$$\text{so that: } \sum_b g_b(\ell_b, c_b) + q_i(L) = Q(C) \tag{1}$$

$$\sum_b \ell_b + L_{CENTRAL} = L \tag{2}$$

$$\sum_b c_b + A_i = C \tag{3}$$

where $\ell_b \geq 0$, $c_b \geq 0$, b = 1,..,B; i is in {1,..,I};
and A_i is the cost of the ith central computer configuration.

0 3 : 0 5 1 9 4 1 8 5

Equation (1) states that the mean number of transactions in the system is the sum of mean queue lengths over all the branches and the mean queue length at the center. Equation (2) states that the total load on the central computer is the sum of the central load requirements over all branches and the load generated at headquarters. Equation (3) states that the total cost of the system is the sum of the branch costs and the central computer cost. The variables are the branch costs for each branch, the central load requirement for each branch, and the choice of the central computer.

We solve this problem recursively (by dynamic programming). The solution procedure provides some insight into the basic design trade-offs and we therefore discuss the procedure here. Our goal is to divide the two resources, central load L and cost C, in an optimal fashion between the branches and the central computer. For every cost c and central load ℓ, we shall first partition the resources c and ℓ between the branches, e.g., branches 1 and 2, so that the sum of the queue lengths of these two branches is minimized (Fig. 11). We might think of this process as creating a "superbranch" {1,2} and determining

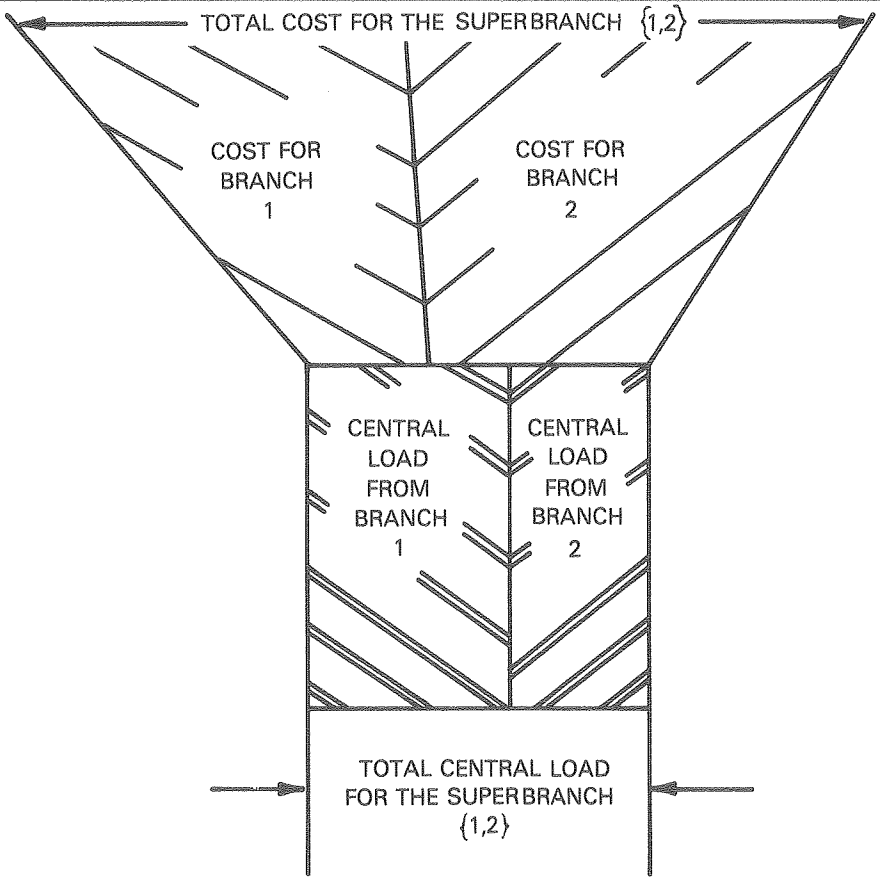


Fig. 11. Creation of Superbranch {1,2}

the minimum branch queue length, given a branch cost and central load requirement for this superbranch. For every cost c and central load ℓ, we next partition these two resources between the superbranch {1,2} and some other branch, e.g., branch 3; this can be thought of as creating an optimum design (i.e., allocation of resources ℓ and c) for superbranch {1,2,3}. This process is continued until the cost and central load have been optimally partitioned among all branches. In the final step, we divide the resources between remote computing equipment (branches) and the central computer.

Let $G_b(\ell,c)$ be the minimum total queue length for branches 1,..,b so that the total central load requirement from these branches does not exceed ℓ and the total cost of these branches does not exceed c. In other words:

$$G_b(\ell,c) = \text{minimum} \sum_{J=1}^{b} g_j(\ell_j,c_j) \tag{4}$$

$$\text{so that} \quad \sum_{J=1}^{b} \ell_j \leq \ell \tag{5}$$

$$\text{and} \quad \sum_{j=1}^{b} c_j \geq c \tag{6}$$

By definition

$$G_1(\ell,c) = g_1(\ell,c) \tag{7}$$

Furthermore, we can show by induction

$$G_b(\ell,c) = \underset{\ell',c'}{\text{minimum}} \{G_{b-1}(\ell',c')+g_b(\ell-\ell',c-c')\} \quad \text{for } b=2,3,..,B \tag{8}$$

Equation (8) corresponds to designing a superbranch {1,..,b}, given central load requirement ℓ and cost c, by a brute force enumeration of all possible ways of dividing the resources between superbranch {1,..,b-1} and branch b. We apply equation (8) for b=2,..,B. Now $G_B(\ell,c)$ is the minimum mean number of transactions in all remote equipment (branch computers and lines), given a remote computing cost of c units and a central load requirement of ℓ units from all remote equipment. Now we divide the total system cost C between central and remote equipment. Assume that the ith central computer configuration is used and that the total central load requirement from

0 0 3 3 5 1 0 4 1 8 6

all remote equipment is ℓ. Since the cost of the central computer is A_i, the cost of all remote equipment cannot exceed $C-A_i$. Hence, the mean number of transactions in remote equipment is $G_B(\ell, C-A_i)$. The total central load $L = \ell + L_{CENTRAL}$. Since we are using the ith central computer configuration, the central computer queue length is $q_i(\ell + L_{CENTRAL})$. Hence,

$$Q(C) = \underset{i,\ell}{\text{minimum}} \{G_B(\ell, C-A_i) + q_i(\ell + L_{CENTRAL})\} \qquad (9)$$

The computation corresponding to all the equations is carried out only at grid points. Hence, the computation of $G_B(\quad)$ (and hence also the computation of the cost-response time characteristic) increases linearly with the number of branches. The computation also increases linearly with the number of branch designs, which increases linearly with the number of alternate line bandwidths H, linearly with the number of local computer configurations E, but exponentially with the number of application functions. Note also that the computation increases as the reciprocal of the grid widths J and K (Fig. 10), and that nonzero grid widths introduce error.

The problem is NP-complete (32) (since we can reduce the knapsack problem to it); hence, given the current state of combinatorics technology, an error-free optimum algorithm must have a computation time that increases exponentially with the number of branches. Such algorithms are generally computationally intractable. As a consequence, a tractable algorithm must permit some error (because of positive grid widths); methods to bound this error are discussed in Yeh (65).

Heuristics for Star Network Designs

To be effective, we found it necessary to develop CAD systems that designers could interact with on a continuous basis. The exact algorithm discussed earlier is too slow to be used in an interactive mode and, hence, we were forced to develop heuristics. We developed three heuristics and compared them against each other and against the exact solution; only the best heuristic is described here; descriptions for the other two are found in Yeh (65). We next describe a heuristic that has two initialization steps and two interactive steps.

Step 1 (Initialization) - Allocation of Data Bases. We consider each of the branches in turn, and for each branch we consider each application data base independently. The data bases used by a branch can be located either at the branch itself or at headquarters. We assign the function to that site (branch or headquarters) between the branch and headquarters. The allocation of functions at this stage is made without regard to the capacity of the central and branch computers,

line bandwidths, or the amount of computation required by the applications. In general, this step tends to place all the branch data bases at the branches.

Step 2 (Initialization) - Assignment of Processors to Regional Centers and Headquarters, Terminals for Branches and Line Bandwidths. Initially, each resource (central or branch computer or line) is assigned the minimum capacity necessary to handle the load specified by the algorithm at the end of Step 1. If there are any funds left over, then all queue lengths (at central and branch computers and lines) are inspected, and the resource with the largest queue length is selected; if there are enough funds available, then this resource's configuration (model) is upgraded to the next higher model number. This step is repeated until the available funds are insufficient to upgrade any resource.

Step 3 (Interactive Mode) - Reallocation of Data Bases. We alter data base locations one branch at a time. We inspect the branches in decreasing order queue length. We consider each application data base for each branch in turn, and we alter its current location (from the branch to headquarters or vice versa) and determine if there is an improvement in the response time that is due to the switch in location. If there is an improvement, then the current configuration is altered. Otherwise, the original configuration is retained. After the data bases for all the branches have been considered, we go to Step 4.

Step 4 (Interactive Mode) - Reallocation of Resource Capacities. We now attempt to improve response time by downgrading one device and using the funds obtained by downgrading this device to upgrade one or more other devices. All devices are inspected for potential downgrading starting with the devices with the smallest queue lengths. Similarly, all devices are inspected for potential upgrading in decreasing order of queue length. If Step 4 results in improvements in response time, then we go back to Step 3. We iterate through Steps 3 and 4 until no further improvement is obtained.

The heuristic has the option of incorporating a certain amount of backtrack capability. At every step in the algorithm we save the two best designs, and in general we may save the k best ($k>1$). We attempt to improve all the saved solutions through the use of Steps 3 and 4.

The entire heuristic is repeated for each specified cost to produce a cost-response time characteristic.

Table I compares optimum and heuristic methods. The columns of the table need some explanation. Maximum deviation is the maximum percentage increase among all test cases in the cost of a design produced by the heuristic in comparison with a design produced by the optimum algorithm. Percentage hit is the fraction of times that the heuristic produced a solution as good as that of the optimum solution.

The small average deviation and the relatively large maximum deviation suggest that the heuristic is good, but that there is still room for improvement. The considerably smaller rates of increase of memory and computation time requirements of the heuristic in comparison with the corresponding rates of the optimum solution are noteworthy. Work is continuing in this area.

TABLE I

COMPARISON OF HEURISTIC AND OPTIMUM SOLUTIONS

METHOD USED	PROBLEM SIZE (NUMBER OF NODES)					
	5		10		20	
	NUMBER OF PROBLEMS COMPARED FOR THIS SIZE					
	10		10		10	
	OPT	HEUR	OPT	HEUR	OPT	HEUR
CPU time in seconds (CDC 6600)	27.7	1.6	58.3	2.4	131	4.2
Space in (octal) words for tables (excluding compiler)	40,500	15,500	40,500	15,600	41,000	15,700
Average deviation (percentage)	-	3.2	-	3.7	-	4.1
Maximum deviation (percentage)	-	21	-	29	-	17
Percentage hit	-	53	-	38	-	21

Fixed Topology With Regional Computing Centers

We will now consider distributed systems with a headquarters computer and one or more regional computing centers (see the section on "Design Issues") where two or more branches may share a regional center. In some cases the regional centers may be selected for managerial purposes not having to do with information systems considerations. However, in general (and the TDMHMR case is an example), the problem of partitioning the system into regions and selecting regional centers is a difficult one. In this section, we assume a fixed topology, i.e., we assume that we are given a fixed headquarters, a fixed set of regional computing centers, and that branches are preassigned to regional centers (Fig. 12). We shall postpone consideration of the problem of partitioning sites into regions to a later section.

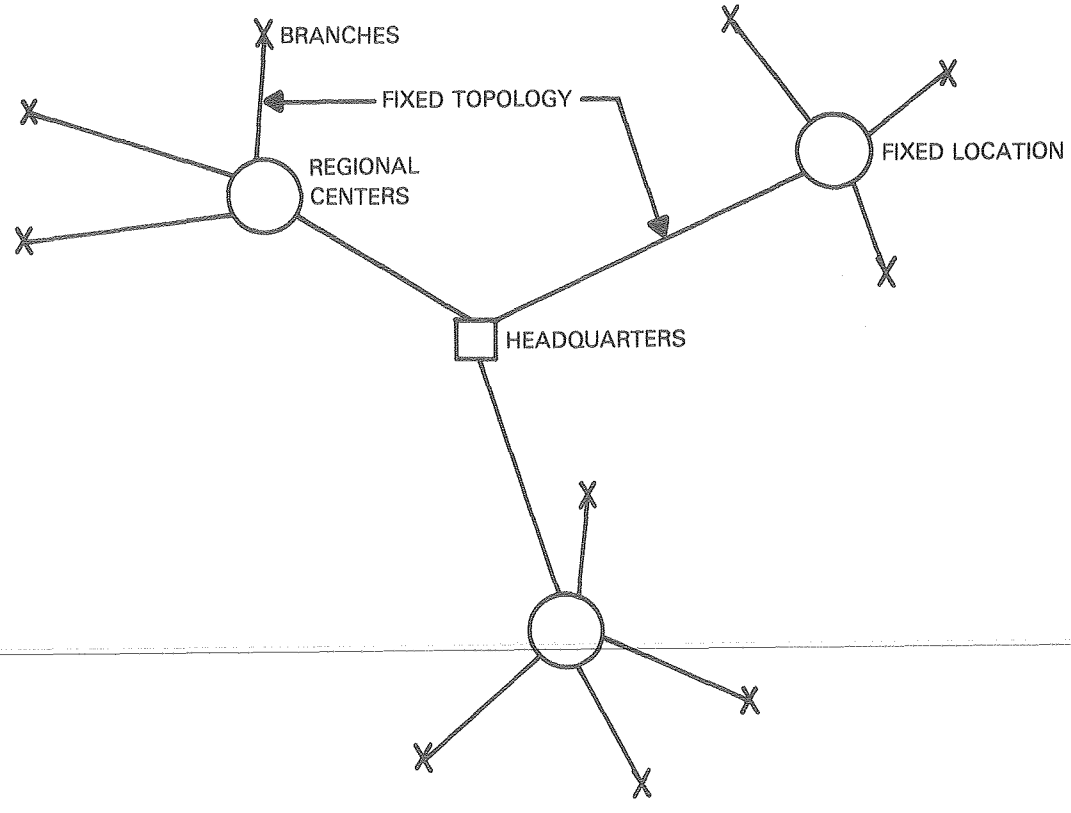


Fig. 12. Designing a System With Regional Centers and Fixed Topology

We think of a "branch" as a terminal that is directly connected to a preselected regional center. There may be several "branches" at the same location. A branch may have either an intelligent terminal or a passive terminal. If a branch has an intelligent terminal, all screens are assumed to be stored at the terminal, and basic editing functions are also carried out at the branch itself. The traffic between a branch and its regional center is determined by the type of terminal (intelligent or passive) at the branch and the rate at which transactions are entered at the branch. Screens and editing functions are also carried out at regional centers. Thus, screens and editing information are transmitted between a passive terminal and its associated regional center (but never to headquarters through a regional center).

The problem is to select (a) terminals (intelligent or passive) for the branches, (b) bandwidths for the lines connecting terminals to regional centers, (c) computers for the regional centers, (d) application functions to be performed at each regional center, (e) bandwidths for the lines connecting regional centers to headquarters, and (f) possibly the computer for the headquarters as well.

Assigning Functions to Regional and Central Computers. In general, a transaction generated at a branch can be processed at a regional center or at headquarters independent of where transactions

from other branches (served by the same regional center) are processed. For instance, if a regional center at San Angelo also serves terminals at Lubbock and Abilene, then BCP transactions from Lubbock could be processed at headquarters in Austin, while BCP transactions from Abilene are processed at the regional center in San Angelo (Fig. 13).

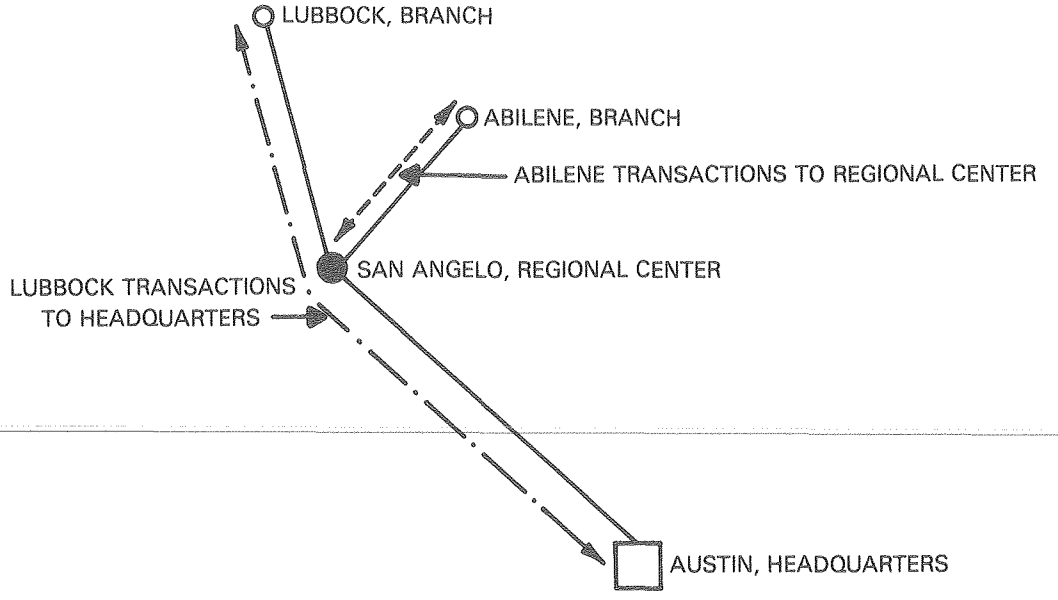


Fig. 13. Splitting an Application Function Between a Regional Center and Headquarters

In this paper, we will restrict attention to systems where an application function is either entirely at a regional center or not in the regional center at all. Thus, the previous example, with Lubbock BCP transactions processed at Austin and Abilene BCP transactions processed at San Angelo, will not be considered.

We will allow different application functions to be partitioned between regional and central computers. For instance, a regional center may serve all BCP transactions generated by the terminals it serves and none of the TAS transactions. Another center may serve all TAS transactions and none of the BCP transactions.

The disadvantage of limiting regional centers to handle all or none of an application function is that it limits control of the degree of load-sharing between regional and central computers. Advantages are that data base administration is simplified and decisions regarding routing transactions to their appropriate destination are trivial.

In some cases, functions may be allocated to central or regional computers prior to the design. (For instance, when a new function is developed, it may be initially offered at the central computer. Then, with increasing experience and funds, the function may be gradually pushed out to remote sites until it is processed entirely at remote

locations.) The design problem in this case is essentially the same as when functions are not preallocated to sites, except that the number of variables is reduced.

Design Algorithms for Networks With Regional Centers. We will not discuss design algorithms for this class of systems here, since the algorithms are essentially recursive applications of the star algorithms discussed in the section on "The Algorithms"; the star algorithms are applied in designing the network between terminals and their regional computers and then again in designing the network between regional computers and headquarters. For details, see Yeh (65).

Selecting Regional Centers

The problem of designing a network including the selection of regional centers is a very difficult one. (It is as difficult as the concentrator location problem whose complexity has been analyzed in Van Sickle and Chandy [62]). We developed a (slow) optimal algorithm that we used to solve very small problems, and we also developed heuristics. The optimal algorithms are implicit enumeration schemes based on dynamic programming. They are not discussed here; see Yeh (65) for details.

In general, any two branches could belong to the same region. However, in specific instances the design team may choose not to consider certain alternatives. Consider the TDMHMR example (Fig. 14). The designers may choose to ignore topologies in which Richmond and Lubbock belong to the same region. In some cases, it may be possible to define clusters (Fig. 14) and to preassign terminals to groups where a terminal belongs to one or more groups. We permit any set of terminals within one group to belong to the same region, but we do not allow two terminals (which do not belong to the same group) to belong to a common region. Grouping is merely a scheme to reduce the number of alternatives that need to be considered. The design team defines groups before using the algorithms. We will now discuss heuristics for the overall design problem.

The heuristic works in the following way. We first partition the set of all sites into clusters or regions; the clustering algorithm attempts to put sites that are close together into a single region; the algorithm is described later in greater detail. We then design the communication system for each region, i.e., select the regional center, line bandwidths, and line topology (if multidropping is permitted). We next select the processors at headquarters and the regional centers and design the communication system between regional centers and headquarters. To do this, we treat an entire region as being a single site located at the regional center, and this site is assumed to generate transactions at a rate equal to the sum of the rates of all sites within the region. Thus, this part of the design reduces to the previously discussed problem of connecting several branches to headquarters. The heuristic repeats these steps until no further improvement is obtained. We now describe the algorithm in more detail.

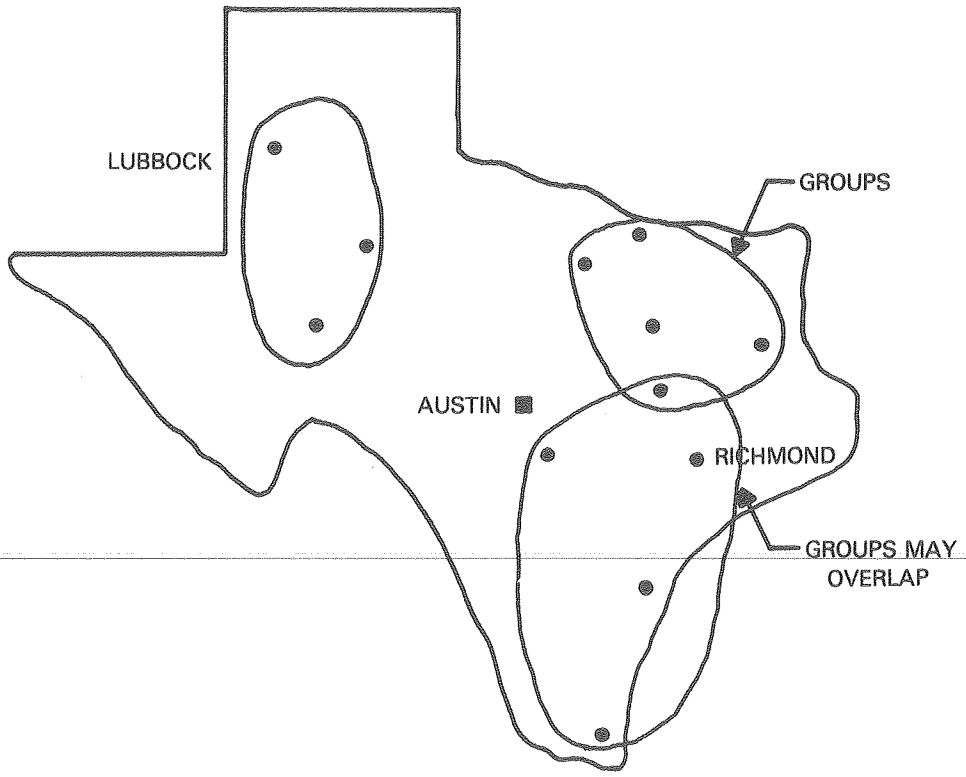


Fig. 14. Sites Grouped Together by the Designer

Clustering Heuristic. First a threshold distance D is specified; if a site S is separated by a distance exceeding D from every other site, then S is not attached to any region, or equivalently, S is assumed to be a "region" by itself; such sites are called singletons. For each site that is not a singleton, we determine its nearest neighbor. A site and its nearest neighbor are placed in the same region. Thus, if site B is the nearest neighbor to site A, and site C is the nearest neighbor to site B, then A, B, and C are assigned to the same region. However, the headquarters site is not assigned to any region. If some site's nearest neighbor is headquarters, then the connection between that site and headquarters is not made. All the connected components that exist after all the sites have been considered are tentative regions (Fig. 15). These regions may be modified as the algorithm proceeds.

It is possible that the regions constructed in this way have a very large number of sites. Given a threshold value N, any region with more than N sites is partitioned into two or more regions where no region contains more than N sites. Rather than using a threshold on the number of sites, a threshold on the total volume of traffic in a region could also be used.

The heuristic was compared against the optimum solution (see Table II and Fig. 15) for a number of very small problems. It must be emphasized that the value of the comparison is limited because of the restricted sizes of the problem.

TABLE II

RESULTS FOR TEN PROBLEMS UNDER SIX NODES

Method	Optimal Solution	Heuristic
Time (sec)	117.52	4.97
Space (byte)	61,000	32,600
Accuracy	-	5.8%
Max Dev	-	14%
Hit	-	24%

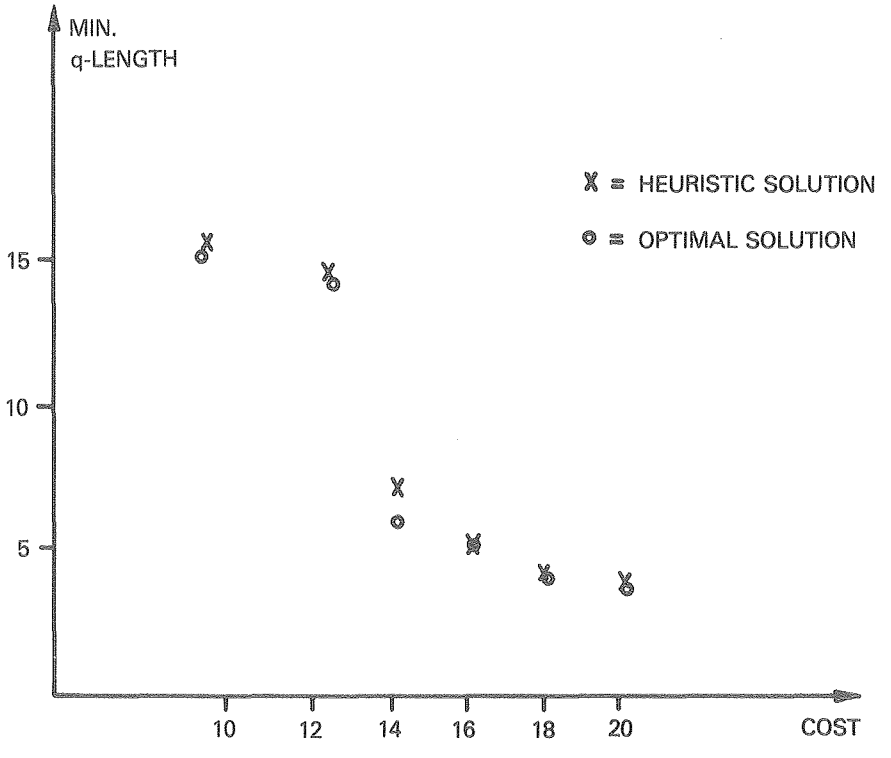


Fig. 15. Graphical Comparison of Optimum Solution and Heuristic for a 5-Node Problem

0 0 - 3 5 1 0 4 1 9 0

PRACTICAL PROBLEMS IN SYSTEMS DESIGN

The most difficult problem is that of obtaining data on existing and proposed traffic patterns so that the inputs to the design algorithms are representative of reality. It is helpful to consider three subclasses of this problem.

Existing Application Functions

It is possible to make measurements on transactions that are currently being run on the central computer. The problem is to extrapolate from these measurements to predict the behavior of the same transactions on the minicomputer systems that may be purchased for the branches. Ideally, measurements should be made directly on the minicomputer systems running benchmarks of typical peak loads, but rewriting portions of the applications software to run on the minis can be a time-consuming process. Admittedly, benchmarks must be run before a system is purchased. However, while we are searching through the myriad design alternatives offered by several vendors, a compromise solution is to make a gross representation of system resources required by transactions. For each class of transactions, we determine the number of disk accesses, the sizes of the files transferred, and the average amount of CPU time. From this data, we estimate the average CPU time, the number of disk accesses, and the average time that a disk is held on the proposed minicomputer configuration. From this data, we can estimate the expected number of transactions that can be processed by a minicomputer configuration running under saturation conditions. For details, see Yeh (65).

What Is the Throughput of a Branch Computer?

The input to the design algorithms describes a computing system by its cost, disk capacity, and its throughput capacity (transactions per second). This is, of course, a very gross representation of a computing system; yet a more detailed representation would make the design algorithms extremely complex. Each possible computing system configuration (CPUs, main memory, disks) is represented by a different "computer" in the input to the design algorithm. For each configuration, we carry out a queueing network analysis to determine the maximum throughput that the system can deliver; this throughput is assumed to be the capacity of the computing system.

Estimating the Behavior of Proposed Application Functions and Estimating Future Load

This is the most difficult aspect of the entire design process. The estimates must be provided by application programmers and data center managers. Because of the uncertainties associated with this data, it is important that it be possible to change a distributed system in a modular fashion, and that it be relatively insensitive to changes in traffic patterns.

Transaction Mixes

Even with just one application function, there may be two or more classes of transactions that exhibit significantly different patterns in the use of system resources. For instance, in the TDMHMR study, it was found that for a single application (BCP), there were two classes of transactions: (a) "regular" BCP transactions, which search through a relatively small portion of the data base, and (b) BCPS transactions, which carry out a much more extensive search of the data base. The former class executes an average of 8 I/O operations per transaction; the corresponding figure for the latter class of transactions is 147.

If there are many application functions, it is likely that there are several classes of transactions, each with its own characteristic resource usage patterns. It is a gross approximation to use a single "composite" throughput figure for a computing system and to restrict attention to a single response time figure. Further work needs to be carried out on incorporating detailed models of transactions into the design process. We see the design taking place in two steps: in the first step, a single composite transaction is used to reduce the large number of design options; and in the second step, the different classes of transactions are considered.

Personnel Costs

Creating regional computing centers could result in dispersion of data processing personnel and increased personnel and managerial costs. Indeed, this cost element could be the key factor in deciding whether to distribute processing power. Costs of personnel should be added to costs of hardware and software development in figuring out the overall cost of a regional computer.

CONCLUSION

We have described approaches to designing hierarchical distributed systems. There are several such systems in existence. Design problems associated with such systems are more easily understood and resolved than problems that arise in designing general distributed systems. However, the practical problems that arise in designing even simple distributed systems are formidable. Indeed, it is the practical problems (rather than the mathematics) that make the design of distributed systems so difficult.

In our opinion, it is helpful to develop optimum algorithms and precise models to understand the problems that arise in design and to validate heuristics. In practice, the design process requires continuous human interaction because the data used in the design process is "soft," there may be several (and sometimes contradictory) objectives, and it is difficult to state some objectives in a precise, quantitative fashion. For instance, a manager might say that he wants a near-optimum system now and that he does not want the system to change "much" to handle the load in 1980. Though it is possible to quantify and prioritize such objectives, it seems more helpful to us to give the manager a heuristic program that he can interact with.

REFERENCES

1. N. Abramson and F. Kuo, Computer Communication Networks (Prentice-Hall, Englewood Cliffs, N.J., 1973).

2. L. R. Bahl and D. T. Tang, "Optimization of Concentrator Locations in Teleprocessing Networks," Proc. Symposium Computer-Communication Networks and Teletraffic (1972), pp. 355-362.

3. F. Baskett, "Mathematical Models for Computer Systems," Ph.D. Dissertation (Univ. of Texas at Austin, 1970).

4. J. C. Browne and IRA Staff, "ASQ: User's Manual" (Information Research Associates, Austin, Texas).

5. R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Ryder, and L. A. Wehr, "A Back-End Computer for Data Base Management," CACM 17:10, 575-582 (Oct. 1974).

6. A. F. Cardenas, "Evaluation and Selection of File Organization - A Model and System," CACM 16:9, 540-548 (Sept. 1973).

7. R. G. Casey, "Allocation of Copies of a File in an Information Network," AFIPS Conf. Proc. 40 (1972 Spring Joint Computer Conf.), 617-625 (May 1972).

8. R. G. Casey, "Design of Tree Networks for Distributed Data," Proc. AFIPS 1973 NCC 42, 341-348 (AFIPS Press, Montvale, N.J., 1973).

9. K. M. Chandy, "Models of Distributed Systems," Proc. 3rd Int. Conf. Very Large Data Bases (1977) pp. 105-120.

10. K. M. Chandy, "Bayesian Models of Design Based on Intuition," Proc. 2nd Int. Conf. Reliability (1976), pp. 281-285.

11. K. M. Chandy, "The Analysis and Solutions for General Queueing Networks," Proc. 6th Princeton Conf. Information Sciences and Systems (1972).

12. K. M. Chandy and R. A. Russell, "The Design of Multipoint Linkages in a Teleprocessing Tree Network," IEEE Trans. Computers C-21:10, 1062-1066 (Oct. 1972).

13. K. M. Chandy and T. Lo, "The Capacitated Minimum Spanning Tree," Networks 3, 173-181 (1973).

14. K. M. Chandy and J. E. Hewes, "File Allocation in Distributed Systems," Proc. Int. Symposium Computer Performance Modeling, Measurement, and Evaluation (March 1976).

15. K. M. Chandy, J. Hogarth, and C. H. Sauer, "Selecting Capacities in Queueing Network Models of Computer/Communication Systems," IEEE Trans. Software Eng. (July 1977).

16. S. K. Chang, "A Model for Distributed Computer System Design," IEEE Trans. Systems, Man, and Cybernetics smc-5:6, 344-359 (May 1976).

17. S. K. Chang and D. T. Tang, "Processor Allocation in a Distributed Computer System," Research Report RC4954, IBM (July 1974).

18. W. W. Chu, "Optimal File Allocation in a Multi-computer Information System," IEEE Trans. Computers C-18:10, 885-889 (Oct. 1969).

19. W. W. Chu, "Optimal File Allocation in a Computer Network," Computer-Communication Network (Prentice-Hall, Englewood Cliffs, N.J., 1973), pp. 82-94.

20. W. W. Chu, "Performance of File Directory Systems for Data Bases in Star and Distributed Networks," NCC (1976), pp. 577-587.

21. C. A. Ellis, "Consistency and Correctness of Duplicate Database Systems," Xerox PARC (1977).

22. C. A. Ellis, "A Robust Algorithm for Updating Duplicate Databases," 2nd Berkeley Workshop Distributed Data Management (1977).

23. L. R. Esau and K. C. Williams, "On Teleprocessing System Design: Part II, A Method for Approximating the Optimal Network," IBM Systems Journal 5:3, 142-147 (1966).

24. K. P. Eswaran, "Placement of Records in a File and File Allocation in a Computer Network," Information Processing 74 (North-Holland Publishing Co., Amsterdam, 1974), pp. 304-307.

25. K. P. Eswaran and D. D. Chamberlin, "Functional Specifications of a Subsystem for Database Integrity," Proc. 1st Int. Conf. Very Large Data Bases (1975).

26. M. Gerla, "The Design of Store-and-Forward (S/F) Networks for Computer Communications," Ph.D. Dissertation (Computer Science Dept., UCLA, Jan. 1973).

27. S. P. Ghosh, "Distributing a Data Base With Logical Associations on a Computer Network for Parallel Searching," IBM Report No. RJ 1439 (22109) (IBM Research Laboratory, San Jose, Calif., Aug. 1974).

28. D. A. Greenberg, "A New Approach to the Optimal Placement of Concentrators in a Remote Terminal Communications Network," IBM Technical Report (White Plains, N.Y., 1973).

0 0 / 0 5 1 0 4 1 9 2

29. HP 3000 Data Center, Hewlett-Packard Database Systems Seminar Material (1975).

30. Jackson, Management Science 10:1, 131-142 (1963).

31. D. Johnson "Approximation Algorithms for Combinatorial Problems," JCSS 9:3, 256-278 (Dec. 1974).

32. R. M. Karp, J. Computer and Systems Sciences (June 1974).

33. L. Kleinrock, Communication Nets: Stochastic Message Flow and Delay (McGraw-Hill, New York, 1964).

34. L. Kleinrock, Queueing Systems, Vol. I: Theory (Wiley Interscience, New York, 1975).

35. L. Kleinrock, Queueing Systems, Vol. II: Computer Applications (Wiley Interscience, New York, 1975).

36. J. R. Knight, "Case Study: Airlines Reservations Systems," Proc. IEEE (1972), pp. 1423-1481.

37. H. Kobayashi, "Application of the Diffusion Approximation to Queueing Networks, I: Equilibrium Queue Distributions," 21:2, 316-328 (April 1974).

38. H. Kobayashi, "Application of the Diffusion Approximation to Queueing Networks, II: Nonequilibrium Distributions and Applications to Computer Modeling," 21:3, 459-469 (July 1974).

39. T. Kunii and H. Kunii, "Design Criteria for Distributed Database Systems," Proc. 3rd Int. Conf. Very Large Data Bases (1977), pp. 93-104.

40. A. Lavia and E. G. Manning, "Topological Optimization of Packet-Switched Data Communication Networks," CCNG Report E-35 (Univ. of Waterloo, Oct. 1975).

41. K. Levin, "Organizing Distributed Data Bases in Computer Networks," Ph.D. Dissertation (Wharton School, Univ. of Pennsylvania, 1974).

42. K. Levin and H. L. Morgan, "Optimizing Distributed Data Bases - A Framework for Research," Proc. SJCC (AFIPS Press, 1975) pp. 473-478.

43. E. T. Lowenthal, "Computing Subsystems for the Data Management Function," NCC (May 1974).

44. S. A. Mahmoud and J. S. Riordan, "Optimal Allocation of Resources in Distributed Information Networks," ACM TODS 1:1 (March 1976).

45. E. Manning and R. W. Peebles, "A Distributed Operating System Nucleus for Transaction Processing on Distributed Data Bases," Proc. 4th Texas Conf. Computing Systems (IEEE Comp. Soc., 1975), pp. 5A-2.1 - 5A-2.5.

46. T. Marill and D. Stern, "The Datacomputer - A Network Data Utility," Proc. AFIPS Nat. Comp. Conf. (1975), pp. 389-395.

47. J. Martin, Telecommunications and the Computer (Prentice-Hall, Englewood Cliffs, N.J., 1972).

48. J. Martin, Design of Real-Time Computer Systems (Prentice-Hall, Englewood Cliffs, N.J., 1967).

49. K. Maruyama and D. T. Tang, "Discrete Link Capacity Assignment in Communication Networks," Proc. 3rd ICCC (Aug. 1976), pp. 92-97.

50. K. Maruyama, L. Fratta, and D. T. Tang, "Flow Assignment Algorithm for Computer Communication Network Design With Different Classes of Packets," Trends and Applications Computer Networks (IEEE, Nov. 1976), pp. 51-57.

51. K. Maruyama, L. Fratta, and D. T. Tang, "Heuristic Design Algorithm for Computer Communication Networks With Different Classes of Packets," IBM Systems Journal 21:4, 360-369 (July 1977).

52. R. Peebles and E. Manning, "A Computer Architecture for Large Distributed Data Bases," Proc. 1st Int. Conf. Very Large Data Bases (1975), pp. 405-427.

53. C. V. Ramamoorthy, G. S. Ho, T. Krishnarao, and B. W. Wah, "Architectural Issues in Distributed Data Base Systems," Proc. 3rd Int. Conf. Very Large Data Bases (1977), pp. 121-126.

54. J. B. Rothnie and N. Goodman, "A Study of Updating in a Redundant Distributed Database Environment," TR CCA-77-01 (Computer Corporation of America, Feb. 1977).

55. J. B. Rothnie and N. Goodman, "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases," Berkeley Workshop Distributed Data Management and Computer Networks (1977), pp. 39-57.

56. J. B. Rothnie and N. Goodman, "A Survey of Research and Development in Distributed Database Management," Proc. 3rd Int. Conf. Very Large Data Bases (1977), pp. 48-62.

57. S. Sahni, "Approximate Algorithms for the 0/1 Knapsack Problem," JACM 22:1, 115-124 (Jan. 1975).

58. S. Sahni, "General Techniques for Combinatorial Approximation," ORSA/TIMS, Las Vegas (Nov. 1975).

0 0 0 3 5 1 0 4 1 9 3

59. M. Schwartz, Computer-Communication Network Design and Analysis (Prentice-Hall, Englewood, Cliffs, N.J., 1977).

60. V. L. Shatz, "Computer Networks for Retail Stores," Computer (1973) pp. 21-25.

61. D. T. Tang, L. S. Woo, and L. R. Bahl, "Optimization of TP Networks With Concentrators and Multiconnected Terminals," IBM Research Report (1974).

62. L. Van Sickle and K. M. Chandy, "Computational Complexity of Network Design Algorithms," IFIPS (1977), pp. 235-239.

63. L. Woo and D. T. Tang, "Optimization of Teleprocessing Networks," Proc. National Telecommunications Conf. 2, 37C1-C5 (1973).

64. S. B. Yao, "Evaluation and Optimization of File Organizations Through Analytical Modeling," Ph.D. Dissertation (Univ. of Michigan, 1974).

65. R. T. Yeh, "Configuration of Distributed Data Base Systems With Tree Topologies," Ph.D. Dissertation (Univ. of Texas at Austin, 1978).

SYNCHRONIZATION IN DISTRIBUTED SYSTEM MODULES*

Gregor V. Bochmann

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, Canada

Abstract

The communication delays between the different components of a distributed system often create problems for the logical consistancy of the overall system behaviour. The ideas presented in this paper suggest how to eliminate these problems by observing certain regularity constraints during the system design, which guarantee that the logical behaviour of the system is independent of the communication delays. The paper presents a descriptive model for the specification of distributed systems, and defines system properties which imply regular system behaviour. Several criteria for checking the regularity of a given system are discussed. The application of the concepts presented is illustrated by several examples.

1. INTRODUCTION

It is common practice to subdivide complex systems into a number of modules. Usually, a module is characterized by a given data structure and a set of operations that may be executed on this data. Such a module may be considered an instance of an abstract data type. Seen from the outside, a data type is characterized by the operations that may be invoked on the module by other modules, i.e. the order in which these operations may be executed and the parameter values exchanged. The other point of view considers the implementation of these operations inside the module, for which it is usually necessary to specify the data structures of the module and the procedures that implement the externally available operations. There may also be some internal book-keeping operations invoked inside the module.

---

*This work was performed while the author was on leave at the Département de Mathématiques de l'Ecole Polytechnique Fédérale de Lausanne, Suisse.

0 0 0 3 5 1 3 4 1 9 4

Most systems contain some kind of parallel activities. Thus, in principle, it is possible that the operations of a given module be invoked by several other modules, in an arbitrary order. However, many module types only function correctly if the operations are invoked in a specific order. Therefore, the correct synchronization of the module operations must be enforced in some appropriate way, perhaps delaying the calling modules. Many different tools have been developed for the specification of synchronization mechanisms, for example semaphores, monitors, conditional critical regions, conditions on history counter variables (1), etc. In this paper, we use a type of conditional critical region (see section 2).

The discussions in this paper are illustrated by examples showing distributed implementations of a queue, and mutual exclusion. The queuing module considered in this paper, has the following two operations that may be invoked by other modules :

- *enter* a block of data into the queue, and
- *remove* a block of data from the queue.

The queuing behaviour of the module is characterized by the requirement that the sequence of data blocks obtained by the execution of *remove* operations, from the moment the system is initialized be a prefix of the sequence of data blocks *entered* into the queue. We also assume a maximum buffer size of N data blocks.

The other examples consider N processes sharing a resource module. We assume that mutual exclusion is enforced for the use of the resource, i.e. at most one user module may use the resource at any given time.

The problem considered

In this paper we consider the problems that arise from the distribution of a module over several physical components, which may be located at different places. Because of the communication delay between the different components, the sharing of information between the different components of a module becomes more complicated. For example, reading a variable in a distant component generally involves a delay implying that the variable may already have been updated when the value read is obtained. Another example is the potential deadlock resulting from call collision, which occurs when two communication stations call one another at the same instant and, due to the transmission delay, each finds the other one busy. In a local context, these problems may be resolved by the introduction of critical regions for the access of shared variables, but this approach is difficult in a distributed context.

The approach

In the following sections we approach these problems using a descriptive model for system specification, which avoids the use of varia-

bles shared at a distance, but uses instead actions, executed in a given component, which may be initiated by remote components, as explained in section 2. In section 3 we show that the analysis of a distributed module may be reduced to the analysis of an equivalent non-distributed case (i.e. ignoring the communication delays), if the module structure satisfies certain constraints, which we call regularity. We think that the concept of regularity leads to a better understanding of the problems of distributed systems, to the design of simpler systems and more reliable implementations. It is clear that many existing distributed systems already satisfy some regularity constraints. Relations to other aspects of distributed systems and synchronization algorithms are also discussed.

2. THE DESCRIPTIVE MODEL

In the following we only give a brief explanation of the descriptive model used for the specification of distributed modules, since it is similar to models already described elsewhere (2,3).

2.1. The basic model

In the basic model, the distribution aspect is ignored. A module is characterized by a set of variables (declared within the module) and a set of operations. The (internal) state of the module is characterized by the values of these variables. Each operation specifies a set of possible state transitions. An operation is either "external", i.e. it is part of the module interface and initiated by other modules, or "internal", i.e. it is initiated by the module itself. An operation is characterized by an enabling predicate, a boolean function of the variables; and an action which updates the variables, which is usually written in some high-level programming language. Only when the enabling predicate is true may the operation be fired, i.e. the associated action is executed, thus performing a state transition. We assume mutual exclusion between the firing of different (external and internal) operations.

As an example we show below an implementation of ensuring mutual exclusion between N *User* modules accessing a shared resource. We use a notation similar to Pascal (4) to specify the variables, enabling predicates, and actions. We write *name.operation* to indicate the initiation of an operation in another module or component. To control the resource access, the example uses the customary *request* and *release* operations which must be called in a consistant order by the user modules. A cyclic order of execution is enforced for each *User* module by the *state* variable.

Resource Module (the resource manager)

Variables

```
    free : boolean ;
```

External operations

```
    request ;
        when free
        do   free:=false ;
    release ;
        free:=true ;
```

Initially

```
    free:=true ;
```

User i Module (the i-th user module of the resource)

Variables

```
    state : (start, reserved, done) ;
    ...
```

Internal operations

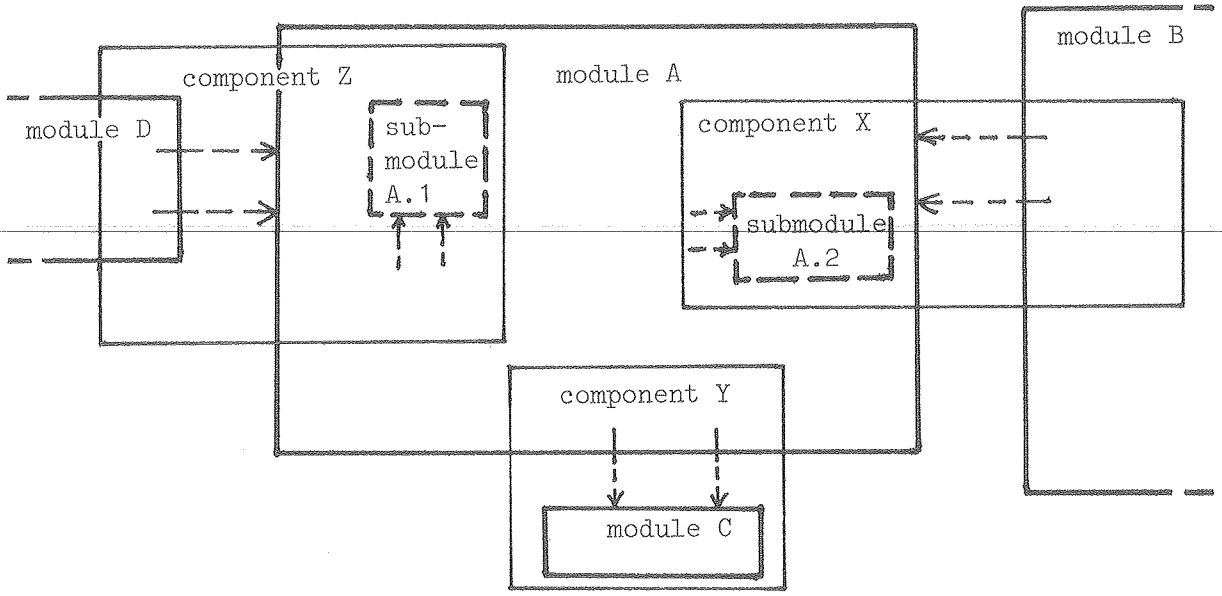
```
    request ;
        when     state = start
        do begin Resource.request ; state:=reserved end ;
    usage ;
        when     state = reserved
        do begin ... {use resource} ; state:=done end ;
    release ;
        when     state = done
        do begin Resource.release ; state:=start end ;
    ...
```

Initially

```
    state:=start ; ...
```

2.2. The distributed model

We will now consider a module to be distributed over several physical "components". A typical situation is shown in the figure below, where the module A is distributed over the physical components, X,Y,Z (possibly placed in different geographical locations). Its external operations are invoked by the modules B and D in components X and Z, respectively. A , in turn, invokes the external operations of the module C , which is completely implemented in component Y . It also invokes the (external) operations of submodules located in components X and Z .



For the description of such distributed modules, we propose in the following a distributed model related to the basic model above. The model is a kind of language containing simple primitives for specifying synchronization and interactions between actions executed in the different components of a module, but it does not indicate how such a module may be obtained. The question of systematic methods for designing distributed modules is outside the scope of this paper, although the examples indicate some possible strategies.

To deal with the distribution aspect in the description of a module, we adopt the following conventions (in addition to those of the basic model) :

(a) Each variable of a distributed module is located in a single component.

(b) Each operation is assigned to a single component, where it is initiated (either internally by the module, or as an external operation), and its enabling predicate depends only on variables located within that component.

0 0 - 0 5 1 0 4 1 9 6

(c) The action of an operation is partitioned into a certain number of localized actions, each updating the local variables of a particular component, such that the new values depend only on the previous values of the local variables of that component and (possibly) on the variable values of the initiating component. We call "local action" an action localized in the initiating component, and "remote action" an action localized in another component.

(d) Mutual exclusion between local and remote actions of different operations is ensured separately for each component.

(e) When an operation is fired, its local action is executed immediately, and each corresponding remote action is executed some finite time later (possibly using the values of the variables in the initiating component as updated by the execution of the local action). During the intermediate time intervals, other actions may be executed on the different components.

It is clear that an implementation of this model may be obtained by using an inter-component communication system for the transfer of messages. When an operation is fired in the initiating component a message, containing the identification of the operation and the necessary variable values of the initiating component, is sent to each component where a remote action must be executed. Variations on point (e) above may be assumed, corresponding to different message communication services such that, for example,

- messages may get lost (remote actions might never be executed),
- messages arrive sequentially, depending on the (logical) interconnection structure between the components.

In the above discussion, we have considered a single system module and ignored the fact that a given physical system component will, in general, contain components of several modules which may interact locally through external module operations.

As an example, we show below an implementation of a queuing module distributed over two components which may be used to transfer data from the *sender* component to the *receiver* component. It is derived from a straight-forward non-distributed queuing module by duplicating the variable *length* in both components, and assigning the *buffer* variable to the *receiver* component. This corresponds to the design strategy of assigning a variable to the component where it is used, possible creating several copies of the same "logical variable" in different components. (We note that the module assumes sequential message transmission between the components).

Sender Component

Variables

```
length : 0..N ;
```

External operation

```
enter(data:block) ;
    when length < N
    do begin
        length:=length + 1 ;                          } local action

        Receiver.length:=Receiver.length + 1 ;        } remote action
        Receiver.buffer.put(data)                     }
    end ;
```

Initially

```
length:=0 ;
```

Receiver Component

Variables

```
length : 0..N ;
buffer : queue of block ; {submodule with the primitive operations
                           put, get and empty}
```

External operation

```
remove(var data:block) ;
    when length > 0
    do begin
        length:=length - 1 ;                          } local action
        buffer.get(data)   ;                          }

        sender.length:=sender.length - 1 ;            } remote action
    end ;
```

Initially

```
length:=0 ;
buffer.empty ;
```

3. REGULAR SYSTEMS

In this section we consider a certain class of constraints which may be satisfied by distributed systems, described in the model outlined above, and which, when satisfied, allows a logical system validation ignoring the possible message transmission delays between the components. We call these constraints "regularity" constraints. The following subsection explains some useful concepts and notations needed for the following discussion.

3.1. Traces of operation execution

We consider a system module with a certain set of possible (internal and external) operations. The (local and remote) actions of these operations are identified by action symbols. A "trace" is a string of action symbols, and indicates a possible sequence of executions of these actions. Such a trace includes all local and remote actions of the internal and external operations of the module, in the order in which they are executed.

Given that the module is in a certain (internal) state s, only a certain set of traces represent the possible execution sequences starting from this state. (The enabling predicates of the operations impose restrictions on the execution of local actions in each state, and a remote action may be executed any time after the corresponding local action has been executed). We write $(s)\sigma \neq \perp$ to indicate that σ is a possible trace starting in state s. We assume that all actions are deterministic. Thus the execution of a trace σ starting from a state s leads to a (single) state which we write "$(s)\sigma$".

We call an "initial trace" a trace which specifies an execution sequence starting from the module's initial state s_0. There is a correspondance between the initial traces of a module and its internal states : each initial trace σ leads to a particular state $(s_0)\sigma$, which we often simply write "σ". We also write $\sigma \neq \perp$ to indicate that σ is a possible initial trace. We say that a state s is "reachable" iff there is an initial trace σ which leads to s, i.e. $\sigma = s$.

We say that two module states s_1 and s_2 are "equivalent", which we write $s_1 \simeq s_2$, iff the set of possible traces starting from the states s_1 and s_2 are the same. We say that s_2 includes the traces of s_1, which we write $s_1 \subset s_2$, iff all possible traces starting from state s_1 are also possible traces from s_2.

3.2. Regularity

Roughly speaking, a system is regular if in the presence of message transmission delays it "behaves in the same way as" in the absence of any delays (except for the speed of operation). In the absence of delays, all traces are of the form $a_L\ a_R\ b_L\ b_R^{(1)} \ldots b_R^{(n)}\ c_L\ d_L\ d_R \ldots$

(we distinguish local and remote actions by the indexes L and R respectively. Here we have assumed that the operation c has no remote actions, the operations a and d have one, and b has n remote actions), i.e. each local action is immediately followed by the corresponding remote actions. We call such a trace "delayless". In the presence of delays, distant actions also follow the corresponding local action, but other actions may occur in between.

The following properties exactly describe what we mean by the same system behaviour in the presence and absence of transmission delays. We call these properties "regularity". For the purpose of understanding and verifying a module that is known to be regular, we may then ignore the transmission delays and assume that only "delayless" traces, i.e. of the above form, may occur. This assumption will usually considerably simplify the analysis of the module.

Definition : A module has a "regular set of traces" (RST) iff for each initial trace $\sigma_1\sigma_2$, where the subtrace σ_1 leads to a state with no messages in transit, $\sigma_1\hat{\sigma}_2$ is also a possible initial trace, i.e. $\sigma_1\sigma_2 \neq \perp$, where $\hat{\sigma}_2$ is the delayless trace which contains the same local actions occuring in the same order as in σ_2 , and all corresponding distant actions.

For any reachable state of a module with this property, the operation sequences which are possible in the presence of arbitrary (finite) delays are also possible in the absence of delays. Taking the empty sequence for σ_1 , this implies that transmission delays cannot render possible any operation sequences which are not possible without delays. Conversely, it is clear that for any module described in the model of section 2.2, an operation sequence possible without delays is also possible in the presence of delays. Therefore the possible operation sequences of a module with RST are independent of the delays.

Definition : A module has a "regular state reachability" (RSR), or a "regular equivalent state reachability" (RESR), iff it has RST and for each $\sigma_1\sigma_2$, there exists a "supplement trace" σ_3 such that $\sigma_1\sigma_2\sigma_3 = \sigma_1\hat{\sigma}_2$, or $\sigma_1\sigma_2\sigma_3 \simeq \sigma_1\hat{\sigma}_2$ respectively.

This property ensures, in particular, that whatever state is reached by an initial trace σ_2 (we assume σ_1 = empty sequence), it is always possible to reach a state (i.e. by executing σ_3) which is also reachable in the absence of delays (i.e. by executing $\hat{\sigma}_2$) or equivalent to such a state. From this, it follows that if the module contains a deadlock or is not live (5) in the general case (i.e. in the presence of arbitrary transmission delays) then the same problems must be present in the case with no delays.

3.3. Regularity conditions and commutation relations

The conditions discussed in this section may be used to decide whether a system is regular or not.

3.3.1. The non-restricting condition

Definition : An action a is "non-restricting" iff for all actions b of the module and all reachable states s, $(s)\ b\ a \neq \perp$ implies $(s)\ a\ b \neq \perp$. If this is true for all states of the module, we say a is "strongly non-restricting".

For many regular systems, the condition that all remote actions be non-restricting is satisfied. Generally, it is easy to check whether this condition is satisfied or not. However, this condition is not sufficient for a system to be regular. A counter example is given by a module component which contains the local operation

$$b_L : \text{when } Y \neq 0 \text{ do if } X = 0 \text{ then } Y := 0 ;$$

and the distantly initiated remote action

$$a_R : X := 0$$

where the action a_L is executed on a different component. Clearly a_R is strongly non-restricting, but starting from the state $X=Y=1$, the trace $a_L b_L b_L a_R$ is possible, but not $a_L a_R b_L b_L$, which rules out both RST and RSR.

3.3.2. Semi-commutation

Definition : An action a "semi-commutates with an action b" in respect to state equality, equivalence, or trace inclusion iff for all reachable states s, $(s)\ ba \neq \perp$ implies $(s)\ ba = (s)\ ab$, $(s)\ ba \simeq (s)\ ab$, or $(s)\ ba \subset (s)\ ab$ respectively (which by convention implies $(s)\ ab \neq \perp$). We say that a "semi-commutes" iff it semi-commutes with all actions b, and that it "strongly semi-commutes" if the condition is satisfied for all states s.

Lemma : Semi-commutation, with respect to state equality, implies the same with respect to state equivalence, which in turn implies the same with respect to trace inclusion, which, finally, implies the non-restriction condition.

3.3.3. Regular remote actions

Definition : A remote action a_R is "regular" with respect to state equality, equivalence, or trace inclusion iff for all traces σ_1, σ_2, $\sigma_1 a_L \sigma_2 a_R \neq \perp$ implies $\sigma_1 a_L \sigma_2 a_R = \sigma_1 a_L a_R \sigma_2$, $\sigma_1 a_L \sigma_2 a_R \simeq \sigma_1 a_L a_R \sigma_2$, or $\sigma_1 a_L \sigma_2 a_R \subset \sigma_1 a_L a_R \sigma_2$ respectively.

Lemma : If a remote action a_R semi-commutes with all actions b then it is regular. (With respect to state equality, equivalence, or trace inclusion respectively. Proof : by induction over the lenght of σ_2).

Lemma : If a remote action a_R is regular then it semi-commutes with all actions b . (Valid with respect to state equality or equivalence, but not with respect to trace inclusion).

Proof : We may assume $s=(s_o)\sigma_1 a_L \sigma_2$ and $(s)\ b\ a_R \neq \perp$. Then the regularity of a_R implies

$(s_o)\sigma_1 a_L \sigma_2 b\ a_R \simeq (s_o)\sigma_1 a_L a_R \sigma_2\ b \simeq (s_o)\sigma_1 a_L \sigma_2 a_R\ b$, i.e. $(s)b\ a_R \simeq (s)a_R b$.

Definition : A remote action a_R is "semi-regular" with respect to state equality (or equivalence) iff for each initial trace σ_1 which leads to a state with no messages in transit, and any traces σ_2 and σ_2' , $\sigma_1 a_L \sigma_2 a_R \sigma_2' \neq \perp$ implies that there is a supplement trace σ_3 such that $\sigma_1 a_L \sigma_2 a_R \sigma_2' \sigma_3 = \sigma_1 a_L a_R \sigma_2 \sigma_2' \neq \perp$ (or $\sigma_1 a_L \sigma_2 a_R \sigma_2' \sigma_3 \simeq \sigma_1 a_L a_R \sigma_2 \sigma_2' \neq \perp$ respectively).

Lemma : The regularity of a remote action implies the corresponding semi-regularity. (Proof : take σ_3 = empty sequence).

Proposition 1 : If all distant actions of a module are regular (in respect to state equality, equivalence or trace inclusion) then the module has RST .

Proposition 2 : If all distant actions of a module are semi-regular or regular with respect to state equality or equivalence then the module has RSR , or RESR respectively, which implies RST .

Proposition 3 : If all distant actions of a module are regular with respect to state equality or equivalence then the module has RSR , or RESR respectively, with the empty supplement trace σ_3 , which implies RST .

The propositions are proved by induction over the distant actions in σ_2 .

4. EXAMPLES

In this section we consider several examples of distributed modules, and analyse their regularity using the tools developed in the preceding section.

4.1. The queuing module

We consider the distributed queuing module of section 2.2. We note that a remote action executed in a given component always semi-commutes with an action executed in a different component (except with the corresponding local action). For an analysis of the regularity of the module,

it is therefore sufficient to investigate the commutation of $enter_R$ with $remove_L$, and $remove_R$ with $enter_L$. We find strong semi-commutation in both cases, therefore the module is regular.

4.2. An erroneous mutual exclusion algorithm

An adaptation of the mutual exclusion algorithm of section 2.1 to a distributed environment is not easily obtained. If the strategy of variable assignment and duplication, which was successful in the case of the queuing module, is applied in this example we obtain a distributed module such as the following (the variable *free* is duplicated) :

User i Component

Variables

```
state : (start, reserved, done) ;
free : boolean ;
...
```

Internal operations

```
request ;
    when state = start and free
    do begin
        state:=reserved ;                          } local action
        free:=false ;                              }
        for all j≠i do user_j.free:=false ;        } remote actions
    end ;
usage ;
    when state = reserved
    do begin ... {use resource} ; state:=done end ;
release ;
    when state = done
    do begin
        state:=start ;                             } local action
        free:=true ;                               }
        for all j≠i do user_j.free:=true ;         } remote actions
    end ;
```

Initially

```
state:=start ; free:=true ;
```

It is easy to see that this module is not regular. For example, the

trace $user_1.request_L$ $user_2.request_L$ $user_1.request_R^{(2)}$
is a possible initial trace, whereas the corresponding delayless
trace $user_1.request_L$ $user_1.request_R^{(2)}$ $user_2.request_L$
is not, which is a contradiction to the RST property. (We note that $user_1.request_R^{(2)}$ is the remote action of the *request* operation initiated in the $user_1$ component executed in the $user_2$ component). By the way, this also results in a violation of the mutual exclusion in the case of transmission delays.

4.3. A centralized mutual exclusion algorithm

Similar to the algorithm of section 2.1, the following distributed resource sharing module uses a centralized resource manager. Mutual exclusion is maintained by a rather more complex exchange of "messages" between the user components and the manager.

Resource Component (the resource manager)

Variables

```
free : boolean ;
requests : queue of userid ; {user identification, i.e. 1..N}
```

Internal operation

```
allocate ;
    var next : userid ;
    when free and not empty (requests)
    do begin
        free:=false ;              } local action
        requests.get(next) ;       }
        next.state:=reserved ;     } remote action
    end ;
```

Initially

```
free:=true ;
requests.empty ;
```

User i Component

Variables

```
state : (start, reserved, done) ;
...
```

Internal operations

```
request ;
    when state = start
    do begin
        Resource.requests.put(myid) } remote action
    end ;
usage ;
    when state = reserved
    do begin
        ... {use resource}
        state:=done
    end ;
release ;
    when state = done
    do begin
        state:=start ;          } local action

        Resource.free:=true ;   } remote action
    end ;
```

Initially

```
state:=start ;
```

To prove the regularity of the algorithm we consider, as in section 4.1, the commutation of each remote action with the local actions executed in the same component. This leads to the following pairs of actions :

- $allocate_R$ commutes with $request_L$ strongly,
- $allocate_R$ semi-commutes with $usage_L$ strongly,
- $allocate_R$ semi-commutes with $release_L$ (there is no initial trace of the form "... $release_L$ $allocate_R$") ,
- $request_R$ semi-commutes with $allocate_L$ strongly,
- $release_R$ semi-commutes with $allocate_L$ (there is no initial trace of the form "... $allocate_L$ $release_R$").

This shows that the module is regular.

4.4. A distributed mutual exclusion algorithm

Dijkstra has described self-stabilizing algorithms with distributed control (6), which provide mutual exclusion in the stabilized situation. His solution with K-state machines may be re-written in the following form :

User i Component

Variable

```
S,L : 0..K-1   {own machine state, and latest knowledge about
                the state of the left neighbour}
```

Internal operations

```
send-status ;
    do user_{i⊕1}.L:=S ; } remote action
usage (for i=0) ;
    when L=S
    do begin
        ... {use resource} ;
        S:=S⊕1
    end ;
usage (for i>0) ;
    when L≠S
    do begin
        ... {use resource} ;
        S:=L
    end ;
```

where nothing is assumed about the initial values of the variables (the algorithm is self-stabilizing (8)) and ⊕ stands for addition modulo N .

To show the regularity of the algorithm, we consider the following pairs of actions :

- *send-status*$_R$ and *usage*$_L$ (for i=0) ,
- *send-status*$_R$ and *usage*$_L$ (for i>0) , and
- *send-status*$_R$ and *send-status*$_L$ (the remote action is executed in the same component as the local action, but belongs to a different operation).

The last pair commutes strongly. The first two pairs semi-commute (assuming as initial state L=S=0 in all components), but not strongly, i.e. not during the self-stabilization phase when nothing can be assumed about the variable values. Once the system is stabilized, the semi-commutation relation ensures regularity.

4.5. Recovery from message loss

So-called time-out operations are usually introduced to prevent deadlock in the case of a lost message. Such an operation is included in an example of a simple data transmission protocol with message numbering and retransmission (11), presenting the same external interface as the queuing module of section 2.2. For this example, some remote ac-

tions do not semi-commute with all the local actions within the same component, as was the case in the above examples. However, semi-regularity can be shown for these actions, thus proving the regularity of the protocol module.

4.6. Synchronization based on counters

The variable duplication strategy for obtaining distributed implementations may be applied to the counter-based synchronization mechanism described by Robert and Verjus (1). Only under certain conditions are the resulting implementations regular (11).

5. EXTERNAL REGULARITY

In this section we take the view that we are not interested in the internal operation of a module as we were above, but only in the interface it presents to its environment. This interface is characterized by the external operations of the module and the possible order in which these operations may be executed. Therefore the interface may be described by the set of possible "external traces", which are initial traces, as defined in section 3.1, ignoring all actions except the local actions of external operations. To complete the interface description, these external traces must be complemented with the parameter values that are exchanged between the module and its environment during the execution of the external operations. Bartussek and Parnas (8) describe an interface specification method based on external traces.

Definition : A distributed module is "externally regular" iff the inter-component communication delays have no influence on the interface of the module (except for the speed of its operation).

More precisely, we propose that an externally regular module should satisfy the following two conditions :

(a) The module has a "regular set of external traces" (RSET) which is defined as for RST where $\hat{\sigma}_2$ is the delayless trace which contains the same local actions of external operations as σ_2 .

(b) The module has the same deadlock and liveness properties in the presence and absence of delays, which is ensured if the module has RSR or RESR with supplement traces σ_3 containing no local actions of external operations.

6. RELATED WORK

The concepts developed in this paper are related to many algorithms designed for distributed implementation, in particular communication protocols, routing algorithms in data networks, or other algorithms for

distributed processing, such as the mutual exclusion algorithms described by Dijkstra (6) (see section 4.4).

In a recent paper (9) Lamport presented some synchronization algorithms which rely on a particular property of shared multi-digit variables, namely that the value read is always smaller or equal to the last written value. As in the case of synchronization based on counter variables (1), he complies with the restriction that the values of the shared variables are never decreased. Assuming individual increments are always one, this leads to the situation where the value read from a shared multi-digit variable is more or less "old". The same situation holds in our distributed model where the variables of distant components contain "old" values until they are updated by remote actions. When re-written in our model, with an appropriate distribution of variables, most of Lamport's algorithms turn out to be regular, and may therefore be used in a distributed context, although not originally intended for this purpose.

So far we have considered algorithms or modules the logical behaviour of which is independent of the inter-component communication delays. Vissers (10) has described several methods to turn a system, designed for negligeable communication delays and therefore in general not regular, into what we call a regular system. He considers a "turn method" which results in a system where each component in turn does an operation, similar as in the mutual exclusion algorithm of section 4.3. He also considers "freeze" and "grant" methods which result in systems where two components interact in an inherent asymmetry.

7. CONCLUSIONS

The communication delays between the different components of a distributed system often cause problems for the logical consistancy of the overall system behaviour. The ideas discussed in the paper suggest to eliminate these problems by observing certain regularity constraints during the system design which guarantee that the logical behaviour of the system is independent of the communication delays.

The paper presents a descriptive model for the specification of distributed systems and several conditions which guarantee the regular behaviour of the system. Whereas several examples are given which show how these concepts apply for some simple systems, more work is required to study the applicability of the ideas for larger and more complex systems. Other interesting questions not covered in this paper are systematic methods for designing regular distributed systems; and the impact of message losses, non-sequential message delivery, and failures of system components on the overall system behaviour.

ACKNOWLEDGEMENTS

I would like to thank Jan Gecsei and Chris Vissers for interesting discussions related to this topic, and Marcel Berthoud, André Schiper and Simon Waddell for suggesting many improvements of a preliminary draft of this paper.

REFERENCES

1. P. Robert and J.P. Verjus, "Toward autonomous descriptions of synchronization modules", Proc. IFIP Congress 1977, pp. 981-986.
2. G.V. Bochmann and J. Gecsei, "A unified method for the specification and verification of protocols", Proc. IFIP Congress 1977, pp. 229-234.
3. E. André and P. Decitre, "On providing distributed application programmers with control over synchronization", Proc. Computer Network Protocols Symposium, Université de Liège, Febr. 1978, pp. DI-1. to DI-6.
4. K. Jensen and N. Wirth, "Pascal user manual and report", Springer Verlag, Berlin, 1974.
5. R.M. Keller, "Formal verification of parallel programs", Comm. ACM 19, 7 (July 1976), pp. 371-384.
6. E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control", Comm. ACM 17, 11 (Nov. 1974), pp. 643-644.
7. J. Mossière et al., "Sur l'exclusion mutuelle dans les systèmes informatiques", Internal report No 75, IRISA, Université de Rennes, France, Oct. 1977.
8. W. Bartussek and D.L. Parnas, "Using traces to write abstract specifications for software modules", UNC Report No TR 77-012, Dec. 1977, University of North Carolina at Chapel Hill.
9. L. Lamport, "Concurrent reading and writing", Comm. ACM 20, 11 (Nov. 1977), pp. 806-811.
10. C.A. Vissers, "Interface : Definition, design and description of the relation of digital system parts", Technische Hogeschool Twente, The Netherlands, 1977 (section 5.2).
11. G.V. Bochmann, "Regular synchronization in distributed system modules", draft, 1978.

PERFORMANCE LIMITS OF THE TMM PROTOCOL : MODELLING AND MEASUREMENT

Erol Gelenbe*, Jean-Louis Grangé**, Patrice Mussard**

Abstract :

The TMM protocol uses a stop-and-wait procedure with acknowledgements for error control. The purpose of this paper is to investigate the basic limits of TMM with respect to its utilisation of available line capacity. Measurement results are obtained and a mathematical model is presented in order to predict TMM's performance and to derive optimum average lengths for random length or fixed length packets.

* IRIA-LABORIA, B.P. 105 -78150 Le Chesnay.
** IRIA-Projet CYCLADES, B.P. 105 - 78150 Le Chesnay.

0 0 1 0 5 1 9 4 2 0 3

1. Introduction

CIGALE [1] is the packet switching network of CYCLADES [2]. The basic service offered by CIGALE simply consists of transporting independent packets to their destination : this type of service has been called Datagram service within CCITT*.

In order to reach their destination, packets are forwarded from node to node through point to point synchronous lines. Since lines introduce delays and errors, line handling protocols are needed to provide for efficient and reliable transmissions. Standard manufacturer's protocols are made available in CIGALE so as to avoid any special piece of hardware or software for host connections. The purpose of this paper is to present some results on performance measurement and evaluation of the TMM protocol.

TMM is a CII standard protocol [3] . It allows full duplex symmetrical transmission, i.e. physical transmission in both directions at the same time. It uses a stop-and-wait scheme to control the packet transmission : a new packet can be transmitted over the line only when the previous one has been acknowledged. A timout is set before transmitting each packet. If it expires before receiving the acknowledgement, the packet is retransmitted. A simple packet numbering scheme allows to detect and discard duplicated packets. It is to be noted that, since the transmission of a new packet cannot occur before receiving the acknowledgment of the previous one, any delay in acknowledgment transmission leads to the waste of a portion of the bandwith available for the reverse traffic. In particular, short packet traffic may be considerably slowed down by long packets transmitted in the reverse direction : short packet acknowledgments are delayed due to the transmission time of long packets.

---

* Comité Consultatif International Télégraphique et Téléphonique

Figure 1 shows how a part of the bandwidth is wasted if packets are not of the same length in both directions of the line.

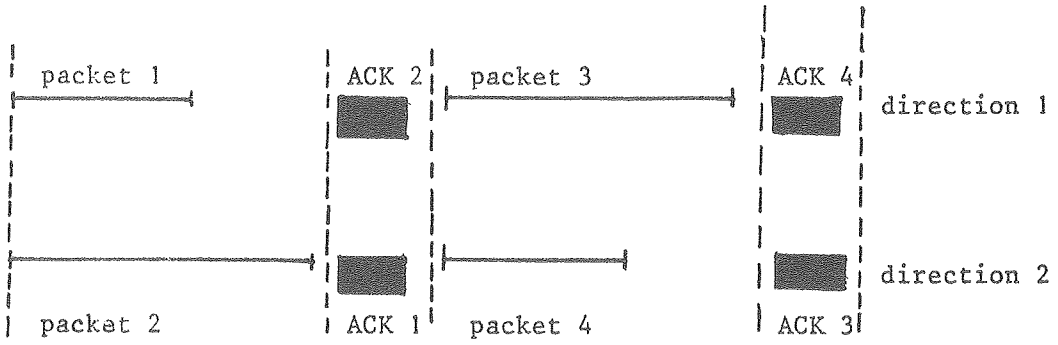
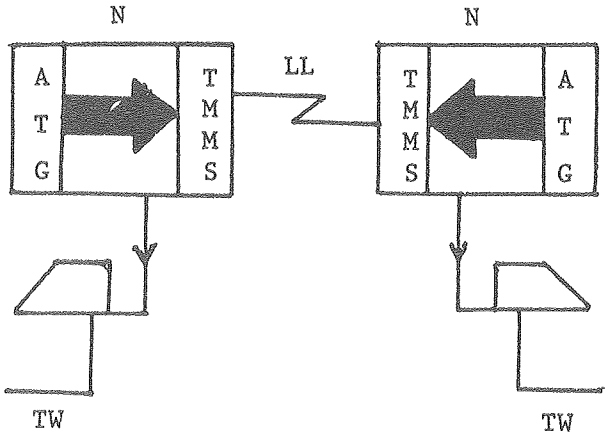


FIGURE 1

Transmission diagram in both directions for the TMM protocol

2. The measurements

In order to evaluate the performance of the TMM protocol various measurements have been performed. For that purpose, the configuration shown in Figure 2 has been set up.



N = CIGALE node
LL = local synchronous line
TMMS = TMM protocol software
ATG = Artificial traffic generator
TW = Control Typewritter.

FIGURE 2

0 0 : 0 5 1 0 4 2 0 4

Two nodes are connected by a local telephone line at 19 200 bits/sec. Base band modulation modems are used and the line length is extremely short (about 2 meters) : propagation delays through the hardware are negligible compared to the transmission delays.

Each node is loaded with the regular CIGALE node system including the TMM protocol software (TMMS) and an artificial traffic generator (ATG).

In all experiments, ATG is used to generate fixed length packets at the maximum rate in both directions, i.e. as fast as TMMS can accept them.

A typewriter is connected to each node and is used to control the ATG (e.g, start, stop) and to extract measurement data from the node system.

A number of experiments have been conducted with various combinations of packet length at each end. Throughput results obtained are summarized in Figure 3 and 4.

In the following sections of this paper we present a mathematical model of TMM and validate it with the measurement data obtained. We then obtain some basic properties of this protocol. In particular we compute the maximum throughput of TMM in the presence of packets of variable lengths.

A study of the time-out optimization for stop-and-wait schemes is presented elsewhere [6]. In this study we assume that error rates can be neglected.

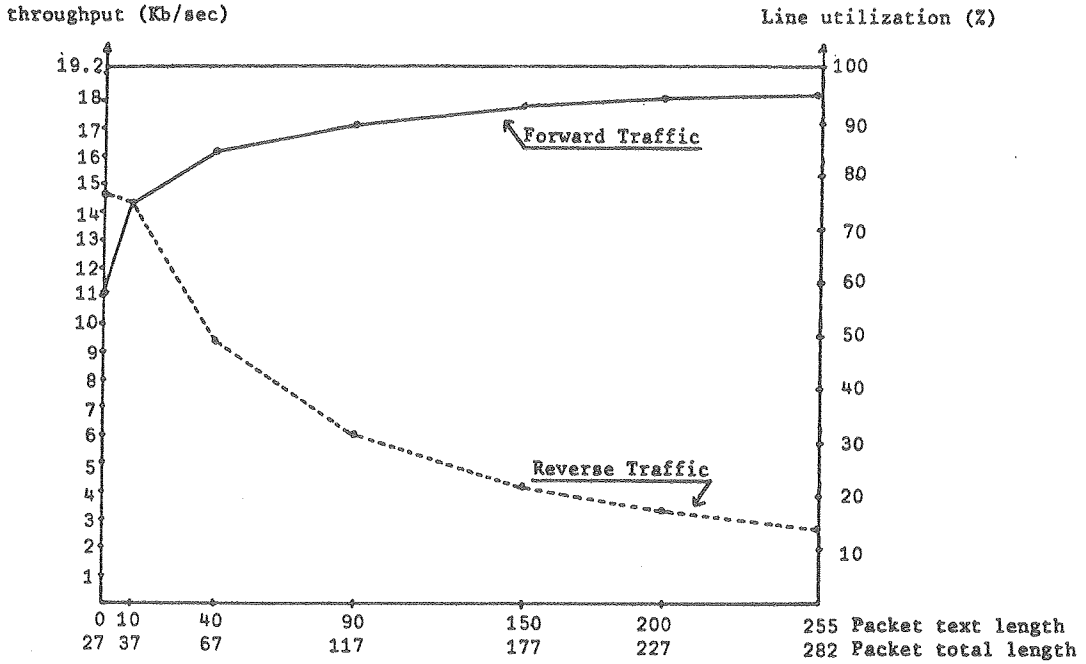


Figure 3 : TMM Performances (*) (Reverse Traffic Packet Length : 37 octets)

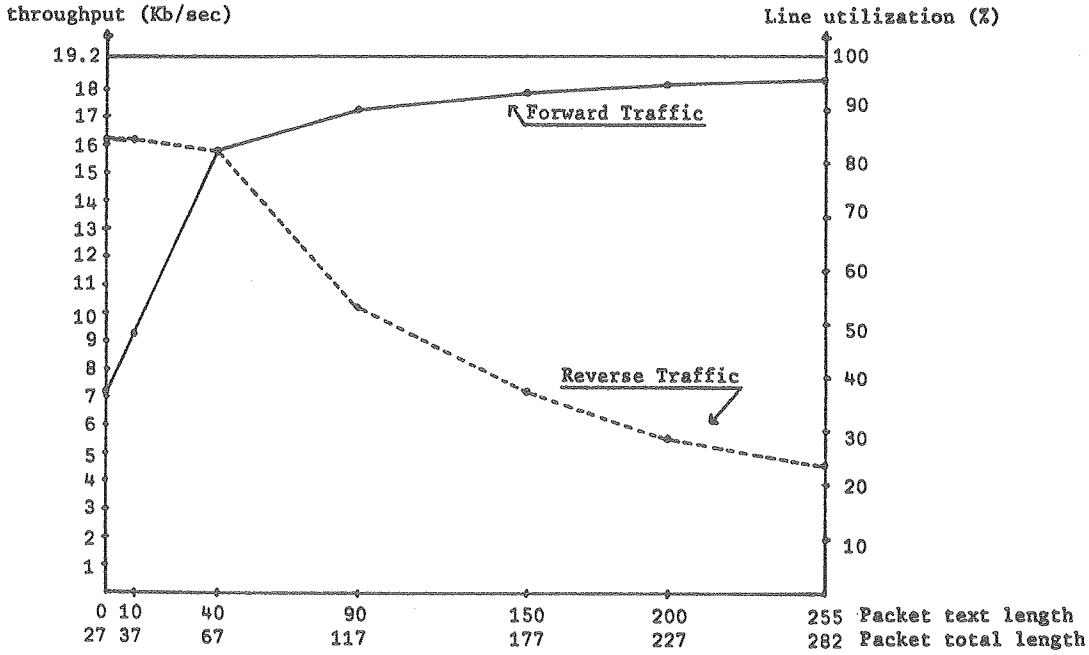


Figure 4 : TMM Performances (*) (Reverse Traffic Packet Length : 67 octets)

(*) The plotted throughput and line utilization include overhead (i.e. synchronization characteres, packet header, acknowledgements).

0 0 0 0 5 1 0 4 2 0 5

3. The model and its validation

The behaviour of the protocol is modelled by examining the state of each of the two nodes : let S_1 and S_2 be the state node 1 and 2 respectively. S_1 can take one of three values :

- state P corresponds to a packet transmission from 1 to 2
- state A corresponds to the transmission of an acknowledgement from 1 to 2
- state W is the state in which the sender of node 1 waits for an acknowledgement to arrive from node 2.

The states P, A, W of node 2 are defined in similar fashion.

We are assuming here that pure software times between state transitions of the senders are negligible. We shall see via our measurement results that this assumption is justified.

If both nodes are considered we obtain the state transition diagram of Figure 5.

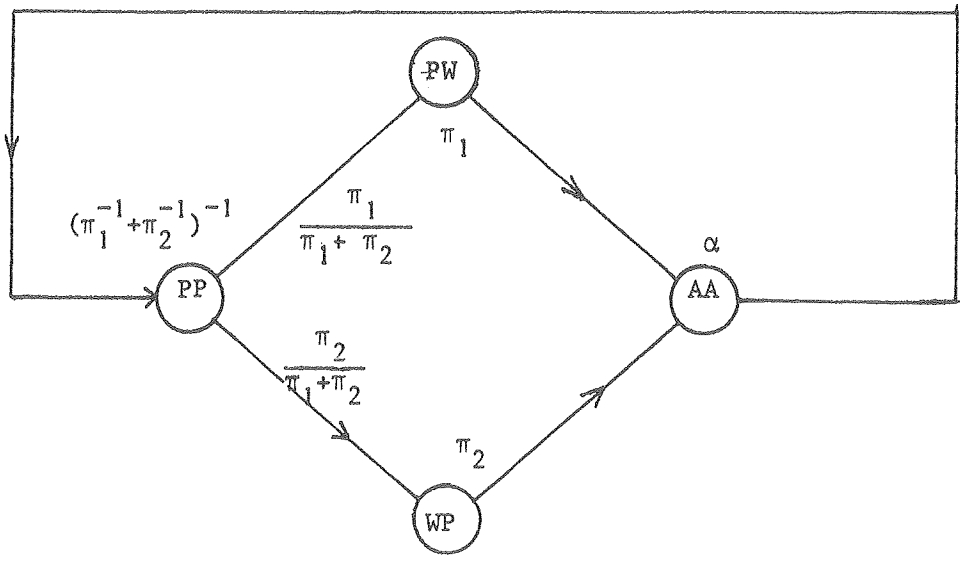


FIGURE 5

In state PP both senders are transmitting user packets, and a packet is travelling on each of the two lines. Depending on which packet transmission terminates first, state PW or state WP is entered. For instance, in state PW the packet transmission of sender 2 is complete and it enters the wait state (waiting for an acknoledgement to arrive from node 1). From state PW the system moves to state AA when the packet transmission from 1 to 2 is complete, since receiver 1 has received a packet and now proceeds to acknowledge it and so does receiver 2. Finally, after this last acknowledgement is transmitted the system will once more enter state PP.

Notice that the system cannot be in state PA or in state AP : if an acknowledgement is being transmitted from 1 to 2 then 2 must be waiting for it in state and vice-versa. Similarly, we do not allow state WW since we have excluded the possibility of software waiting times (such times have been assumed to be negligibly short compared to packet or acknowledgement transmission times; an estimate is obtained in Section 5).

The time spent in state AA is α (i.e. the transmission time of an acknowledgement). In the computations everything will be normalized so that α will also be the length in bytes of the acknowledgement packet. Let π_1 and π_2 be the average length of user packets sent from 1 to 2, and from 2 to 1, respectively. We shall assume in this section that user packets are exponentially distributed ; this restriction, which simplifies the analysis is not essential and is removed in the computations given in the Appendix. Our purpose in keeping this restriction in this section is to avoid complications which may confuse the reader.

With this assumption we show (in the Appendix) that the average time spent by the system in state PP at each passage through that state is $(\pi_1^{-1} + \pi_2^{-1})^{-1}$. The average time spent in state PW and WP is π_1 and π_2, respectively. Similarly we show that from state PP the system will next enter state PW with probability $\pi_1/(\pi_1 + \pi_2)$ or state WP with probability $\pi_2/(\pi_1 + \pi_2)$.

0 0 : 0 5 1 0 4 2 0 6

We notice that the model shown on Figure 5 is a semi-Markov process [4]. The steady-state probability that it will be in state $\mathcal{S}$ is denoted by $p(\mathcal{S})$: we shall need $p(PP)$, $p(PW)$, $p(WP)$ since these states are the only ones which contribute to the throughput. We define D_1 the throughput of sender 1 as

$$D_1 = p(PP) + p(PW)$$

and D_2 the throughput of sender 2

$$D_2 = p(PP) + p(WP)$$

Notice that in state PP both senders are busy. The total throughput is

$$D = D_1 + D_2 \tag{1}$$

These quantities are easily obtained as [3]

$$D_1 = \frac{1 + \pi_1/\pi_2}{1 + \frac{\pi_1}{\pi_2} + \frac{\pi_2}{\pi_1} + \frac{\alpha}{\pi_1}\left(1 + \frac{\pi_1}{\pi_2}\right)} \tag{2}$$

$$D_2 = \frac{1 + \pi_2/\pi_1}{1 + \frac{\pi_1}{\pi_2} + \frac{\pi_2}{\pi_1} + \frac{\alpha}{\pi_1}\left(1 + \frac{\pi_1}{\pi_2}\right)} \tag{3}$$

To simplify the notation we shall use $x = \pi_1/\pi_2$, $y = \alpha/\pi_1$. We notice the following after some simple algebra :

- For fixed π_2, D_1 is a monotone <u>increasing</u> function of π_1 which becomes 1 as $\pi_1 \to \infty$. . Similarly, for fixed π_2 it is monotone decreasing in π_2 tending to 0 as $\pi_2 \to \infty$.

- The above statements can be made also for D_2 if we interchange π_1 and π_2 (see Figure 6).

These theoretical results are illustrated on Figure 6. We have taken three values of π_1 (10, 20 and 100 bytes) and have set α = 7 bytes (this is the length of an acknowledgement packet in CIGALE [1]. We show D_1 and D_2, the forward and reverse traffic rate as a function of π_2 (the average length of packets in the reverse traffic. We also show D the total throughput (sum of forward and reverse traffic rate). Notice that equations (1), (2), (3) give the normalized traffic rate ; the true traffic is obtained by multiplying the normalized traffic rate by the link capacity in bits per second.

On Figure 7, 8, we validate the analytical model by providing a comparison with measurements on the CIGALE packet switching network.

Consider first the measurement results presented on Figure 5. The values are π_1 = 37 bytes (27 bytes of header, plus 10 bytes of text), and π_2 taking the values 27, 37, 67, 117, 227, 282 bytes. What is shown is the total traffic in one direction. $\hat{D}_1$ and in the other $\hat{D}_2$ in bits per second. $\hat{D}_1$, $\hat{D}_2$ include the acknowledgements.

We also show the measured number of acknowledgements per second A_m, and the theoretical values A_t computed from the following formula :

$$A_t = \frac{\hat{D}_1 - 56\ A_t}{8\ \pi_1} \tag{4}$$

where 56 is the number of bits in an acknowledgement. Notice that we could also use the formula

$$A_t = \frac{\hat{D}_2 - 56\ A_t}{8\ \pi_2} \tag{5}$$

The small discrepancy between A_t and A_m indicates the existence of measurement errors. In Figure 6 we show the useful traffic D_1', D_2' given in bits/second, where

$$D_1' = \hat{D}_1 - 56\ A_m \tag{6}$$

0 5 1 0 5 1 0 4 2 0 7

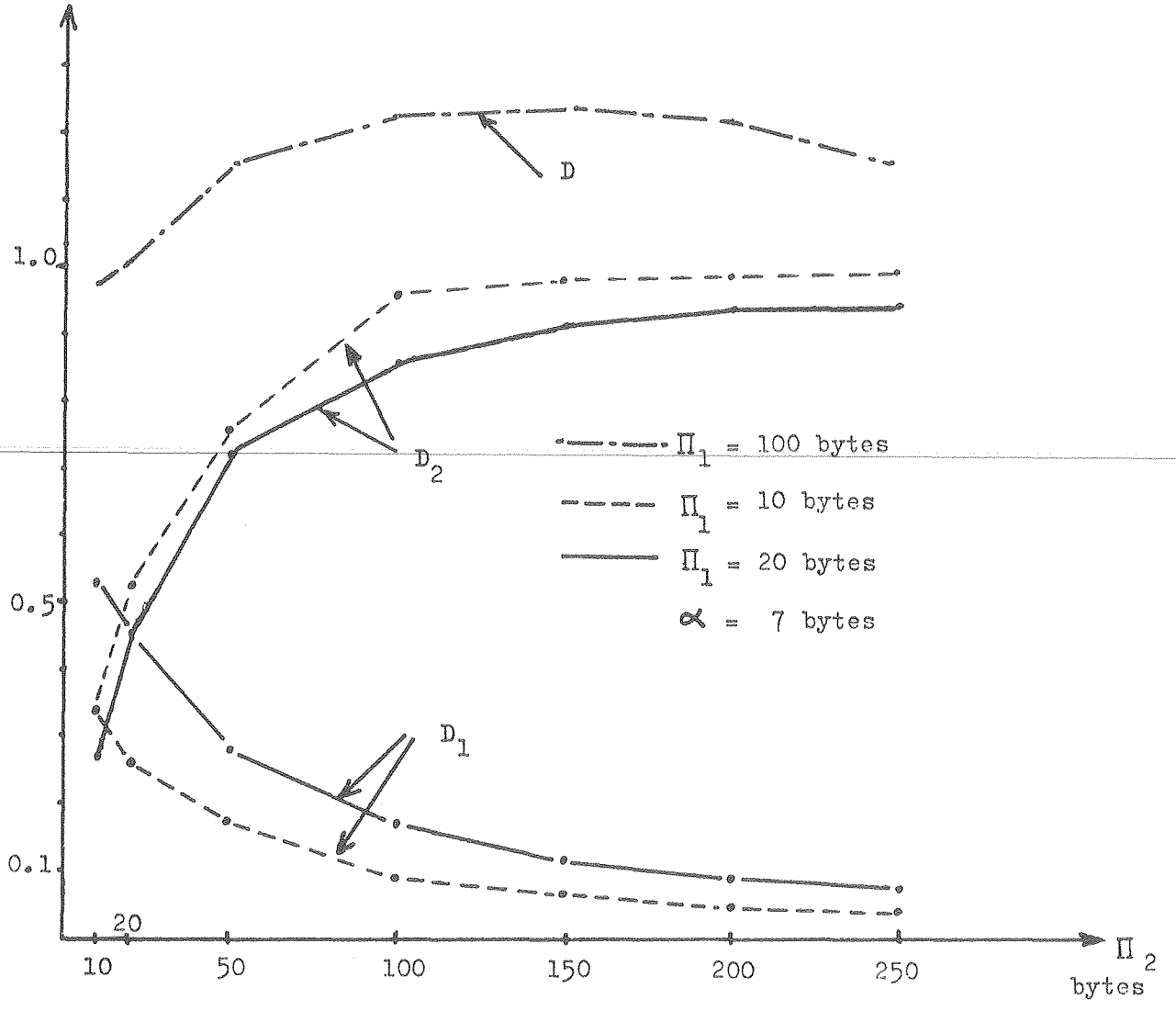


FIGURE 6.

π_2 bytes	27	37	67	117	177	227	282
A_m ACKS/sec.	41.6	41.6	27.3	17.1	12	9.7	7.9
A_t ACKS/sec.	41.5	40.6	26.2	17.33	11.8	9.66	8.09
D_1' bits/sec.	14320	12000	8000	5100	3600	2800	2300
D_1 bits/sec.	12590	11366	7400	5345	3733	2968	2422
D_2' bits/sec.	10720	12000	14700	16250	16800	17500	17800
D_2 bits/sec.	9185	11366	15401	16906	17849	18225	18462
D' bits/sec.	25040	24000	22700	21350	20400	20300	20100
D bits/sec.	21775	20442	22801	22251	21582	21193	20884
$\hat{D}_1$ bits/sec.	14600	14300	16200	17200	17400	18093	18270
$\hat{D}_2$ bits/sec.	11000	14300	9500	6040	4212	3342	2731

π_1 = 37 bytes, π = 7 bytes

Figure 7

π_2 bytes	27	37	67	117	177	227	282
A_m ACKS/sec.	27.3	27.3	27.3	17.1	12	9.7	7.9
A_t ACKS/sec.	27.35	26.7	26.72	17.36	12.07	9.55	7.84
D_1' bits/sec.	14665	14631	14297	10143	7054	5580	4578
D_1 bits/sec.	15740	14764	11970	8659	6361	5159	4269
D_2' bits/sec.	5711	7845	14282	17167	17809	17969	18231
D_2 bits/sec.	6345	8154	11970	15148	16790	17488	17935
D' bits/sec.	20375	22476	28576	27310	24863	23549	22809
D bits/sec.	22085	22918	23940	23807	23151	22647	22204
$\hat{D}_1$ bits/sec.	16194	16160	15823	10280	7150	5658	4641
$\hat{D}_2$ bits/sec.	7240	9374	15811	17304	17905	18047	18294

π_1 = 67 bytes , α' = 7 bytes

FIGURE 8

(7) $$D_2' = \hat{D}_2 - 56\ A_m$$

For the links being measured, the maximum capacity is 19 200 bits/second. In Figure 7 we also show the theoretical traffic obtained from equations (1), (2), (3). We see that the maximum error in the model prediction is 14 % (for the case $\pi_1 = 27$), while generally the error is of the order of 5 %. Similar results for other sets of measurements are given on Figure 8 , where the largest error is 16 %.

The high degree of accuracy of our model predictions confirm the validity of our simple mathematical model as well as its robustness : in the measurements all packet lengths are constant (and not exponential), and software wait times exist. The agreement between the theory and the measurements may be due to the fact that due to lack of synchronisation between nodes : a jitter is introduced which is perceived by the "other" node as a random packet length.

If we examine the data of Figure 5 we obtain that D, the total two-way throughput has a definite maximum value which is in the neighbourhood of $\pi_2 = 67$ bytes. This indicates that if we have $\pi_1 = 37$ we should set $\pi_2 = 77$ in order to maximize the total throughput. This does not seem to have an intuitive explanation and motivates our optimization study of the following section. We also notice that the maximum value of D, which is 22 801 bits/second, is well below the 38 400 bits/second which is the theoretical upper bound of the two-way link in this case. This point merits some analysis.

4. Packet size for throughput maximization

We have already noticed that D_1 is an increasing function of x, while D_2 is a decreasing function of $x(= \pi_1/\pi_2)$. We shall presently see that D given by (1) :

(8) $$D = \frac{2 + x + x^{-1}}{1 + x + x^{-1}} = y(1 + x)$$

has a maximum as a function of x. It is easily obtained by taking the derivative of D with respect to x and setting it to zero. D is maximized by

(9) $$x^* = \frac{1 - y}{1 + y}$$

and its maximum value is

$$(10) \qquad D^* = \frac{4}{3 + y(2 - y)}$$

assuming $y < 1$, i.e. that the length of the acknowledgement is shorter than the average length of a packet. From (9) and (10) we have that :

- the maximum throughput cannot exceed 2/3 of two-way channel capacity (obtained when $y = 0$) ; this is similar to Pouzin's predictions [5].
- when $y > 0$, the optimum value of π_2 is larger than π_1. That is, the system is optimized when the traffic is not identical in both directions.

Let us apply these results to the example of Figure 7 ; we see that the optimum value of π_2 computed from (9) is 54.35 bytes, and $D^* = 22\ 978$ bits/second. On Figure 8, the optimum value of π_2 is close to 67 bytes and the corresponding value of the traffic is 22 700 bits/second.

On Figures 9 and 10 we illustrate the theoretical results of this optimization study.

5. Long constant packet lengths

In the previous analysis we have examined the case of asymmetric traffic with exponentially distributed packet lengths. This model seems to be satisfactory when either :

(i) packets in each direction are of different length (most of the measurements given pertain to this case), or

(ii) packets in both directions are of equal length, but are small enough so that the random fluctuations introduced by the system become significant.

Our previous model does not seem to account accurately for the case when both nodes are transmitting long packets of equal length. This is the case we will treat here.

0 0 : 0 5 1 0 4 2 0 9

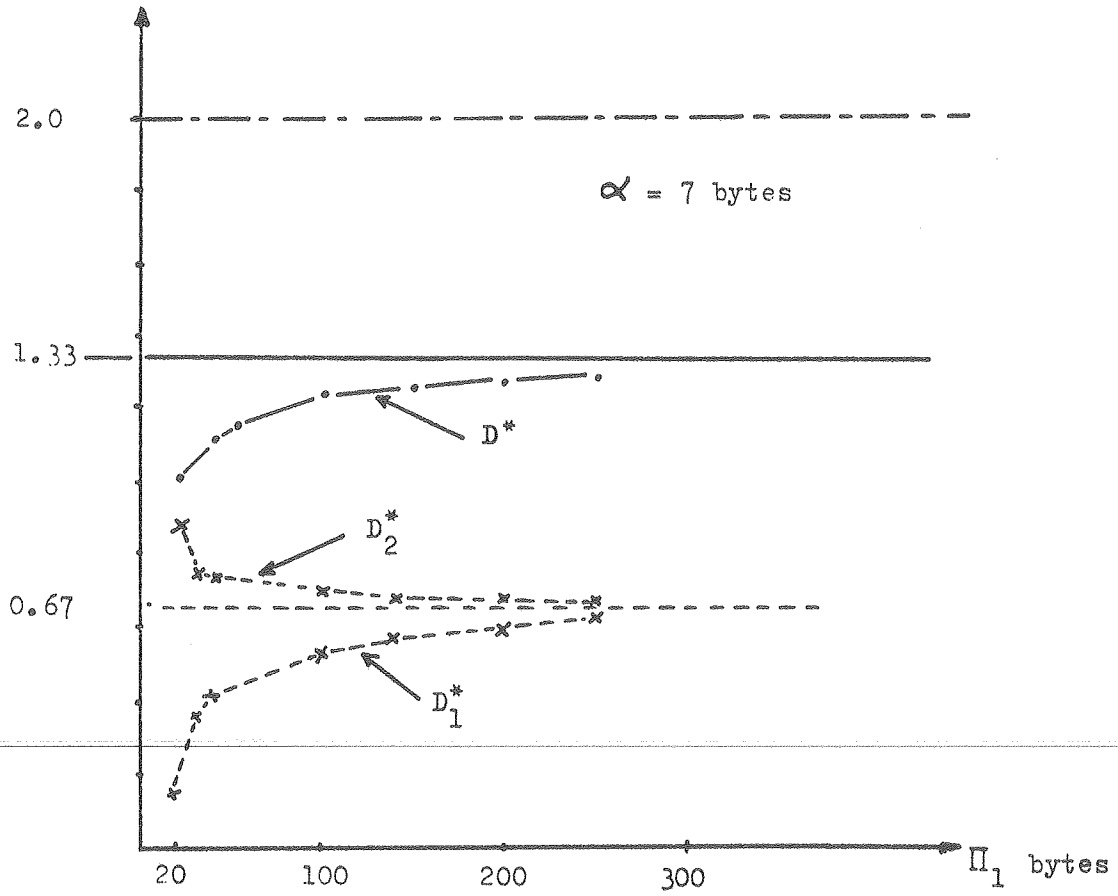


FIGURE 9

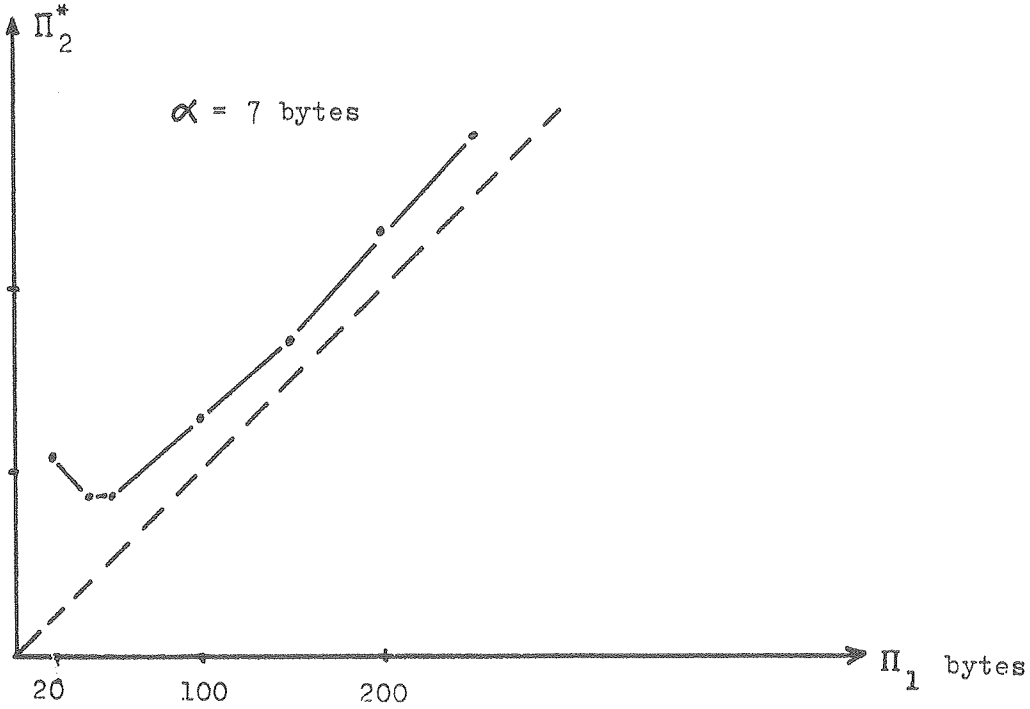


FIGURE 10

Suppose that $\pi \equiv \pi_1 = \pi_2$, and that packets transmitted in each direction are of constant length. If π is sufficiently large, it is reasonable to suppose that the time spent by the system in states PW or WP (see Figure 1) can be neglected. The state transition diagram is now given on Figure 11.

FIGURE 11

In this case

$$D_1 = D_2 = \frac{\pi}{\alpha + \pi} \quad , \quad D = D_1 + D_2 = \frac{2\pi}{\alpha + \pi}$$

~~For instance, for π = 282 we have $D_1 = D_2$ = 18 735 bits/sec. (if channel~~ capacity is 19 200 bits/sec.). A measurement experiment yielded $D_1' = D_2'$ = 17 200 bits/sec. The difference between the theoretical prediction and the measurement result can be explained by the existence of software wait times or of propagation delays. This phenomenon can be represented by a W state in the transition PP to AA and a W' state in the transition AA to PP ; let ω and ω' be the times spent in W and W', respectively. We shall provide an estimate of $\omega + \omega'$ by comparing the theoretical and experimental results. We now must have

$$D_1' \cong \frac{\pi}{\alpha + \pi + (\omega + \omega')} \times 19.200 \text{ bits/sec.}$$

so that $(\omega + \omega')$ = 25.8 bytes. That is, $(\omega + \omega')$ is the time equivalent to the transmission time of 25.8 bytes along the line. In time units we have $(\omega + \omega') = 25.8 \times 8/19.200 = 10.75$ milliseconds. This time would include all software times in the nodes as well as propagation delays.

6. Conclusions

In this paper we have given a model of the stop and wait protocol (TMM in CIGALE [2]) and have validated it by using measurements on the actually implemented system.

0 0 7 3 5 1 0 4 2 1 0

We then show that the maximum throughput of this protocol cannot exceed 2/3 of total two-way throughput for exponentially distributed packet lengths. We also obtain the optimum value of relative packet sizes.

The case of long constant packets of equal length has been presented, and we have provided an estimate of software wait times.

In the Appendix we have presented a general model of the protocol, in which the exponential assumptions concerning packet lengths are not made.

This analysis shows clearly that the send and wait protocol is unable to exploit fully the capacity of the communication links if packets are not of fixed and equal length in both directions, and long enough to render the randomizing effect of software wait times and propagation delays negligible. Our study also shows the effectiveness of simple mathematical models for the study of network protocols.

APPENDIX

We shall treat the case where the packet lengths are distributed according to some general distribution. Let $F_1(t)$, $F_2(t)$ be the probability distribution function for packets transmitted from 1 to 2, or 2 to 1, respectively. Then the probability, in the state transition diagram of Figure 1, of the transition "PP to PW" is :

$$p_1 \equiv \int_0^\infty dF_2(t)\ [1 - F_1(t)]$$

while the transition probability "PP to WP" is :

$$p_2 \equiv \int_0^\infty dF_1(t)\ [1 - F_2(t)]$$

The average time spent in state PP is :

$$\tau \equiv \int_0^\infty t[dF_2(t)\ (1 - F_1(t)) + dF_1(t)\ (1 - F_2(t))]$$

The average time spent in state PW is :

$$\tau_1 \equiv \int_0^\infty dx[1 - \int_0^\infty dF_2(t)\ (F_1(t+x) - F_1(t))]$$

while the average time spent in stape WP is :

$$\tau_2 \equiv \int_0^\infty dx[1 - \int_0^\infty dF_1(t)\ (F_2(t+x) - F_2(t))]$$

Therefore

$$D_1 = \frac{\tau + p_1\tau_1}{\tau + p_1\tau_1 + p_2\tau_2 + \alpha} \quad , \quad D_2 = \frac{\tau + p_2\tau_2}{\tau + p_1\tau_1 + p_2\tau_2 + \alpha}$$

The results of Section 3 are obtained if we assume that

$$F_1(t) = 1 - e^{-t/\pi_1}, \quad F_2(t) = 1 - e^{-t/\pi_2}$$

0 0 1 0 5 1 0 4 2 1 1

REFERENCES

[1] J.L. GRANGE - "CIGALE the packet switching subnetwork of CYCLADES" - Journées Informatique Médicale - Toulouse - Mars 1976 - Réseau CYCLADES, MIT 617.

[2] L. POUZIN - "Presentation and major design acpects of the CYCLADES computer network" - 3rd Data Comm. Symp. TAMPA - Nov. 1973 - 80-85 - Réseau CYCLADES SCH 511.

[3] N. NAFFAH - "Spécification d'interface TMM avec le réseau CIGALE" - Déc. 1975 - Réseau CYCLADES TRA 521.

[4] S. ROSS - "Applied probability models with optimization applications" - Holden-Day - San Francisco - 1970.

[5] L. POUZIN - "Efficiency of full-duplex synchronous data burk procedures" - IFIP WG6.1 - INW 35 - June 1973 - Réseau CYCLADES IRA 510.

[6] G. FAYOLLE, E. GELENBE - G. PUJOLLE - "An analytic evaluation of the performance of the stop-and-wait prptocol" - Rapport de Recherche n° 207 - IRIA-LABORIA - Dec. 1976.

DISTRIBUTED SIMULATION USING A NETWORK OF MICROCOMPUTERS

J. K. Peacock, J. W. Wong, and Eric Manning

Computer Communications Networks Group
University of Waterloo
Waterloo, Ontario

Abstract

Simulation, particulary of networks of queues, is an application with a high degree of inherent parallelism, and is of great practical interest. We hope to exploit this parallelism by performing simulations using a network of cheap, but slow, micro-processors.

We define a taxonomy by which we classify a number of parameters of a distributed simulation method, and seek solutions to problems of synchronization, deadlock prevention, and inter-process communication arising with various methods in each class of the taxonomy. We concentrate in particular on the class (loose, event-driven) which seems to possess the greatest potential parallelism, and on the class (tight, time-driven) which allows mock-up studies of real systems. We give algorithms for deadlock prevention and detection, and briefly discuss the design of a distributed operating system to support the simulation application.

This research forms part of a joint program with the Dept. of Computer Sciences, University of Texas at Austin, directed by Prof. K. M. Chandy.

INTRODUCTION

Why Simulate?

Simulation provides the means for observing a system's behaviour over time when direct measurement of the system parameters is inconvenient, or when it is desirable to view the system behaviour at an increased or decreased rate of time passage, relative to the observer.

Two separate classes of simulation exist: discrete and continuous. As the terms suggest, in discrete simulations, changes in the state of the system take place at discrete points in time and are instantaneous, whereas in continuous simulations, changes in state occur continuously in time. A queue at a bank teller is an example of a discrete system, while the planetary weather system is an example of a continuous system. The behaviour of a continuous system is usually modelled in terms of systems of simultaneous

0 0 0 0 5 1 0 4 2 1 2

differential equations. Solving such systems leads us into the realms of numerical analysis and analog computing. Since work on the solution of numerical problems using the Torus multi-microprocessor is currently underway (6), we restrict ourselves to considering the simulation of networks of queues, which represents a large class of discrete simulation problems.

Hardware Configuration

Due to the current availability of relatively slow but very cheap micro-processers, we wish to define and compare a number of different methods of performing simulation of networks of queues, with particular emphasis on exploiting computer architectures consisting of networks of micro-computers without shared memory. These form a class of so-called distributed systems.

Rather than using shared memory for inter-process communication, we use "thin-wire" communications (7,8). This approach makes shared variables rather difficult to implement, but in doing so prevents memory contention problems and encourages message passing for inter-process communication. It is also expected to enforce a more careful approach to the design of our software, since we are not allowed the luxury of shared global variables. On the other hand, it raises some interesting synchronization problems which are discussed by Le Lann (2).

In particular, we anticipate making use of 10 LSI-11's and 1 PDP-11/45 connected by a fixed time-division multiplexed bus developed at the Computer Communications Networks Group, called the SIBUS (10). The SIBUS allows 16 subscriber machines to each send a one word, addressed message every 3.2 microseconds. Unlike Cm*'s network of K.maps and intercluster busses (9), it does not support shared variables (of which we disapprove anyways), but the complexity and cost are rather smaller, the parts costing on the order of $1000.

Performance Measures

We hope to develop methods employing multi-computer architectures which offer substantial improvement in performance over single-processor methods. There are a number of performance criteria which we can consider. Since simulations tend to require large amounts of processor time to produce statistically meaningful results, we expect that one good performance measure will be the speedup in real time over single-processor simulation.

Another measure is the cost-effectiveness of the method employed. It will be interesting to determine these measures as a function of such parameters as the power of each processor, the number of processors used, the

interconnection method used, and, of course, the particular algorithm employed.

TAXONOMY

Definitions

There are a number of aspects to be considered in defining the characteristics of a simulation method. We therefore define a set of terms which will be applied to the description of the methods to be examined. These will have to do with such factors as the local or global nature of simulation time, the synchronization of changes in simulation time, and the event-driven or time-driven nature of each method.

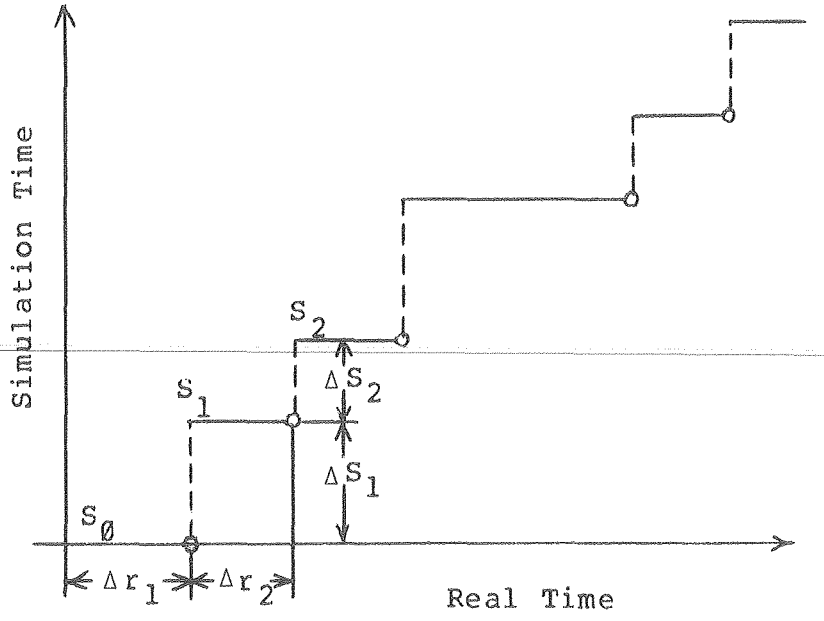
We first introduce the concept of simulation time and then define what is meant by event-driven and time-driven simulation. Simulation time is the abstraction of real time in the real system, in that the state of the real system at any real time will correspond to an abstracted state of the simulated system at the corresponding simulated time. Since we are interested in digital simulation methods, we also note that the values assumed by simulation time must be discrete. The simulated time need not, however, behave as real time does, that is, it need not change at a fixed rate relative to the computer's real time, nor be monotonic.

With this definition, we can characterize event- and time-driven simulation methods with respect to the sequence of values assumed by simulation time. In time-driven simulation the sequence of simulation time values forms an increasing arithmetic sequence, that is, $\Delta tsim$ = constant and $\Delta tsim > 0$. In event-driven simulation the simulation time sequence is still monotonically non-decreasing, but is not an arithmetic sequence, that is, $\Delta tsim \geq 0$; the sequence values represent times at which the state of the system changes, and we call such state changes "events".

The availability of one or many processors is the second aspect considered. The idea of mapping each component of a simulated system onto a separate processor has great intuitive appeal, since we can then manipulate the processors as we would the components of the real system, and parallelism is exploited. Also, the interactions between the components of the system must be reflected in the interactions between processors. For example, if we consider a queuing network with each server mapped onto a processor, then the interactions between the processors would have to allow the movement of customers from one server to the next, and each processor would be required to model the corresponding real server. In doing this on a single-processor computer, the parallelism just described ends up being hidden, and overhead to multiplex the single

0 0 : 0 3 1 0 4 2 3

CPU is introduced. Events are simulated one-by-one according to their sequence in simulation time, without considering that there are instances where many events could be simulated at once without making the simulation indeterminate. With multiple processors, there exists the possibility of exploiting this parallelism, but at the expense of creating a synchronization problem among the components.



ΔS_i = constant => Time-driven

Δr_i = q & ΔS_i = kq => Scaled Real Time

Figure 1.

At this point, we define tight and loose variants of the multi-computer method. We say that a method is tight if simulation time at any instant of real time is the same for all processors, and loose if each processor may have its own distinct value for simulation time. We can consider the processors to occupy points in space, and consider tsim as a function of the spatial coordinates. If we put processors along a one-dimensional axis, say x, then if tsim(x) = constant, we have the tight variant, and if tsim(x) $\neq$ constant, we have the loose variant.

The reason for introducing the possibility of allowing different simulation times at different processors is so that those components whose sequence of states do not depend on events at other components are not prevented from proceeding forward when they could do so. Each of these two kinds of methods has its own peculiar synchronization

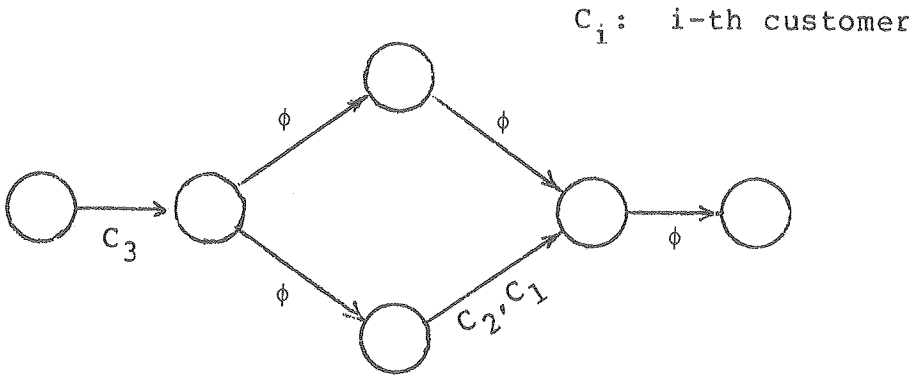


Figure 2. Blocked node example

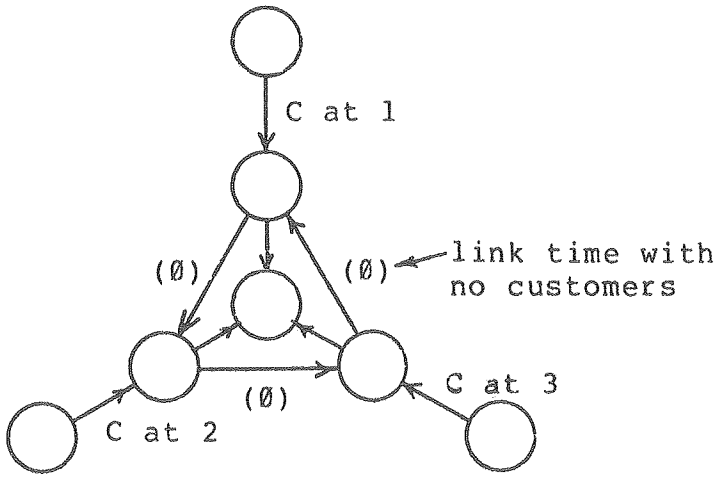


Figure 3. Deadlock example

0 0 1 0 5 1 0 1 2 1 4

problem. The tight variant requires that the simulation time sequence be the same in all of the processors, and that it should not be incremented until all of the processors have completed events scheduled at the current simulation time. The loose variant requires some sort of algorithm in each processor which decides, based on information received from processors which influence its state changes, whether or not it can step its simulation time forward . Such algorithms must ensure that there is no possibility for a deadlock situation to occur, that is, a situation where no processor can step its simulation time forward. (The Scaled Real Time technique is discussed below.) Figure 1 gives a graphical representation of the behaviour of simulation time as a function of real time.

LOOSE EVENT-DRIVEN METHODS

The loose event-driven methods give the greatest individual autonomy to each processor by allowing a server to step its simulation time forward to the time of the earliest event which may occur at that server. This section deals with the development of algorithms for computing the earliest event time at a node using information from the nodes which supply it with customers.

To aid in discussion, we shall consider the network of servers and queues to be a directed graph with each server and its queue represented by a node, and each path between servers represented by an edge. External arrivals to the network will be modelled as nodes with no input edges, and will generate a continuous stream of customers for their output edges. Similarly, departures from the network will go to nodes with no output links. Note that there will be queuing of customers on the edges of the graph as well as inside the node, since the node must merge the customers arriving on each input edge into correct time sequence. We require that the sequence of simulation arrival times of customers on the edges of the graph be monotonic non-decreasing. Without this stipulation, we would not be able to perform the straightforward merging of customers at each node.

This merging could easily be accomplished by accepting the customer with the minimum simulation arrival time across all input edges, except for the fact that there may not be any customers on some of the input edges. If there is an empty edge, then the node is not able to decide whether to wait for an arrival on the empty edge or accept the minimum from the edges which have customers, and we say the node is blocked. To resolve this problem, we have adopted the solution that every edge has a link time, which is a lower bound on the arrival time of the next customer on the edge. When an edge is not empty, the link time is simply the arrival time of the first customer waiting on

the link. When an edge is empty, the source end is responsible for maintaining the link time as the greatest lower bound on the next departure across the link, according to the information which it possesses. The destination node of an empty edge can use the link times in its merging process, and when the edge with the minimum link time is empty, the node is blocked. A blocked node becomes unblocked only when the link times of all empty edges become greater than the minimum link time. This happens only when the source ends of empty edges are unblocked; the node either generates a customer for the empty edge, or is able to compute a greater lower bound, due to the processing of a customer. We say a node is deadlocked if the link time on an edge which blocks a node becomes fixed, and hence the node never unblocks. Figure 2 gives an example of a situation in which a node has to wait because of an empty link.

Here we have the makings of a possible deadlock situation. If the source node of the empty link is blocked also, and the activity into the source node ultimately depends on the activity of the original blocked node, then we have a circular dependency and a deadlock situation exists. Figure 3 gives a simple example of a deadlock which occurs at initialization, when all of the node times are Ø. The following Theorem, discovered independently by Chandy and Holmes (1) and presented here without proof, gives necessary and sufficient conditions for deadlock to occur.

Theorem. A deadlock exists if and only if there is a cycle consisting of empty links which all have the same link time, and nodes which are all blocked because of these links.

The importance of this theorem is that it gives us the means to find a solution to the deadlock dilemma. First, we note that acyclic graphs present no problem, since the necessary condition that a cycle exists is not fulfilled. Secondly, we note that if we can somehow force the relationship between input edges and output edges so that $t_i \neq t_{i+1}$ at any node in a cycle, we can break the necessary condition of identical time values around a cycle. A simple way to do this is to insist that a customer must spend a non-zero amount of time at any server it visits. Hence, the lower bound on the empty output edges will be the minimum input time plus the minimum service time at the server, and will thus be strictly greater than the minimum input time. We note that an intuitive interpretation of the conditions which allow deadlock is that each node in the cycle is waiting for a customer on the empty input link to arrive at the lower bound time. Since zero service times are allowed, the potential exists for this possible

0 0 : 0 3 1 0 4 2 1 5

customer to traverse the cycle in zero time, and hence to traverse the cycle an infinite number of times before time is advanced at any of the nodes in the cycle.

The solution given above successfully prevents deadlock from occurring, yet it causes a kind of time-driven behaviour to occur in parts of the network which contain cycles. This removes some of the advantage of using event-driven timing, so we seek a solution which will allow the time in a component of the graph with feedback to advance directly to the next event to be performed in that component.

To aid in developing a distributed algorithm which finds the next event in a strongly-connected component, we distinguish two kinds of events at a node: conditional and unconditional. Suppose a node is at time t and the next event on its list is at time a. If it is possible for another event to occur between time t and a, then the event at time a is conditional, otherwise it is unconditional. (The conditional case implies that the node is blocked and must wait until the event at a is unconditional, or there is another unconditional event between time t and time a.)

These definitions give another way of looking at the deadlock problem. We can say that a deadlock occurs if and only if there exists a cycle whose edges are causing all of the events at the nodes to be conditional. Of the blocked nodes, there will be one or more whose event times are minimal. If a blocked node could determine that its next event time were minimal, it could change the status of its event from conditional to unconditional, since it would then know that no other events could intervene before the minimum time.

We have now reduced the deadlock problem to one of finding the minimal event time in a graph. One solution to this problem is to have a central controller which keeps track of the next event times at all the nodes, and at each stage selects the node with minimal event time to proceed. Since we prefer to keep the algorithm as distributed as possible, this is not a satisfactory solution, although it provides a good benchmark against which to compare distributed algorithms. Algorithms for the case of the cyclic graph have been given and analyzed by Le Lann (2), and by Chang and Roberts (3). These algorithms require O(N) time per event to synchronize N processors and N*N and N * Log N message passages, respectively. Unfortunately, these algorithms do not seem to generalize easily to the case of the general graph, although we could use their solutions if we imposed a cycle on the nodes of the given graph. Since the complexities of these algorithms depend on cycle lengths, some improvement on this appears possible if we impose a cycle on each strongly-connected component, rather than imposing one cycle on the whole graph.

In the next section, we explore algorithms for finding the minimal set of event times using information available from previous synchronizations of the network. By way of analogy, we explore algorithms which perform a kind of merging or sorting, whereas the Le Lann and Chang et al algorithms simply find the minimal element.

Virtual Ring Synchronization Algorithm

We consider the case of N processors interconnected in a ring topology, where each processor has a list of events with associated times. We denote the time associated with event i as ti. We require that for any two events p and q in the system that if tp < tq, event p is to be processed before event q. At processor k, the minimum of the event times is denoted mk. (mk = min{ti|i is an event at processor k}). We also maintain at processor k a rank value rk which is the number of processors j such that mj < mk. It follows from this definition that if rk = Ø, then processor k may proceed to process the event corresponding to time mk, since there is no mj, and hence no ti, such that ti < mk.

We now have a measure, rk, at each processor with which we can determine whether or not processor k may proceed. We now address the problem of how the value of rk at each processor can be computed efficiently, giving an algorithm which is shown elsewhere (4) to be correct, based on the assumptions that messages which are sent around the ring do not pass one another, and that the resynchronization of a processor receiving a customer is completed before the sender advances its minimum event time.

The algorithm is as follows, using the C language (5):

At processor k:

```
struct message
{ int src; /* Source processor of message */
  int rnk; /* Will contain updated rk */
  int min; /* Value of mk for compares */
  int omin; /* Value of previous mk */
} msg;

int k,mk,omk,rk;

main()
{ mk = rk = Ø;
  while (1)
  { if (rk == Ø)
    { proc_event();
      omk = mk;
      mk = min_ev_tim();
      rk = rk-1;
```

0 0 0 0 5 1 0 4 2 1 6


```
        msg.src = k;
        msg.rnk = 0;
        msg.min = mk;
        msg.omin = omk;
        send(msg); /* Send to next processor in ring */
      }
      wait_msg(msg); /* Receive message in msg */
      if (msg.src == k)
      { rk = msg.rnk; } /* Now have a valid value for rk */
      else
      { if (mk < msg.min) msg.rnk = msg.rnk+1;
        else if (mk > msg.omin && mk <= msg.min) rk = rk-1;
        else if (mk <= msg.omin && mk > msg.min) rk = rk+1;
        send(msg); /* Send message on */
      }
    }
}
```

The algorithm is started with ri = mi = 0 for all i, with a null event at time 0 which we might call the "startup event". With M events to be processed, randomly distributed in time and among the processors, we see that the time required to synchronize the events in the total run is O(N+M*N/2) if all the events have unique times. This follows from the fact that startup takes time N and the next event from the current one is on the average halfway around the ring. The number of messages passed is exactly (N+M)*N, since each event requires that a message go around the loop and there are N+M total events.

This algorithm is not optimal, since a processor may have to wait for some other processors which cannot influence the sequence of event times at the waiting processor. In the next section, we present an algorithm which takes the topology of the graph into account.

Blocking Table Algorithm

This algorithm accomplishes synchronization by constructing at each blocked node j a list of nodes which could cause a pre-emption of the conditional event at node j. This is done by transmitting a "request", consisting of the time of the conditional event at j and j's identity, to the sources of all empty input edges to j. The identities of the receiving nodes are also entered into the blocking table at node j. Each node i receiving a request replies with a list of nodes which are in turn at the source end of its empty input edges, and which are to be added to the blocking table at node j. Also, the conditional event time is entered into a table of request times at node i. When node j receives an addition to the blocking table which is not already recorded there, it adds it to the table and sends a request to the added node. A node i in j's

blocking table sends a message informing j that it is no longer blocking it when the next event time at i exceeds the request time stored for node j. Note that since the next event may be conditional, and hence is pre-emptible, it is possible for node i to reintroduce itself as a node which is blocking j. This happens if a customer arrives at i which pre-empts the conditional event and whose simulation arrival time precedes the request time for node j.

This algorithm has the desirable property that it blocks a node from proceeding only when the possibility of pre-emption exists. We describe it in more detail and show it to be correct elsewhere (4).

SCALED REAL TIME METHOD

Scaled real time is a tight, time-driven simulation technique satisfying the following scaling property:

real time in processors = real time in system * k

where k is a time scaling factor. With this property, a service of x units is simulated by keeping the customer record in the processor for kx units of time.

This approach allows the experimenter to observe the dynamic behaviour of the system as the simulation progresses in much the same way as he does with simulation using an analog computer. The simulation can be speeded up or slowed down by adjusting the scaling factor k.

A hardware feature required for the efficient implementation of scaled real time is a programmable interrupt timer. The timer is decremented at regular intervals of q units of real time and an interrupt is generated when the timer register goes to zero. To simulate a service time of x units, the timer register is set to int(kx/q). Service completion is then signalled by the timer interrupt. Since the timer interrupt has a resolution of q units, the above method implies that

real time in processors = int(real time * k /q) * q

A smaller q will therefore give better accuracy in the simulation.

The simulation algorithm executed by each processor is a function of the service facility being simulated. It depends on characteristics such as the number of servers, the service time distribution, and queueing discipline. For the case of a single server queue, the algorithm is given below:

```
struct message
{} msg;
int server;

main()
{ init();
```