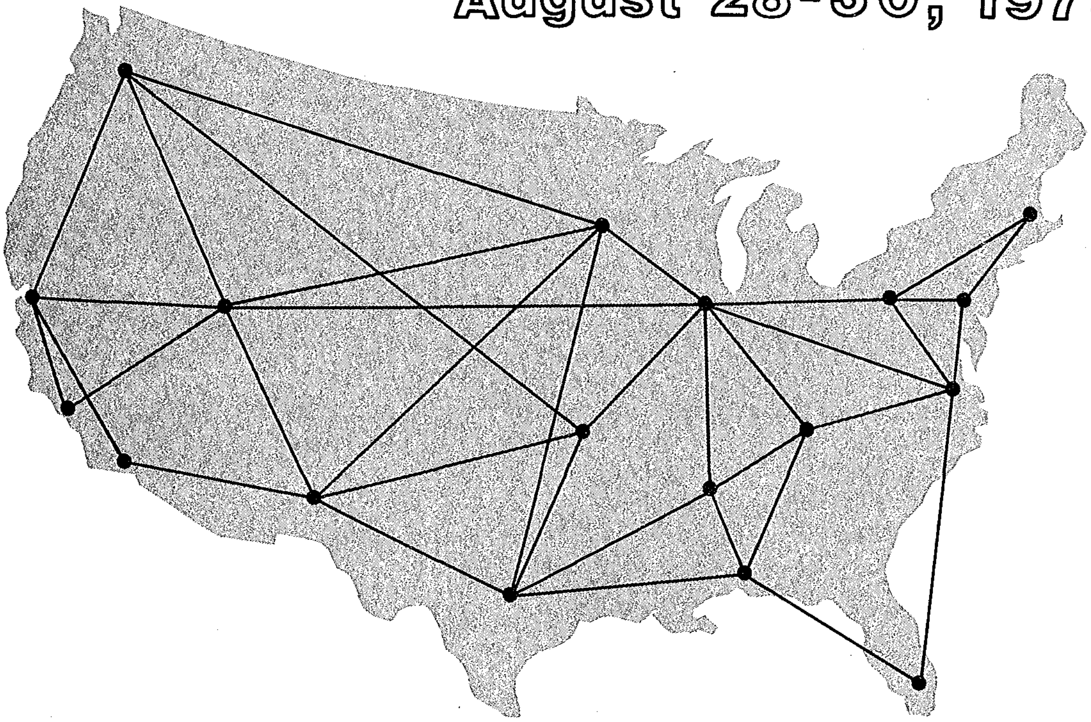
Fredric C. Gey / LBL

PROCEEDINGS of the

LBL-9433
UC-32
CONF-790834

FOURTH BERKELEY CONFERENCE on DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS

August 28-30, 1979



LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA, BERKELEY

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

LEGAL 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.

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: A18

LBL-9433

PROCEEDINGS OF THE FOURTH

BERKELEY CONFERENCE ON DISTRIBUTED DATA MANAGEMENT

AND COMPUTER NETWORKS

Sponsored by

Computer Science & Applied Mathematics Department	Applied Mathematical Sciences Research Program
Lawrence Berkeley Laboratory	Office of Energy Research
University of California	U. S. Department of Energy

And in cooperation with ACM and IEEE

AUGUST 1979

General Chairman:	Dennis Hall, Lawrence Berkeley Laboratory
Program Cochairmen:	Michael Stonebraker, University of California, Berkeley Carl Sunshine, RAND Corporation
Program Committee:	

Ed Birss, Hewlett-Packard Corporation
Gregor Bochmann, University of Montreal
Stephen Crocker, Information Sciences Institute
Yogen Dalal, Xerox SDD
John Day, Digital Technology Inc.
Ivan Frisch, Network Analysis Corporation
James Gray, IBM Research
Paula Hawthorn, Britton-Lee
Gerald Popek, UCLA
Lawrence Rowe, University of California, Berkeley
Daniel Sagalowicz, SRI International
Patricia Griffiths Selinger, IBM Research
David Shipman, Computer Corporation of America
Stewart Schuster, TANDEM Computers Incorporated
James White, Xerox SDD
John Wong, University of Waterloo

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 conference.

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

E. Birss
G. Bochmann
D. Clark
S. Crocker
Y. Dalal
J. Day
I. Frisch
L. Garlick
J. Gray
P. Hawthorn
S. Kimbleton
G. Popek
L. Rowe
D. Sagalowicz
J. Shoch
P. Griffiths Selinger
D. Shipman
S. Schuster
J. White
J. Wong

CONTENTS

IMPLEMENTATION OF DISTRIBUTED SYSTEMS - I

DISTRIBUTED DATA BASE INTEGRITY

PROTOCOL MODELING

MULTIPLE COPY CONTROL TECHNIQUES

NETWORK RESOURCE ALLOCATION

IMPLEMENTATION OF DISTRIBUTED SYSTEMS - II

IMPLEMENTATION OF DISTRIBUTED SYSTEMS — I

XNDM: AN EXPERIMENTAL NETWORK DATA MANAGER

Stephen R. Kimbleton
Pearl S.-C. Wang
and
Elizabeth N. Fong

National Bureau of Standards
Washington, D.C. 20234

ABSTRACT

Data base access is increasingly important in a networking environment. Two alternative approaches can be identified: i) implementation of distributed databases presenting the user with one logical database implemented across a collection of computers or, alternatively, ii) development of network data managers providing a uniform user and program viewpoint across heterogenous DBMSs. While the first approach is the most natural extension of the concept of an individual DBMS, its utilization imposes certain requirements including the necessity for converting existing DBMSs if their data is to be supported in the distributed environment. The second approach minimizes or eliminates conversion problems; however, it has not yet been shown feasible. This paper describes an ongoing research project concerned with establishing the feasibility, issues, alternatives, and a technical approach for supporting a network data manager. Although implementation has not been completed, the initial evidence is positive and suggests that network data managers may well prove either an acceptable alternative or useful intermediate stage to a distributed database.

1. INTRODUCTION

Computer networks support the sharing of remote programs and data. The gradual maturation of networking technology, as measured by the increasingly sophisticated protocols and applications being implemented [ARPAN 76], [INWG 77], has resulted in increasing demands for supporting remote access to data.

This work is a contribution of the National Bureau of Standards and is not subject to copyright. Partial funding for the preparation of this paper was provided by the U.S. Air Force Rome Air Development Center (RADC) under Contract No. F 30602-77-0068. Certain commercial products are identified in this paper in order to adequately specify the procedures being described. In no case does such identification imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the material identified is necessarily the best for the purpose.

An individual user, interacting with a remote database management system (DBMS), issues queries and updates in the data manipulation language (DML) used by the system and receives data in response. Because of differences in: i) the data model used in constructing DBMS supported data structures, ii) the functionality provided by the software even if the underlying data models are the same, iii) data structure, e.g. data base semantic differences which are also likely even if the same underlying data model is employed, iv) DML differences, and v) computer system differences, the user wishing to access multiple remote databases is faced with a substantial learning burden.

This paper argues that this learning burden can be substantially offloaded from the user. Accomplishing this requires a network data manager providing a uniform user viewpoint across multiple remote heterogeneous DBMSs. The feasibility of this approach is being explored through constructing an Experimental Network Data Manager (XNDM) at the National Bureau of Standards.

The basic assumption underlying the design of XNDM is heterogeneity of data models, data structures, DBMSs, DMLs and computer systems on which these DBMSs reside. Superimposing a uniform user viewpoint in such an environment clearly requires a substantial amount of software and may be a significant source of delay in processing user requests.

To explore this issue, recall that information processing requirements can be divided into three categories [ANTHR 65]: operational control, managerial control and strategic planning. As one passes from operational control to strategic planning, the bandwidth of the application decreases as does its predictability. Intuitively, we believe that network data managers are inappropriate for operational control, highly appropriate for strategic planning, and may be of help in managerial control. For example, handling inventory out-of-stock conditions could be simplified through a means for querying remote DBMSs to determine an alternative source of supply when an out-of-stock is indicated by the local DBMS.

The preceding suggests that strategic planning and exception reporting constitute two likely applications for a network data manager. Moreover, the nature of these applications suggests that the additional overhead of supporting a network data manager is likely to prove very acceptable in comparison with the burden of manually performing the necessary translation processes in response to unpredictable and non-recurrent demands.

The remainder of this paper provides a more detailed discussion of XNDM. To provide context, section 2 establishes some comparisons between a network data manager and a distributed database. Section 3 describes the user's view provided by XNDM. Section 4 discusses translation technology required to support this view and observes that it differs substantially from that currently discussed in the data translation literature. Section 5 describes the current XNDM implementation status and presents some concluding remarks.

2. NETWORK DATA SUPPORT OPTIONS

A distributed DBMS (DDBMS) is usually viewed as one logical DBMS implemented across several host computers. Thus, excluding performance differences, there is no apparent difference to the user in accessing a DDBMS and accessing a DBMS resident on a single host using the same data structures and data manipulation language. Moreover, through redundancy, the DDBMS potentially permits increased reliability and decreased access times to frequently used portions of the database. Redundancy does require care in ensuring consistency of multiple data copies and in synchronizing updates [ROTHJ 77], [STONM 77].

Using a DDBMS poses the need for conversion of existing DBMSs. The current state of database conversion suggests that non-trivial costs are associated with this process [NAVAS 76]. Moreover, even if these costs were insignificant, the resulting organizational dislocation in adapting to the new DDBMS is likely to be extensive. Consequently, the DDBMS approach may prove infeasible given the environment in which it is to be implemented.

A network data manager is intended to provide an alternative to the DDBMS through providing an easy means for simplifying network access to multiple, heterogeneous DBMSs. The basic relationship between a network data manager and the individual DBMSs is illustrated in Figure 2-1 in the context of the NBS Experimental Network Data Manager (XNDM). Thus, a process represented as a circle within one computer (PHOST) interacts in a uniform way with multiple independent DBMSs located in one or more computer systems.

Our working hypothesis is that the network data manager approach is likely to prove very acceptable in handling unpredictable and non-recurrent requests. Moreover, given the cost of database conversion, it is also likely to be the only feasible way of easily adapting to the opportunities for sharing information which are provided by networking. Thus, we are motivated to consider its design and development in greater detail.

3. THE NETWORK USER ENVIRONMENT

The two essential functions of a Network Data Manager are provision of a uniform user environment across individual (heterogeneous) local DBMSs (LDBMSs), and translating between this user environment and the LDBMSs. The remainder of this section structures the basic components of the XNDM supported user environment while the following section addresses translation technology.

3.1 Data Model/Data Language Selection

Developing a data language and data model for XNDM can be approached either as a problem of developing a 'best' data model and data language and then considering the issues in translating to existing data models and languages or through selecting one of the existing data models and languages. The former is a problem of independent interest. Requiring its solution as the prerequisite to analyzing network data managers seems undesirable. Instead, we have chosen to examine the existing alternatives, select a reasonable candidate, and place primary emphasis on the data manager specific aspects of the problem. This has expedited our consideration of the basic nature of the problem. It will be interesting to see if future data model/data language research can be easily accommodated as we expect or, instead, will require substantial revision.

Selection of a data model for XNDM has been driven by three basic assumptions. The first is that the network user is naive vis-a-vis the access requirements of local DBMSs. The second is that the network user should be assisted to ensure that queries and updates are meaningful. The third is that the local DBMS should be provided with relatively tight guarantees that the network user will not be able to adversely affect its operations through ignorance or intent. Note that the second and third assumptions are closely interrelated.

The first assumption motivates selection of a data model and data language minimizing the knowledge and effort required to support access. That is, the data model should present data in a way which is easy for the user to understand. Further, the Data Manipulation Language (DML) should minimize procedural (extent to which the user must specify how rather than what is to be retrieved or updated) and navigational (need for explicitly specifying interrelationships between data elements) requirements.

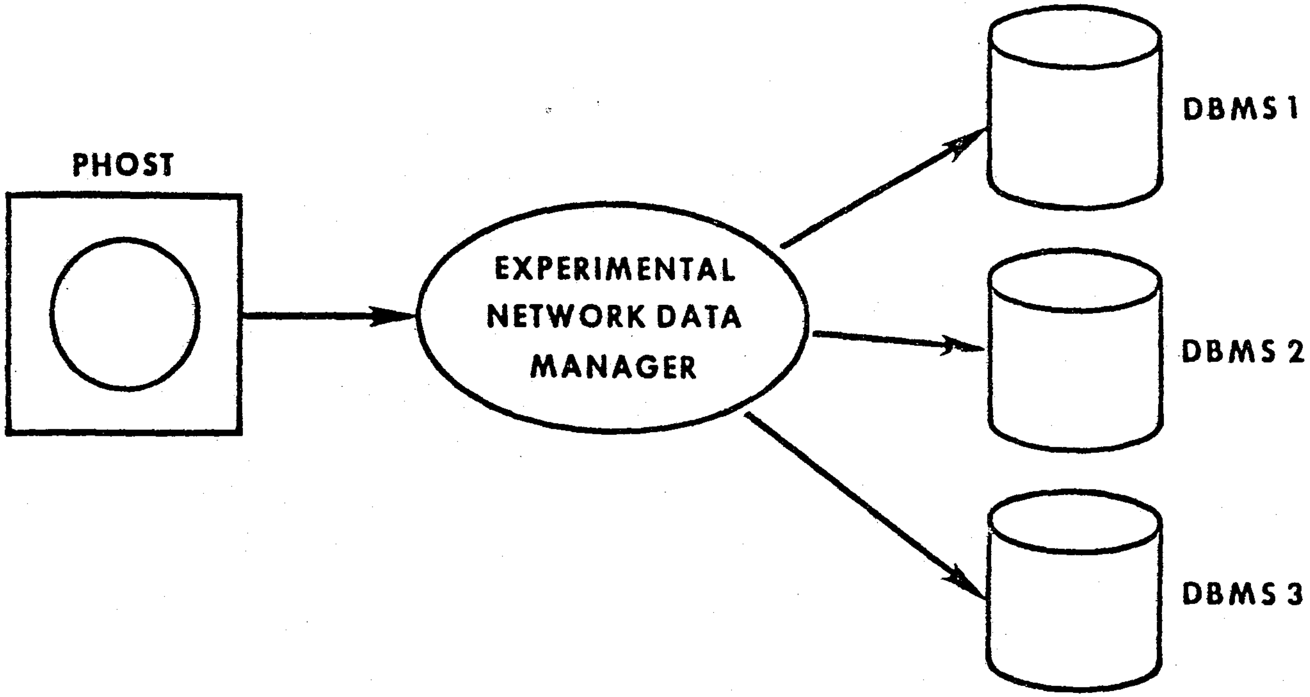


Figure 2-1. XNDM Interface between User Program and Multiple Remote DBMSs

Of the three basic data models: relational, hierarchical, and network, it is our opinion that the relational model is the simplest to understand. Accordingly, we have chosen tables as the basic mechanism for representing data. Although a properly chosen user schema can result in an appropriately simple user viewpoint regardless of the particular global schema employed, the static nature of such a schema conflicts with the random and unpredicatable nature of arriving requests.

The requirements for our second assumption are met through a semantic integrity system that ensures meaningful queries and updates as discussed below. Moreover, an access control mechanism is also being implemented to ensure that the network user is only permitted to access data appropriate to his/her access rights. This is the basic tool for meeting the third requirement.

3.2 Global Schema Specification

Central to the specification of an XNDM global schema is the balancing of the conflicting requirements of the network users so as to provide a design that can satisfy the need of the "community" of users - as opposed to the need of any individual user.

As discussed above, a basic XNDM assumption is that a uniform user environment is to be superimposed on a highly heterogeneous collection of existing local DBMSs. This requires: i) a common view of data to be presented to the network user, and ii) a means for mapping from this common view to the target systems. Note that this common view need not contain all of the data in the local DBMSs. Rather, it will probably comprise only that data thought to be of common interest. This, in turn, is likely to be a subset of the data which local DBMS management is willing to make available to the network user. Since both of these selection processes are judgmental, we assume that the selection of data and its attributes is performed by a team (of database administrators?) responsible for the overall utilization of the network data manager.

Given this selection, and the resulting structuring using the described data model, the need arises for a suitable translation process. This translation process proves to be substantially different from that currently discussed in the data translation literature. It is discussed in some detail in the following section.

3.3 Experimental Network Data Language

The Experimental Network Data Language consists of three major components: i) Experimental Network Data Manipulation Language (XNDML), ii) Experimental Network Data Control Language (XNDCL), and iii) Experimental Network Data Definition Language (XNDDL).

Since the basic XNDM objective was to explore the feasibility of providing a uniform environment for the network user, we decided to adopt an existing DML and add any extensions which proved necessary. After some consideration, we have chosen SEQUEL [CHAMD 76] to provide the basic framework for XNDL since: i) it is a table based DML, and ii) it has been subjected to human factors oriented investigations which have improved the quality of its user interface [REISP 75].

Currently, the design of both the query and update portions of XNDML has been completed and implementation of the query portion is underway. Implementation of update capabilities is being deferred pending completion of the design of XNDCL and XNDDL.

XNDML is both a subset and extension of SEQUEL. XNDML is a subset since it does not contain the SEQUEL sorting facilities and certain alternative ways of stating predicates. Sorting was eliminated because it adds little to demonstrating the feasibility of a network data manager and can be an expensive consumer of processing time on the host containing the LDBMS. XNDML is invoked via subroutine CALLs. Thus it does not have a host language interface corresponding to that provided by SEQUEL. Table 3-1 lists the six major categories of XNDML query commands

XNDML extends its SEQUEL subset to meet the need for specifying the target database. Three major alternatives can be identified: explicit specification, implicit specification, and specification of location as a virtual attribute.

The target database can be explicitly specified by using the statement DATABASE IS 'DATABASENAME'. The effect of this statement is to make all subsequent XNDML statements refer to this DATABASE until another target specification is encountered.

Implicit specification of the target database occurs when the user issues an XNDML statement without any target database specification. In this case, XNDM maintained information is used to identify the relevant databases (those containing information about the entities and relationships identified in the XNDML statement). The statement is then applied against each such database and the results aggregated.

The third and most sophisticated specification is through treatment of location as a virtual attribute. This logically attaches a location column to each relation seen by the user. This permits one to construct queries in which the predicate applies to location as well as to entities and their attributes. Thus, assuming that the distance between sites is known, one can specify the site of the location to replenish an out-of-stock condition as a function of conditions prevailing at each relevant location. For instance, an out-of-stock replenishment rule might be to replenish in an amount inversely proportional to distance and directly proportional to stock on hand. Distance proportionality can be used to lower shipping cost overhead while stock on hand proportionality could be used to avoid unduly impacting a site with a low stock level.

Two required XNDM support functions are data location and access path determination. Data location uses the Network Wide Directory System contained within the NBS Experimental Network Operating System [KIMB 78]. Access path information is provided by the XNDDL processor.

TABLE 3-1. XNDML Query Categories.

C1 SELECT (columns)

C2 SELECT....WHERE (rows)

C3 PARTITION

C4 SET OPERATIONS

C5 AGGREGATION

C6 COMPOSITION

3.4 Semantic Integrity

Semantic integrity is a significant issue in the context of an individual DBMS since it provides a means of assuring that the database is a valid representation of the application environment. Two major reports [MCLED 76] and [BRODM 78] have appeared on this subject as well as a variety of papers. The general objective is ensuring that if one starts with a valid DBMS configuration, subsequent updates will not impair this validity.

Semantic integrity is of greater importance in the context of a network data manager since local DBMS management is likely to want strong assurances that remote, and therefore presumably less knowledgeable users, will not affect DBMS integrity. This problem varies somewhat from that for an individual DBMS. XNDM cannot assure that the database is, initially, in a consistent state. Thus, the major concern is that updates are semantically correct. A lesser concern is facilitating the correct structuring of queries through supporting strong domain typing.

XNDM semantic integrity concerns also differ from the corresponding problem for an individual DBMS because the network user's view of data is virtual. Thus, there is a premium on performing all non-data dependent integrity checking before proceeding with the data dependent checks. This may ultimately result in a partitioning of integrity checking functions between XNDM and the LDBMS. In any event, the major issues can be divided into two major categories: i) assurance of integrity at the network level, and ii) assurance of integrity at the local DBMS level.

Although work on the XNDM Semantic Integrity System is in its preliminary stages [FONGE 79], some initial observations can be made. Semantic integrity can be expressed at the global schema level through the (virtual) tabular data model. Assuring integrity within an individual table can be subdivided into assurance of attribute integrity, row integrity, column integrity, and predicate integrity.

Assurance of semantic integrity is provided via two facilities: strong domain typing and predicate-based assertions. Strong domain typing facilities of XNDM permit the user to define: i) the format of the data, ii) the acceptable range of values, iii) the collection of legal (arithmetic, logical and string) operations, and iv) the interrelationships among data elements in terms of the collection of legally acceptable operations.

Predicate-based assertions specify validity criteria which are to hold in the application environment. The facility provided in XNDM will permit: i) specification of rules for consistency and correctness of data bases, ii) the time at which the assertion is to be enforced, and iii) the actions to be taken when the assertions are not satisfied.

Assuring predicate-based integrity for either an individual relation or for a collection of relations can imply significant overhead depending on the amount of data involved and the types of checks which must be performed.

3.5 Access Controls

A second major support function required for acceptance of XNDM is provision of an appropriate access control mechanism. Currently, many DBMSs provide access controls via passwords on files [DATEC 77]. This is clearly insufficient for the level of functionality intended to be provided by XNDM. The issue is whether a significantly better system can be implemented. This issue has been discussed in [WOODH 79]; the following summary considerations are based on the discussion contained therein.

Access control mechanisms can be divided into two major categories [KARGP 77]: i) non-discretionary access control mechanisms which support organizational constraints on the sharing of information, and ii) discretionary access control mechanisms which permit user directed controlled sharing of information.

Security levels and compartments constitute a major example of non-discretionary access control mechanisms. Conceptually, a user is labelled with security level(s) and compartments, e.g. level is SECRET, compartment is NATO, and is entitled to access all information having the same, or lower levels, e.g. level is SECRET or CONFIDENTIAL, compartment is NATO.

System-R provides an example of a sophisticated DBMS discretionary access control mechanism [GRIFP 76]. Through its use, an individual user is permitted to grant a subset of his/her access rights to another user. The supported functionality permits READing, INSERTing, DELETEing, UPDATEing, and DROPing (of an entire table). Moreover, a GRANT command permits one user to provide another user with the ability to GRANT rights. These mechanisms are supported for both an entire table and for individual columns of a table.

XNDM provides both discretionary and non-discretionary access controls. Their combined support requires a mechanism for checking that discretionary grants do not conflict with non-discretionary controls. This checking process has been implemented using the lattice security model [DENND 76].

4. TRANSLATION TECHNOLOGY

This section: i) establishes the differences between data translation required to support XNDM and that currently considered in the data translation literature, ii) discusses the two major alternatives in implementing a translation capability, and iii) describes the translation process which we have selected. Currently, translation has only been implemented for the query portion of XNDML which, for simplicity, we refer to as the Experimental Network Query Language (XNQL).

4.1 The Nature of the Translation Process

Data translation can be characterized in two different dimensions: i) online vs. offline, and ii) constraints on source and target data structures. XNDM translation requirements differ from those usually discussed in the data translation literature since: i) it is a real-time, online process, and ii) it is dependent upon both source and target data structures.

The requirement that the translation process be real-time and online forces a substantially different translation process than that usually considered in the context of database translation [NAVAS 76]. Specifically, the need for explicit consideration of physical representations of data is eliminated while the need for an online and realtime level of functionality cannot be avoided.

XNDM translation also differs from that usually associated with database front ends and database terminals. (A database front end presents the user with data structures differing from those actually employed by the DBMS being accessed and often based on a different data model. Thus, there is substantial interest in relational front ends to DBTG DBMSs. For a front end, the data structures presented to the user are 'fixed' and the data structures employed by the target DBMS are derived from the user presented data structures. Database terminals, in contrast, provide the user with a constant data model and DML across heterogeneous DBMSs. The target data structures are fixed and the data structures presented to the user are derived from these target data structures [KLUGA 78].)

In both of these cases only one set of data structures is fixed while the other is derived from this fixed set. This allows substantial freedom in tailoring data structures to simplify the translation process. Such freedom is not available in constructing a network data manager in which the data structures presented to the network user are fixed (recall that they were chosen by a committee) and the data structures of the target systems are also fixed.

4.2 XNDM Translation Alternatives

An XNQL statement specifies the sequence of operations to be performed on the underlying information structures. It is a high-level language, and by its very nature, does not specify the step-by-step, system-specific actions needed to evaluate the query by a given target DBMS. It is the function of the translator to supply these details.

Since XNQL is a query language, the primitive information structures of the language are aggregated, not simple, data. That is, the basic 'atoms' of data expressed in an XNQL statement are relations rather than individual data elements. The translator interprets these data objects in terms of the primitive data constructs provided by the particular target DBMS and its data structuring rules.

Construction of the XNQL translator is further complicated by the fact that different target systems support different primitive operations and data structures; therefore we need not a single translator but a family of translators. Two approaches to their realization can be identified: construction of a collection of source-target specific translators or, alternatively, construction of a single translator for the bulk of the translation process common to all translators together with custom tailored front ends handling the source specific portion of the translation process and custom tailored back ends handling the target specific portion of the translation process.

Construction of independent translators has the advantage that design unity and run-time efficiency is more achievable with a single translator for each target DBMS. However, an entire translator is needed to support each additional target, whereas in the family approach all the translators share a core design which defines the common (source and target-independent) part of the translator. Each new translator in the family is obtained by building source and target-oriented specialities on top of the basic design. Therefore the bulk of the implementation effort is available across different target systems and new developments need not start from scratch.

An important side-effect of the family approach is the insight it provides for DBMS data manipulation and structuring facilities. That is, a simple, coherent design for a translator family is impossible without abstracting the essential properties of target systems and recognizing their commonalities and differences. Thus, we have chosen the approach of designing a good general framework, i.e. a consistent, efficiently implementable translator allowing effective use of target system facilities. The insights provided by this framework are augmented by those developed in preparing the mappings to and from specific target systems.

4.3 XNQL Translation

The complex semantic manipulations required for translation are achieved by means of step-by-step transformations of an appropriately chosen internal representation of the input text. We have chosen a tree as the intermediate representation because of the requirement for flexibility in handling a wide range of target DML's and data structures.

Each transformation takes us somewhat closer to the target query by either changing the original form of the input text to uncover the underlying "basic structure" of the query tree which characterizes the system-independent organization of queries, or reshaping the basic tree to incorporate the surface structure of the target language. The value of this transformational approach is that it reduces the overall translator complexity and also supports a simple, consistent, modular design [DEREF 76].

The translation process is (vertically) segmented into five phases as illustrated in Figure 4-1. A more extensive discussion is contained in [WANGP 79].

Lexical and Syntactic Analysis

The tasks of the lexical and syntactic analysis modules are conventional [GRIED 69]. They produce a source(XNQL)-specific syntax tree representation of the input query. This tree contains all the information originally present in the source text as well as all the information that is inherent in the XNQL grammatical description. The source syntax tree is the first of a sequence of trees used in the translator as intermodular data structures. Each later module takes as input the tree produced by the previous module and leaves a tree that is closer to the target query by reshaping the tree, pruning source-specific information from the tree and/or incorporating target-specific information into the tree. The basic task facing the translator writer is disentangling those aspects of the source and target queries that reflect "essential" (language-independent) logical structures frcm those that characterize "incidental" (language-specific) representational details.

Standardization

Processing beyond the syntactic level can be made simpler if the source syntax tree is transformed into a standard form where each WHERE clause is represented as a binary tree of predicates connected by AND and OR nodes arranged in conjunctive normal form [STONM 76].

Static Semantic Processing

Since each XNQL query interacts with a data space which is the Cartesian product of several relations subject to the restriction of the WHERE clause, and frequently these restrictions are such that the Cartesian product becomes an equi-join (merging of two relations based on a common column), differences in source and target structures at the record level imply different join conditions in the queries.

The static semantic level of the translator does the processing needed to account for data structure differences at and below the record level by first resolving data item name differences and then the differences in the joins.

The "data item renaming" module traverses the source syntax tree from the top down, replacing all leaf references to source(user) data items with corresponding references to target data items and depositing their attribute information at these nodes. The "record structure mapping" module then deletes all predicate nodes representing joins between different source relations and inserts the appropriate join predicates for target records.

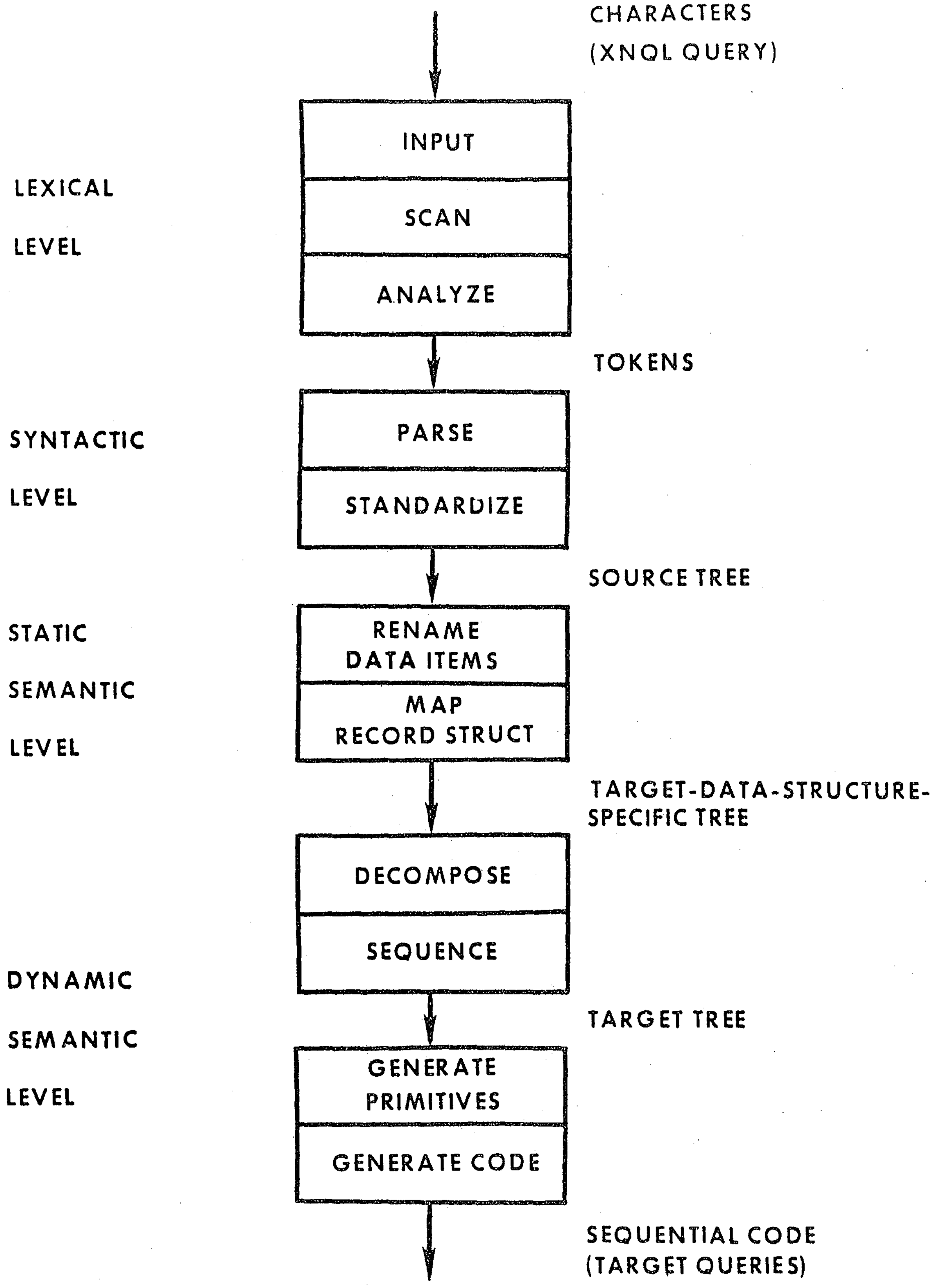


Figure 4-1. The XNQL Translator as a Tree Transformer

Dynamic Semantic Processing

The transformations happening at this level account for the differences in the logical structures of the source and target query languages. Since the unit of data structure for each target query may be smaller than for XNQL (e.g. each Codasyl DML statement can only involve a single record (or set) type, whereas there is no limitation to the number of different tuple types (relations) an XNQL statement can manipulate), we first decompose the query tree into sub-trees, each of which involves a single unit of data structure that a target query can handle. The "sequence" module then chains the sub-trees together in the order that the corresponding queries should be sequenced for the target DBMS and selects the execution sequence of these chains that minimizes the amount of intermediate records needed to be processed.

Code Generation

This is the final phase of the translator and outputs the desired target DML statements that can be executed by the local DBMSs. The first module interprets each of the subtrees along the chains produced by the Sequencer and generates CALL statements to primitive target database operations. The second (code generation) module then expands these CALLs into sequences of actual target DML statements.

The exact form of the primitives depend upon the particular target system we are considering. Their behavior characteristics fall, in general, into the following categories: search or return the first/next instance of a specified record type, test the truth value of some predicate expression of the record type, partition all instances of a record type on the basis of some data item values and evaluate aggregate functions for the specified record type. (These correspond roughly to the information algebra operations [CODAS 62] of searching/returning the first/next point of a line, bundling, glumping and evaluating functions of lines.)

This extra level of indirection before the actual code generation allows us to separate out the representational details of the target DMLs and makes it possible to have a standard set of primitives for each general class of target systems, that is, Codasyl, relational calculus and relational algebra systems.

The decision to set the primitives at a fairly procedural level (namely, one record instance at a time) was driven by the flexibility it provides for expressing a variety of access strategies. This allows easy incorporation of optimization modules which selects the "best" access paths for the input query based upon knowledge of how the records are stored (keys, inversion indices, etc.). This is particularly important since the value and usefulness of XNDM in a real environment depends critically upon its performance and experiences with current relational DBMSs indicate that some form of optimization is essential in bringing the performance to an acceptable level [SMITJ 75].

5. IMPLEMENTATION STATUS AND CONCLUDING REMARKS

This paper has described the design and ongoing implementation of a collection of functions for providing a uniform network view of data across a heterogeneous collection of network accessible DBMSs. Our experience to date suggests that XNDM is a realistic and pragmatic approach for achieving the advantages of networking given a significant, in place, collection of DBMSs.

Perhaps the three key issues in ensuring user acceptance of a network data manager are: i) access controls and semantic integrity, ii) developing more sophisticated translation capabilities optimizing the allocation of the translation process among NDM and LDBMS, and iii) performance. We believe the basic issues and a reasonable approach for (i) have been discussed in this paper. Developing a more sophisticated translation capability is of obvious importance and closely relates to the performance issue. Implementation of translators should be paralleled with research directed toward a better understanding of the nature of the translation process. Some work is beginning to appear in this area [KLUGA 78] establishing the theoretical limits of translation feasibility.

5.1 Implementation Status

XNDM translation is performed on a PDP-11/45 attached to the Arpanet as are the other host computers. The operating system for the PDP-11/45 is UNIX [THOMK 74] and the translator is programmed in C. To provide a more uniform interface to the translator, small support modules termed envelopes are implemented on the system on which each LDBMS resides. Basic communications support between systems and the ability to preserve meaning in transporting structured records between heterogeneous systems is provided by an Experimental Network Operating System (XNOS) [KIMBS 78]. Work on the XNQL translator is still in progress. The current version handles two out of the six XNQL constructs (selections of columns and rows), for the following target systems: the Multics Relational Data Store (MRDS) [HONEY 77], a relational calculus system, and the Honeywell 600/6000 Integrated Data Store (IDS) [HONEY 71], a Codasyl-like system. For MRDS, the translator can handle all target data structures in general, but for IDS, target records with multiple owners and multiple members are excluded.

5.2 Implementation Approach

Two different approaches to implementing XNDM can be considered. The first distributes the implementation across the supported host systems while the second, which we have adopted, offloads the implementation, to the extent possible, onto a separate satellite computer.

The tradeoffs between these two approaches are essentially those of evaluating the cost of supporting an additional computer versus the cost of implementing common modules on several different systems. Given the opportunity for centralized design, implementation and support afforded by offloading and the increasingly high cost of software, we believe that offloading is the natural approach in an evolving technology. The alternative might be appropriate for an extremely static environment.

6. ACKNOWLEDGMENTS

The authors would like to express their appreciation to Gary Sockut, Helen Wood, and Fran Nielsen who provided many helpful comments and substantial assistance in preparing this paper.

7. REFERENCES

[ANTHR 65] Anthony, R., "Planning and Control Systems: A Framework for Analysis," Division of Research, Graduate School of Business Administration, Harvard University, 1965.

[ARPAN 76] Arpanet Protocol Handbook, Network Information Center, Stanford Research Institute, Menlo Park, CA, April, 1976.

[BRODM 78] Brodie, Michael, L. "Specification and Verification of Data Base Semantic Integrity", University of Toronto, Computer System Research Group, Technical Report CSRG-91, April, 1978.

[CHAMD 76] Chamberlin, D.D., et al., "SEQUEL 2: A Unified Control", IBM Journal of Research and Development, Nov. 1976, pp. 560-575.

[CODAS 62] Codasyl Development Committee, "An Information Algebra", Comm. of the ACM, Vol. 5, No. 4, April, 1962, pp. 190-204.

[DATEC 77] Date, C.J., An Introduction to Database Systems, Addison-Wesley, Second Edition, 1977.

[DENND 76] Denning, Dorothy E., "A Lattice Model of Secure Information Flow," Comm. of the ACM, Vol. 19, No.5, May, 1976, pp. 236-243.

[DEREF 76] DeRemer, F. L., "Transformational Grammars", in Bauer, F. L., and Eickel, J. (eds) Compiler Techniques - An Advanced Course, Springer-Verlag 1976, pp. 121-145.

[FONGE 79] Fong, Elizabeth, "Semantic Integrity System for an Experimental Network Data Manager", in preparation.

[GRIFP 76] Griffiths, Patricia P. and Bradford W. Wade, "An Authorization Mechanism For a Relational Data Base System", IBM Research RJ 1721, Feb. 1976.

[HONEY 71] Honeywell Information Systems, Inc., Integrated Data Store, Order No. Br69, Rev.1, December, 1971.

[HONEY 77] Honeywell Information Systems, Inc., Multics Relational Data Store (MRDS) Reference Manual, Order No. AW53, Rev.0, September 1977.

[INWG 77] "A Network Independent File Transfer Protocol," prepared by The High Level Protocol Group, INWG Protocol 86, HLP/CP(78)1, December, 1977.

[KARGP 77] Karger, Paul, "Non-Discretionary Access Control for Decentralized Computing Systems," SM Thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, May, 1977. (Also available as MIT/LCS/TR-179, Laboratory for Computer S_ience,M.I.T., May, 1977, NTIS AD A040808.)

[KIMBS 78] Kimbleton, Stephen R., Helen M. Wood, and M. L. Fitzgerald, "Network Operating Systems - An Implementation Approach", Proc. National Computer Conference, AFIPS Press, Anaheim, CA, Vol. 47, June, 1978, pp.773-782.

[KLUGA 78] Klug, Anthony C. "Theory of Database Mappings", Ph.D. Thesis, Department of Computer Science, University of Toronto, 1978.

[MCLED 76] McLeod, Dennis, "High Level Expression of Semantic Integrity Specifications in a Relational Data Base System", MIT Report MIT/LCS/TR-165, available from DDC AD-A034184.

[NAVAS 76] Navathe, S.B. and J.P. Fry, "Restructuring for Large Databases: Three Levels of Abstraction," ACM Transactions on Database Systems, Vol. 1, No.2, June, 1976, pp. 138-158.

[REISP 75] Reisner, P., R.F. Boyce and D.D. Chamberlin, Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL", Proc. National Computer Conference, Anaheim, CA, Vol. 44, May, 1975, pp. 447-452.

[ROTHJ 77] Rothnie, James B. and Nathan Goodman, "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley, CA, May, 1977, pp.39-57. (Also available from Computer Corporation of America, 575 Technology Square, Cambridge, MA 02139, as Technical Report No. CCA-77-04).

[SMITJ 75] Smith, John Miles and Philip Yen-Teng Chang, "Optimizing the Performance of a Relational Algebra Database Interface", Comm. of the ACM, Vol. 18, No. 10, October 1975, pp. 568-579.

[STONM 76] Stonebraker, M., Eugene Wong, Peter Krepts, and Gerald Held, "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Vol. 1, No. 3, September, 1976, pp. 189-222.

[STONM 77] Stonebraker, M., and E. Neuhold, "A Distributed Database Version of INGRES", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977, pp. 19-36.

[THOMK 74] Thompson, K. and D. Ritchie, "The UNIX Time-Sharing System," Comm. of the ACM, Vol. 17, No. 7, July, 1974, pp. 365-375.

[WANGP 79] Wang, Pearl S.-C., "Common Query Language for the Networking Environment: Design, Translation Structure, and Initial Implementation," in preparation.

[WOODH 79] Wood, Helen M. and Stephen R. Kimbleton, "Access Control Mechanisms for a Network Operating System," to appear, Proc. National Computer Conference, June, 1979.

AN ARCHITECTURE FOR SUPPORT OF NETWORK OPERATING SYSTEM SERVICES

Richard W. Watson
John G. Fletcher

Lawrence Livermore Laboratory
Livermore, California

This paper argues that network architectures should be designed with the explicit purpose of creating a coherent network operating system (NOS). The resulting NOS must be capable of efficient implementation as the base (native) operating system on a given machine or machines, or of being layered on top of existing operating systems as a guest system.

The goals and elements of a network architecture to support a NOS are outlined. This architecture consists of a NOS model and three layers of protocol: an interprocess communication (IPC) layer, with an end-end protocol and lower sub-layer protocols as needed to support reliable uninterpreted logical-message communication; a service support layer (SSL), abstracting logical structures and needs common to most services, including naming, protection, request/reply structure, data-type translation, and session support; and a layer of standard services, (file, directory, terminal, process, clock, etc.).

0. INTRODUCTION

Most current network architectures consist of one or more function-oriented protocols, such as virtual terminal or file transfer protocols, built on top of an interprocess communication (IPC) protocol layer [6,7,19,22,26,35,41]. The potential of computer networking for resource sharing and distributed computing cannot be realized with such architectures because [17,24,44,49]:

- No basis is provided for easily creating, in a layered fashion, new resources or services out of existing ones.
- Each programmer desiring to provide or use a new network sharable resource must face anew all the issues of data-type translation, command and reply formatting and parsing, naming, protection, and interfacing to the IPC protocol layer.
- The terminal user or programmer must know the different naming and other access mechanisms required by the network, each host, and each service.
- The setting up of accounts and other administrative procedures are awkward.

These problems can be eliminated if a network architecture is explicitly designed to support the evolution of a network operating system (NOS). Three important NOS design goals are the following.

The prime design goal is that a process (program), terminal user, or programmer have a uniform coherent view of distributed resources. Processes, programmers, and terminal users should not have to be explicitly aware of whether a needed resource is local or remote. This does not mean that programs or users have no control over where a process is to be run or other resource is to be located or that they cannot learn the locations of resources. It means that a user need not (although he may) program differently or use different terminal procedures depending on resource location and that network operations and the idiosyncrasies of local hosts can be largely or completely hidden. There may however, depending on resource location, be performance differences. One consequence of this goal is that if a resource or its controlling service is relocated for economic, performance, or other reasons to another system in the network, then at most a new name (address) is required, but no changes are required in the program logic or resource access mechanisms.

A second goal is that the NOS structure be efficiently implementable and usable as the base (native) operating system on a single system of common current architecture, as well as be implementable as a "guest" layer on existing operating systems that support appropriate interprocess communication [21]. By the former condition we mean that, when implemented as the native operating system, access by local user processes to local services should be as efficient and no more involved in terms of the number and kind of messages or system calls exchanged than is common on existing single systems OS's. Initially NOS's will likely be implemented, as guest systems, on top of existing OS's, but over time, as part of the evolution toward distributed computing, we expect that the structure of base OS design to evolve toward that required for a NOS.

A third important goal is extensibility, implying:

- That users can easily add new services built on existing services without requiring system programmers to add new resident or privileged code. (Some services may be made resident or privileged for performance enhancement, but that is a separate issue.)

- That the basic NOS structure not require the NOS to spring full blown into existence with all possible services to be useful; in other words that it can start with a few services and evolve.

- That systems desiring to participatge in the NOS as users of or providers of a single service be able to do so with minimal implementation.

A NOS must perform the same basic functions as an operating system on a single host:

- Turn a collection of hardware/software resources into a coherent set of abstract objects or resources (such as processes, files, directories, clocks, accounts, etc.) and support their naming, access, sharing, protection, synchronization, and intercommunication (including error recovery).

- Multiplex and allocate these resources among many computations.

An NOS must solve the problems that exist for single host OS's and must deal with the problems arising from its distributed nature and the heterogeneous systems on which it is based: translation problems due to different encodings and data representations, distributed service and resource structures, potentially more complex error recovery, multiple copy file or database update problems, multiple controlling administrations, and special efficiency problems arising from distance and bandwidth between components. Creating an extensible, coherent set of services or resources in an environment of distributed and heterogeneous systems requires a NOS model and supporting structure to handle the above problems. This paper is focused on such a NOS framework and the areas where we see coding, communication, and other standard conventions to be required or useful to support the services that will reside within an NOS. It is beyond the scope of this paper to discuss design of specific NOS services, or many of the critical implementation issues of an NOS.

The Lawrence Livermore Laboratory's high performance local network (Octopus) [13,14] is currently undergoing a change in its hardware interconnection to increase performance and be more modular, is being extended to interconnect hundreds of local micro/mini/midi computers with each other and the high performance central facilities, and interconnect with other networks [46]. The network architecture under development described here will provide the new software base for this evolution. A prototype operating system for a single machine using many of the features of the NOS structure to be described is also presently being implemented.

1. NETWORK OPERATING SYSTEM MODEL

Model Structure

We believe that the first step in creating a network architecture is to choose an NOS model [10,17,24,44]. One approach is to use an existing operating system as the NOS model and extend it into a distributed environment. The RSEXEC work at BBN is a pioneering example that extended some of the facilities of an existing OS (TENEX) to distributed homogeneous systems and was later layered on OS's of other systems as well [43]. We do not believe this to be the preferred approach, because most existing OS's have monolithic structures and weaknesses in their inteprorcess communication mechanisms [21] that inhibit their easy extension into a distributed environment. The National Software Works (NSW) [28,39] and the later BBN works on the ELAN system represent documented approaches to designing NOS's from scratch for explicit distribution [17,44].

The NOS framework we have chosen is based on the object or resource model of an operating system [23,38]. All communication among processes is by message passing. The model is shown in Figure 1.

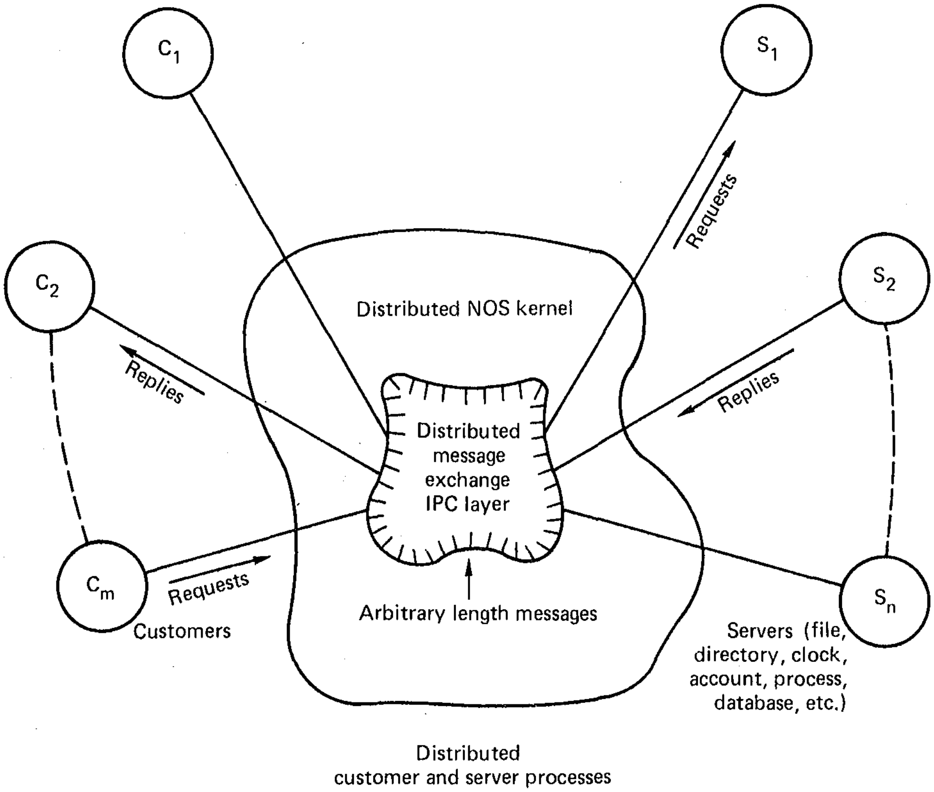

C_1
S_1
C_2
S_2
Requests
Replies
Replies
Distributed NOS kernel
Distributed message exchange IPC layer
C_m
Requests
S_n
Customers
Arbitrary length messages
Servers (file, directory, clock, account, process, database, etc.)
Distributed customer and server processes

Figure 1 NOS Structure

Objects or resources are entities such as processes, files, directories, virtual I/O devices, databases, etc. Resources can be accessed or manipulated only in terms of well-defined functions or operations. Each type of resource is specified by 1) a logical set of data structures, and 2) a set of operations that can be performed on these data structures. Two resources are of the same _type_ if they have the same specification. The abstract representation of a resource and the operations on the representation are implemented by one or more modules called _servers_.

The implementation details of a resource representation are of concern only to the server. Two different servers of a resource of type, say file, might internally represent the files they manage quite differently, while presenting externally the same representation and operations. This characteristic is important as we want to build the NOS on top of existing operating systems or implement it as the base operating system on many vendors' hardware. The system can be extended by creating new resources, using existing ones as components.

A given process can operate in either or both _server_ and _customer_ roles at different times. A customer process accesses a resource by sending _requests_ containg operation specification and parameters to the appropriate server. The server may then satisfy the request by accessing data structures local to it or by sending additional requests to other servers to aid it in carrying out the original request. When a request is satisfied, the server sends _replies_ containing an indication of success or failure and results (if any).

Requests and replies consist of control and data parts. Besides the customer and server processes being distinct, the handler of replies _may_ be a different process from the requester, or a different address port on the requester than that used to send the request. Further, the sources and sinks for data _may_ be at different locations or addresses from the above as shown in Figure 2. The basic NOS request/reply model supports the following distributed roles for processes communicating by messages.

- Requester - The requester is the customer process desiring some service, such as the copying of information from a source to a sink. The requester controls the data source/sink C.

- Server - The server is the process providing a service in terms of abstract resources. The server controls the data source/sink E.

- Source - There are a variety of possible sources: a file, an input device, the memory of a process, etc.

- Sink - there are a variety of possible sinks: a file, an output device, the memory of a process, etc.

- Reply-handler - The reply-handler is where control information associated with the transfer is to be sent. Normally this would be a port of the requester, but in a distributed system this may not be the case. Replies may be desired at different times: only when the request is completed, or also when the legality of the request and parameters has been verified, or also when some intermediate point in the processing of the request has been reached, etc. A parameter of data movement requests, the reply-option, indicates when replies are to be sent and another the reply-capability, indicates where.

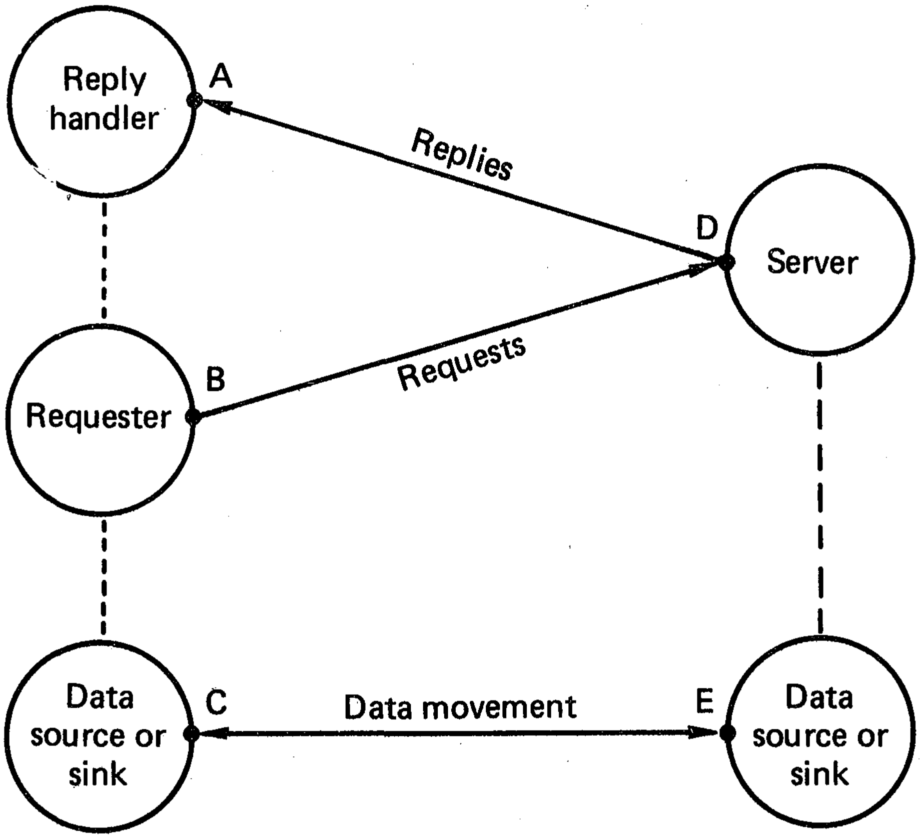


Figure 2 Request/Reply Data Movement Model

Another feature of the data movement model desired is that data not move until source and sink are both ready. Besides normal end-to-end flow control, each end may be unwilling to allocate needed resources until the other end has reached some state of "readiness". For example, a sink may be unwilling to allocate disk space until just before a transfer can take place. If the source is a tape in a vault which must first be fetched and then assigned a tape drive, this could take some time. A readiness negotiation ability and other higher level conventions necessary to support the request/reply, data movement model are presented in Section 4.

The NOS structure above has the following desirable properties needed for a distributed system.

It places no a priori restrictions on which processes can communicate with which others. Knowing a process's address is sufficient to communicate.

It allows all components, including data sources and sinks, to be distributed.

It provides for user extensibility, location independence, and a uniform user view, because communication among all processes use the same mechanism and form, whether local or remote, user or system provided. Logical addressing, as the other aspect of location independence, is discussed later.

It allows a system to participate in the NOS by minimally supporting the basic NOS message passing service defined below.

Interprocess Communication and Syncronization Services

The structure shown in Figure 1 supports an interprocess communication (IPC) service of logical messages (letters). Logical messages, or just messages, can be of arbitrary length. The IPC provides for transmission of beginning-of-message (BOM) and end-of-message (EOM) marks (as part of its "headers") between.source and destination processes. These marks allow the source and destination processes to use different buffer sizes and management strategies, allow messages to be fragmented in transmission, provide data resynchronization after a failure and provide well defined points in the data stream for starting parsing or other operations. Messages are reliably delivered (not lost, misaddressed, missequenced, damaged, or duplicated). Messages are exchanged between network addresses, viewed at this level as ports on processes. (All communicating entities are loosely thought of as processes.) At the interface to the IPC layer there is no concept of establishing connections or virtual circuits, only that of sending and receiving messages, possibly in pieces, bounded by BOM and EOM marks where appropriate.

The information at source and destination transmitted across the interface to the IPC layer in the Send and Receive primitives includes:

Destination and source address,
BOM,EOM marks,
Security level of the message,
Uninterpreted message content.

Wait and Abort are also primitives. Wait is the basic process synchronization mechanism in the system. A process can Wait or not, at its option, for any of its pending Sends or Receives to complete. Servers to support semaphores or other higher level synchronization mechanisms can be constructed on top of this primitive service [25,36]. Abort allows any pending or active Sends or Receives to be cancelled or stopped.

The following subsections discuss general NOS model issues above the IPC layer.

Resource Naming

Two kinds of resource names are needed in a NOS, one convenient for people and one convenient for machines [38]. The latter should have the same form across all resources, be machine-oriented, contain communication level. Rather, each service may need none at all, a quite simple one [19,22] or one that is quite complex [20,39]. This issue is discussed further in reference [47], in the context of a file service. The conventions outlined below, we believe, should allow a range of error recovery strategies to be supported. In particular, within certain assumptions, conventions for crash detection and separation of data and control are provided.

Resource Location or Placement

One of the NOS architecture goals is that user or user processes should not have to be explicitly aware of where a resource is located. For example, in a distributed file system, the location or level of storage that information resides on can be made invisible for many applications, and files can migrate as appropriate. Reasonable file copying/caching and control strategies can also be envisioned as outlined in references [44,47]. (There are many difficult problems associated with updating multiple copies [2].) Global resource placement strategies across different types of servers may enventually prove needed, but the issues here are not well understood [17,37,44]. Our initial assumptions are that each host computer system will multiplex its own local hardware resources, and, that in early versions of the NOS, users can explicitly select where resources reside or execute. Later, when these issues are better understood, automatic allocation or location on a global basis can be added to the framework presented here. The National Software Works represents pioneering work in this area [28,39]. The difficult problems associated with automatic handling of distributed directory structures is discussed in reference [44].

Resource Allocation Limitation and Accounting:

These issues are not necessarily related. Even on single systems, they are confused by organization politics. The desires for autonomy of remote systems under local control, while yet allowing participation within the larger NOS, further complicate these issues. The NSW represents initial work on this problem [28]. What is required here is a clean separation between basic mechanism and policy decision and implementation. The use of account capabilities, in addition to principal capabilities to represent users, and provision of account and authentication servers using the basic capability mechanism outlined above should allow a variety of policies to be supported.

2. NOS PROTOCOL STRUCTURE OVERVIEW

The protocol structure, to support the NOS model above, is built on five principles:

- Layered design - A layered design is used to achieve understanding, ease of evolution, and implementation modularity [10,26,35]. The interfaces between each layer are kept as simple as possible.

- Transaction orientation - Most operating system services are transaction oriented: a customer process issues a request, and the server process replies and no additional conversation need ever take place, and the protocol structure should not require the overhead of additional messages. The structure permits, however, the creation of extended conversations, called sessions, where they are useful. The IPC transaction oriented service that we wish to support is different than that of conventional datagrams [34] in that we want a reliable service, which is not usually guaranteed for datagrams, and we want the messages to be of arbitrary length, also not usually supported for datagrams.

- Symmetry - Processes can operate both in customer and server roles during a conversation. The protocol structure must allow for this shift in roles [49].

- Abstraction of commonality - Common aspects of servers and resources, such as their logical structure, naming, protection, and the common operations applicable to them should be abstracted and standardized.

- Provision of a complete set of primitive services - It must include those services necessary in single host systems to form a complete set of building blocks. The primitive operations must facilitate, but not demand, their distribution.

For the purposes of this paper a protocol is loosely defined as any agreed set of conventions associated with the exchange of information by peer entities during communication. Definitions of data and message formats are included, as well as rules for control and data interchanges to achieve some defined service.

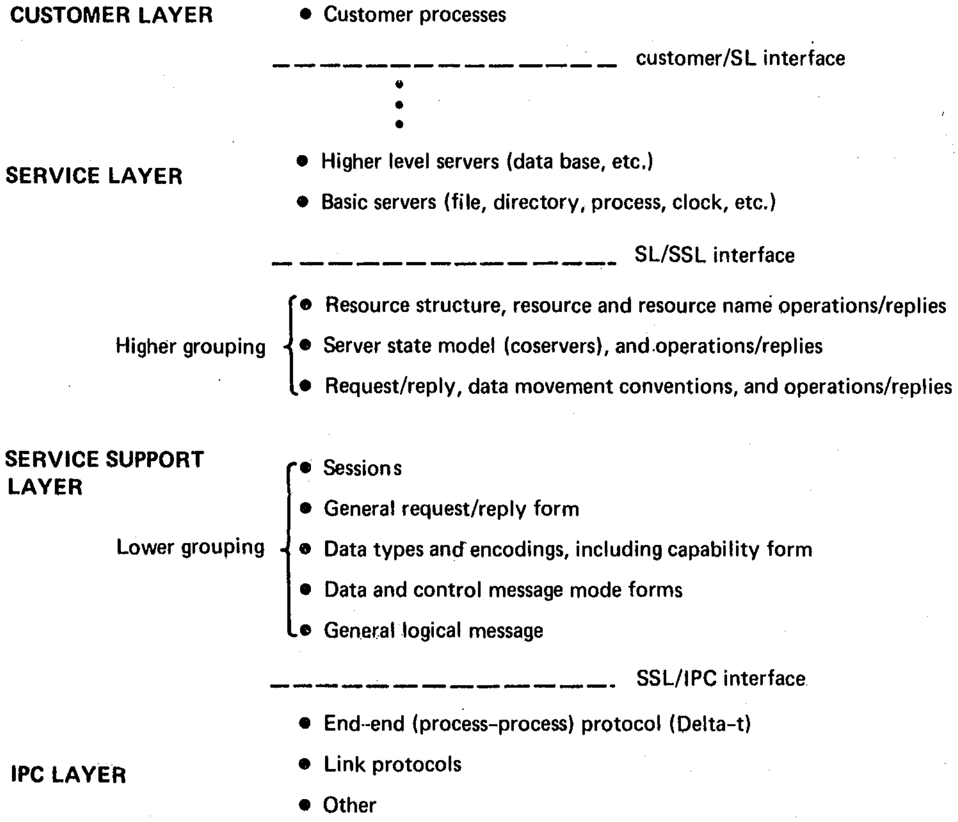
The protocol hierarchy, described bottom up, consists of three layers:

- Interprocess communication layer supplying the IPC service mentioned earlier.

- Service Support Layer - Defines standard server and resource logical structures, resource naming, protection, data formats, request/reply functions and form, and sessions.

- Service Layer - supports basic resources and services, authentication, logging, files, directories, processes, clocks, accounting, terminals, etc.

The structure is depicted in Figure 3. Each of the following sections discusses a layer.



CUSTOMER LAYER

- Customer processes

customer/SL interface

SERVICE LAYER

- Higher level servers (data base, etc.)
- Basic servers (file, directory, process, clock, etc.)

SL/SSL interface

SERVICE SUPPORT LAYER

Higher grouping

- Resource structure, resource and resource name operations/replies
- Server state model (coservers), and operations/replies
- Request/reply, data movement conventions, and operations/replies

Lower grouping

- Sessions
- General request/reply form
- Data types and encodings, including capability form
- Data and control message mode forms
- General logical message

SSL/IPC interface

IPC LAYER

- End-end (process-process) protocol (Delta-t)
- Link protocols
- Other

Figure 3 Structure of Protocols and Conventions

3. INTERPROCESS COMMUNICATION LAYER

Introduction

The basic interprocess communication service was defined in Section 1. To support this service in a distributed environment requires a layered set of protocols. The main protocol relevant here, which rests on link-level protocols, is a transport or end-end protocol providing, addressing from a source (origin) to a sink (destination) process, delivery assurance (information is not lost, damaged, duplicated, missequenced, or misdelivered), and flow control to the sink's rate of acceptance. A (source, sink) address pair is called an association. This layer supports transport of uninterpreted arbitrary length logical messages delimited by BOM and EOM marks. The BOM, EOM marks are carried out-of-band in packet headers. The two major areas of addressing and assurance are now discussed briefly, as these are where our approach may differ somewhat from that used in other transport protocols.

Addressing

Each source and sink is identified by a unique hierarchical address that routing modules parse from left to right. The further away the sink is, in terms of the chosen hierarchy, the sooner the parse is stopped. That is, the address of a process reflects the hierarchical geometry of the network (network, cluster, host, process etc.), which means that every node need not store information about every potential sink individully. Each branch down the tree could contain a different number of levels and a different fan-out at each level. The network address space is large enough that every process can have several addresses (allowing it to have ports), and none of the addresses has to be reused, even after the process is destroyed (assuming reasonable lifetime for the network). This feature is important as one element in achieving the transaction orientation of the architecture, because a process does not first have to go to a well known logger or connection establishment port, present a higher level name, and then be allocated a logical channel, socket or other reusable network address before entering a data transfer phase.

Within this framework, we also provide for logical, generic, or functional addressing [27,31,44]. A portion of the network-address space, characterized by a standard value for the leftmost bits, is set aside for this purpose. The routing tables in each node of the network then point to the nearest "representative" of a generic service. Communication and, if necessary, synchronization among the representatives of a generic service uses non-generic (physical) addresses and is a higher-level problem.

Logical addressing can also be handled at higher levels, with translation from logical to physical address taking place above the end-end protocol level at the source. In some cases, the network address of a service appearing in a capability may actually be to a higher level logical-address server whose only job is to forward messages to the actual server, which it locates by means of records ("yellow pages") that it maintains. Maintaining the distributed or centralized logical to physical address maps when processes move will require forwarding protocols, not currently defined. We expect logical addressing to be important in certain NOS servicees [47]. It has already proved useful within the NSW [28,31].

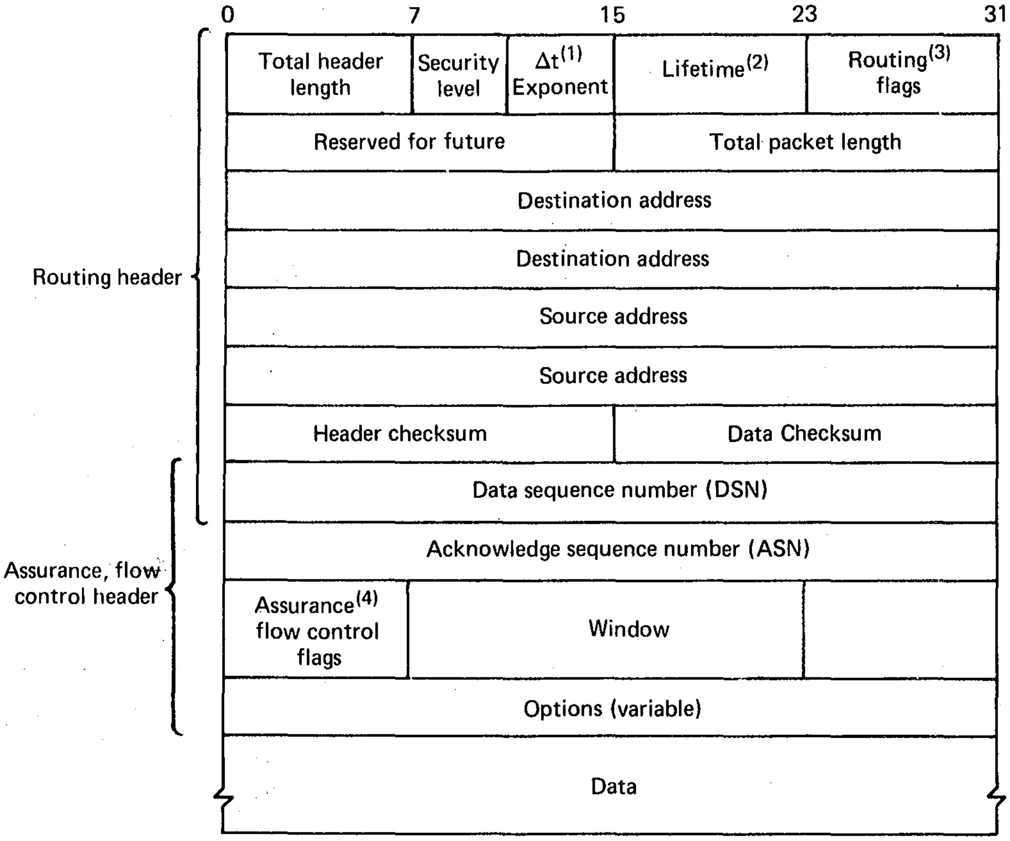
Assurance and Flow Control

To achieve assurance and flow control, while maintaining a transaction or single message orientation, we have had to design our own process-to-process (end-to-end) protocol. Published work on message systems have generally not dealt with reliability [1,12,31,45]. Well-known existing and proposed protocols such as INWG, and TCP [4,32,33], and X.25 all require overhead messages to be sent between source and sink in order to reliably set up and tear down a connection or virtual circuit, even if only one request and one reply are to be exchanged [18,42]. In some network architectures this overhead has to be borne at each of several levels. This overhead, we believe, is unacceptable, not primarily because of the raw bandwidth consumed, but because of the cost and delay involved in generating messages, forming them into packets, placing them onto the transmission media, and buffering and handling them along the way. Our experience and that of others [50] show that general OS overhead for packet handling may require several times the time required for actual protocol processing.

It has been demonstrated by Belsnes [3] that for reliable single message transmission, the reliable connection set up overhead is unavoidable unless the state information kept by the two ends of a conversation is under timer control. Accordingly our protocol depends on the use of timers and is called Delta-t. Delta-t is based on the fact that the total time of existence of a packet, including the interval between its first and last transmission, its maximum lifetime within the routing network, and the delay before it is acknowledged by the sink, can be bounded. This bound is expressed in terms of an interval Δt, hence the name of the protocol.

Briefly, Delta-t works as follows. The state information used for generating sequence numbers at the source, packet acceptance at the sink, acknowledgement, and flow control (normal window flow control) are kept in connection records at each end, as for any non-timer protocol. These records have a lifetime under control of a Send-timer at the source, and a Receive-timer at the sink. When either of these timers go to zero, the corresponding record can be destroyed. These timers do not have to be synchronized, but are expected to run at the same rate. When initialized or refreshed, these timers are set to multiples of Δt. The rules for timer intervals, control of the timers, setting of header control flags, sequence number selection, and packet acceptance are given in references [16,48]. The protocol header for the Delta-t protocol is shown in Figure 4.

Simplified, the lifetime of a packet is strictly controlled by including a field in the packet header that is initialized by the source and counted down by intermediate nodes. Each node, including the end protocol module, must count at least once, more if it holds the packet longer than one time unit (tick). Retransmissions start partly counted down. The packet is discarded and nacked if the count reaches zero before delivery. Our link protocols, on links that have internal buffering that could hold a packet for an indefinite period of time, have been augmented by a feature that guarantees knowledge of the transit time. An entire network could be such a logical link. The idea is that each link frame is time-stamped by the sender so that the transit time can be computed by the receiver. The two logical link clocks, send and receive, are simply synchronized whenever necessary by the receiver sending its clock value to the sender. This mechanism assures that the transit times are always overestimated, never underestimated; the details are presented in references [40,48]. Routing nodes also destroy all packets on recovery from a crash.



Lengths and window are in octets. Sequence numbers are for octets.

(1) Δt-exponent allows the receiver to calculate senders Δt. $\Delta t = K \times 2^{\Delta t\text{-exponent}}$

(2) Lifetime equals number of "ticks" remaining for oldest data octet in this packet. Tick = $\frac{\Delta t}{256}$ secs

(3) Routing flags: F/R fragmentation allowed, BOM, EOM, NAK

(4) Assurance flow control flags: DRF (all previous DSN's acted), ARF (ASN field valid), WOF (window overflow)

Figure 4 Delta-t Protocol Header

4. SERVICE SUPPORT LAYER

Introduction

The service support layer (SSL) defines a hierarchy of conventions consisting of two main groupings shown in Figure 3. The first of these (lower grouping) contains conventions to establish data and control message separation, provide for data and control parameter translation, establish the general syntax of request/reply messages, and provide crash detection. The second of these (higher grouping) supports the request/reply, data movement model of Figure 2, and abstracts common server and resource structures and requests/reply semantics.

The purposes of the service support layer are the following:

- Each new service should not have to be designed from scratch, dealing with the above issues anew. This facilitates the introduction of a new service. A run-time environment can be created embodying the common service support features in terms of library routines, utility processes, or other building block mechanisms [49].

- A uniform user view is created that eases the learning time and other difficulties of a customer trying to use a new service. As seen by processes, an operating system or protocol interface is a language and should meet good language design criteria such as uniformity and compactness.

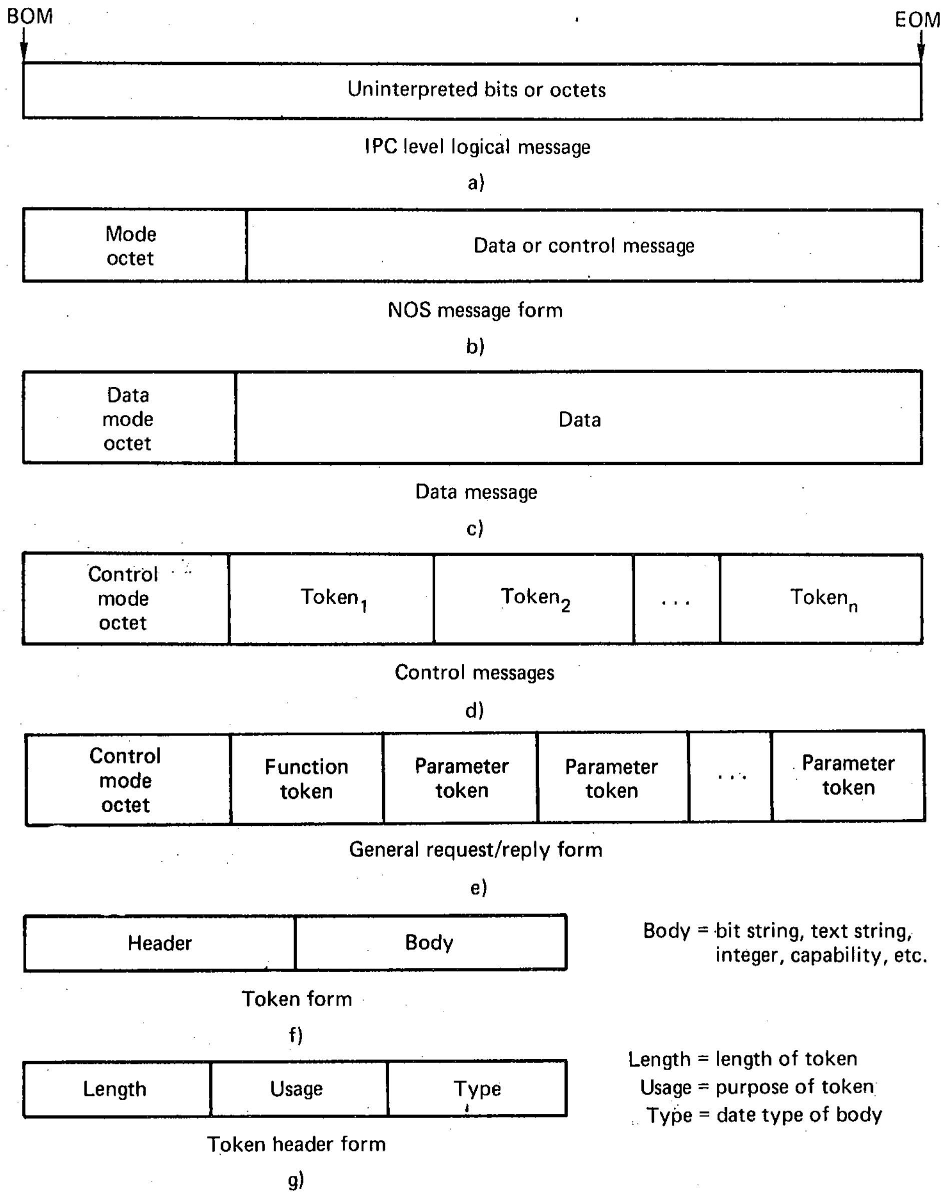
We now present the SSL issues and conventions in the order shown in Figure 3.

Separation of Data and Control

The lowest convention of the SSL identifies each message as being in one of at least two modes: control or data. Control messages, in general, are requests or replies in a standard encoding that contain the semantics of the customer-server dialog. Data messages are, in effect, parameters that are too large to be conveniently or efficiently enclosed within a control message; an obvious example is the contents of a file being transmitted. We want to be able to support data or control messages on the same or different associations. For example, in Figure 2, association (C,E) may be as shown, or in fact be the same as associations (B,D) or (A,D).

Knowing the mode enables a process to quickly and unambiguously separate what it must interpret from what it must simply store, print or pass on. It greatly reduces the danger that after a loss of state information (e.g., at deadstart) it will treat raw data as a command. It permits control information, such as a statement of an error condition, multistream synchronization mark, or checkpoint number to occur in contexts where data is expected, without causing confusion or data scanning. Finally, it cleanly separates control translation issues from data translation issues. Control needs to adhere to a standard format so that all processes may understand one another, while it is often desirable that data be shipped in its raw form or be translated in an application-dependent way, the latter being a higher-level issue.

The mode indication could be either an "out-of-band" signal in the IPC protocol heading (as, for example, the qualifier bit of X.25) or it could be the first few bits of the message. We have chosen to make it the first byte (the mode byte) of the message so that the SSL can we used with transport protocols other than Delta-t. The SSL hierarchical structuring of logical messages is shown in Figure 5.



IPC level logical message

a)

NOS message form

b)

Data message

c)

Control messages

d)

General request/reply form

e)

Token form

f)

Body = bit string, text string, integer, capability, etc.

Token header form

g)

Length = length of token
Usage = purpose of token
Type = date type of body

Figure 5 Logical Message Structure

Translation and Control Mode Message Structure

In order for translation to be performed, explicit or implicit data type information must exist. For the parameters in requests and replys, explicit typing of each parameter is provided as described below. It would be inefficient for each data item in a data message to be typed. For data messages, the data type information can be known in three ways: implicitly by the nature of the service or address, conveyed in control messages, or explicitly encoded in the message mode code.

Let us now consider the structure of control messges. Desirable goals are to allow parameters in functions to be omitted and defaulted (achieving data compression), appear in any order (allowing services to evolve by adding new parameters to functions), to be variable length, and to be automatically translated to and from a servers internal representation from and to standard network encodings. To achieve these goals control messages are considered a string of tokens. A token consists of two main parts: a header followed by a body. The header describes the body, while the body represents the actual value conveyed by the token. The header in turn consists of three parts:

. The length defines the number of bytes included in the entire token.

. The usage defines the purpose of the token, such as function code, source (resource) identifier, source label (first-bit-address, for example), count, etc.

. The type indicates the data type of the body, such as integer, bit string, character string, capability, etc.

Type has been separated from usage because there are examples of usages that may be of various types; it permits a simpler common translator to be designed for each programming language or system that translates tokens to and from their internal representations to and from the standard. The translator's decisions are based only on the type. The token encoding that we are developing is expected to make the most commonly-occurring, token headers only one byte long, with an escape to two bytes for most of the remaining cases.

The tokens of a message are grouped into statements. Each statement begins with a token of usage "function code" and ends just before the next token of that usage or at the end of message. (The first token of a message should be of usage "function code.") The tokens of a statement following the function code are parameter tokens. The function code token defines an operation to be performed or indicates a reply; the parameter tokens supply arguments or results. Allowing (not demanding) multiple statements per message helps reduce message traffic.

Each function, in general, expects parameters of several different kinds of usage. If a needed usage is omitted, then a default value is assumed; for example, the default for usage "count" is one. The concept of usage thus permits a form of information compression. More importantly, it permits new options to be added to a function, expressed by new parameters, without impacting existing customers.

When a parameter of a function actully occurs in a preceding or following data message (possibly on a separate association), a parameter is placed in the control message, in effect an "indirecting pointer," indicating this fact.

Standard Data Types, Capability Form

Among the standard data types defined by the SSL is a standard capability as shown in Figure 6. Human-oriented names are handled at a higher level by naming graphs as mentioned earlier. A standard capability is a token body that identifies a resource and confers right of access to a particular resource. It consists of the following fields.

. The *address* is the network address (logical or physical) of the server that manages the resource. This would be D of Figure 2. Often the customer uses the address of one of the capabilities in a request message to determine where to send the request.

. The *properties* are a set of standard bits and fields that indicate to the customer the nature of the capability, such as controlled/uncontrolled, resource type, access mode, resource lifetime, security level, etc.

. The *unique identifier* is used by the server to identify and locate the specific resource named, and possibly for other server-dependent purposes.

. The *password*, if present, guards the unique identifier part of the capability against forgery. The idea is that, if any process or user tried to forge a capability, it would not be accepted by the server unless the password were correct. Encryption can also be used for this purpose [5,29].

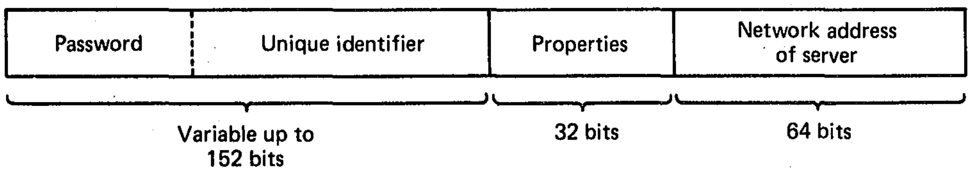


Figure 6 Standard Capability Form

Other standard data types to be supported include at least integer, bit string, and character string.

Sessions

All servers maintain state information for an association for some length of time, depending on the nature of the server: during a single operation, multiple operations (statements) in a single logical message, or across multiple messages. In addition some servers may want to support parallelism such as parallel operations, parallel streams, etc. on a single association. While active state information is being maintained for an association a session is in existence.

The IPC layer end-end protocol with its BOM, EOM marks allows customer or server processes to detect crashes with loss of memory during a logical message, but cannot aid detection of crashes between messages, although its rules protect against lost, damaged, duplicate, missequenced packets across crashes. Detection is achieved because after deadstart the server expects a BOM, and a customer generates a BOM when the server is expecting a EOM first. If state information is being maintained across messages, tied to an association, there is a need to provide a mechanism for customer or server crash detection between messages. This is the purpose of a session. Sessions are delimited explicitly or implicitly with beginning-of-session (BOS) and end-of-session (EOS) function tokens, depending on whether or not a service supports multiple message sessions. This allows crash detection, as now described.

When a server crashes, it deadstarts with inactive sessions logically on all associations, which expect a BOS as the first token received. If the customer thought a session was in progress it will not include a BOS in the message sent the server, and the server will generate an error reply, forcing the customer to enter an error recovery procedure. Similarly when a customer process deadstarts it sends a BOS as part of its recovery procedure and will be informed by the server if a session was in progress. Then the cutomer can take whatever recovery action is appropriate.

Request/Reply, Data Movement Model support

To support the request/reply, data movement model of Figure 2, conventions are required so that all communicating entities can know each others address, authenticate the right of a partner to send them a message, and detect a partner crash. These needs are met as follows:

- The requester obtains the server address D from a capability or some a priori way. The requester can detect a server crash by the session mechanism above.

- The server obtains the address A of the reply-handler from a capability passed as a parameter in the request. This address defaults to that of the requester B, always provided by the IPC layer interface. The server may require a capability passed as a parameter in the request to authenticate the requester's right to make the request. Address B can be used by the server to protect controlled capabilities as mentioned in Section 1.

- The reply-handler and requester are working together and the reply-handler can be sent the address D of the server if needed. The reply-handler receives the reply-handler capability in replies from the server, authenticating the server as the process with the right to send it messages. Failure to receive a reply could result from a server or reply-handler crash. A duplicate request could then be sent with an attendent risk of a duplicate operation not detectable by lower level IPC layer mechanism. Therefore, duplicate requests are not recommended unless requests are formulated such that duplicates can not cause harm. Instead, when a reply fails to arrive within some timeout period or the reply-handler detects it has crashed, the state of the appropriate coserver-state-record (see next subsection) can be interrogated for status to determine whether or not to reissue a request.

- For servers supporting or requiring data movement in data mode messages (normally only those involved in bulk data movement such as the file server), a mechanism is required to exchange data source/sink addresses C and E. Because these addresses cannot, in general, be known ahead of time, a simple "open" protocol is required to be used before data movement can begin. The requester sends the server an appropriate resource capability (such as to a file), address C, and other parameters to intialize state. Address C could also be provided in data movement primitives also so that several cooperating customer processes could serve as sources or sinks at different times. The server returns a capability to the "open-resource" with E in its address field. Addresses E and C will only accept messages from each other. Note that therefore operations involving bulk data movement are not of the single request/reply form. This seems acceptable because the "open" exchange is small overhead relative to the expected large data movement. Information in control messages can be sent on the association (C,E) for checkpoint restart or higher-level checksums if desired for error detection and recovery mechanisms. A crash at either end would be detected by the IPC layer failing to get a message through or failure to receive the expected amount (count) of data.

Data is actually moved with standard "read" or "write" operations defined for sequential and random open-resources. These operations are sent to address D. (We are considering whether or not to extend the model to allow a different control address for the read and write operations so that the module serving actual data movement could be distributed without indirection through address D.) Besides the normal parameters for reads and writes (open-resource capability, first element address, count etc.), there is an additional parameter for "readiness" negotiation. Normally the customer process is ready and so no negotiation takes place. If the customer desires to begin a readiness negotiation it sends a read or write with the readiness parameter indicating its current state of readiness. The server sends a reply indicating its readiness when it reaches a state "more" ready than the requester. This cycle continues until the customer sends a request indicating fully ready.

There is also a standard "copy" operation explicitly specifying two resources as source and sink to be used for "third party" data movement requests. This allows transfers directly from one file to another or special servers to support copying from one arbitrary resource to another without having to involve the original requester. The "copy" server would issue "opens" and successive reads or writes to the source and sink, or if the source or sink supported "copy", then a "copy" could be forwarded to one of them, which would in turn "open" the other, and then perform the reads or writes.

Coservers

As part of the goal of providing users (customer processes) as uniform a view of servers as possible, the coserver concept has been developed. The idea of a coserver is quite parallel with the conventional idea of a process and is motivated by the desire to:

- Support server state information across many types of servers in a consistent manner.
- Support state information across messages for data compression.
- Allow state information after one operation to be defaulted as input parameters for succeeding operations.
- Share state information across two or more associations.
- Be able to operate on state records even when an association is blocked by lower level flow control.
- Support parallel services on a single association.
- Be able to interrogate the state of an operation while it is in progress from the same or a different association.
- Be able to distinguish and specify when and where replies for an operation are to be sent and from which parallel entity the reply is coming from.
- Be able to abort, suspend, restart an operation.
- Provide for the above services in general, but only require a minimal implementation when, as is expected to be common, a server only supports sequential operations, and does not require state to be saved across messages.

The coserver mechanism or protocol briefly is the following. For a given association, a server may in some cases be viewed by the customer process as logically providing independent parallel servers. It seems useful to make this notion explicit and to talk about server processes that multiplex themselves to run abstract servers called coservers (which are like coroutines), each represented by a coserver-state-record (CSR). The CSR consists of two parts, a set of parameter-registers (PR) readable and writable, and a set of execution-state-registers (ESR), read only. The way a coserver is viewed as working is as follows:

- It receives its operation stream from logical messages. The operations allowed are any accepted by a server and permitted by capability access rights.
- The parameters in the message are loaded into the PRs named by usage. When end-of-message (EOM) or the next operation token is

reached, execution begins, the parameters in the PRs needed by the operation are used. Parameters are thus defaulted from values in these registers if they are not included in the message.

- As the operation proceeds its state evolves and is recorded in the ESR as advertised by the server.

- Replies are sent when specified in the <u>reply-option</u> parameter.

- Replies are sent to the <u>reply-handler</u> represented in the capability in the reply-handler usage parameter.

- A given coserver is sequential, that is, it can perform only one operation at a time.

- At any given time one and only one CSR is selected as attached to an association. The CSR is in one of two states <u>active</u> or <u>inactive</u>. A session is in progress if the selected CSR is active.

- At time 0 on an association a default CSR, containing initial default values for the PRs, is logically tied to it and is marked inactive. The only acceptable operation on an inactive CSR is the BOS token. Any other token will cause an error return. The BOS operation makes the CSR active and now any advertised server operation is acceptable. There is a corresponding EOS token which detaches the current CSR from the association (but does not destroy it--therefore it can continue executing its current operation) and attaches an inactive default CSR to the association.

- A BOS will not be accepted on an active CSR and an error message will be returned.

- All coservers are named by capabilities either explicitly returned on CSR creation or left in the CSR for return if interrogated.

- If two or more associations are sharing a CSR (which is permitted), then they are assumed to be synchronizing themselves at a higher level.

- There are a set of conventions for dealing with replies from detached coservers. Requirements are to provide options that would either 1) send such replies into a "block hole" not requiring the coserver to block, 2) require the coserver to block, if a reply is generated, until it is reattached to an association, 3) allow all replys but the last to enter the black hole, but allow the last reply to be obtained by an interrogation.

- Coservers can also be explicitly created, destroyed, interrogated, reattached, suspended restarted, aborted by a standard set of operations.

- Conventions are required to allow a command affecting the CSR currently attached to an association not to be blocked by flow control on that association or the fact that the attached coserver is executing a normal resource operation. A number of mechanisms to meet this need are under consideration.

Uniform Resource View

We want to provide a uniform and compact language for manipulating resources. This requires a uniform view of resource structure. The following uniform resource model is under consideration.

A typical resource can be viewed as a data structure (possibly distributed) consisting of two major parts:

. The heading or resource state record contains named fixed fields of information of varying length and type, such as creation time, last access time, account capability, security level, access rights, identity verification, mnemonics or other commentary, etc.

. The body is the resource proper. Its structure varies depending on the nature of a resource.

For example, a file could be an array of bits or records labelled by consecutive natural integers, while a directory is a list of capabilities labelled by character strings. For some resources, such as most printers, only one item of the body is accessible at a time, and a label is not needed. We believe all possibilities can be treated as special cases of one or a few general forms. A resource usually is named by a token of type capability, while the items in its body are labelled by tokens of various types.

Only a few functions are required to cover the vast bulk of operations performed on resources. All operations involving querying or modifying coserver state records and resource headings, or reading or writing resource bodies are actually special cases of generic read and write functions. Functions are needed to "create" and "destroy" entire resources and to "enter" and "delete" items of a resource (as, for example, in a directory where the items are neither fixed in number nor strictly consecutive). Another group of functions is needed for validating or invalidating controlled capabilities and creating a new capability with different access privileges. Some important functions apply to only certain kinds of resources; active resources, such as coservers or processes, need to be "started," and "stopped," while synchronizers, such as semaphores, have their own specialized operations. Standard operations for coserver state record handling were mentioned earlier.

Specifying an essentially complete small set of functions seems a language goal well worth pursuing, provided that we exclude servers that perform primarily a processing function, such as editors, compilers, and applications in general, although we would expect server designers to use the standard operations where appropriate.

One more standard function needs mention, it is the one that usually appears as the only function in a reply. The parameters following it define the results or the status to be conveyed including, if appropriate, residual count and address. The most important of these parameters is one that indicates either no error or the nature of an error, such as invalid capability, access denied, improper label, insufficient funds, inadequate security level, excessive count,, server fault, resource destroyed, etc.

Other aspects of a uniform resource model that need specification include the following: (1) standard access rights, as indicated in the properties field of a capability or the heading of a resource, such as read, write, execute; (2) standard token usages that categorize the parameters of functions; and (3) standard token types.

5. SERVICE LEVEL

The service level (SL) defines standard kinds of servers, the structure of the resources they manage, and those formats and protocols that do not seem widely applicable to many servers. Examples of issues we believe to be server dependent are error handling and recovery; optimal resource location or placement strategies and protocols, such as automatic file caching; and internal server structure, centralized or distributed [17,44,47].

The main goal of the service level is to try to assure a complete set of basic standard servers is defined, and that, for example, all servers of a given resource type are compatible with one another and present the same external appearance no matter where in the network they are located or from where they are accessed. A discussion of issues and our current plans associated with a standard file server is contained in reference [47].

We are initially planning the following standard servers: file, directory, process, terminal, authentication, clock, account, synchronization. Most of the operations for these servers will be the standard ones mentioned in the last section.

6. CONCLUSION

We have outlined our goals for a NOS, a NOS model, and a protocol structure to support this model. Our curent status is that the transport level of the protocol structure is designed [48]; the message format sublayers of the service support layer are complete except for minor details; the coserver and data movement models are still being refined; and we are beginning specification of the standard servers.

We believe strongly that an integrated approach to NOS and protocol design is required if true resource sharing, multiprocessing, and distributed computing are to evolve. We have further argued that protocol structures must be built on a message or transaction base. We have shown the main elements required to provide the transaction base, adequate address space so that addresses do not have to be reused and can be permanently assigned, timer based IPC layer assurance mechanism, explicit data typing, capability based naming, and a request/reply dialog structure. On top of this, single or multiple message sessions can be built.

The elements of a uniform customer/server model were presented; which included a distributed request/reply data movement model, server state model (coservers), and resource model. Using such an approach should; 1) provide a firm basis for distributed application or service design, and 2) allow a simpler, more consistent, easier to learn operating system language, which we believe will be important for a extensible NOS with many services. The ideas presented here also seem useful for development of portable as well as distributed operating systems. Increased integration of protocol, OS, and language design concepts should be encouraged.

We do not believe that a NOS must spring fully grown into existence. Even if ones initial need is for a single service such as virtual terminal service or file transfer, if protocols for providing these services are designed on the type of structure outlined in this paper, then a foundation will exist for smooth evolution toward a fuller NOS as additional services are required.

There is a large amount of work yet to be done to fully specify the protocols outlined above, create implementations both as a base OS and layered on existing OS's, and write new distributed applications and servers. Only when these tasks are completed will we believe we really have a handle on all the NOS issues.

Acknowledgements

We wish to acknowledge the many valuable past and continuing discussions with Garret Boer, Sam Coleman, Jed Donnelley, Bob Judd, Dan Nessett, Lansing Sloan, Bing Young, and Mary Zosel. We particularly wish to acknowledge Jed Donnelley's central role in the evolution of the NOS model. The work reported here was supported by the U.S. Department of Energy under the contract number W-7405-ENG-38

References

1. E. Akkoyunlu, A. Bernstein, and R. Schantz, "Interprocess Communication Facilities for Network Operating Systems," Computer 7, 6, 1974.

2. P.A. Bernstein, N. Goodman, "Approaches to Concurrency Control in Distributed Data Base Systems," AFIPS Conference Proceedings, Vol. 48, 1979 NCC, pp.813-820.

3. D. Belsnes, "Single-Message Communication," IEEE Transactions on Communications COM-24, No. 2 (1976).

4. V. Cerf, A. McKenzie, R. Scantleburg, H. Zimmerman, "Proposal for an Internetwork End-to-End Transport Protocol," INWG 96.1, also in Proceedings Computer Network Protocols, Liege, February 1978.

5. P. L. Chaum, R. S. Fabry, "Implementing Capability-Based-Protection Using Encryption," University of California, Berkeley, Electronics Research Laboratory, Memorandum UCB/ERL M78/46 July 17, 1978.

6. S. D. Crocker et al., "Function Oriented Protocols for the ARPA Computer Network, AFIPS-SJCC, Vol. 40, May 1972, pp. 271-279.

7. J. Davidson, N. Mimno, R. Thomas, D. Walden, W. Hathaway, and J. Postel, "The Arpanet Telnet Protocol: Its Purpose, Principles, Implementation and Impact on Host Operating System Design," Proceedings Fifth Data Communications Symposium, Snowbird, Utah, September 1977, pop. 4-10-3-18.

8. P. J. Denning, "Fault Tolerant Operating Systems," ACM Computing Surveys, Vol. 8, No. 4, Dec. 1976, pp. 361-386.

9. J. B. Dennis, E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," CACM Vol. 9, No. 3, March 1966, pp. 143-155.

10. R. desJardins, G. White, "ANSI Reference Model for Distributed Systems," IEEE CompCon 78, Fall September 1978, pp. 144-149.

11. J. E. Donnelley, "A Distributed Capability Computing System," Proceedings Third International Conference on Computer Communication, August 1976.

12. J. A. Feldman, J. R. Low, P. D. Rovner, "Programming Distributed Systems," Proceedings ACM 1978 Annual Conference, December 1978, pp. 310-316.

13. J. G. Fletcher, "The Octopus Computer Network," Datamation, Vol. 19, No. 4, April 1973, pp. 58-63.

14. J. G. Fletcher, et al., "Computer Storage Structure and Utilization at a Large Scientific Laboratory," Proceedings of the IEEE, Vol. 63, No. 8, August 1975, pp. 1104-1113.

15. J. G. Fletcher, "Software Protection of Information Networks," Infotec State-of-the-Art Report, Future Networks, Vol. 2, 1978, pp. 149-164.

16. J. G. Fletcher, R. W. Watson, "Mechanisms for a Reliable Timer-Based Protocol," Computer Networks, Vol. 2, No. 4-5, September/October, 1978, pp. 271-290. Also in Proceedings Computer Network Protocols Symposium, Liege, Belgium, February 1978, p. C5-1/C5-17.

17. H. C. Forsdick, R. E. Schantz, R. H. Thomas, "Operatng Systems for Computer Networks," Computer, Vol. 11, No. 1, January 1978, pp. 48-59.

18. L. Garlick, R. Rom, and J. Postel, "Reliable Host to Host Protocols: Problems and Techniques," Proceedings 5th Data Communications Symposium, IEEE/ACM, September 1977.

19. M. Glen, "A File Transfer Protocol," Proceedings Computer Network Protocols Symposium, Liege, Belgium, February 1978, pp. (D5-1)-(D5-7).

20. J. V. Gray, "Notes on Data Base Operating Systems," in Operating Systems an Advanced Course, Springer Verlag, Berlin, N.Y., 1978, pp. 393-381.

21. J. F. Haverty, R. O. Rettberg, "Inter-process Communication for a Server in UNIX, "Proceedings CompCon 78, September 1978, pp. 312-315.

22. High Level Protocol Group, "A Network Independent File Transfer Protocol," INWG86, December 1977. Available through Computer Aided Design Centre, Cambridge, England.

23. A. K. Jones, "The Object Model: A Conceptual Tool for Structuring Software," in Operating Systems and Advanced Course, Springer-Verlag, Berlin/N.Y., 1978, pp. 7-16.

24. S. R. Kimbleton, R. L. Mandel, "A Perspective on Network Operating Systems," AFIPS-NCC, Vol. 45, 1976, pp. 551-559.

25. K. Legally, "Synchronization in a Layered System," In Operating Systems an Advanced Course, Springer-Verlag, Berlin/Heidelberg/N.Y. 1978, pp. 252-278.

26. J. M. McQuillan, V. G. Cerf, Tutorial: A Practical View of Computer Communications Protocols, IEEE Catalog No. EHO 137-0, 1978.

27. J. M. McQuillan, "Enhanced Message Addressing Capabilities for Computer Networks," Proceedings IEEE, Vol. 66, No. 11, November 1978, pp. 1517-1526.

28. R. E. Millstein, "The National Software Works: A Distributed Processing System," Proceeding ACM Annual Conference, 1977, pp. 44-52.

29. R. M. Needham, "Adding Capabilities Access to Conventional File Services," ACM Operating Systems Review, Vol. 13, No. 1, January 1979, pp. 3-3.

30. R. M. Needham, M. D. Schroeder, "Using Encryption for Authentication in Large Networking Computers," CACM, Vol. 21, No. 12, December 1978, pp. 993-998.

31. NSW Protocol Committee, "MSG: The Interprocess Communication Facility for the National Software Works," BBN Report No. 3483; also available as Massachusetts Computer Associates Document No. CADD-7612-2411, December 1976.

32. J. B. Postel, "Internetwork Protocol Specification," Version 4, February 1979, IEN 80, Available through Defense Advanced Research Projects Agency, IPTO, Arlington, VA.

33. J. B. Postel, "Specification of Internetwork Transmission Central Protocol," TCP Version 4, February 1979, IEN 81, Available through Defense Advanced Research Projects Agency, IPTO, Arlington, VA.

34. L. Pouzin, "Virtual Circuits vs. Datagrams - Technical and Political Problems," AFIPS NCC, June 1976, p. 483.

35. L. Pouzin and H. Zimmermann, "A Tutorial on Protocols," Proceedings IEEE, Vol. 66, No. 11, November 1978, pp. 1346-1370.

36. D. P. Reed, R. K. Kanodia, "Snychronization with Eventcounts and Sequencers," CACM Vol. 22, No. 2, Feb. 1979, pp. 115-123.

37. J. H. Saltzer, "Research Problems of Decentralized Systems with Largely Autonomous Nodes," ACM Operating Systems Review, Vol. 12, No. 1, January 1978, pp. 43-52. Also in Operating Systems an Advanced Course, Springer-Verlag, Berlin N.Y., 1978, pp. 583-591.

38. J. H. Saltzer, "Naming and Binding of Objects," in Operating Systems an Advanced Course, Springer-Verlag, Berlin N. Y. 1978, pp.99-208.

39. R. M. Shapiro, R. E. Millstein, "NSW Reliability Plan," Mass. Computer Assoc. CA-7701-1411, June 1977.

40. L. J. Sloan, "Limiting the Lifetime of Packets in Computer Network," Prepared for 4th Conference on Local Computer Networks, Minneapolis, October 1979. LLL Report UCRL 82825.

41. R. F. Sproull, D. Cohen, "High Level Protocols, "Proceedings IEEE, Vol. 66, No. 11, November 1978, pp. 1371-1385.

42. C. A. Sunshine and Y. K. Dalal, "Connection Management in Transport Protocols, "Computer Networks, Vol. 2, No. 6, 1978, pp.454-373.

43. R. H. Thomas, "A Resource Sharing Executive for the Arpanet," AFIPS Conference Proceedings, Vol. 42, 1973, SJCC, pp. 155-163.

44. R. H. Thomas, R. E. Schantz, H. C. Forsdick, "Network Operating

Systems," Rome Air Development Center Technical Report TR-78-117, March 1978, also Bolt Beranek and Newman Report 3796.

45. D. Walden, "A System for Interprocess Communication in a Resource Sharing Network," CACM , March 15, 1972, pp. 221-330.

46. R. W. Watson, "The LLL Octopus Network: Some Lessons and Future Directions," Proceedings Third USA-Japan Computer Conference, San Francisco, October 1978.

47. R. W. Watson, "Network Architecture Design Issues: With Application To Backend Storage Networks," To appear in IEEE Computer late 1979.

48. R. W. Watson, "Delta-t Protocol Specification," In preparation.

49. J. E. White, "A High-Level Framework for Network-Based Resource Sharing," AFIPS Conference Proceedings, Vol. 45, 1976, pp. 561-570.

50. M. A. Wingfield, Unpublished experience with TCP implementation, June 1979.

The ADAPT Data Translation System and Applications

Maurice J. Bach

Nancy H. Goguen

Michael M. Kaplan

Bell Laboratories

ABSTRACT

The ADAPT (A DAta Parsing and Transformation) system provides an efficient generalized language driven approach towards data translation. Its high-level languages are easily learned and understood. The data descriptions and transformations can be easily modified as the conversion requirements evolve. It provides transformations on an inter-record level as well as the power of standard text editors for intra-record transformations.

ADAPT has other uses besides that of a one-time data translation. Since the process of data conversion may cover a long time frame, logically consistent copies of the source and target data bases must be maintained. The ADAPT system can be used as a tool to insure consistency of the source and target data bases, even if they exist on different machines. Another use of the ADAPT system is in a distributed data base context. Logical records which are distributed to different nodes of the data base can be "collected" by ADAPT and presented as a single physical record to a user at one node. This paper presents a functional overview of the ADAPT system and discusses applications of the ADAPT system to computer network problems.

1. Introduction

The traditional approach to data conversion requires development of independent hard-coded conversion systems for every conversion process. Such systems consume valuable resources in development and maintenance. The need for generalized high-level data translation systems has been well documented over the last few years. Such systems can make the conversion process much simpler, as the conversion code is easier to develop, easier to understand, easier to maintain, and easier to modify. Unfortunately, the appearance of generalized translation systems in a production environment has lagged far behind research into the conceptual problems surrounding data conversion.

Significant work in the area of data conversion has been done at the University of Michigan [3-9]. But the emphasis in that work was to provide a foundation for future research and development in data translation. Other significant work in the area has been done by Smith [1Ø-11], Ramirez [12-13], Sibley and Taylor [1], Shoshani [14], and Bakkom and Behymer [15]. Work by Housel, Shu, and Lum [16-2Ø] at IBM is based on two descriptive languages which drive their translation system. The IBM work is principally geared towards logical restructuring of hierarchical data structures, but it is one of the only generalized translation systems being used in a production environment.

The ADAPT (A DAta Parsing and Transformation) system provides an efficient generalized language-driven approach towards data translation. ADAPT provides the user with a language for describing the source and target data formats and structures, and a language for specifying the mappings between the source and target data structures. ADAPT allows transformations involving multiple record types, follows a generative approach towards data conversion, provides logical restructuring and reformatting operations including those performed by the UNIX* text editor, and provides a neat modular scheme for crossing over machine boundaries. Further, ADAPT was designed to be a production environment translation tool. As such, efficiency and functional completeness for handling production translation requirements were prime design criteria.

The ADAPT system lends itself to quick and simple modifications of the data descriptions and transformations, as the source data and conversion requirements become better understood by the user. ADAPT can be used for other applications besides a one-time translation system. It can be used for consistency control between the source and target data bases during the conversion period. It can also be used to control access to a distributed data base system. In short, it can be used dynamically by any application requiring transformation of a data stream from one format to another. This paper presents a functional overview of the ADAPT system and discusses applications of the ADAPT system to data base network problems.

2. System Configuration

All components of ADAPT are written (or generated) in the C language [23]. ADAPT was originally designed to run

* UNIX is a Trademark of Bell Laboratories.

on the PDP 11/70 computer under the MERT/UNIX operating system [24]. ADAPT runs as a single process in that environment; communication with other (UNIX) processes is a natural extension of this environment. ADAPT is also portable to any machine/operating system which supports a C compiler. If ADAPT is used as a sub-module of a larger process in those environments, the appropriate inter-process communication protocols must be followed. Currently, ADAPT is portable to the IBM 370 and UNIVAC 1100 series computers, and it will soon be ported to the VAX 11/780 computer.

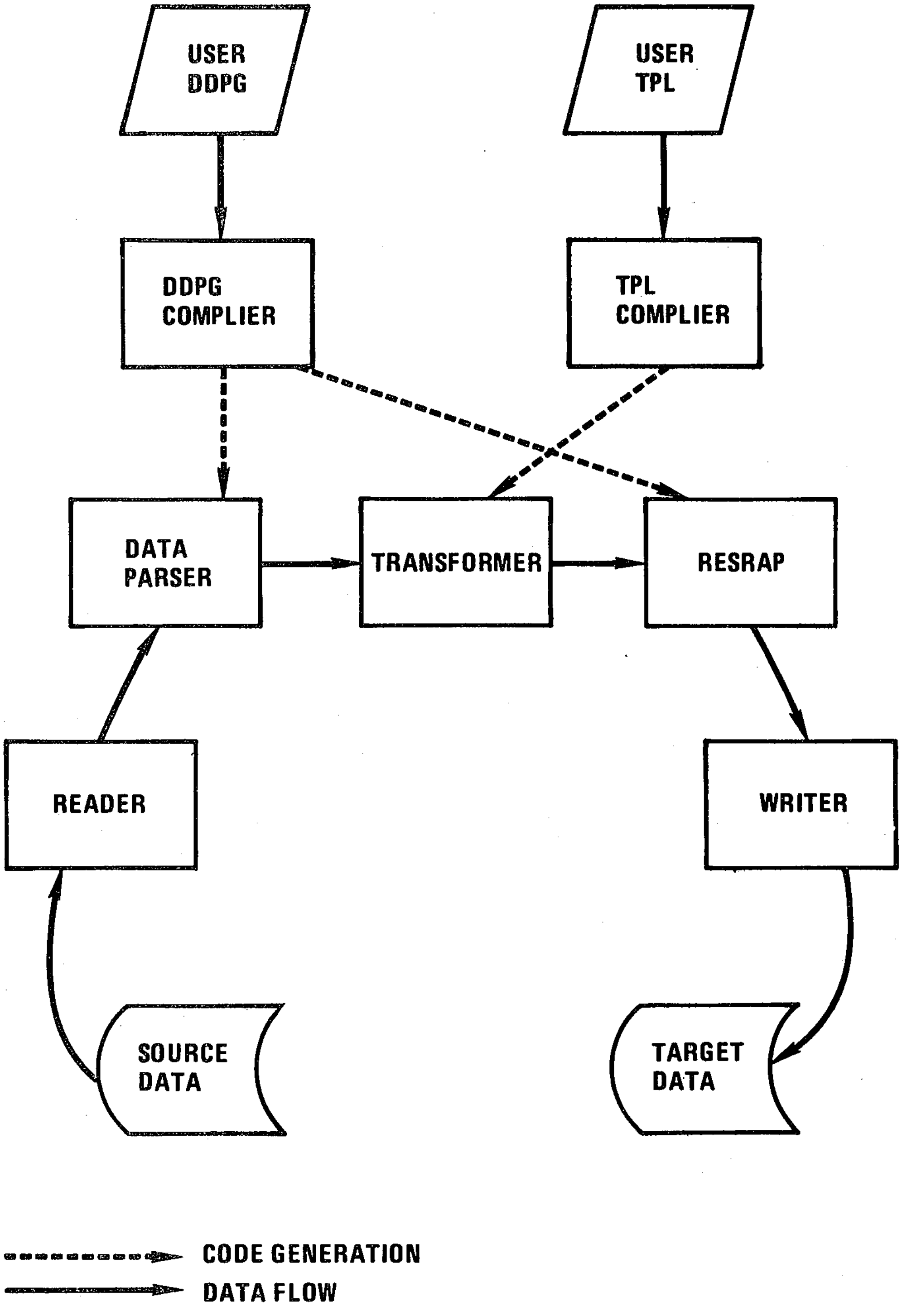
The ADAPT system consists of two compilers, and a run-time system consisting primarily of code generated by the two compilers. The user describes the format and structure of the source and target data using the Description Language for Data Parsing and Generation (DDPG). The Transformation Programming Language (TPL) is then used to describe the mappings between the source and target data. Subsequent sections of this paper will present the DDPG and TPL languages in more detail.

Based on the user data descriptions, the DDPG compiler generates two data parsers - corresponding to the source and target data descriptions. The target data parser is called the Resrap module. The TPL compiler is then run on the user's TPL specification and, using the user data descriptions, generates the Transformer module. As can be seen, with this generative approach, each executable ADAPT system is automatically tailored to the particular application's conversion requirements, thus optimizing the performance of the conversion system for each application environment.

The run-time data flow through the ADAPT system is shown in Figure 1. The Translation Controller acts as the main routine, controlling the execution of the other modules, collecting statistics and performing error handling. The Reader prepares the input data for the rest of the system. After the data has been read, the Data Parser parses the source data, matching it to the user source description. The Data Parser also performs hardware-dependent data conversions. The Transformer then applies the user-specified transformations to the source data and produces the target data. The Resrap module does a "reverse" parse of the target data, formatting it according to the target description, and then sends the target data to the Writer for target hardware-dependent conversions and final output.

3. DDPG Compiler and Data Parser

As mentioned above, the DDPG language is a high level language used to describe the format and structure of the



ADAPT SYSTEM ARCHITECTURE

FIGURE 1

source and target data bases. The user-supplied DDPG description contains separate sections for the source and target data. Each section is further subdivided into an environment section, a cluster definition section, a data filtering section, a table definition section, and a data section. The environment section specifies such information as the application machine and character code set. The data filtering section allows the user to specify certain conditions under which data should not be translated. This is described further in the discussion below on the DUMP and DISCARD commands. The table definition section contains the descriptions of user supplied tables. Since these tables can also be used by the TPL, their description will be given in the section on TPL operators. The cluster section and the data section contain the complete logical description of the user data.

An item is the elementary data unit. A group is a named ordered collection of items and/or other groups. The named set of multi-level hierarchical structures formed by nesting and concatenating groups and items is a record type. A record is a collection of data conforming to a record type. The complete DDPG data section consists of multiple record types. The records described in the data section can occur in different run-time combinations, called clusters. The cluster section specifies the run-time conditions under which records occur, as well as the number of times they occur. The ADAPT system processes clusters of records sequentially.

Some of the major data attributes which can be described in the DDPG language are the following:

- specification of a variety of data types (e.g. character, integer, packed decimal).

- fixed or variable length data fields, where the field length can be expressed as an arithmetic expression or can be determined by a character terminator.

- character justification, pads, null values, and string terminators which can be expressed globally in the record header or overridden at the item level. The record header can also contain blocking information and a record type indicator.

- specification of self-defining data using the MATCH function. Match provides the user with the ability to "look ahead" at data, returning "true" if a pattern is matched, "false" otherwise.

  Match takes the form

```
MATCH(off1,off2,pattern)
```

where off1 and off2 are the byte offsets relative to the current position of the Data Parser within the record, and pattern is a character string expression to be used as the pattern matching criteria. The character string expressions used in patterns are equivalent to those used in the UNIX text editor.

For example, the boolean expression

```
MATCH(2,4,"[ABC][Ø-9]{2}")
```

instructs the Data Parser to look ahead to byte position 2 through 4 relative to the current position in the record. If the characters in those bytes consist of an A or B or C followed by any two numeric digits, then the match is true; otherwise it is false.

- optional data (at the GROUP or ITEM level) specified with a conditional expression via the EXISTS clause. For example,

```
GROUP gname EXISTS (boolean expression)
```

means that the group identified by gname exists in the data stream if the boolean expression evaluates to true.

- mutually-exclusive descriptions of the same data using the VIEW construct. Views can be used at the group or item level, and they can be nested.

```
e.g.
   GROUP gname
      VIEW vname1 (a == "YES")
      [
            other DDPG constructs
      ]
      VIEW vname2 (a == "NO")
      [
            other DDPG constructs
      ]
      VIEW vname3 (a == "MAYBE")
      [
            other DDPG constructs
      ]
   ;
```

In this example, for any instance of the gname group, one and only one of the three views apply depending on whether item a (previously parsed) is "YES", "NO", or "MAYBE". The "other" DDPG constructs associated with the view describe the data for that particular instance of gname. If no views apply, a run-time error results.

- mutually-exclusive descriptions of data using the SET-ELEMENT construct.

```
   e.g.
        SET listing until ( match( Ø, 3, "EOF") )
           ELEMENT name ( match( Ø, 2, "NA") )
           [
                other DDPG constructs
           ]
           ELEMENT address ( match( Ø, 2, "AD") )
           [
                other DDPG constructs
           ]
           ELEMENT phone_num ( match( Ø, 2, "PN") )
           [
                other DDPG constructs
           ]
        ;
```

  Syntactically, this is similar to views within repeating groups. Semantically, however, elements in a set have a closer relationship to each other than views; elements are later referenced in the TPL independent of the order they were parsed, whereas views must be referenced via subscripting. The set will be parsed until its match expression ("EOF") is true. Elements name, address and phone_num apply when their respective match expressions are true.

- specification of repeating items or groups via the OCCURS clause. This specification can be fixed or variable.

- specification of characters which must be stripped from the middle of data fields, via composite items. For example, an ADAPT application requires that a number of physical lines, each of length 8Ø characters, be treated as a single logical line. But the blank characters at the beginning of each physical line must be omitted. The DDPG specification for this application is

```
        composite item logi_line until ( ! (match(Ø,1," ") )
                physi_line char (8Ø) just right pad " ";
        end logi_line;
```

  The Parser parses 8Ø character fields, stripping off the left-most blanks, until it encounters a line starting with a non-blank character. The concatenation of the physical lines is treated as a single logical line.

- record filtering criteria expressed via the DUMP and DISCARD commands.

```
DUMP rname to fname (boolean expression)
```

The boolean expression is evaluated at data parse time. If true, the associated record or cluster is dumped to the named file, an associated logging message is written out, and processing continues with the parsing of the next record or cluster. DISCARD performs a similar function as DUMP except, in this case, the record or cluster is thrown away rather than dumped to a file. Thus, DUMP can be used to control the order in which records get translated (since the files built via the DUMP commands could be translated at a later point in time) without separately pre-processing the source data and applying pre-determined translation selection criteria.

- data validation criteria via the FORMAT clause.

```
iname CHAR(arithmetic expression) FORMAT (pattern)
```

Here, as a record is being parsed, a character string item whose length is given via an arithmetic expression, is validated according to the user-supplied pattern. If the validation test fails, the user can filter the record or cluster to appropriate files.

- special constant data generation for the target data through use of the ATTACHL, ATTACHR, and VALUE clauses. The ATTACHL and ATTACHR clauses can be used to attach special user-supplied field identifiers to the left or right of the actual target field value. The VALUE clause can be used to specify a fixed value to be assigned to a target field.

The Data Parser and Resrap modules can perform more extensive validation than the FORMAT clause allows by means of the elegant table handler provided by ADAPT (see next section).

4. TPL Compiler and Transformer

The Transformation Programming Language (TPL) is a high level language used to perform the actual translation of data from the source data base to the target data base. The TPL compiler generates the Transformer module based on the DDPG descriptions and the user-specified TPL transformations.

The user's TPL code is divided into several translation blocks. Each block consists of a set of many-to-one transformations. That is, each block contains all of the transformations involving one and only one target record type regardless of how many source record types map into it. Thus, the user has the ability to combine fields from different record types to produce fields in the target record type.

```
e.g.  TRANSLATE RECORDS a,b,c TO d;

                           .

                        tpl code

                           .

                    END_BLOCK;
```

would take source records a, b, and c and produce target record d. If the target record is only produced under certain conditions, the translation block header would have the following form:

```
TRANSLATE RECORDS a,b,c TO d WHEN (boolean condition);
```

The operators currently supported by the TPL compiler include assignment, selection, concatenation, extraction, control flow, explicit type conversion, table handling, user specified termination, looping mechanisms, and user supplied functions. These operators interact among themselves and with the usual Boolean and arithmetic operators to form the expressions referred to below. A brief description of each operator follows.

- ASSIGN. The assignment operator correlates the transformed data with the appropriate target field(s).

  ```
  ASSIGN TO field_name (expression);
  ```

  ASSIGN is designed to work in conjunction with the other operators, in that these operators form TPL expressions which are evaluated at run time and assigned to the target field(s).

- SELECT. The selection operators retrieve source data entities: that is, they retrieve either source items or groups of items. There are three variations of the SELECT operator:

  a. retrieval of subtrees from the source data structures.

```
SELECT subtree
```

  b. retrieval of subtrees satisfying certain conditions using the WHERE clause.

```
SELECT subtree WHERE (boolean expression)
```

  c. retrieval of entire data entities without nesting other operators, i.e. retrieval where no further transformations are to be performed other than ASSIGN.

```
SELECT AS IS field_name
```

  This is a more efficient form of

```
ASSIGN TO field_name2 (SELECT field_namel)
```

- CONCAT. The concatenation operator is used to concatenate any number of data fields, constant values, or other expressions.

```
CONCAT (expl, exp2, ... , expn)
```

- EXTRACT. The field extraction operator is used to match a pattern in a character data field.

```
EXTRACT FROM (exp) pattern
```

The class of patterns which can be extracted is equivalent to that of the UNIX text editor. For example,

```
EXTRACT FROM (field_a) "[A-Z]{2}[Ø-9]*"
```

returns the string in field_a which has two upper case alphabetic characters followed by any number of numeric characters.

The fixed field extraction operator SPLIT is used to extract that portion of a data field lying between specified byte offsets.

```
SPLIT (offset1, offset2, exp)
```

- IF-ELSE. The control flow operator allows blocks of TPL statements to be executed dependent upon the evaluation of a boolean expression.

```
IF (boolean expression) tpl statement list;

        ELSE tpl statement list;  (optional)
```

- Explicit Type Conversion. These operators allow the user to specifically convert data from one type to a second type.

```
type (expression)
```

For instance, if line_num were defined as a character field, then

```
INT (line_num)
```

converts line_num to its integer representation.

- Table Handler. In the table definition section of the DDPG, the user specifies the structure of a table, sort keys, and the file containing the table data in the following manner:

```
      TABLE t {KEY type field1;...;type field_n;}
                                    FILE file_name;
```

where "type" is the data type such as "character".

ADAPT reads the table into the system and generates a function to access it. The user references the table by indicating the field whose value is to be returned, qualified by values of the key fields.

```
      t.return_field SUCH THAT (boolean condition on key
                                         fields of table t)
```

For instance, if the user had a file "directory" containing a table "listings" with field names "name", "address", and "phone" sorted by "name", then it would be defined by

```
        TABLE listings (KEY char(16) name;
                char(40) address;
                char(7) phone;
               } FILE directory;
```

An expression such as

```
listings.phone SUCH THAT (name == 'kaplan')
```

retrieves the phone number of someone named 'kaplan'. This facility can be used to return data values, or it can be used by the Data Parser and Resrap modules to perform data validation.

- User Specified Termination. This will cause an immediate return from the Transformer, presumably when some error condition has been discovered.

```
      IF (error condition) ABORT;
```

- Run-time Variables. Special run-time variables can be assigned values in the TPL code. These improve efficiency since repetitive calculations need only be performed once, then assigned to the variables. In addition, run-time variables can be assigned from the "command" line which starts execution of the ADAPT system. These variables can participate in all arithmetic and boolean expressions. Execution of entire blocks of code may depend on their values. This gives the user greater flexibility in running ADAPT from a uniform set of descriptions and transformations without rewriting and recompiling ADAPT code.

- Looping Mechanisms. The TPL allows two kinds of "for" loops. The first kind uses explicit user indices, as in

```
FOR (i = Ø;   i < LIM;   i++)
{
        loop body
}
```

Here, the variable i is initialized to Ø and incremented by 1 (i++) after each execution of the body of the loop until it is no longer less than LIM. The user must explicitly subscript field names in this scheme. The second kind of "for" loop uses implicit indices, as in

```
FOR each gname
{
        loop body
}
```

Here, gname was subject to an occurs clause in the DDPG description. For each occurrence of gname, the loop body is executed; the appropriate indices are supplied automatically to all field names in the loop body which are in the scope of the gname structure.

- User Functions. The user may supply a set of specialized routines which perform operations particular to the given application but not supported by ADAPT. They are called from the TPL code by the CALL operator.

```
CALL function_name(expl,exp2,...,expn)
```

The source code of the function can be written in any language supported by the translation machine. It is compiled on the translation machine, then linked with the other modules of the ADAPT system.

5. Clustering

As mentioned above, ADAPT processes clusters of records sequentially. Applications frequently require that more than one source record be utilized to produce target records, and that several target records be produced from a single set of source records. The specification of source records appearing in a cluster and the target records output in a cluster is given in the cluster section of the source and target data descriptions.

Records can be described as conditionally existing and occurring a multiple number of times per cluster using the same EXISTS and OCCURS constructs described in the data section of the DDPG. EXISTS and OCCURS expressions in the source cluster may depend on the values of fields in records which were already parsed, or they may depend solely on their existence in the data stream. The existence of records in the data stream is determined by examining the record type indicator as specified in the record header. For instance, records named "BEE" may have an indicator "B" in its third character position. When a record is read, the reader checks if it is a "BEE" record, and if so, calls the correct parse routine.

The target cluster is constructed by the Transformer module. Conditional translation blocks control the creation of target records. The target cluster section is then used to validate the integrity of the records produced by the Transformer.

Records in a cluster bear an implicit relationship to other records in the same cluster, but for purposes of the data translation process, they are considered unrelated to records in other clusters. Assignment of run-time variables, however, allows information to be "remembered" between clusters. When translating between arbitrary source and target data bases, the record structures must be "linearized" into the cluster format before entering the ADAPT system.

Records in a cluster can be in different physical files as specified in the DDPG description. ADAPT accesses the correct file when reading (writing) records of particular types. The only restriction is that all records of a particular type must be in the same physical file, and they must appear in the order in which they are to be read. The "logical" record stream input to (output from) the ADAPT system is thus identical to the sorted physical record stream input from (output to) the files.

6. Applications of ADAPT

Data base translation is a relatively infrequent operation, and hence, data base translators are usually only thought of in terms of performing this one-time translation. However, they could have much wider use in applications involving transformations of data streams. ADAPT, in particular, can be used as a dynamic translation module in a larger software system. Two such applications will be discussed in this section.

Data conversion takes a long period of time even when an efficient data translation program is used. The translated data must be examined and tested against live data before the original source system is replaced. During this time, the source data base cannot always be frozen since it must be constantly updated to reflect the real-world situation. If source data which was already converted is updated, the corresponding target data must also be updated. For any significant volume of update activity, manually updating both data bases is difficult and subject to error.

The ADAPT system can be utilized to overcome many of these transition problems. ADAPT accepts the update request to one data base and outputs two update requests: one for each data base. The input data to the ADAPT system consists of the update command to the first data base. ADAPT outputs a cluster consisting of two target records. The first target record is identical to ADAPT's input record and is passed directly to the update facility of the first data base. The second target record is the semantic equivalent of the input (update) record, but is reformatted to conform to the syntax and semantics of the update facility of the second data base. Note that even though the input and output records are really command lines, ADAPT treats them as streams of data.

Since ADAPT can convert data from one machine format to another, this scheme can also be used when converting across machine boundaries. An application requiring conversion of directory assistance products uses ADAPT in this fashion. Communication between the machines is provided by an independent computer network facility called BANCS [26], which handles all of the physical machine interfaces, and has its own independent queuing facility.

For example, suppose the user made the following request to the data base:

```
SET PHONE_NUMBER EQUAL TO '4769' IN DIRECTORY
          SUCH THAT NAME IS 'KAPLAN'
```

Two configurations exist depending on which machine the ADAPT program resides. If it resides on the same machine where the user request is made, then ADAPT accepts this request as an input record. It outputs the exact record to the first data base, whose update facility accepts this stream of data as a command, and performs the appropriate operation on the data base. ADAPT's second output record may have the following form:

```
UPDATE PHONE_NUMBER '4769' (DIRECTORY.NAME = 'KAPLAN')
```

This record is sent to the BANCS communication facility which transmits the record to the second data base. The update facility of the second data base then treats this data stream as a command and performs the appropriate operation on the data base.

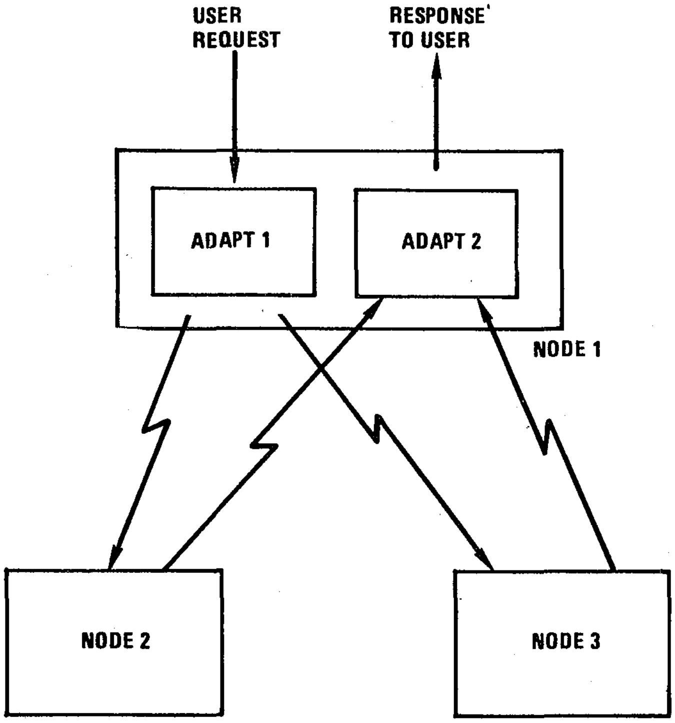
Alternatively, if ADAPT does not reside on the machine where the user request originates, then that request is immediately transmitted via BANCS to the second machine and input to ADAPT. ADAPT produces the two target records in the manner described above but sends the original request back to the first machine via BANCS.

The extension of this idea to distributed data bases is straightforward. The data in a distributed data base exists at all nodes of the network, but a user at any node of the distributed data base has no knowledge of the underlying structure of the data base. For security reasons, the data base administrator may not want particular users to access certain fields of data so these data fields are invisible to them. Using ADAPT, the data base administrator has an efficient, easy way to accomplish these aims.

The data base administrator writes an ADAPT program for each set of users, specifically geared to the users' application requirements. The source description for the ADAPT program accepts all allowable user queries to the distributed data base. Based on the type of user command, the associated data values supplied by the user, and the known location of data types in the distributed data base network, the ADAPT program outputs a cluster of records which are really commands to be sent to different nodes of the data base network. User requests which differ with regard to command type or associated data values are automatically routed by ADAPT to the proper nodes of the network. When the nodes return the data to the user node, another ADAPT program accepts all of these records as an input cluster. From these records, a single output record is built and returned to the user.

Since ADAPT can convert data across machine boundaries, the distributed data base can exist on different machines, providing a true computer-data base network. As opposed to "standard" distributed data base systems, individual data bases at different nodes of the network can be of different types. Synchronization of data transfer is handled by an inter-machine communication facility such as BANCS, mentioned above. The user interfaces with the distributed data base through individualized ADAPT systems. This has the added security advantage that the user is totally unaware of other data in the data base.

An example of the use of ADAPT in a distributed data base environment is depicted in Figure 2.



USE OF ADAPT IN DISTRIBUTED DATA BASES

FIGURE 2

Suppose a user enters a request at node 1 of a computer network. The request is sent to the ADAPT 1 module which formulates the request as (possibly different) queries to nodes 2 and 3 of the network. The two queries represent two record types of an ADAPT target cluster. The ADAPT 1 module directs the queries to the correct "channels" of the communications link, from where they are sent to the appropriate nodes. When the responses from nodes 2 and 3 are received at node 1, they are treated as records in the input cluster to the ADAPT 2 module. The output from the ADAPT 2 module is the response to the original user query.

7. Conclusions

The need for an efficient high-level language approach to data conversion has been proven historically by the large and expensive conversion effort experienced by almost every data processing application. The ADAPT system provides a generalized, efficient approach towards meeting the needs of many of these applications. The functional capabilities provided by the ADAPT system are those that appear to be most often required by applications. These facilities have been tuned over a period of time and now operate in a manner that provides a system throughput rate which is well within the operational requirements of most applications.

ADAPT's first major application was to translate a data base comprising a set of directory assistance products. The source data base resided on an IBM 370/168, and the target data base was to reside on a PDP 11/70. The source records had an average size of 340 bytes and the target records had an average size of 165 bytes. For this application, ADAPT was able to achieve a throughput rate of 30 records per second running on a PDP 11/70, and a throughput rate of 75 records per second running on an IBM 370/168.

The ADAPT system can also be used in a computer network environment where a data translation step is required for inter-node communication. For instance, concurrent copies of source and target data bases can be synchronized during the conversion process, using ADAPT. On a larger scale, ADAPT can be linked with an inter-machine communication facility to support many of the concepts of distributed data base systems.

8. Acknowledgements

The authors wish to acknowledge the contributions made to the development of the ADAPT system by M.E. Mahon, S.C. Stein, and E.M. Sondheim.

9. References

[1] Sibley, E.H., and Taylor, R.W., "A Data Definition and Mapping Language", Comm. ACM 16,12 (Dec.1973).

[2] Fry, J.P., Smith, D.P., and Taylor, R.W., "An Approach to Stored Data Definition and Translation", Proc. ACM SIGFIDET Workshop on Data Description, Access and

Control, Denver, Colo., Dec. 1972.

[3] Taylor, R.W., "Generalized Data Base Management System Data Structures and Their Mapping to Physical Storage", Ph.D. Diss., University of Michigan, Ann Arbor, Mich., 1971.

[4] Fry, J.P., Frank, R.L., and Hershey, E.S.III, "A Developmental Model for Data Translation", Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif., Nov.1971.

[5] Merten, A.G., and Fry, J.P., "A Data Description Language Approach to File Translation", Proc. ACM SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[6] Birss, E.W., and Fry, J.P., "Generalized Software for Translating Data", Proc. AFIPS 1976 NCC, AFIPS Press, Montvale, N.J.

[7] Deppe, M., Lewis, K., and Swartwout, D., "Operational Software for Restructuring Network Data Bases", Working Paper DT3.2, Data Translation Project, University of Michigan, Ann Arbor, Mich., 1976.

[8] Navathe, S.B., and Fry, J.P., "Restructuring for Large Databases:Three Levels of Abstraction", ACM Transactions on Database Systems 1,2 (June 1976).

[9] Swartout, D., "An Access Path Specification Language for Restructuring Network Databases", Proc. ACM SIGMOD International Conference on Management of Data, Toronto, Canada, August 1977.

[10] Smith, D.P., "An Approach to Data Description and Conversion", Ph.D.Diss., University of Pennsylvania, Philadelphia, Pa., 1971.

[11] Smith, D.P., "A Method for Data Translation Using the Stored Data Definition and Translation Task Groups Languages", Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, Denver, Colo., Dec.1972.

[12] Ramirez, J.A., "Automatic Generation of Data Conversion Program Using a Data Description Language (DDL)", Vols.I, II, University of Pennsylvania, Philadelphia, Pa., May 1973.

[13] Ramirez, J.A., Rin, N.A., and Prywes, N.S., "Automatic Generation of Data Conversion Programs Using a Data Description Language", Proc. ACM SIGMOD Workshop on

Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[14] Shoshani, A., "A Logical-Level Approach to Database Conversion", Proc.ADM SIGMOD Conference on Management of Data, San Jose, Calif., 1975.

[15] Bakkom, D.E., and Behymer, J.A., "Implementation of a Prototype Generalized File Translator", ibid.

[16] Housel, B.C., Lum, V.Y., and Shu, N.C., "Architecture to An Interactive Migration System (AIMS)", Proc.ACM SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[17] Housel, B.C., and Shu, N.C., "A High-Level Data Manipulation Language for Hierarchical Data Structures", Proc. Conf. on Data Abstraction, Definition, and Structure, Salt Lake City, Utah, March 1976.

[18] Housel, B.C., Smith, D.P., Shu, N.C., and Lum, V.Y., "DEFINE-A Nonprocedural Data Description Language for Defining Information Easily", Proc. ACM Pacific 75, San Francisco, Calif., April 1975.

[19] Lum, V.Y., Shu, N.C., and Housel, B.C., "A General Methodology for Data Conversion and Restructuring", IBM J. Res. and Develop.2Ø, 5(1976).

[2Ø] Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., and Lum, V.Y., "EXPRESS: A Data EXtraction, Processing, and REstructuring System", ACM Transactions on Database Systems, Vol.2, No.2, June 1977.

[21] Bracchi, G., Fedeli, A., Paolini, P., "A Language for a Relational Data Base Management System", Proc. Sixth Annual Princeton Conf. on Information Sciences and Systems, Princeton, N.J., March 1972.

[22] Ritchie, D.M., and Thompson, K., "The UNIX Time-Sharing Sytem", Comm.ACM 17,7 (July 1974).

[23] Kernighan, B.W., Ritchie, D.M., "The C Programming Language", Prentice Hall Inc., Englewood Cliffs, N.J., 1978.

[24] MERT Programmer's Manual, Bell Telephone Laboratories, Internal Publication.

[25] Date, C.J., "An Introduction to Database Systems", Addison-Wesley Publishing Co., Reading, Mass., 1975.

[26] Leung, S., "The Architecture of a Modern Network - The BANCS Network", COMPCON Proceedings, Fall 1978.

DISTRIBUTED DATA BASE INTEGRITY

The Effects of Concurrency Control on the Performance of a Distributed Management System

Daniel R. Ries
Electronics Research Laboratory
University of California, Berkeley

ABSTRACT

Simulation models for four concurrency control algorithms were used to study the effects on a distributed database. In a distributed database, the data and transactions are distributed over several computer sites connected through some type of network. Some transactions access data at only one site, while others access data at several of the computer sites.

The concurrency control algorithms simulated can be divided into two general classes: primary site control and decentralized control. In the primary site control models, all of the locking takes place at one of the nodes designated the primary site. Note that even "local transactions" (transactions that just access data at their originating sites) must send lock requests to the primary site.

In the decentralized control models, the locking of the data items takes place at the site where the data being accessed is stored. In these models, then, local transac-

tions need not send any messages over the computer network.

1. INTRODUCTION

Recently, considerable attention has been devoted to the development and use of distributed databases. A distributed database is a database which is stored at multiple computer sites connected by some type of computer network. In this environment, a transaction originates at one of the computer sites and potentially accesses data at other sites as well as at the originating site.

One of the primary advantages of a distributed database over a centralized database is that increased parallelism is possible because multiple sites can be simultaneously processing transactions. However, the distributed concurrency control mechanism may have to expand additional overhead to guarantee database consistency [ESWA76, GRAY76] during this simultaneous processing. This additional overhead is due to the costs required to set locks at remote sites and/or the costs which may be required to resolve deadlock between transactions at different sites.

Several solutions to the concurrency control problems for distributed databases have been proposed ([BERN77], [ROSE77], [GRAY78], [MENA78], [STON78] and [THOM78]). Often, one performance goal of such proposals is to minimize the number of concurrency control messages which have to be sent across a computer network. In [BERN77], it is shown that if the transactions are known in advance, different types of concurrency control can be used for different types of transactions and thereby reduce even further the overhead network traffic.

Unfortunately, the count of overhead message traffic does not, by itself, determine the effects of the concurrency control on the overall performance of a distributed database system. Other factors such as the processing load at each site, the overall network load and the types and sizes of the transactions must also be considered.

Thus, simulation models were developed to more adequately invesigate the performance trade offs between increased parallelism and increased overheads of a distributed database. These models simulated four concurrency control algorithms and were used to study the effects of locking granularity ([GRAY75], [RIES77], [RIES79]), the effects of the proportion of transactions requiring non-local or remote resources, and the effects of different network throughputs and bandwidths on the overall performance of a distributed management system.

In the next section, the basic model of a distributed database that was simulated is described. In section 3, the four different concurrency control algorithms are discussed. In section 4, the simulation results for each of the four algorithms are reported. In the final section, the major conclusions are reiterated.

2. The Simulation Model

The model of a distributed database system that was simulated closely follows the basic model being implemented in distributed INGRES [STON77]. In the simulation model, the database was assumed to be distributed among a number of different computer sites or nodes connected by some type of network.

Transactions were submitted to the database management system at each site. Some of the transactions, called 'local' transactions, only accessed data at the site where they originated. Other transactions, called 'non-local' transactions, required some database access at other than the originating sites.

Such a non-local transaction was realized by a 'MASTER' transaction at the originating site and 'SLAVE' transactions at the other sites where processing was required. The MASTER transaction initiated all of its SLAVES and waited for those slaves to complete. In the simulation, transactions were cycled around a closed loop model (shown in Figure 1) for each node or site in the distributed database. Each of these site models was very similar to the simulation models in [RIES77], [RIES79]. At each site, the transactions initially arrived one simulation time unit apart and went

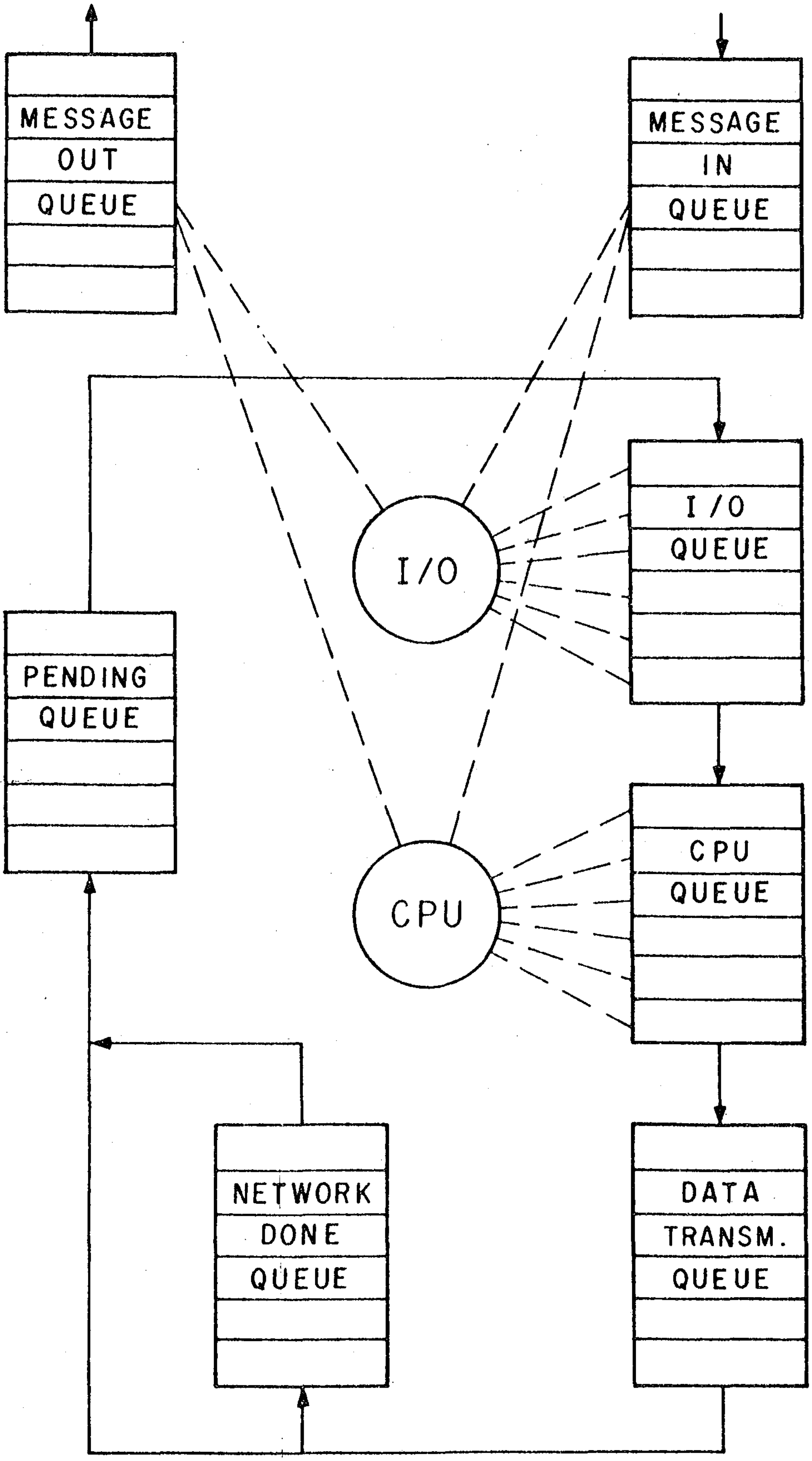


Figure 1: Node or Site Model

through the following steps: 1) left the pending queue, 2) I/O processing, 3) CPU processing, 4) data transmission, 5) local processing completion, and 6) distributed processing synchronization. Each of these steps is described in more detail below.

1) When a transaction left the pending queue it was placed on the I/O queue. If the transaction was a MASTER, it sent SLAVE create messages to the appropriate nodes.

2) The I/O server was multiplexed among the transactions on the I/O queue. When a transaction had received its share of I/O resources, it was placed on the CPU queue.

3) The CPU server was multiplexed among the transactions in the CPU queue. When a transaction had received its share of CPU resources, its next action depended on whether or not the transaction was local.

4) Local transactions were considered complete at this point and were recycled to the pending queue. Non-local transactions (both SLAVES and MASTERS) were placed on the data transmission queues. If any data was to be transmitted to another node, a data transmission message was sent.

5) When the data transmission message had been delivered (or if no data was to be transmitted), the non-local

transaction proceeded to the Network done queue. At this time, SLAVE transactions sent a SLAVE complete message back to the MASTER transaction.

6) Depending on the concurrency control strategy, a SLAVE either waited on the Network done queue or was simply released. The release of a slave is discussed in more detail in section 3. The MASTER transaction waited on the Network done queue until it had received "slave complete" messages from all its slaves. At that point, the transaction was recycled back to the pending queue.

Several simplifying assumptions should be noted about the model. First, all of the SLAVEs were identical to the originating MASTER in terms of the proportion of database accessed and whether or not data needed to be transferred. In distributed database applications, the actual characteristics of the SLAVEs could be quite different from the MASTER and from each other. Second, the only synchronization between the SLAVEs and their MASTER transaction occurred at the beginning and end of the transaction. Some applications would require additional synchronizations on the data being transmitted [WONG77, EPST78].

Also note that a transaction is on each of the I/O, CPU and data transmission queues once in the indicated serial order. The total processing required is the same as if the transaction cyclically accessed the I/O, CPU and data transmission queues. To send a message, a transactor would place the message on the message- out queue together with a message destination and length. Messages were taken from the message-out queue and given to the Network Manager as shown in Figure 2. When a message had received the needed amount of network service, it was placed on the destination message-in queue.

Both a speed and a bandwidth are associated with the Network Manager. The network speed was represented by the minimum time a message of any type spent in the network where time was measured in the time units of the simulation. The bandwidth was represented by the maximum number of messages which could be serviced in one of those time units.

The flow of a message in the Network Manager can be described as follows:

1) When a message entered the network manager, the time remaining for that message was initialized to the message length in the time units of the simulation. The message length can vary depending on whether or not data is being sent but is at least equal to the minimum length mentioned above.

2) If MESSBDWH was the bandwidth of the Network Manager, the times remaining of the first MESSBDWH messages in the Network queue were reduced by one time unit.

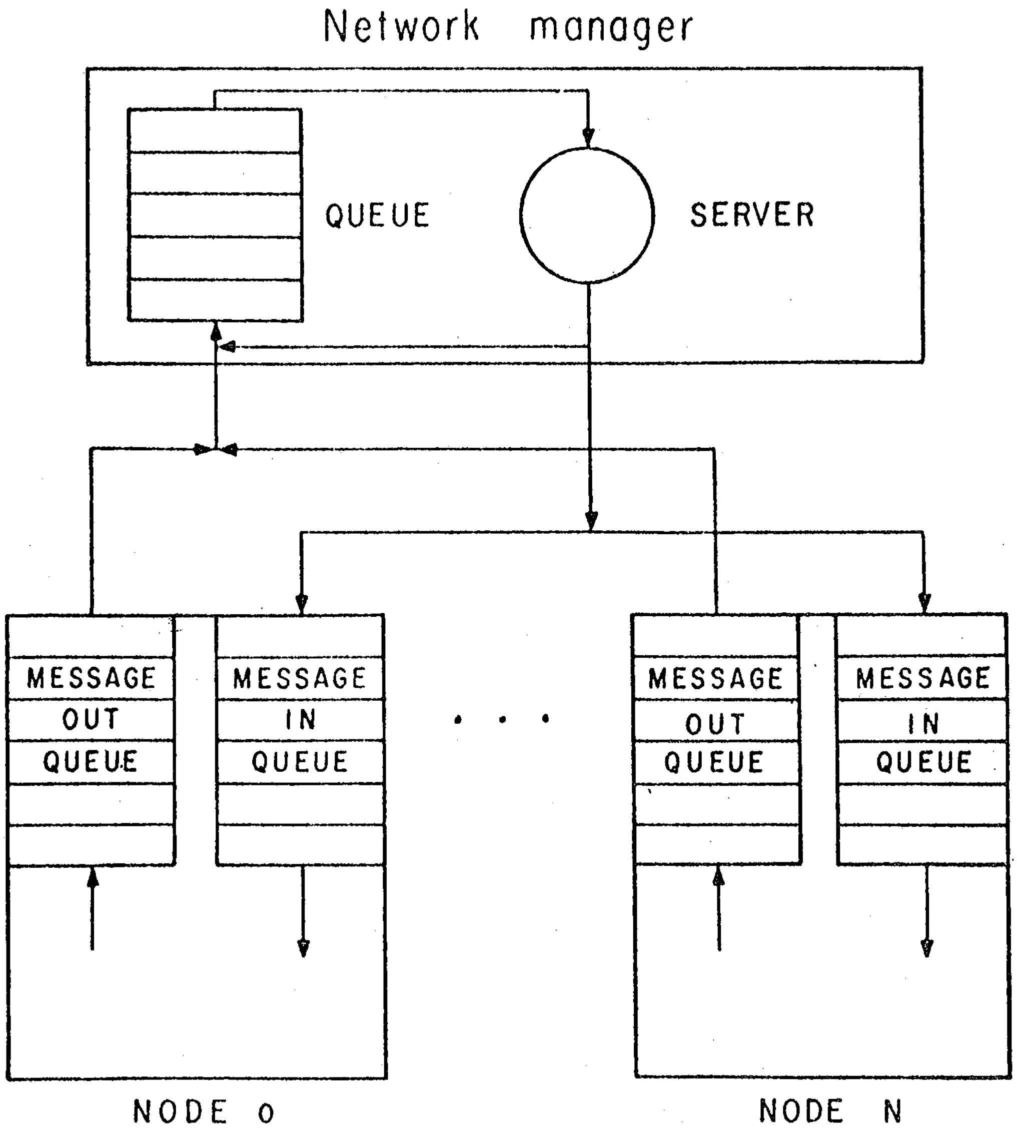


Figure 2: Network Model

3) If the time remaining for any message was zero, it was delivered to the message-in queue of the destination node.

In several of the concurrency control schemes, a site was allowed to send messages to itself. In these cases, the network manager was bypassed and the message went directly from the message-out queue to the message-in queue.

3. Concurrency Control Algorithms

Four concurrency control algorithms were simulated. All of the algorithms required that transactions 'lock' the parts of the database they access and obey a 'two- phased' locking protocol [ESWA76]. A 'lock' on a certain portion or granule of the database was granted to one transaction and prevented any other transactions from accessing that portion of the database until the given transaction released the lock. Note that in the simulation models, each lock was assumed to be exclusive in that it could only be held by one transaction at a time. Thus, in the simulation models, no distinction was made between read and write access to the database.

The 'two-phased' protocol required that a transaction first acquired all of the needed locks (called a 'growing' phase) before releasing any locks (during a 'shrinking' phase). This protocol, together with the requirement that all accessed parts of the database be locked, insured that the effect of the transactions would be equivalent to the effects of running the transactions one at a time in some serial disorders. This 'serializability' [ESWA76] of the transactions insures a certain level of database consistency [GRAY76].

The four concurrency conrol algorithms simulated can be divided into two general classes: primary site concurrency control ([ALSB76], [MENA78]) and decentralized control ([ELLI77], [GRAY78], [ROSE77], [STON77]). In two primary site models, one site was chosen to manage the locking for the entire database. In both of these models, when any transaction (local or MASTER) left the pending queue (see Figure 1), a global lock request was sent to the 'primary' site. The transaction then waited until it received an 'all locks granted' message and proceeded to the I/O, CPU, and data transmission queues. Also at this point, a MASTER translation, which was smart enough to request locks for all its slaves, sent the 'SLAVE create' messages to the appropriate nodes.

Thus, when a 'SLAVE create' message was received at a site, the SLAVE transaction went directly to the I/O queue. When the SLAVE transaction was through with the I/O, CPU and data transmission queues, it sent an 'all done' message back to the MASTER transaction and did not wait on the Network Done Queue. When a local transaction completed, it sent a 'release locks' message to the PRIMARY site. When a MASTER had completed, however, it had to wait for all of the SLAVES to complete before sending the 'release locks' message to the primary site.

The two primary site models differed by the activities at the primary site. In the primary site one model (denoted PS1), a fixed ordering was placed on all of the sites and locks, for a transaction was acquired one site at a time in that order. In other words, a transaction would be granted locks for the first site, then the second site, etc. If the required locks for a given site were already held by a second transaction, the first transaction would wait for the second transaction to complete and re-request the locks for the given site. When the locks for all of the sites had been acquired, the primary site sent a 'locks granted' message back to the requesting transaction. Note that the fixed ordering of sites is used to prevent deadlock. Also note that for 'local' transactions locks were only requested at one site.

The primary site two model (PS2) differs from the PS1 model in only one respect. In the PS2 model, if the locks needed by a given transaction for a given site were already held, all of the locks a lower numbered sites (in the fixed ordering) that were granted to the given transaction were released. When the locks in contention were eventually released, the acquisition of the given transaction locks for all of these nodes had to be re-requested. Note that in the PS2 model, no transactions could hold locks for one node while waiting for other locks for another node. Also note that the PS2 model would favor those transactions which required locks at a fewer number of nodes. Thus the difference in the two primary site models is essentially one of transaction scheduling.

The other two concurrency control algorithms simulated were decentralized in that a concurrency control mechanism at each site managed the locks for the portion of the database at that site. In those models, a MASTER transaction sent the lock requirements for the SLAVE along with the 'SLAVE create' messages. A transaction requested its locks for a site when it left the pending queue (see Figure 1). If the locks were granted, the transaction could proceed. If the locks were denied, the requesting transaction waited for the blocking transaction to release its locks. Note that at a site, the locks for a transaction were either all granted or all denied.

When a local transaction had completed, it would simply release its locks and be recycled back to the pending queue. A non-local transaction, however, had to wait until its processing had completed at all of the nodes. Thus, in the decentralized concurrency control models, the SLAVE transactions had to wait on the Network done queue (again, see Figure 1) until they had received a 'release locks' message from their MASTER. At that point, the SLAVES could release their locks. The MASTER transaction waited for the 'all done' messages from each of the SLAVES before it could send those 'release locks' messages.

Unfortunately, in the above decentralized concurrency control models, deadlock is possible. Two transactions could each be waiting (directly or indirectly) at different sites for the other to complete. The two concurrency control models simulated differed in the way they solved the deadlock problem.

A wound-wait model (denoted WW), based on the algorithm presented in [ROSE77], prevented deadlock by using a unique timestamp for each transaction to resolve conflicts between distributed transactions (note that SLAVES had the same timestamp as their MASTER). In the WW simulation implementation, the following actions took place if one distributed transaction, say T1 was blocked by another distributed transaction, say T2: if T1 was older than T2, T2 was "wounded". When a transaction was wounded, the MASTER and all of the SLAVES were notified. If T1 is younger than T2, T1 simply waited for T2 to release its locks.

When a wounded transaction (a SLAVE or MASTER) was itself blocked by an older distributed transaction, the wounded transaction "killed" itself. The killing of a transaction involved the release of all locks by both SLAVES and MASTERS and the reincarnation of the MASTER transaction back on the pending queue.

The second decentralized control algorithm is based on the SNOOP [STON78] or the global detector [GRAY78] algorithms. In the SNOOP simulation implementation, a conflict between distributed transactions were reported to a "SNOOP" site which checked for deadlock. If deadlock was detected, a transaction was "killed" and reincarnated as in the wound-wait model.

4. Simulation Results

The simulation models were highly parameterized in order to provide insights into the effects of concurrency control on the performance of a wide variety of distributed databases. Simulation experiments were conducted varying the locking granularity, the number of nodes in the network, the number of SLAVES for each distributed transaction, and the number of distributed transactions. Different networks evirnonments were represented by varying the network speed, the network bandwidth, the messages hankdling overhead at the nodes and the percentage and rates for data transfer. The details of these experiments for all four concurrency control algorithms and two classes of transactions are presented in [RIES79].

In this paper, the discussion is limited to the parameters of greatest significance including the locking granularity, the percentage of distributed transactions, the network bandwidth, the types (or classes) of transactions and the concurrency control algorithms.

For all of the experiments reported, the other parameters were set to simulate the following scenario. Ten transactions were active at each of the six nodes in the distributed database. Each node contained 10,000 entities of the database where an entity can be thought of as the unit of data moved between the operating system and the database management system. It took a transaction 30 milliseconds on both the I/O and CPU queues to process one entity and 3 milliseconds of CPU time to set one lock. It took 15 milliseconds of CPU time to check for a deadlock condition. The number of entities and locks required by a transaction depended on the transaction class and is discussed below.

Each distributed transaction had 5 SLAVES and was thus active at all nodes in the network. Forty percent of these transactions transferred 25% of their entities across a megahertz data transfer network. To transfer a simple message across the network took 90 milliseconds.

Under the above scenario, the parameters shown in Table 1 were varied. Locking granularity, the LGRAN parameter, refers to the number of locks at each node. A value of 1 would imply that there were 6 locks - one for each node in the database. A value of 10,000, on the other hand, implies that each entity has its own lock and allows for the maximum potential parallelism.

The transaction class parameter, TCLASS, actually represents a set of parameters governing the transaction sizes and lock placement assumptions. With "Class 1" transactions, a hyperexponential distribution of the number of entities accessed by the transactions was used. Ninety percent of the transactions accessed on the average 5 entities of the database while the other 10% accessed on the average 250 entities. With "Class 1" transactions, the locks were considered to be well-placed, in that a tranasaction required the minimum number of locks that could cover the entities accessed by the transaction. This class of transactions implies that the transaction access paths are all sequential, most of the transactions are small, but a few are relatively large in terms of the proportion of the database they access.

With "Class 2" transactions, all of the transactions are accessed, on the average, only 5 entities in the database and a random placement of locks was assumed. The Class 2 transaction environment implies that the data access patterns are primarily random and that all of the transactions are small.

The PREDIST parameter controls how many transactions were non-local. A value of zero, for example, would imply that all of the transactions were local. A value of 100 percent implied that each of the 10 transactions at each node was a MASTER transaction and would spawn 5 SLAVE transactions at the other 5 nodes.

The message bandwidth parameter, MESBDNT, represents the number of messages which can be simultaneously processed by the network manager. The four concurrency control algorithms have already been discussed.

The results of varying the locking granularity, the percentage of distributed transactions, and the message bandwidth for the four concurrency control algorithms and the two transaction classes is discussed below.

1.1. Locking Granularity

Figure 3 shows the effects of varying the locking granularity on the "Useful I/O" utilization for each of the four concurrency control algorithms when 10% of the transactions were distributed and Class 1 transactions were

Table 1

Parameter	Description
LGRAN	No. of locks at each node
TCLASS	Transaction class
PREDIST	Percentage of the Transactions which are distributed
MESBDWT	Message Bandwidth
CCALGORITH	Concurrency Control Algorithm

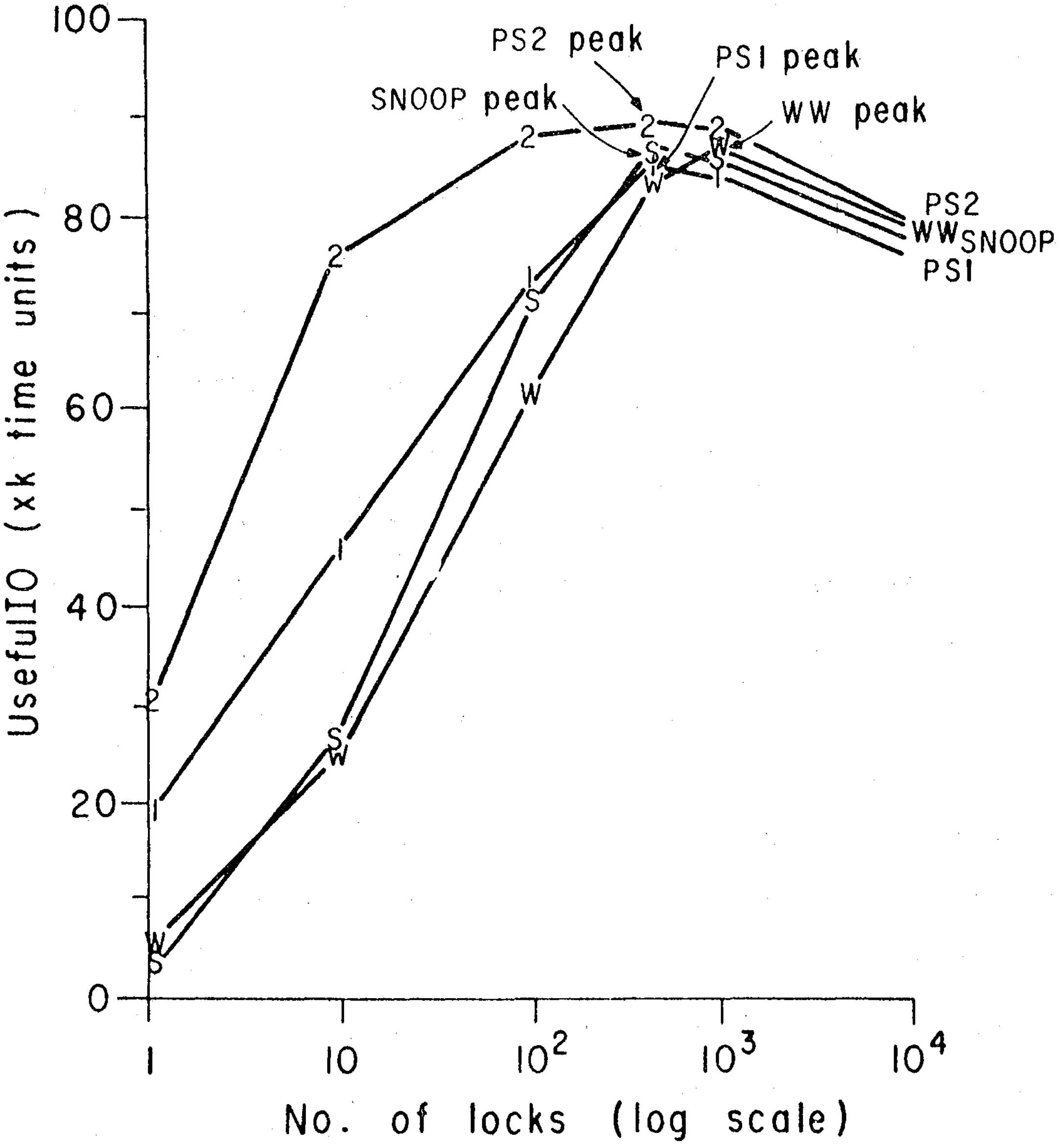


Figure 3: Class 1 Transactions
: PREDIST 10
: MESBDWT ∞

assumed. The "Useful I/O" refers to the net utilization of the I/O resources for processing transactions. The curves for "Useful CPU" utilization were similar and not shown. For all four concurrency control algorithms, the maximum useful I/O occurred with 500 to 1000 granules. For the primary site 2 (PS2), the primary site 1 (PS1), and the global deadlock detector (SNOOP) models, the peak occurred at 500 granules. For the wound-wait (WW) models, 1,000 granules were optimal. In either case, 99% of the maximum I/O utilization was reached with 500 or 1000 granules.

Several observations about figure 3 should be noted. First, the primary site two model (PS2) achieved 98% of the maximum I/O utilization with 100 granules and 90% of that maximum with as few as 50 granules. Each of the other three models required at least 250 granules to reach 90% of its respective maximum. In the primary site 2 model, no transactions held locks at one node while waiting for locks at another node. In each of the other models this condition was not true. Also note that the differences in the performances of the differenct concurrency control models was very small at the optimum granulaties.

The computer utilization for each of the four concurrency control algorithms for class 2 transactions are shown in Figure 4. Under the randomly placed locks with only small transactions, the finest granularity, 10,000 locks in this case, was optimal. With this optimal granu-

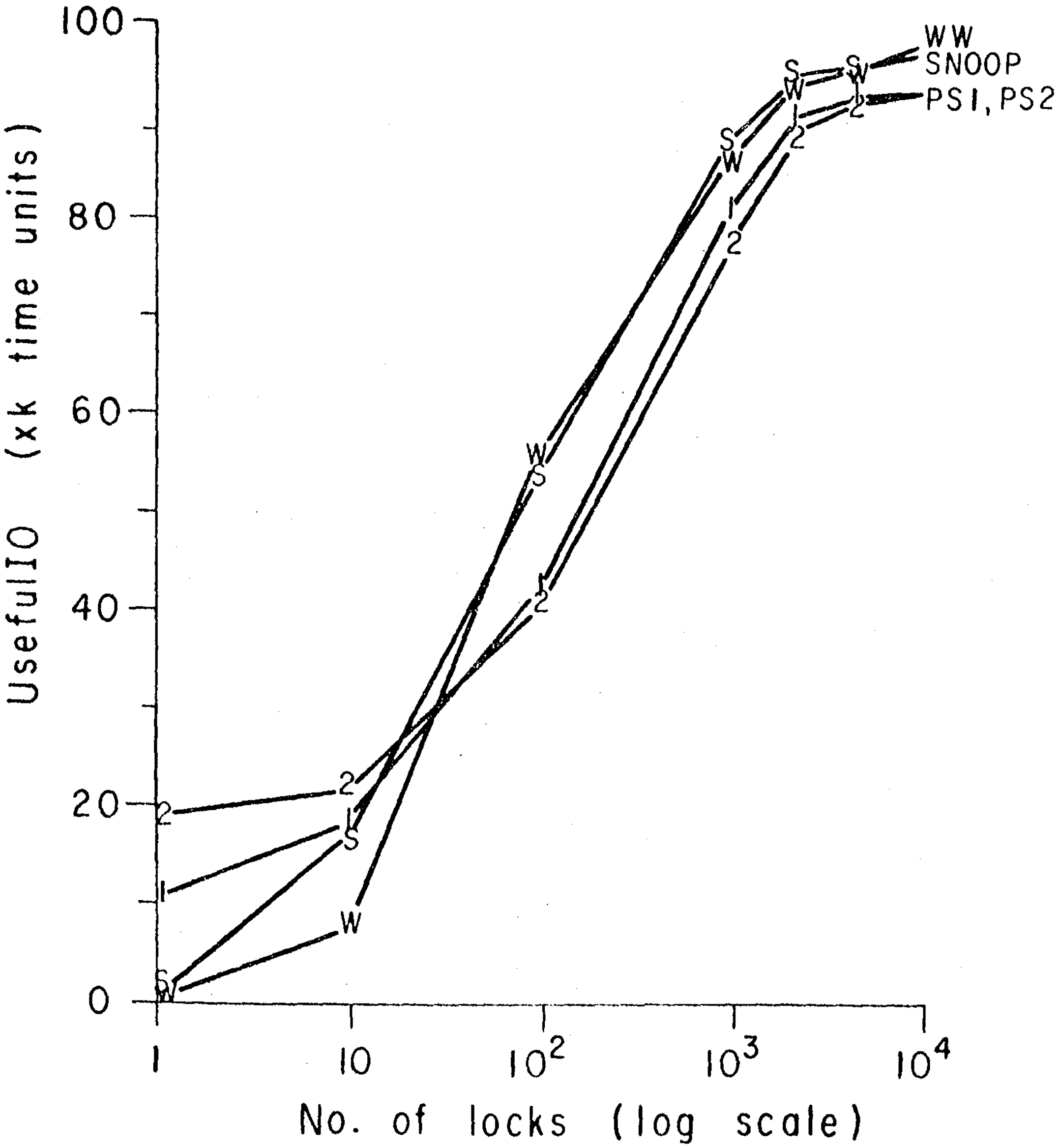


Figure 4: Class 2 Transactions
: PREDIST 10%
: MESBDWT ∞

larity, as with class 1 transactions, only slight differences in computer utilizations were due to the concurrency control algorithms.

However, the wound-wait and global deadlock detector algorithms did consistently produce somewhat better results than the primary site algorithms over a wide variety of granularities. In fact, only with fewer than 50 locks at each node, were the primary site models advantageous.

No difference in computer utilization was observed between the two primary site models once the granularity became fine enough. This result was true for class 2 transactions, since the probability of success on a lock request was extremely high. Thus, very few of these transactions waited for locks at one node, while holding locks at another node.

Similarly, once the granularity was less coarse (about 50 granules), little difference in computer utilization is realized between the two decentralized algorithms. This result was also realized because of the high probability of success on a lock request.

1.2. Percentage of Distributed Transactions

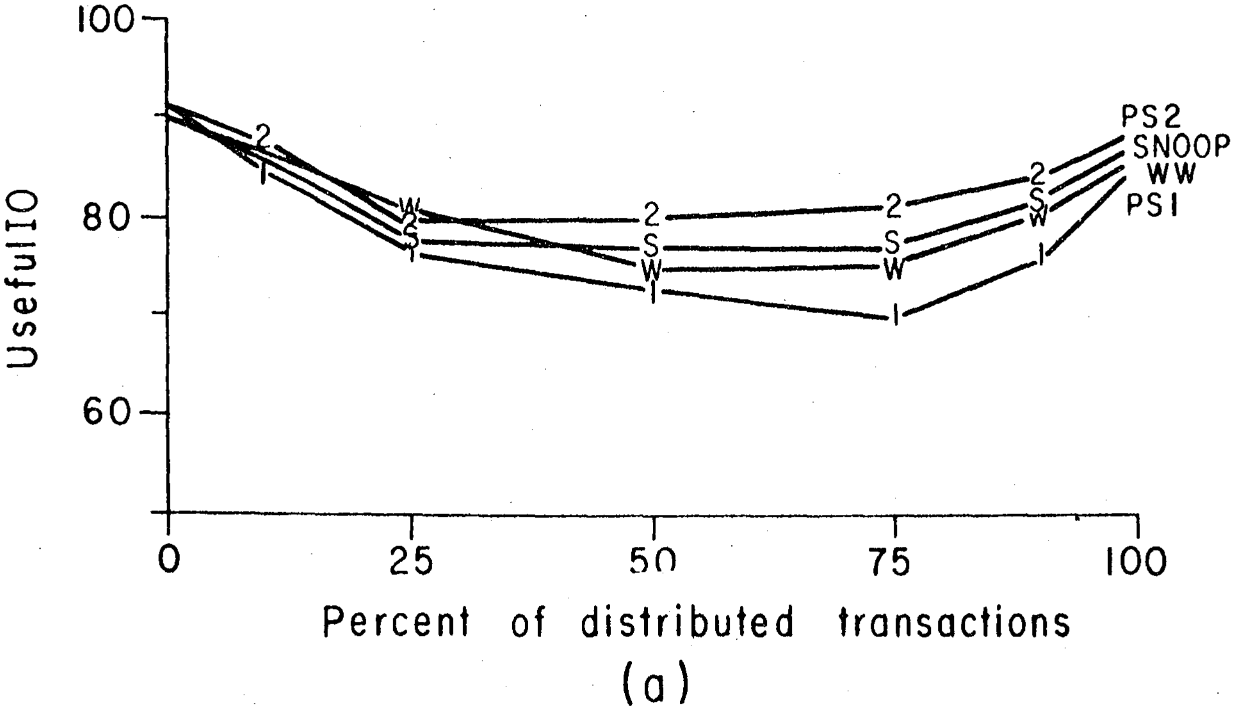
Changes in the percentage of distributed Class 1 transactions affected the optimum granularitios and the choice of a "best" algorithm. In general, finer granularity was required to achieve the best computer utilization and response times for the PS1, WW and SNOOP models. However, with the PS2 model 500 granules was always close to optimal.

Figure 5 shows the effects on the useful I/O and the average response time of the percent of distributed transactions for each of the four concurrency control algorithms. (For each percentage, and for each algorithm, the best useful I/O and average response time regardless of granularity was plotted.)

The 'dish' shaped curves for I/O utilization were surprising. As the percentage of distributed transactions was increased up to 50%, all four models showed decreases in useful computer utilization due to the additional overhead (message handling and locking) required to run distributed transactions. However, as the percentage increased beyond 75%, the useful computer utilization significantly increased.

That increase was due to two factors. First, the number of transactions running at each node was greatly increased. For example, when all of the transactions were distributed, parts of transactions were active at each node. Second, the average transaction size at each node was smaller as more and more transactions were distributed.

The simulation parameters were modified to keep the number and sizes of active transactions at each node constant as the percentage of distributed transactions



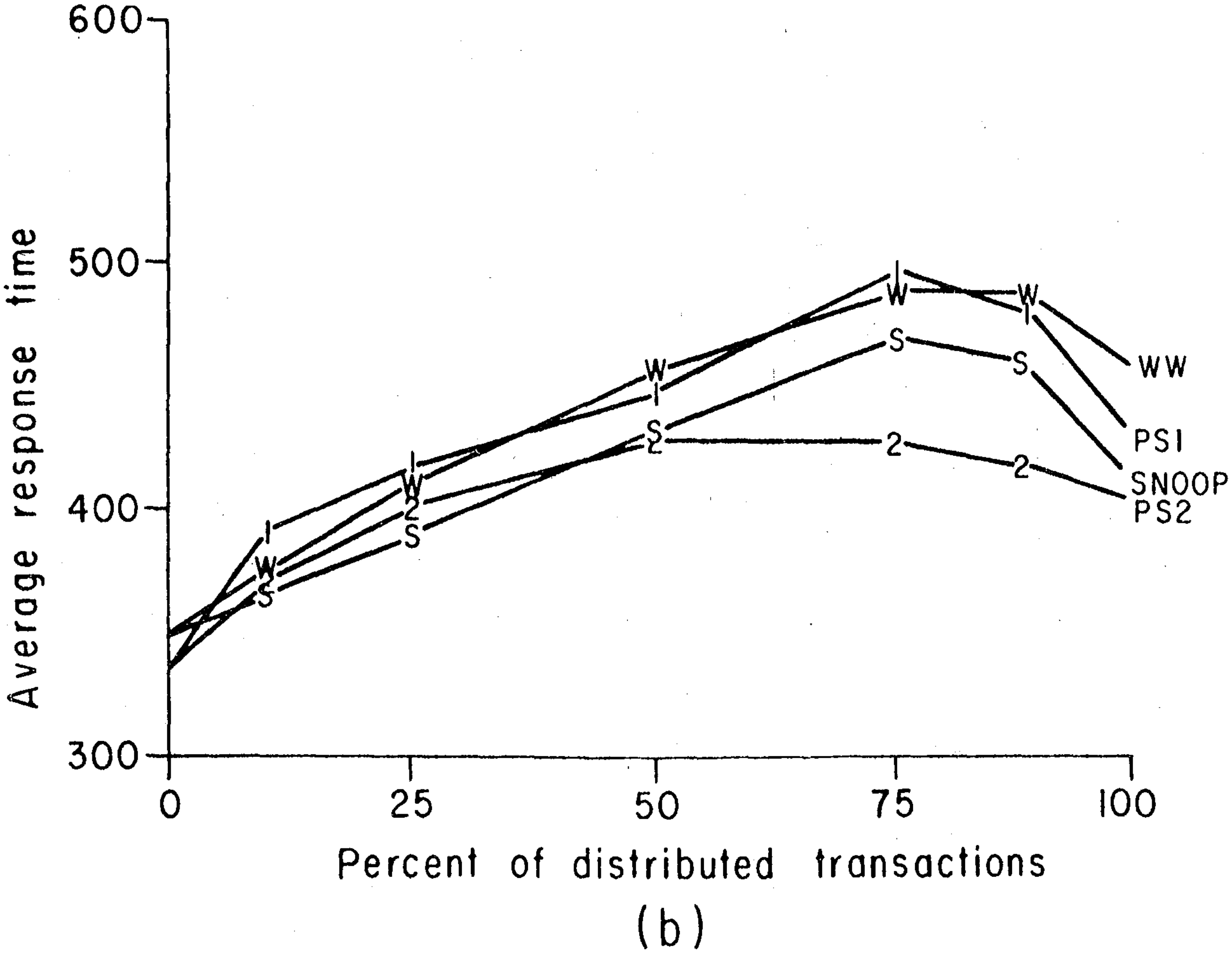


Figure 5: Class 1 Transactions
: MESBDWT

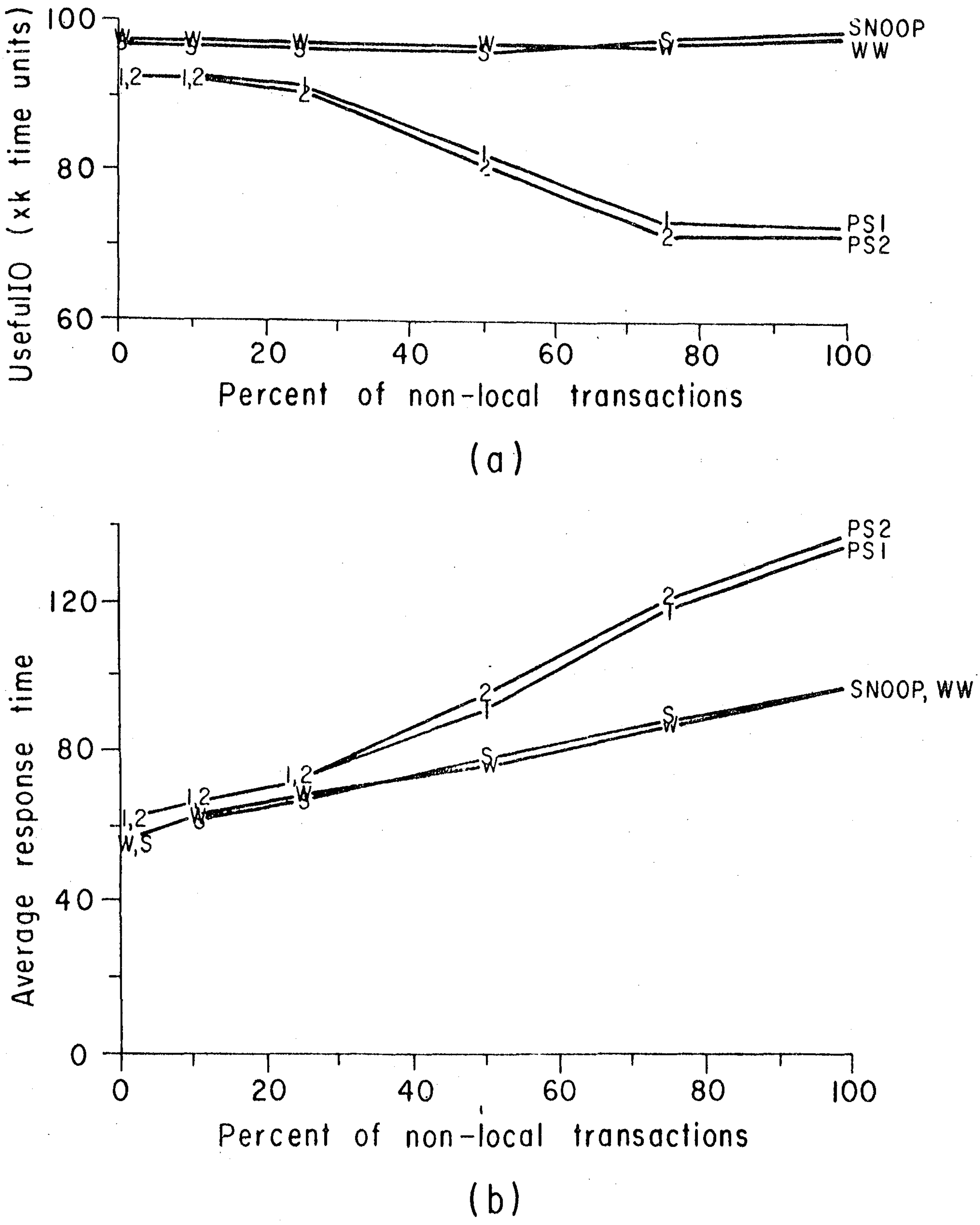
increased. Only when both parameters were held fixed did the 'dish' shaped curves disappear. When only one of the parameters were held constant, having all transactions distributed produced more useful I/O (and CPU) than when only 50% of the transactions were distributed.

The average response time curves also demonstrated dish shaped curves. In almost all cases, the second primary site model (PS2), produced the best average response time of the four models. The holding of locks at one node while waiting for locks at another was quite detrimental to the throughput of the system and occurred with increasing frequency in the other three models as the percentage of distributed transactions increased.

With class 2 transactions, the finest granularity was optimal, regardless of the percentage of distributed transactions. Furthermore, the performance of the concurrency control algorithms also changed consistently as the percentage of distributed transactions increased.

Figure 6(a) shows the I/O utilization for the four algorithms as that percentage increased. The utilization with the decentralized algorithms was affected very little by the increase in non-local transactions. Again, a slight increase in useful computer utilization was realized due to the increased distribution of transaction processing.

In the primary site algorithms, on the other hand, the overall computer utilization decreased as the percentage of



(a)

(b)

Figure 6: Class 1 Transactions
: MESBDWT ∞
: LGRAN 10,000

non-local transactions increased. The decrease was most dramatic between 25 and 75 percent.

The same advantage for the decentralized algorithms over the primary site algorithm appeared in the average response time, as shown in figure 6(b). For all four algorithms the response times increase as the percentage of distributed transactions increased. However, the increase was much less for the decentralized concurrency control algorithms than for the primary site concurrency control algorithms.

Two factors caused the dramatic difference between the primary site and decentralized models for class 2 transactions: the transactions were all small and the primary site created a bottleneck.

The transactions of class 2 were all small and the results in Figure 6 were for the finest granularity. Under those conditions, the probability of success on a lock request was extremely high, which considerably reduced the advantage that the primary site 2 model exhibited for class 1 type transactions.

The second factor which affected the performance of the concurrency control algorithms was the bottleneck at the primary site. Over 7,000 time units out of a possible 20,000 were used for locking at the primary site when all of the transactions were non-local. Moreover, all transactions required some database processing at that primary site and were thus all delayed by the locking overhead. This bottleneck became increasingly worse as the percentage of distributed transactions increased.

One solution to the bottleneck problem would be to offload the primary site concurrency control to a separate processor. The primary site 2 simulation was modified to test this strategy and in fact then produced results very similar to the decentralized models.

1.3. Message Bandwidth

The above observations changed when a lower network bandwidth was assumed. MESBDWT settings of 100, 50, 10, 6 and 1 were tested for each of the four concurrency control algorithms and each class of transactions.

For Class 1 transactions, 10% of which were distributed, MESBDWT settings of 100 and 50 produced useful computer utilizations and average response time identical to the infinite setting previously used. Slight drops in the useful I/O and CPU utilizations were realized with message bandwidths of 10 and 6. The drops with a message bandwidth of 10, however, were less than 1% and not considered significant.

A message bandwidth of 6 did produce more noticeable reductions in the useful I/O and CPU utilizations. The drops in useful utilization were only about 2-3% with the primary site and SNOOP models. The wound-wait model, on the other hand, realized a drop of almost 7%. Although the primary site models sent more lock messages, they were mainly sent one message at a time. A wound or kill, however, resulted in 5 messages being sent, or broadcast over the network. These "bursts" of messages were effected more by the lower bandwidth than the greater number of individual messages in the primary site models. In the SNOOP model, on the other hand, a conflict only required 1 message. A kill still required 5 messages, but occurred very rarely.

Figure 7 shows the effects of the PREDIST parameter on Class 1 transactions on a reduced bandwidth network. With fewer than 40% of the transactions being non-local, the global deadlock detector algorithm produced more useful I/O utilization than the other algorithms. When 45% or more of the transactions were distributed, the primary site 2 model again produced better results. In these cases, the extra two messages for locking were not that significant; a distributed transaction required at least 2 * 5(no. of SLAVES) messages anyway.

Note also that the 'dish' shape curves for Useful I/O have practically disappeared with a limited bandwidth network. In these cases the extra network delay overhead caused by an increased PREDIST parameter more than offset the increases in transaction parallelism.

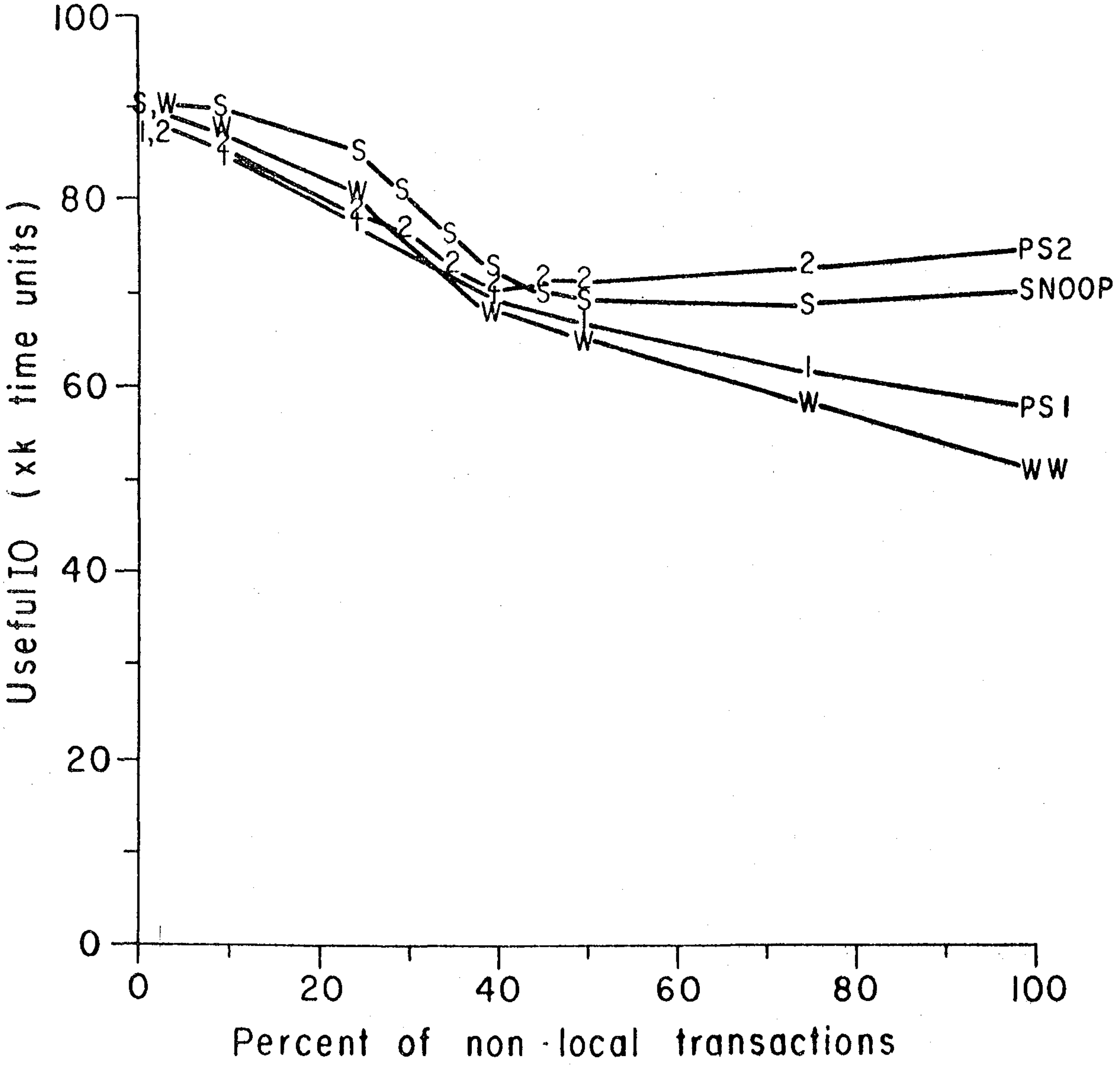


Figure 7: Class 1 Transactions
MESBDWT $\geqslant$ 6

For Class 1 transactions, only the message bandwidth parameter significantly affected the performance of the database under the four concurrency control algorithms. For Class 2 transactions, however, some of the other network parameters did effect the choice of concurrency control algorithms. The results are thus repeated for the message speed or time to send a simple message, MESRATE; the CPU time to process (send or receive) a message a site, MESCPURATE; as well as the network bandwidth, MESBDWT.

The I/O utilization and the average response time (in parenthesis) is given in Table 2 for each of the four concurrency control algorithms. In the first set, the MESRATE parameter was varied while the MESCPURATE and MESBDWT were fixed at .01 and 1000 respectively. As the message rate increases, the gap between the primary site and decentral-

Table 2: Effects of Network Parameters

	PS1	PS2	WW	SNOOP
MESRATE				
1	94994(63)	94720(63)	96839(61)	97037(62)
3	93996(64)	93319(64)	97134(61)	96204(62)
10	87998(67)	88078(67)	96037(63)	96875(62)
MESCPURATE				
.01	93996(64)	93319(64)	97145(65)	96204(62)
.05	88953(67)	88767(68)	95048(63)	94710(64)
.1	83273(72)	83086(73)	92394(65)	91860(65)
.3	58676(102)	58372(102)	83313(72)	82690(73)
MESBDWT				
1000-50	93996(64)	93319(64)	97145(61)	96204(62)
10	82804(72)	83234(72)	96827(62)	96979(62)
6	55200(108)	55692(108)	95948(63)	96242(62)

ized control models widened. A MESRATE of 1 can be interpreted as requiring 30 milliseconds to send a message.

A more dramatic change occurred when the message CPU rate was varied. During these experiments, the MESRATE and MESBDWT were fixed at 3 and 1000 respectively. With a 3 millisecond cost (MESCPURATE = .1) for sending a message, the primary site models produced only 89% of the useful computer utilization that was realized with the decentralized concurrency control algorithms. With a 9 msec message rate (MESCPURATE = .3) this percentage drops to 72%.

A dramatic change in computer utilization and response time for the primary site models and Class 2 transactions was realized as the message bandwidth was restricted. While the performance of the primary site models was heavily affected by the restricted bandwidth, the decentralized models were hardly affected at all. This result is due to the fact that with the primary site models, almost 40,000 more messages were sent than with the decentralized algorithms.

The PREDIST simulation experiments for class 2 transactions were repeated with a limited bandwidth network. In these experiments, the primary site models were best if more than 50% of the transactions were distributed. In those cases, the primary site models actually sent fewer locking messages than the decentralized algorithms. However, if fewer slaves were used, the decentralized algorithms would send fewer messages even in 100% of the transactions were distributed.

5. Summary

Four concurrency control algorithms thus were simulated in order to study their effects on the performance of a distributed database management system under a variety of database and network conditions. Which model was best in terms of the overall database system performance is application dependent as shown in Table 3. Class 1 transactions refer to a workload environment where the locks are assumed to be well-placed with respect to the accessing transactions and that those transactions are of mixed sizes. Class 2 transactions refer to workloads where all of the transactions are small and random placement of locks is assumed.

In some cases, it appears that the concurrency control mechanism is not a significant factor in the database system performance. For class 2 transactions, additional simulation runs showed that the preference for decentralized concurrency control could be offset by reducing the database load at the primary site. Thus in these cases, the choice of concurrency control algorithm may again not be significant.

For class 1 transactions, when most of the transactions only required local processing and a slower, lower bandwidth

Table 3: Concurrency Control Models

	Class1 Transactions	Class2 Transactions
Fast Net. Most trans. local	Primary Site or Decentralized	Primary Site or Decentralized
Slow Net. Most trans. local	SNOOP	Decentralized
Fast Net. Most trans. non-local	Primary Site 2	Decentralized
Slow Net. Most trans. non-local	Primary Site 2	Primary Site

network is assumed, the SNOOP algorithm is preferred. In this case, the SNOOP model was favored because of the lower number of messages required.

REFERENCES

ALSB76 Alsberg, P.A., Belford, G.G., Day, J.D. and Grapa, E., "Multi-copy Resiliency Techniques", CAC Doc.202, Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1976.

BERN77 Bernstein, P.A., Shipman, D.W., Rothnie, J.B., and Goodman, N., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases", Technical Report CCA-77-09, December 1977.

ELLI77 Ellis, C.A., "A Robust Algorithm for Updating Duplicate Databases", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California, pp. 146-158.

EPST78 Epstein, R., Stonebraker, M., and Wong, E., "Distributed Query Processing in a Relational Data Base System", ACM SIGMOD International Conference on Management of Data, Austin, Texas, pp. 169-180.

ESWA76 Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, L. I., "On the Notions of Consistency and Predicate looks in a data base System ", Communications of the ACM, Vol.19, No.11, November, 1976. pp. 624-633.

GRAY75 Gray, J.N.,Lorie, R.A., and Putzolu, G.R. "Granularity of Locks in a Shared Data Base", Proc. 1975 VLDB Conference, Framingham, Mass., Sept., 1975. pp. 428-451.

GRAY76 Gray, J. N., Lorie, R. A., Putzolu, G. R. and Traiger, I. L., "Granularity of Locks and Degrees of Consistency in a Shared Data Base." Proc. IFIP Working Conference on Modelling of Data Base Management Systems, Freudenstadt, Germany, January 1976. pp. 695-723.

GRAY78 Gray, J., "Notes on Data Base Operating Systems", IBM Research Report, RJ 2188, San Jose, California, 1978.

MENA78 Measce, D.A. and Muntz, R.R., "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August, 1978, San

Francisco, California, pp. 215-232.

RIES77 Ries, D. R., Stonebraker, M. "Effects of Locking Granularity in a Database Management System", ACM Transactions on Database Systems, Vol.2, No.3, September, 1977 pp. 233-246.

RIES79 Ries, D.R., Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems, Vol.3, No.2, June, 1979.

RIES79A Ries, D.R., "The Effects of Concurrency Control on Database Management System Performance", Ph.D. Thesis, University of California, Berkeley, March, 1979.

ROSE77 Rosenkrantz, D.J., Teams, R.E., and Lewis, P.M., "A system Level Concurrency Control for Distributed Database Systems", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California, pp. 132-145.

STON77 Stonebraker, M. and Neuhold, E., "A Distributed Database Version of INGRES", Proceedings of the Second Berkeley Workshop on

Distributed Data Management and Computer Networks, May, 1977, Berkeley, California, pp. 19-36.

STON78 Stonebraker, M., "Concurrency Control of Multiple Copies of Data in Distributed INGRES", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August, 1978, San Francisco, California, pp. 235-258.

THOM78 Thomas, R.A., "A Solution to the Update Problem for Multiple Copy Databases which uses Distributed Control", BBN Report 3340, July 1978.

WONG77 Wong, E., "Retrieving Dispersed Data from SDD1: A System for Distributed Databases", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California. pp. 217-275.

A CONCURRENCY CONTROL MECHANISM FOR DISTRIBUTED DATABASES WHICH USES CENTRALIZED LOCKING CONTROLLERS

Hector Garcia-Molina*

Computer Science Department
Stanford University
Stanford, California 94305

Abstract

In this paper we present a new efficient concurrency control mechanism for distributed databases. This general concurrency control mechanism is based on the idea of having a centralized locking controller for each replicated fragment of data. The independent centralized controllers operate without explicit backup controllers. A simplified two phase commit protocol is used to perform updates. In this protocol, only a majority of acknowledgments from the copies of a fragment for the "prepare" (first phase) messages is required before committing new data. The major protocols required for the concurrency mechanism are outlined. These include the transaction cancelling protocol and the new controller election protocol.

1. THE MODEL.

In order to discuss transaction processing and concurrency control, we first define a simple model of a distributed database [5]. We view the database as a collection of named items. Each item has a name and some values associated with it; each value is stored at a different node in the system. In addition, each item i has associated with it a set S(i). Set S(i) is the set of nodes which have a value for item i stored in them. We assume that all sets S(i) are not empty. We represent the values associated with item i by d[i,x], where x is a node in S(i). (For nodes y not in S(i), d[i,y] is undefined.) The values for a given item i at different nodes should be the same (i.e., d[i,x] should equal d[i,z] for all nodes x, z in S(i)). However, due to the updating activity, the values may be temporarily different.

We can group items that have identical storage characteristics into "fragments". A fragment F is a set of items that have the same S sets. We use the notation S(F) for the set of nodes where F is stored. (That is, S(F) = S(i) for all items i in F).

----

* Author's current address: Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N. J. 08540

Operations on the data are grouped into transactions [2]. A transaction T first specifies a subset of items it wants to read. The transaction does not indicate where the items are to be read; it is up to the system to select one of the available values for each item specified by T. Based on the values read, transaction T performs some computations and proceeds to update some items. In this final step, T produces a set of new values for a subset of items. For each item i updated by T, the system must make sure that the new value for i produced by T is stored at all nodes in S(i). Notice that the data reading and computing phases of T may be interleaved. Also notice that transactions do not necessarily update data. However, to simplify the discussion, we assume that all transactions are update transactions. The concurrency control mechanism of the system must guarantee that the effect of running transactions concurrently is as if the transactions were run one at a time.

In this paper we concentrate on the concurrency control issues of transaction processing. We avoid two other important issues: directory management and transaction optimization. That is, we assume that the S set for each item (which is part of the directory) is known at all nodes. We also assume that a transaction can read the items it needs in any order and at any node that has the values available. The directory information, which constitutes a distributed database in itself, can be updated (e.g., a new node can be added to an S set). However, the concurrency control mechanism for this directory information is different from the concurrency control mechanism we discuss in this paper because more safeguards must be taken when modifying the directory. We will not discuss directory updating here.

2. A CONCURRENCY CONTROL MECHANISM.

In this section we will illustrate a common concurrency control mechanism for transaction processing [7] through an example. (The description is simplified and we omit many details.) Suppose that item i is duplicated at nodes x1 and x2, while item j is replicated at nodes x2, x3 and x4. That is, S(i) = {x1,x2} and S(j) = {x2,x3,x4}. A transaction T wishes to read item i and then update item j. The way T is processed is by having T "visit" nodes x1, x2, x3 and x4 requesting locks for the referenced items. Each node in the system has locks associated with each value stored at the site. When a node grants a lock to a transaction, it gives the transaction exclusive access to the value (until the lock is released). Thus, after T obtains locks for d[i,x1], d[i,x2], d[j,x2], d[j,x3] and d[j,x4], it can compute the new values for item j without any interference from other transactions.

When transaction T has computed the new value for item j, the system updates j and releases the locks through a two phase commit protocol. In the first phase of this protocol, a "master" node (which can be any node) sends out "prepare" messages with the value for j and the lock release information to all the nodes that participated in T (i.e., x1, x2, x3 and x4). When these sites receive the information, they save it but do not update j or release any locks. Instead, they acknowledge receipt of the information to the master. After having received acknowledgments for all participating nodes, the master starts the second phase of the protocol by sending out "commit" messages to all sites involved. (The time when T obtains all the necessary acknowledgments is called the commit point.) When a node receives a "commit" message for T, it actually releases the locks held by T and stores the new value for item j (except node x1 which does not have a value of j). The two phase commit protocol guarantees that T terminates correctly at all nodes.

3. A NEW CONCURRENCY CONTROL MECHANISM.

We propose a variation of this transaction processing mechanism which we believe has several important advantages over the mechanism we have just described. The main difference is in the way we propose to handle replicated data in the system. The motivation for such a mechanism comes from a performance analysis [6] which indicates that a centralized control strategy for managing replicated data is superior to a distributed control strategy. Notice that in the mechanism of section 2, the control of an item i is distributed among the nodes in S(i). That is, each node in S(i) has a lock for the value of item i stored at the node, and in order to access the item, a transaction must secure all locks for the item. We will replace this control structure by creating a central "controller" for item i which can grant exclusive access to the values of item i at all nodes.

The idea of centralized control is not new. Alsberg and Day [1] suggested having a primary site for executing all update transactions. In the mechanism we are proposing, only the control of the data (i.e., the locks) is centralized; reading the data needed and performing the computations for a transaction can be done at other nodes in the system. This reduces the load at the central site. In turn, this can improve performance because the central site is usually a "bottleneck". Menasce et al [8] have also suggested the use of a central controller. Their lock controller is a unified control structure for the entire system; here we propose a collection of independent controllers. The lock controller in [8] has "local" controllers which act as backups for the main controller. In our system, we do not have backup controllers. When one of the controllers fails, we do not reconstruct its lock information. Instead, we either cancel or successfully terminate all pending transactions that involve the failed controller. This strategy eliminates the overhead associated with backups.

4. AN EXAMPLE.

Before we proceed, we illustrate how we propose to process transactions with the example we used in section 2. Recall that item i is replicated at two nodes (S(i) = {x1,x2}), while item j is replicated at three nodes (S(j) = {x2,x3,x4}). We select a controller for item i, C(i). Controller C(i) is a "module" which can be located anywhere in the system, but for convenience we will assume that it is located at a node in S(i). Suppose that C(i) is located at node x2. Similarly, assume that the controller for item j, C(j), is located at node x4.

Transaction T reads item i and updates item j. To process T, we make T "visit" controllers C(i) (at node x2) and C(j) (at node x4) and request locks for those items. After obtaining locks at both controllers, T has exclusive access to the two items and can proceed. (Notice that controllers C(i) and C(j) grant their locks without sending any messages to backup nodes.)

Once T has computed the new value for item j, the system performs the update and releases the locks using a modified two phase commit protocol. In this protocol, the master (which can again be any node) sends out prepare messages informing all nodes involved in T (i.e., x1, x2, x3, x4) that T has completed. But now, the master only has to wait for a majority of acknowledgments from each S(i) set involved. For example, if the master gets acknowledgments from nodes x1, x2 and x3, then it can send out the commit messages because a majority of nodes in each set S(i), S(j) have responded. When a node receives a commit message, it updates item j if it has a copy of the item. If the node has a controller involved in T, then the commit message also causes the locks to be released. Notice that no acknowledgment is necessary for the commit message.

Due to failures, some nodes that participated in T may not find out about T's completion (e.g., node x4). These nodes will eventually discover that they missed this information because of a sequence number mechanism. (See section 6.3.) When a node discovers this, it obtains the missing information from other nodes. If the information cannot be found, the node attempts to cancel T. (See section 6.6.)

5. ADVANTAGES OF THE PROPOSED CONCURRENCY MECHANISM.

The main advantages of the concurrency control mechanism we propose are: (1) There is no need to lock an item at all nodes where a copy of its value exists, (2) In the two phase commit protocol, only a majority of acknowledgments (for each item referenced) are required, (3) No explicit backup of the controllers and their lock information has to be maintained, and (4) Operation with missing nodes is straightforward because a transaction that references item i can complete even if a minority of the nodes in S(i) are unavailable.

The main disadvantages of our concurrency mechanism are: (1) When a central controller fails, transactions involving the controller are temporarily halted until a new controller is elected. In the process, some transactions may be cancelled or aborted, and (2) The mechanism is more complex than the one described in section 2. Thus, we are not proposing our solution as the best for all cases. Our solution is an interesting alternative which may be well suited for some cases. In particular, our mechanism seems to be attractive for cases where performance is important, where data replication is common, and where we expect failures to be rare.

6. AN OUTLINE OF THE MECHANISM.

Up to this point, we have only given a very informal description of the concurrency control mechanism, omitting most of the details. In the rest of this paper, we will attempt to convince the reader that such a mechanism can operate correctly even in the face of (detectable) failures. In the limited space available, we will give an extremely brief outline of the major concepts and protocols that are required in our mechanism. In [4] we discuss these ideas in detail. In that report we also give a fairly detailed description of the concurrency mechanism for the case of a single controller. The mechanism we present here is simply an extension of the one controller case given in [4].

6.1 Controllers.

The basic idea in our concurrency control mechanism is that each item i in the database has associated with it a controller C(i). Several items can have the same controller. In other words, each controller J resides at a node N(J) and manages the locks for the items in the set I(J). For simplicity, we assume that all items that share a controller (i.e., the items in I(J)) are replicated at the same set of nodes. That is, a controller is always in charge of a fragment of the data. (See section 1.) We use the notation C(F) for the controller of fragment F (i.e., C(F) = C(i) for all i in F).

Each node in S(F) must know where the (current) controller C(F) is located. (This location amy change in the controller node crashes. See section 6.7.) The location of C(F) may also be placed in the system directory so that nodes not in S(F) may find C(F). However, this directory information need not be current because if the controller is not found, any node in S(F) can be interrogated to discover the true location of the controller.

6.2 The majority of nodes requirement.

In order to avoid the serious problems that arise when a network is partitioned, we will require that a majority of nodes in S(F) be active and able to communicate with each other before any transactions involving F are processed. This restriction is embedded in the commit protocol because a transaction needs a majority of acknowledgments from nodes in F before any update involving F can be committed. This restriction is also embedded in the new controller election protocol (section 6.7) because only a majority of nodes in S(F) can elect a new controller C(F) in case the old one fails. No controller C(F) can be in operation if it cannot communicate with a majority of the nodes in S(F).

6.3 Sequence and version numbers.

Another important concept in the concurrency control mechanism is the use of sequence and version numbers. Each transaction T that requests locks from C(F) receives a sequence number. This number must be appended to all messages generated by T. This sequence number plays an important role because it is used to order the operations of T with respect to the operations of other transactions. For example, if T received sequence number 15 from C(F), T must wait until all transactions with a sequence number less than 15 are processed at node x before T can read data from F at node x. To eliminate unnecessary delays, additional sequencing information can be assigned to T by C(F). For example, C(F) can give T a "wait for" list which includes the sequence number of all previous transactions that conflicted with T. This way, nodes that perform operations of T only have to wait until they finish processing transactions in this list [3]. (Sequence numbers also play an important role in crash recovery. See sections 6.4 and 6.7.)

Since a fragment F may have several different controllers over time, it is necessary to distinguish between these controllers and the transactions that they authorized. (Of course, at any given instant, there can only be one controller for F.) Version numbers are used to differentiate controllers of F. A unique version number is associated to each controller of F, and this number is appended to each sequence number generated by the controller. All active nodes in S(F) are aware of the current version number, and are thus able to detect any transactions whose locks were not granted by the current controller. (See section 6.7.)

When a transaction T spans several controllers, all the version and sequence numbers obtained by T at the controllers are included in the messages generated by T. Each sequence, version number pair carries with it an indication of what fragment it corresponds to.

6.4 Update logs.

Any distributed database system needs a mechanism for recording completed transactions. To see this, consider what happens when a node in S(F) crashes. (Assume that the controller C(F) was not at that node.) Since this node will be out of operation, it will miss a set of updates. This means that somehow the rest of the system will have to save these updates for the crashed node. There are many alternatives for doing this.

One solution is to use update logs. An update log is a collection of performed updates that is kept safely at a node. Each log entry contains the database values that were modified by a transaction, plus the sequence and version numbers of the transaction. For simplicity, in our system we assume that a log is kept at each node. Each such log keeps track of all the updates processed at that node. (It is also possible to operate with fewer logs but we do not consider this case here.) When a node x recovers from a failure, it brings each fragment F stored in x up to date by requesting the missed updates from the logs at other nodes in S(F). Sequence and version numbers are very helpful here because the recovering node knows exactly what updates it missed.

6.5 The two phase commit protocol.

When a transaction T is ready to store values into the database, it uses the modified two phase commit protocol described in section 4. This guarantees that either no values are stored at all or that all values produced by T are eventually stored at all nodes involved. When a node in S(F) acknowledges receipt of the prepare message for T, it makes a commitment to remember T (and the values it produced) and to do everything in its power to see that T completes correctly. The node remembers T by placing the information in the prepare message in a "prepare" list. We assume that the information in this list cannot be lost. (Log entries can be made to make the prepare list safe. In [7] we discuss what happens when this and other state information is destroyed.)

When the master node for T receives a majority of acknowledgments from the nodes in S(F), it knows that the update to F cannot be lost. In the case of failures, we know that at least one member of any working majority of nodes in S(F) will have a record of T and will "speak up" for T. Thus, after receiving a majority of acknowledgments from the nodes in each S set involved in T, the master node can send out the commit messages. When a node in S(F) receives a commit message, it adds T's sequence and version numbers to its list of performed transactions (which is kept by all nodes); it writes out a log entry; it performs the update on F indicated by T; and finally it removes T from the prepare list.

Due to failures, a transaction may be unable to get the majority of acknowledgments needed for committing. In such a case, the transaction "times out" and the system attempts to cancel the transaction. This cancelling protocol is described in the next section.

6.6 The transaction cancelling protocol.

In many cases a transaction will have to be cancelled. One such instance is when a deadlock occurs and a transaction must be backed out. Another case occurs when a transaction which holds locks fails to release its locks. For example, a transaction may have been computing at a node which crashed. In this case, the transaction must be cancelled and its locks reclaimed.

A transaction will only be cancelled if no data has been committed by the transaction. Thus, the first step in the cancelling protocol is to verify that the transaction had not reached the commit point. Notice that if a transaction T has reached the commit point, then a majority of nodes in each S(F) set, for each fragment F referenced, have a record of T. Hence, if a single fragment F can be found where a majority of nodes in S(F) have no record of T, then T can be cancelled.

To cancel a transaction T we proceed as follows. First, a node w is selected to be the master node for the cancellation. Any node can be the master, and several such nodes may be attempting to cancel T concurrently. We assume that node w knows that T referenced fragments F1, F2, ..., Fk. (The protocol can easily be modified to handle the case where only one fragment is known initially.) Node w sends out messages to controllers C(F1), C(F2), ..., C(Fk) asking them if they can cancel T. Each controller responds either that T can be cancelled or that it does not know if T can be cancelled. Controllers do not take any action on T at this point. However, if a controller says that T can be cancelled, it makes sure that T can not reach the commit point in the future.

When node w receives answers from all controllers, it decides if T will be cancelled. If at least one controller said that T could be cancelled, then T has not committed and is cancelled. If all controllers say that they do not know if T can be cancelled, then T may have committed and node w attempts to complete T. (Notice that in this case all controllers found a record of T. Thus, all the update values produced by T are known and T can be completed.) The decision of node w is broadcast to all controllers, which then carry out the decision.

When a controller C(F) wishes to know if T can be cancelled (in response to node w's first message), C(F) sends out "propose to cancel T" messages to all nodes in S(F). When a node y in S(F) receives the "propose to cancel T" message, it checks to see if it has a record for T. That is, node y checks if it has previously received a prepare or a commit message for T. If y has such a record, it informs the controller. If y has no record of T, then it sends a "have seen proposal to cancel T" message to C(F). With that message, node y makes a commitment not to acknowledge any prepare messages for T it might receive later. Thus, node y remembers the "propose to cancel T" message until it hears from the controller again. (We assume that node y cannot forget its commitment.)

If C(F) receives a majority of "have seen proposal to cancel T" messages, then C(F) knows that T has not committed and that T will not commit in the future. Thus, C(F) can answer node w that T can be cancelled. On the other hand, if C(F) discovered a record of T among the nodes in S(F), then it must answer that it does not know if T can be cancelled because as far as it knows, T could have committed. In this case, T's record (including its update values) is sent to w.

When controller C(F) receives a command from node w to actually cancel T, it does this using a two phase commit protocol similar to the one used by transactions to commit. This guarantees either that T is cancelled at all nodes in S(F) (as far as F is concerned) or that T is not cancelled at all. A node in S(F) finally cancels T by recording a null or dummy update. That is, T is processed as if T has committed, except that no values are stored in the database. Similarly, a command from w to complete T because it could not be cancelled causes C(F) to distribute the update values for T to nodes in S(F) and to commit them using a two phase commit protocol (with a majority of acknowledgments only).

A nice feature of the cancelling protocol we have described is that it can be interrupted and restarted anywhere without undesirable consequences. Thus, if the cancellation master node w or any of the controllers crashes in the middle of the cancellation, the procedure can simply be abandoned and then restarted by any node that notices that T is still pending.

6.7 The election protocol.

When a controller C(F) fails, a majority of nodes in S(F) elect a new controller. As nodes in S(F) detect that the controller is not active, they go into a special state where all normal processing is halted. (If a node x later finds out that C(F) did not really fail, then node x recovers as if it was the one that failed.) When a halted node discovers a majority of other halted nodes, it attempts to become the node with the new controller. One way to do this is to try to "lock out" all other nodes. If a node succeeds, it creates the new controller. If it fails in locking out the other nodes, then it must release all nodes it was able to lock out and must try again later.

A new controller is assigned a new version number different from all previous version numbers. Every node that participated in the election is given and records the new version number. Before the new controller starts operating, it must deal with the unfinished transactions left by the old controller. Since the old controller did not leave any backup information behind, it is impossible for the new controller to reconstruct the locking information that existed before. Hence, the new controller has to force the release of all locks by either cancelling or completing all outstanding transactions involving fragment F.

To do this, the new controller requests copies of all pending prepare messages in nodes in S(F), as well as the list of the committed transactions at these nodes.

Let s be the largest sequence number from the old version observed in this process by the new controller. If the new controller discovers that a commit message has been received somewhere in S(F) for a transaction T, then T has committed and must be completed using a two phase commit protocol (with a majority of acknowledgments only). The new controller attempts to cancel all other transactions with sequence numbers between 1 and s issued by the old controller. This is done with the cancelling protocol of section 6.6. Finally, notice that transactions with sequence numbers s+1, s+2, ... may have been authorized by the old controller, but no nodes in S(F) knew about these transactions before the crash of the old controller. Thus the new controller must also cancel all transactions with sequence numbers s+1, s+2, ... since they have definitely not committed. This is done through the version number mechanism. Since the new controller and all the nodes in S(F) now have the new version number, all uncommitted transactions (if any) with the old version number will be unable to commit because they can no longer get acknowledgments from the nodes in S(F). When these transactions time out because they cannot commit, they will be cancelled entirely.

As a last step, a new central controller makes an entry into the logs indicating what the largest sequence number of the old version was. This information is used by recovering nodes in order to know what updates they missed from older versions. After this, C(F) and the nodes in S(F) can go back to normal operation.

Like the cancelling protocol, the election protocol can be safely interrupted by failures (like the crash of a newly elected controller node). Another working majority of nodes can then restart the protocol at a later time.

6.8 Deadlock detection.

Deadlocks are possible with our concurrency control mechanism. Deadlocks may be avoided by forcing all transactions to request locks from the controllers in the same predefined order. In some systems, this may not be feasible, so deadlocks must be detected and eliminated. Gray [7] (among others) discusses several deadlock detection strategies that may be used. Once we choose a transaction that must be backed out, it can be cancelled with the protocol of section 6.6. Also notice that a transaction may make several lock requests to the same controller, but this should not cause any problems.

7. CONCLUSION.

We have proposed a new concurrency control mechanism for distributed databases. We believe that this control strategy has some advantages over the other well known strategies. Work is currently underway to evaluate the performance of this proposed mechanism, as well as to verify its correctness.

8. ACKNOWLEDGMENTS.

Several useful suggestions and ideas were provided by Clarence Ellis, Ramez El-Masri, Bruce Lindsay, Toshimi Minoura, Daniel Ries, Tom Rogers, Gio Wiederhold, and others.

This work was partially supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA903-77-C-0322 (KBMS), 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 (SUMEX).

9. REFERENCES.

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

[2] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System"; Communications of the ACM, Vol. 19, No. 11, November 1976.

[3] H. Garcia-Molina, "Performance Comparison of Update Algorithms for Distributed Databases, Part II"; Technical Note 146, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, December 1978.

[4] H. Garcia-Molina, "Crash Recovery in the Centralized Locking Algorithm"; Technical Note 153, Computer Systems Laboratory, Departments of Electrical Engineeing and Compute Science, Stanford University, April 1979.

[5] H. Garcia-Molina, "Partitioned Data, Multiple Controllers and Transactions with an Initially Unspecified Base Set"; Technical Note 155, Computer Systems Laboratory, Depatments of Electical Engineering and Computer Science, Stanford University, April 1979.

[6] H. Garcia-Molina, "Performance Comparison of Two Update Algorithms for Distributed Databases"; Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, August 1978.

[7] J. Gray, "Notes on Database Operating Systems"; Advanced Course on Operating Systems, Technical University Munich, July 1977.

[8] D. A. Menasce, G. J. Popek, and R. R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases"; SIGMOD Proceedings May 1978, to appear in TODS.

ON EFFICIENT MONITORING OF DATABASE ASSERTIONS IN DISTRIBUTED DATABASES

D. Z. Badal

Computer Science Department, UCLA, Los Angeles, Ca 90024

Abstract

A principal problem with the use of database integrity assertions for monitoring the integrity of dynamically changing database is the high cost due to the evaluation of such assertions. In this paper we analyze and compare the cost and performance of several integrity validation methods in distributed database environment where the communication cost and delay are principal factors.

INTRODUCTION

It is often argued that the users of databases should be able to specify semantic integrity (SI) assertions about their data. Such assertions delimit values in the database in terms of other database values or constants. Although considerable work has been done on the specification methodology for such assertions /McL 76, STO 74, BOY 75, ZLO 74, GRA 75, MAC 76, FLO 74, MIN 74, WEB 76/, there seems to be much less concern with implementation issues /STO 75, ESW 75, ESW 76, STO 76, HAM 78/.

A major problem in validating transactions with respect to a set of SI assertions is the high overhead (or cost) caused by the dependency of transactions and SI assertions on values stored in the database. Such dependencies prevent a priori proofs of transaction correctness with respect to a set of SI assertions. An example of such a database data dependent transaction T and SI assertion A could be as follows:

T: increase the salary of employee J. Johnson by 10 percent

A: salary of employee < MAX (salary of manager, 1.5*average salary)

Transactions whose SI correctness cannot be proven must be dynamically monitored to determine whether their final values violate SI assertions. The subsystem which monitors such SI assertion violations is properly a part of database management and in this paper we analyze the cost and the performance of several methods of semantic integrity (SI) validation of transactions.

SI VALIDATION METHODS

The validation of transactions with respect to SI assertions can occur at compile time (i.e. before transaction execution), during transaction execution (i.e. at run time), after transaction execution; or partially during each of these phases. Each method of SI validation has advantages and drawbacks, and each method introduces overhead. The cost of SI validation consists of three factors:

(1) Accessing database data in order to evaluate SI assertions,

(2) Computation to evaluate SI assertions, and

(3) The communication cost if SI assertion arguments are stored at several sites of a distributed database system.

We assume here that the computational cost for SI assertion evaluation is the same for any SI validation method. Therefore, the major components of SI enforcement cost result from accessing database data for SI assertion evaluation and from communication cost due to SI validation that requires access to several sites of a distributed database system.

Compile Time SI Validation

Compile time SI validation means that a transaction is allowed to execute only after its SI assertions are evaluated and all assertions are found true. Hammer and Sirin /HAM 78/ suggest compile time SI validation based on SI tests. The purpose of these tests is to obtain those values which database data would have had if the transaction had been executed. The values are then used for SI validation of the transaction, i.e. for evaluation of all SI assertions which interact with the transaction. Compile time SI validation has one obvious advantage - it does not require transaction rollback when SI assertions are violated. However, compile time SI validation has the following disadvantages:

(1) Validation and execution are sequential, slowing response.

(2) The database objects on which compile time SI tests are run cannot be modified by any other transaction until the transaction being validated is executed. Effectively, such database data objects have to remain write-locked from SI validation through transaction execution, since compile time SI validation tests must execute on the same database data values as the transaction will during its execution. Otherwise revalidation is required.

Run Time SI Validation

Run time SI validation means that SI validation of transactions is concurrent with transaction execution, where the result of transaction execution is not committed, e.g. the actual update is not performed, i.e. is not transferred to the transaction write site and executed, until transaction validation has been terminated without violations of SI assertions /BAD 79/. Thus, if transaction execution is seen (and implemented) as a sequence of read-compute and write events, then all SI assertions can be evaluated as part of transaction execution. After the transaction executed its read-compute events SI assertions can be evaluated because the result of transaction is known at that time. Then depending on the outcome of SI validation the write events of the transaction can be executed, i.e. the update messages are transferred to the transaction write sites and performed there. The major advantages of this approach result from concurrent execution; there is no need for transaction rollback, the time interval during which the database data must be locked for SI validation is reduced to transaction execution time, and duplicate reads are avoided.

Another proposal based on run time SI validation appears in INGRES /STO 74, STO 75/ where single variable aggregate-free integrity assertions can be efficiently evaluated during transaction run time by appending such assertions to the query. However, the evaluation of integrity assertions involving aggregates occurs after transaction execution, i.e. the resulting (updated) relation is tested for the integrity assertions and then the update is undone if the assertions are not satisfied. The strategy is therefore mixed - partialy run time and partially postexecution time.

Post Execution SI Validation

The conventional method of SI validation is to execute the transaction first and then to validate the results. The proposal by Eswaran et al. /ESW 76/ employs postexecution time SI validation where the violation of SI assertions by the transaction triggers corrective action. Transaction rollback or some other compensating action, depending on the semantics of the SI assertions and the transaction, may take place. One advantage of postexecution time SI validation is its conceptual simplicity. The obvious disadvantage is the need for transaction rollback and the longer time interval during which the database objects modified by the transaction may be locked. If the objects are not locked, then any other transactions which access the database data which were undone would have to be rolled back too.

COST AND PERFORMANCE ANALYSIS OF SI VALIDATION METHODS IN DISTRIBUTED DBS

In centralized database systems the only significant factor of SI validation cost is the number of database accesses due to SI validation.It has been shown /BAD 79,BAD 79a/ that in terms of the cost of database accesses

a) the run time SI validation is superior to any of the other methods for realistic database operations, i.e. for systems without high transaction rejection rates;

b) compile time SI validation yields better performance than run time SI validation when the compile time SI tests are very efficient, i.e. require substantially fewer database accesses compared to the transaction reads and if DBS has a relatively high rejection rate;

c) postexecution time SI validation has consistently worse performance than other SI validation methods;

d) the use of fast access memory to store data for evaluation of some SI assertions results in increased performance that differs for each SI validation method.

However, the principal cost and performance criterion for SI validation in nonlocal (or loosely coupled) distributed database systems is the communication cost and communication delay, and the number of database accesses is of secondary importance.Therefore,in our analysis we neglect the cost overhead due to database accesses required for SI validation at each site of distributed database system and we consider the communication cost only.

We derive the cost of SI validation in distributed databases without considering transaction processing strategy. We consider here only the number of messages needed either to access sites or to set local locks there. We assume that there is one control site which either does evaluation of SI assertions (i.e. SI validation is centralized) or it receives the results of distributed SI validation. However, in both cases such site controls subsequent transaction execution steps. Assuming a two-phase locking /GRA 76, GRA 78, ESW 76a/ we analyze SI validation methods in distributed database in terms of lock and unlock messages.

Let

P be the average number of sites at which the transaction during its execution reads and writes or reads only.

Q be the average number of sites at which the transaction during its execution writes only.

S be the average number of sites counted in P above at which the transaction reads and that are also accessed for SI validation. Clearly, S <= P.

V be the average number of sites counted in P above at which the transaction writes. Clearly , V <= P .

R be the average number of sites not accessed by the transaction but accessed for SI validation only.

The cost of compile time SI validation in distributed database environment can be derived as follows. Since from n transactions only m (m <= n) transactions will be accepted, i.e. n - m transactions are rejected because they violated SI assertions, then the communication cost of executing those m transactions can be derived from the following compile time SI validation algorithm:

Algorithm C1:

1) lock at R and S sites

2) R and S sites either evaluate SI assertions and send the result to a control site or they send data to the control site which does SI validation

3) if SI validation results in SI violation, then reject transaction and terminate, else do 4)

4) lock at Q + P - S sites and execute transaction

5) unlock at Q + P + R sites and terminate

The number of messages generated at each step of algorithm C1 is

1) $n(R + S)$

2) $n(R + S)$

3) $(n - m)(R + S)$

4) $m(P + Q - S)$

5) $m(P + Q + R)$

Thus the total communication cost of the compile time SI validation (i.e. the cost of n transactions employing compile time SI validation) is

$$C1 = 2n(R + S) + (n - m)(R + S) + m(2P + 2Q + R - S) = 3n(R + S) + 2m(P + Q - S) \tag{1}$$

where

n is the number of transactions

m is the number of accepted transactions, i.e. the number of transactions which did not cause any SI violations; $m <= n$

The communication cost of postexecution time SI validation can be derived from the following algorithm C2.

Algorithm C2:

1) lock at P and Q sites and execute transaction

2) lock at R sites, send the SI assertion argument values or SI validation results to the control site and release locks after sending the above message to the control site

3) control site requests and receives SI messages from S sites

4) if SI validation results in SI violation, then send reject messages to all sites at which transaction writes, i.e. to Q and V sites, else do 5)

5) unlock at P + Q sites and terminate

The number of messages generated at each step of algorithm C2 is:

1) $n(P + Q)$

2) $2nR$

3) $2nS$

4) $(n - m)(Q + V)$

5) $n(P + Q)$

Thus the communication cost of postexecution time SI validation is:

$$C2 = 2n(P + Q) + 2n(R + S) + (n - m)(Q + V) \qquad (2)$$

The communication cost of run time SI validation can be derived from the following algorithm C3.

Algorithm C3:

1) lock at P and R sites

2) execute read-compute (i.e. generate final update messages at P sites) and read SI values at R sites

3) send SI assertion argument values or the results of SI validation from R and S sites to the control site

4) if SI validation results in SI violation , then reject transaction via unlock messages to R and P sites and terminate, else do 5)

5) lock at Q sites and execute updates

6) unlock at P + Q + R sites and terminate

The number of messages generated by the algorithm C3 is:

1) $n(P + R)$

2) none

3) $n(R + S)$

4) $(n - m)(P + R)$

5) mQ

6) $m(P + Q + R)$

Thus the communication cost of postexecution time SI validation is:

$$C3 = n(P + R) + m(R + S) + (n - m)(P + R) + mQ + m(P + Q + R)$$

$$C3 = 2n(P + R + S) + 2mQ \tag{3}$$

The communication cost of mixed run time and postexecution time SI validation can be obtained by adapting formulae (2) and (3) :

$$C4 = (2n[1](P + R + S) + 2m[1]Q) + (2n[2](P + Q) + 2n[2](R + S) + (n[2] - m[2](Q + V)) \tag{4}$$

where

n[1] is the number of transactions validated at run time; n[1] <= n

n[2] is the number of transactions validated at postexecution time; n[2] <= n and n[1] + n[2] = n

m[1] is the number of transactions rejected due to run time SI validation; m[1] <= m

m[2] is the number of transactions rejected (rolled back) due to postexecution time SI validation; m[2] <= m and m[1] + m[2] = m

Now that a consistent, straightforward method of expressing the communication cost of the various SI validation methods has been provided, it is useful to compare them.

Lemma 1:

The communication cost of run time SI validation C3 is greater than the communication cost of compile time SI validation C1 only if the accesses for SI validation at transaction read sites or read and write sites are more numerous than the accesses for SI validation at sites not accessed by the transaction, i.e. C3 > C1 only if 0 <= R <= S.

Proof:

Assume C1 > C3. Then substituting from (1) and (3) we obtain

$$3n(R + S) + 2m(P + Q - S) < 2n(P + R + S) + 2mQ \tag{4a}$$

(4a) reduces to the following condition

$$m/n > 1/2(1 + (P - R)/(P - S)) \tag{5}$$

Since m <= n we observe the following:

case 1: if R = 0, then (5) is not satisfied, i.e. C1 < C3

case 2: if R = P, then (5) is satisfied, i.e. C3 < C1 if m > .5n case 3: if $R < P$, then

if $R < S < P$, then (5) is not satisfied, i.e. $C1 < C3$

if $P > R > S$, then (5) is satisfied ,i.e. $C3 < C1$

if $R = S$,then (5) is not satisfied, i.e. $C1 < C3$

case 4: if $R > P$, then (5) is satisfied, i.e. $C3 < C1$

Therefore, $C3 > C1$ if $0 <= R <= S$ and $C3 < C1$ if $S < R$. This concludes the proof.

Lemma 2:

The communication cost of compile time SI validation C1 is less than the communication cost of postexecution time SI validation C2, i.e. $C1 < C2$ only if $0 <= R <= S$.

Proof:

Assume $C1 > C2$. Substitution from (1) and (2) leads to

$$3n(R + S) + 2m(P + Q - S) > 2n(P + Q) + 2n(R + S) + (n - m)(Q + V) \quad (5a)$$

(5a) reduces to

$$m/n > (2P + 3Q + V - R - S) / (2P + 3Q + V - 2S) \quad (6)$$

Since $m <= n$ we observe that if $R = 0$ or $R = S$ or $R < S$, then (6) is not satisfied, i.e. $C1 < C2$, otherwise (i.e. if $R > S$) $C1 > C2$. This concludes the proof.

Lemma 3:

The communication cost of run time SI validation C3 is always less or equal to (if $m=n$) the communication cost of postexecution time SI validation C2, i.e. $C3 <= C2$.

Proof:

Assume $C3 <= C2$.Substituting from (2) and (3) leads to

$$2n(P + R + S) + 2mQ <= 2n(P + Q) + 2n(R + S) \quad (7)$$

$$+ (n - m)(Q + V)$$

(7) reduces to

$$(n - m)(V + 3Q) => 0 \qquad (8)$$

Since all terms are positive, and n => m then (8) is satisfied (for any transaction which affects database consistency i.e.,which updates the database). This concludes the proof.

Lemma 4:

The communication cost of mixed run time and postexecution time SI validation is at best equal to run time SI validation and at worst as costly as postexecution time SI validation.

Proof:

We want to show that C3 <= C4 <= C2.

If in (4) n[1] = n,i.e. n[2] = 0, then C4 = C3. If in (4) n[2] = n, i.e. n[1] = 0 ,then C4 = C2. Since C2 > C3 (Lemma 3),then the lower bound for C4 is C3 and the upper bound for C4 is C2. This concludes the proof.

CONCLUSION.

In this paper we have shown that in the distributed database when there is an extensive global SI validation, i.e. when R > S, then in terms of communication cost the run time SI validation is the least costly and the compile time SI validation has the highest communication overhead, i.e. C3 <= C2 < C1. However, if there is not an extensive global SI validation, i.e. when 0 <= S <= R, then the compile time SI validation has the lowest communication overhead and the postexecution time SI validation has the highest communication overhead, i.e. C1 < C3 <= C2. We would like to point out that the conclusions reached here apply to any type of distributed database, i.e. they apply to fully replicated, partially replicated or nonredundant distributed databases. This is so because the obtained results do not depend on the number of sites at which transaction writes only, i.e. on Q.

REFERENCES

BAD 79 Badal, D. Z. "Semantic integrity, consistency and concurrency in distributed database systems," Ph.D. dissertation, Computer Science Dept., UCLA, March 1979.

BAD79a Badal,D.Z. and Popek,G.J. "Cost and performance analysis of semantic integrity validation methods " , Proc. of ACM SIGMOD 79 International Conference on Management of Data, Boston, May 1979, pp.109-115.

ESW 75 Eswaran, K. P. and Chamberlin, D. D. "Functional specification of a subsystem for data base integrity," IBM Research Report RJ 1601, June 1975.

ESW 76 Eswaran, K. P. "Specifications, implementations and interactions of a trigger subsystem in an integrated database system," IBM Research Report RJ 1820, November 1976.

ESW 76a Eswaran, K. P. etal. "The notions of consistency and predicate locks in database system," CACM 19, 11 (1976),pp.624-633.

FLO 74 Florentin, J. J. "Consistency auditing of data bases," Computer Journal 17, 2 (1974), pp. 52-58.

GRA 75 Graves, R. W. "Integrity control in a relational data description language," Proc. of ACM Pacific Conference, San Francisco, April 1975, pp. 108-113.

GRA 76 Gray, J. etal. "Granularity of locks and degrees of consistency in a shared data base," Modelling in Data Base Management Systems, G.M. Nijssen(ed.), North Holland, 1976, pp.365-395.

GRA 78 Gray, J. "Notes on data base operating systems,"IBM Research Report RJ 2188, February 1978.

HAM 78 Hammer, M. M. and Sarin, S. K. "Efficient monitoring of database assertions," ACM/SIGMOD 78 Int. Conference on Management of Data, Dallas, June 1978, pp.38-48.

MAC 76 Machgeles, C. "A procedural language for expressing integrity constraints in the coexistence model," Modelling in Data Base Management Systems, ed. by G. M. Nijssen, Amsterdam, North-Holland, 1976, pp. 293-301.

McL 76 McLeod, D. J. "High level expression of semantic integrity specifications in a relational data base system," MIT/LCS/TR-165, September 1976.

MIN 74 Minsky, N. "On interaction with data bases," Proc.

of CM SIGFIDET Workshop on Data Description, Access, and Control, ACM, New York, 1974.

STO 74 Stonebraker, M. "High level integrity assurance in relational data management systems," Electr. Res. Lab. Memo ERL-M473, UC Berkeley, August 1974.

STO 75 Stonebraker, M. "Implementation of integrity constraints and views by query modification," Electronics Res. Lab. Memo ERL-M514, UC Berkeley, March 1975.

STO 76 Stonebraker, M. and Neuhold, E. "A distributed data base version of INGRES," Electronics Res. Lab. Memo ERL-M612, UC Berkeley, September 1976.

WEB 76 Weber, H. "A semantic model of integrity constraints on a relational data base," Modelling in Data Base Manamgement Systems, ed. by G. M. Nijssen, Amsterdam, North-Holland, 1976, pp. 269-293.

ZLO 74 Zloof, M. M. "Query by example," IBM Research Report RC 4917, July 1974.

PROTOCOL MODELING

A STUDY OF THE CSMA PROTOCOL IN LOCAL NETWORKS*

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

A consequence of bursty traffic in computer communications is that among a large population of network users, at any one time only a small number of them have data to send (ready users). In this environment, the performance of an access protocol for a broadcast network depends mainly upon how quickly one of the ready users can be identified and given sole access to the shared channel. The relative merits of the access protocols of polling, probing and carrier sense multiple access (CSMA) with respect to this channel assignment delay in local networks are considered. A central controller is needed for polling and probing while CSMA employs distributed control. A specific CSMA protocol is defined which requires that "collisions" in the channel be detected and that the users involved in a collision abort their transmissions quickly. In addition, it is assumed that the contention algorithm is adaptive and gives rise to a stable channel. An analytic model is developed. Our main result is the moment generating function of the distributed queue size (number of ready users). Mean value formulas for message delay and channel assignment delay are also derived. These results on queue size and delay are the major contribution of this paper, since they are not available in prior CSMA models in closed analytical form. Numerical results are given to illustrate the performance of the CSMA protocol. When the channel utilization is light to moderate, the mean channel assignment delay of the CSMA protocol is significantly less than that of both polling and probing; consequently, the mean message delay is much smaller. It is also shown that when queueing of messages is permitted at individual users, the maximum channel throughput of CSMA approaches unity in the limit of very long queues.

1. INTRODUCTION

Multipoint networks have been widely used in local networking for the interconnection of terminals to a central site: either a central computing facility or a gateway to a resource sharing computer network. The terminals are typically unintelltgent and access to the shared data path (channel) is managed by the central site using a polling protocol [1]. With increasing interest in local networking and the availability

* This work was supported by the National Science Foundation under Grant No. ENG78-01803.

of inexpensive microprocessors, other interconnection topologies, transmission media and access protocols have been proposed and investigated. They include loop networks with centralized control [2] or distributed control [3], a digital cable network using time-division multiple access [4], the ALOHANET [5] and Packet Radio Network [6], which pioneered the use of radio channels and contention protocols for multiple access. Recently, considerable interest has been revived in multipoint cable networks (based upon CATV technology) employing a variety of multiple access protocols [7-10].

The multiple access problem in multipoint networks is addressed in this paper. A multipoint cable network such as those in [8,9] can be viewed upon as a broadcast channel shared by a population of distributed users. Two major categories of multiple access protocols may be used: polling and contention protocols [11]. Polling protocols require a central controller. On the other hand, with contention protocols each network user makes his own decision according to an algorithm which is driven by observable outcomes in the broadcast channel. We shall consider multipoint networks that have short propagation delays between users relative to the transmission time of a message. In a short propagation delay environment, carrier sense multiple access (CSMA) protocols have been found to be the most efficient among contention protocols [12-15].

Consider a broadcast channel (the multipoint network) shared by a population of N users (terminals, computers, etc.). There are two problems to be addressed by an access protocol: (1) among the N users, identify those with data who desire access to the channel, the *ready users*, and (2) assign channel access to exactly one of the ready users if at least one exists.

The ready users can be considered as forming a "distributed queue" waiting to use the broadcast channel. We assume that each user generates and holds for transmission at most one message of arbitrary length at a time. (The effect of queuing messages at individual users is discussed in the last section of this paper.) A consequence of the conservation law in queuing theory [12] is that the average message delay performance of an access protocol is independent of the order of service but depends mainly upon the amount of overhead needed for assigning channel access. Thus, when access protocols are compared solely on the basis of average message delay performance for a given channel throughput level, the above two problems reduce to just the following: whenever the channel is free and there are one or more ready users, how quickly can channel access be assigned to a ready user?

In conventional polling protocols [1], the above problem is solved by a central controller that queries the N users one after the other. Let $\bar{w}$ be the average overhead associated with querying one user; $\bar{w}$ includes propagation delay, polling message transmission time etc. To find out who the ready users are, the overhead per polling cycle (querying all N users) is $N\bar{w}$, regardless of the number of ready users present. This overhead is an indirect measure of the responsiveness of the access protocol; Konheim and Meister [16] showed that the mean delay of a polled network is directly proportional to $N\bar{w}$.

Hayes [17] recently proposed and studied the method of probing: polling a group of users all at one. The key idea is as follows. If a group of users is probed and none responds, the whole group can be eliminated. If probing a group produces a positive response, it is subdivided into two groups which are then probed separately. Thus when the network is lightly loaded, with few ready users, significant overhead reduction results through eliminating groups of non-ready users all at once. In the extreme case of only one out of the N users being ready, the number of queries required by probing is $2(\log_2 N) + 1$ instead of N required by polling. However, if all N users are ready, the number of queries required by probing is (N^2-1). (See [17] and [11].) Thus probing is penalized when the channel is heavily utilized. Hayes proposed an adaptive algorithm which optimizes the performance of probing and also avoids the above penalty by reverting to pure polling beyond a certain level of channel utilization.

Unlike polling and probing, which require a central controller and are designed for "passive" users, contention protocols require that each ready user actively seek channel access and make his own decisions in the process. We define below a CSMA protocol and show that the time required by it to assign channel access to a ready user is independent of N. Under this protocol, when there is exactly one ready user and the channel is free, the ready user gets channel access immediately. Thus the average "channel assignment delay" is near zero when the channel is lightly utilized. On the other hand, when the channel is heavily utilized the average channel assignment delay is bounded above by a small constant (see below).

CSMA protocols have been studied extensively in the past within a packet radio network environment by Kleinrock and Tobagi [13, 14] and later by Hansen and Schwartz [15]. Analytic results in these references are mainly concerned with the maximum channel "throughput" achievable by various protocols. Characterization of the number of ready users and message delay is limited to approximate numerical solutions or simulation results.

The main contribution of this paper is an analytic model of a CSMA protocol. The protocol is defined and our assumptions stated in Section 2. In Section 3, the moment generating function of the number of ready users is obtained. Formulas for the average message delay and average channel assignment delay are also derived. In Section 4, numerical results are plotted to illustrate the performance of the CSMA protocol, which is also compared with polling. We conclude by discussing possible extensions of this work in Section 5.

2. THE PROTOCOL AND ASSUMPTIONS

The main difference between the CSMA protocol studied in this paper and the p-persistent CSMA protocol of Kleinrock and Tobagi [13, 14] is as follows. We assume here that collisions in the channel are detected and that users involved in a collision abort their transmissions immediately upon detecting the collision. Mechanisms for detecting collisions and aborting collided transmissions have been implemented in at least two multipoint cable networks [8,9]. However, it appears to be much more difficult to implement a "collision abort" capability in the radio environment of interest in [13, 14].

Like the p-persistent protocol in [13,14] network users are assumed to be time synchronized so that following each successful transmission, the channel is slotted in time. (See Fig. 1.) Users can start transmissions only at the beginning of a time slot. Let τ be the amount of time from the start of transmission by one user to when all users sense the presence of this transmission. It is equal to the maximum propagation delay between two users in the network plus carrier detection time. (The latter depends upon the modulation technique and channel bandwidth. It was considered to be negligible relative to the propagation delay in [14].) In order to implement the collision abort capability described above, the minimum duration of a time slot is $T = 2\tau$, so that within a time slot if a collision is detected and the collided transmissions are aborted immediately, the channel will be free of any transmissions at the beginning of the next time slot.

The slotted channel assumption is made to simplify our analysis. (The practical problem of time synchronizing all users in the network is a classical one and beyond the scope of this paper.) In a real system, either a slotted or unslotted channel may be implemented. We discuss in Section 5 that the performance of an unslotted channel is likely to be approximated by that of the slotted model in this paper.

The CSMA protocol in this paper is defined by the following two possible courses of action for ready users:

(P1) Following a successful transmission, each ready user transmits with probability 1 into the next time slot.

(P2) Upon detection of a collision, each ready user uses an adaptive algorithm for selecting its transmission probability (<1) in the next time slot.

It should be clear at this point that we have effectively reduced the contention problem in CSMA to a slotted ALOHA problem. Slotted ALOHA has been studied extensively in the past [18-25], from which we learned that to prevent channel saturation (with zero probability of a successful transmission), the transmission probability of each ready user must be adaptively adjusted. Various control strategies have been proposed and studied. Experimental results have shown that a slotted ALOHA channel can be adaptively controlled to yield an equilibrium throughput rate S close to the theoretical limit of 1/e (=0.368) for a large population of users [21-24]. With an asymmetric strategy, the achievable S will be even higher [25].

For our analysis in the next section, we shall assume that in (P2) a suitable adaptive algorithm is used so that the probability of a successful transmission (slotted ALOHA throughput) in the next time slot is equal to a constant S. This assumption is an approximation but has been found to be a very good one in simulation studies [21-24].

We shall further assume that errors due to random noise are insignificant relative to errors due to collisions and can be neglected. The source of traffic to the broadcast channel consists of an infinite population of users who collectively form an independent Poisson process with an aggregate mean message generation rate of λ messages per second. This approximates a large but finite population in which each user generates messages infrequently; each message can be transmitted in an interval much less than the average time between successive messages generated by a given user. Each user is allowed to store and attempt to transmit at most one message at a time. Thus the generation of a new message is equivalent to increasing the number of ready users by one. The effect of queuing messages at individual users is discussed later.

Finally, the transmission time of each message is an independent identically distributed (i.i.d.) random variable with the probability distribution function (PDF) $\beta(x)$, mean value b_1, second moment b_2 and Laplace transform $\beta^*(s)$.

3. THE ANALYSIS

The ready users can be considered to form a distributed queue with random order of service for the broadcast channel. We are interested in obtaining the equilibrium moment generating function of the distributed queue size. We shall use an imbedded Markov chain analysis. Under the assumptions of Poisson arrivals and that messages arrive and depart one at a time, the moment generating function of queue size obtained for the imbedded points is valid for all points in time.

A snapshot of the channel is illustrated in Fig. 1. We define the following random variables:

q_n = number of ready users left behind by the departure of the n^{th} transmission, C_n

y_{n+1} = time from the departure of C_n to the beginning of the next successful transmission

u_{n+1} = number of new (Poisson) arrivals during y_{n+1}

x_{n+1} = transmission time of C_{n+1}

v_{n+1} = number of new (Poisson) arrivals during $x_{n+1} + \tau$.

We assumed earlier that x_{n+1} has the PDF $\beta(x)$. We shall let $B(x)$ be the PDF of $x_{n+1} + \tau$. The corresponding Laplace transform is thus

$$B^*(s) = \beta^*(s)e^{-s\tau}$$

The random variable y_{n+1} is the sum of two independent random time intervals

$$y_{n+1} = (I_{n+1} + r_{n+1})T$$

where T is the duration of a slot, I_{n+1} is the number of slots in an idle period immediately following the departure of C_n, and r_{n+1} is the number of slots in the contention period following a collision until the next successful transmission. The slot containing the initial collision is included in r_{n+1}. We note that I_{n+1} is nonzero only if $q_n = 0$. Also, if there has been no collision when C_{n+1} begins, $r_{n+1} = 0$.

Let p_j be the probability of j new arrivals (ready users) in a time slot.

$$p_j = \frac{(\lambda T)^j e^{-\lambda T}}{j!} \qquad j = 0,1,2,\ldots$$

At the start of the next time slot, each new arrival executes (P1) or (P2) in exactly the same manner as all other ready users.

Given our earlier assumptions, we have

$$\text{Prob}[I_{n+1} = k/q_n = 0] = (1-p_0)p_0^{k-1} \qquad k = 1,2,\ldots$$

Also,

$$\text{Prob}[r_{n+1} = k/\text{collision occurred}] = S(1-S)^{k-1} \qquad k = 1,2,\ldots$$

From this last result, the Laplace transform of the probability density function (pdf) of a contention period (given a collision occurred) is

$$C^*(s) = \frac{Se^{-sT}}{1-(1-S)e^{-sT}}$$

which has a mean of T/S and a second moment of $T^2(1 + \frac{2(1-S)}{S^2})$.

The following important relationship is evident from Fig. 1.

$$q_{n+1} = q_n + u_{n+1} + v_{n+1} - 1 \tag{1}$$

where v_{n+1} is an independent random variable with the z-transform $B^*(\lambda-\lambda z)$, while u_{n+1} depends upon q_n in the following manner as a consequence of (P1) and (P2). Given

(1) $q_n = 0$,

$$u_{n+1} = \begin{cases} 1 & \text{with prob. } \frac{p_1}{1-p_0} \\ j + \text{number of arrivals during a contention period} & \text{with prob. } \frac{p_j}{1-p_0} \end{cases}$$

(2) $q_n = 1$, $u_{n+1} = 0$

(3) $q_n \geq 2$, u_{n+1} = number of arrivals during a contention period.

(2)

Given the occurrence of a collision, the number of new arrivals during a contention period is an independent random variable with the z-transform $C^*(\lambda-\lambda z)$.

The equilibrium queue length probabilities

$$Q_k = \lim_{n\to\infty} \text{Prob}[q_n = k] \qquad k = 0,1,2,\ldots$$

exist if $\lambda(b_1 + \tau + T/S) < 1$ (see below). Define the z-transform

$$Q(z) = \sum_{k=0}^{\infty} Q_k z^k.$$

By considering Eqs. (1) and (2) and taking the $n\to\infty$ limit, we obtain after some algebraic manipulations the following important result:

$$Q(z) = \frac{B^*(\lambda-\lambda z)\{Q_1 z[1-C^*(\lambda-\lambda z)] + \frac{Q_0}{1-p_0}[p_1 z(1-C^*(\lambda-\lambda z)) - C^*(\lambda-\lambda z)(1-e^{-\lambda T(1-z)})]\}}{z - B^*(\lambda-\lambda z)\, C^*(\lambda-\lambda z)} \tag{3}$$

where

$$Q_0 = \frac{1-\lambda\,(b_1+\tau+T/S)}{\lambda T[\frac{1}{1-p_0} - \frac{1}{B^*(\lambda)S}]} \tag{4}$$

and

$$Q_1 = (\frac{1}{B^*(\lambda)} - \frac{p_1}{1-p_0})\, Q_0 \tag{5}$$

Using Eqs. (3) - (5), we can obtain the mean queue size. Application of Little's result [12] yields the mean message delay (time of arrival to time of departure) to be

$$D = \bar{x} + \frac{T}{S} + \frac{T}{2} - \frac{1-p_0}{2[B^*(\lambda)S-(1-p_0)]}(\frac{2}{\lambda} + ST - 3T) + \frac{\lambda[\overline{x^2} + 2\,\bar{x}\,\frac{T}{S} + T^2(1 + 2\,\frac{1-S}{S^2})]}{2[1 - \lambda(\bar{x} + \frac{T}{S})]} \tag{6}$$

where

$$\bar{x} = b_1 + \tau$$

and

$$\overline{x^2} = b_2 + 2b_1\,\tau + \tau^2$$

We next consider the channel assignment delay, that is, given that the channel is free and that there is at least one ready user, we want the pdf of the time from when the above conditions are satisfied to the start of the next successful transmission. Let d_n be a random variable representing the channel assignment delay immediately prior to the n^{th} transmission and

$$d = \lim_{n\to\infty} d_n$$

It can be readily shown that

$$\text{Prob } [d = k] = \begin{cases} Q_0 \dfrac{p_1}{1-p_0} + Q_1 & k = 0 \\ [Q_0(1-\dfrac{p_1}{1-p_0}) + \sum\limits_{i=2}^{\infty} Q_i] \; S(1-S)^{k-1} & k = 1,2,... \end{cases} \tag{7}$$

The mean channel assignment delay is thus

$$\bar{d} = \frac{1}{S}(1 - Q_0 \frac{p_1}{1-p_0} - Q_1) \tag{8}$$

Note that $Q_0 \frac{p_1}{1-p_0} + Q_1$ is the fraction of transmissions that incur zero delay in gaining channel access (given that the channel is free).

4. PERFORMANCE OBSERVATIONS

An important performance parameter is the ratio of the carrier sense time τ to the mean message transmission time b_1:

$$\alpha = \frac{\tau}{b_1}$$

The throughput of the CSMA channel is defined to be the fraction of channel time utilized by data messages, which is

$$\rho = \lambda b_1$$

under equilibrium conditions.

In Fig. 2, we show the delay performance of the CSMA channel as a function of α and ρ. The normalized delay D/b_1 is plotted and it is assumed that messages are of constant length. Observe that the delay performance of CSMA improves significantly as α becomes small. A small α may come about either by decreasing the carrier sense time τ or by increasing the duration b_1 of each user transmission.

In these numerical calculations, the probability S of a successful transmission during contention periods is assumed to be 1/e which is the slotted ALOHA throughput rate in an infinite population model. Experience with experimental results [21-25] indicates that S = 1/e is pessimistic when the number of contending ready users is small (small ρ) and optimistic when the number of contending ready users is large (large ρ). Thus the same comments will apply to the CSMA delay results in Fig. 2.

The delay-throughput performance of roll-call polling is also shown using the delay formula in [16]. The delay results shown for polling also assume Poisson message arrivals and constant message length. The ratio of propagation delay to message transmission time is $\alpha = 0.05$. The ratio of data to polling message length is 10. Queuing of messages at individual users is assumed; hence the maximum channel throughput is one. Delay-throughput curves for both 10 users and 100 users are shown. Note that the corresponding delay-throughput performance of CSMA at $\alpha = 0.05$ is independent of the number of users. It also permits no queuing of messages at individual users; hence the maximum throughput is less than 1. We observe that CSMA is superior to polling when the channel throughput is low but becomes inferior when the channel throughput is increased to one. However, if queuing of messages is possible at individual users for CSMA, more than one message may be transmitted every time a user gains channel access. Hence, as the network load ρ is increased from 0 to 1, the delay performance of CSMA is first given by the $\alpha = 0.05$ curve at a small channel throughput but switches to the $\alpha = 0.01$ curve and then the $\alpha = 0.001$ curve and so on as the channel throughput increases and queues become long. The channel throughput of CSMA is one in the limit of infinitely long queues at individual users.

In Fig. 3, we show the mean channel assignment delay $\bar{d}$ as a function of α and ρ. Note that $\bar{d}$ decreases to zero when ρ is small. This is because (P1) in the CSMA protocol permits a ready user to access the channel immediately. In Fig. 4 we plot the fraction of transmissions that incur zero delay in gaining channel access given that the channel is free. For comparison, recall that when only one ready user is present, the polling cycle overhead is $N\bar{w}$ for conventional polling and $[2(\log_2 N)+1]\bar{w}$ for probing.

Referring again to Fig. 3, observe that as ρ is increased, $\bar{d}/T$ increases to the maximum value of $1/S$. This desirable property is a consequence of the presence of an adaptive algorithm that we assumed in (P2) which guarantees channel stability during contention periods.

Another advantage that CSMA has over polling protocols is that the time slot duration T is typically much smaller than its counterpart $\bar{w}$ in polling protocols since $\bar{w}$ must include the transmission time of a polling message.

5. CONCLUSIONS

We considered a CSMA protocol as a distributed control technique for a population of users sharing a multipoint network. The capability of aborting collided transmissions is the main difference between our model and previous models of CSMA. It is also assumed that the channel is stable during contention periods (presence of an adaptive control algorithm). Our main results include the moment generating function of the number of ready users, as well as mean value formulas for message delay and channel assignment delay. These results are new. The modeling of the queue size and message delay has previously been limited to numerical solutions or simulations.

We found that the CSMA protocol as defined in this paper has the desirable property that when the channel is lightly utilized, the channel assignment delay is extremely short. The performance of CSMA when the channel is heavily utilized depends upon the ratio α. We make the following observation. If the number of users is finite and queuing of messages is permitted at individual users, then as $\rho \uparrow 1$, we must have $\alpha \downarrow 0$, since the transmission time of each user increases as a result of long queues. In this case, the maximum channel throughput of CSMA is one (the same as polling with queueing permitted at individual users).

Lastly, we discuss the issue of channel slotting. A slotted channel was assumed in our analysis. In practice, either a slotted or unslotted channel may be implemented. The analysis of an unslotted protocol will be more involved. However, the following observation indicates that the performance of an unslotted protocol should be approximated by our slotted model in this paper. In the analysis of slotted and pure ALOHA [12,18] it was found that the probability of success of a transmission depends mainly upon the duration of its "vulnerable period" to another transmission. The vulnerable period in our slotted CSMA channel is the duration of a time slot T. On the other hand, the vulnerable period in an unslotted version of our CSMA protocol would be 2τ (after a little thought) which is the sames as T. Thus the probability that an attempted transmission is successful during a contention period is approximately the same in both cases.

Acknowledgements

The author is pleased to acknowledge the programming assistance of Luke Lien and typing assistance of Nancy DeGlandon. He is also indebted to an anonymous referee for his helpful comments.

REFERENCES

[1] Schwartz, M., Computer-Communication Network Design and Analysis, Prentice-Hall, Englewood Cliffs, N.J., 1977.

[2] Fraser, A., "A Virtual Channel Network," Datamation, Vol. 21, Feb. 1975.

[3] Farber, D. J. and K. C. Larson, "The System Architecture of the Distributed Computer System - the Communications Systems," Proc. Symp. Computer-Communications Networks and Teletraffic, Polytechnic Institute of Brooklyn, April 1972.

[4] Willard, D. G., "Mitrix: a Sophisticated Digital Cable Communications System," Conf. Rec. National Telecommunications Conference, Nov. 1973.

[5] Binder, R. et al, "ALOHA Packet Broadcasting - A Retrospect," AFIPS Conf. Proc. Vol. 44, 1975.

[6] Kahn, R. E., "The Organization of Computer Resources into a Packet Radio Network," IEEE Trans. on Commun., Vol. COM-25, Jan. 1977.

[7] DeMarines, V. A. and L. W. Hill, "The Cable Bus in Data Communications," Datamation, August 1976.

[8] Metcalfe, R. M. and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Communications of the ACM, 19, no. 7, July 1976.

[9] West, A. and A. Davison, "CNET - A Cheap Network for Distributed Computing," Department of Computer Science and Statistics, Queen Mary College, University of London, Report TR120, March 1978.

[10] Thornton, J.E. et al., "A New Approach to Network Storage Management," Computer Design, Nov. 1975.

[11] Lam, S. S., "Multiple Access Protocols," Dept. of Computer Sciences, University of Texas at Austin, Tech. Rep. TR-88, Jan. 1979; to appear in Computer Communications: State of the Art and Direction for the Future, edited by W. Chou, Prentice-Hall.

[12] Kleinrock, L., Queueing Systems, Vol. 2: Computer Applications, Wiley-Interscience, N.Y., 1976.

[13] Tobagi, F. A., "Random Access Techniques for Data Transmission over Packet Switched Radio Networks," Ph.D. Dissertation, Computer Science Department, University of California at Los Angeles, Dec. 1974.

[14] Kleinrock, L. and F. A. Tobagi, "Packet Switching in Radio Channels: Part 1 - Carrier Sense Multiple Access Modes and their Throughput - Delay Characteristics, "IEEE Trans. on Comm. Vol. COM-23, December 1975.

[15] Hansen, L. W. and M. Schwartz, "An Assigned-Slot Listen-Before-Transmission Protocol for a Multiaccess Data Channel," Conf. Rec. International Conference on Communications, Chicago, June 1977.

[16] Konheim, A. G. and B. Meister, "Waiting Lines and Times in A System with Polling," J.ACM, Vol. 21, July 1974.

[17] Hayes, J. F., "An Adaptive Technique for Local Distribution," IEEE Trans. on Commun., Vol. COM-26, August 1978.

[18] Abramson, N. "Packet Switching with Satellites," AFIPS Conf. Proc., Vol. 42, AFIPS Press, Montvale, New Jersey, 1973.

[19] Kleinrock, L. and S. S. Lam, "Packet-Switching in A Slotted Satellite Channel," AFIPS Conf. Proc., Vol. 42, AFIPS Press, Montvale, N.J.,

[20] Kleinrock, L. and S. S. Lam, "Packet Switching in a Multiaccess Broadcast Channel: Performance Evaluation," IEEE Trans. on Commun., Vol. COM-23, April 1975.

[21] Lam, S. S., "Packet Switching in a Multi-Access Broadcast Channel with Application to Satellite Communication in a Computer Network," Ph.D. Dissertation, Computer Science Department, Univ. of California, Los Angeles, March 1974.

[22] Lam, S. S. and L. Kleinrock, "Dynamic Control Schemes for Packet Switched Multi-access Broadcast Channel," AFIPS Conf. Proc., Vol. 44, AFIPS Press, Montvale, N.J., 1975.

[23] Lam, S. S. and L. Kleinrock, "Packet Switching in a Multiaccess Broadcast Channel: Dynamic Control Procedures," IEEE Trans. on Commun. Vol. COM-23, Sept. 1975.

[24] M. Gerla and L. Kleinrock, "Closed Loop Stability Controls for S-ALOHA Satellite Communications," Proc. Fifth Data Communications Symposium, Snowbird, Utah, September 1977.

[25] Kleinrock, L. and Y. Yemini, "An Optimal Adaptive Scheme for Multiple Access Broadcast Communication," Conf. Rec. ICC'78, Toronto, June 1978.

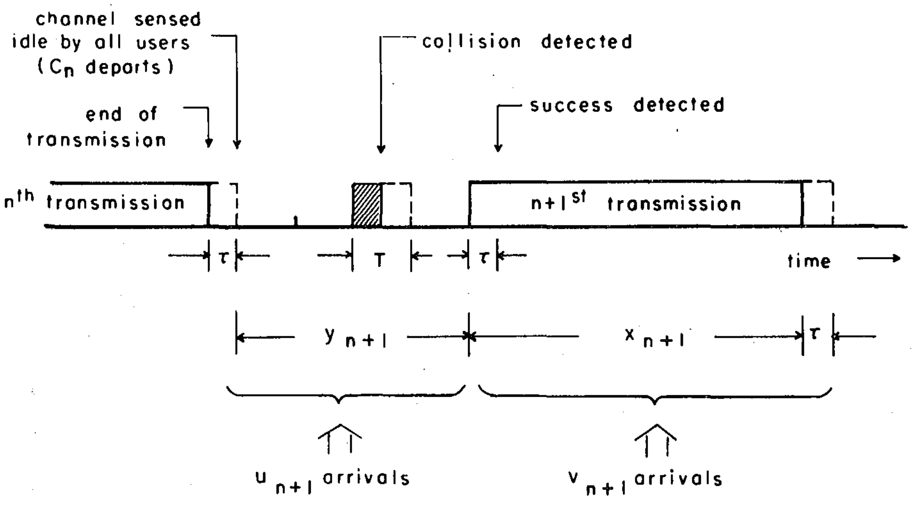


Figure 1. A snapshot of the broadcast channel.

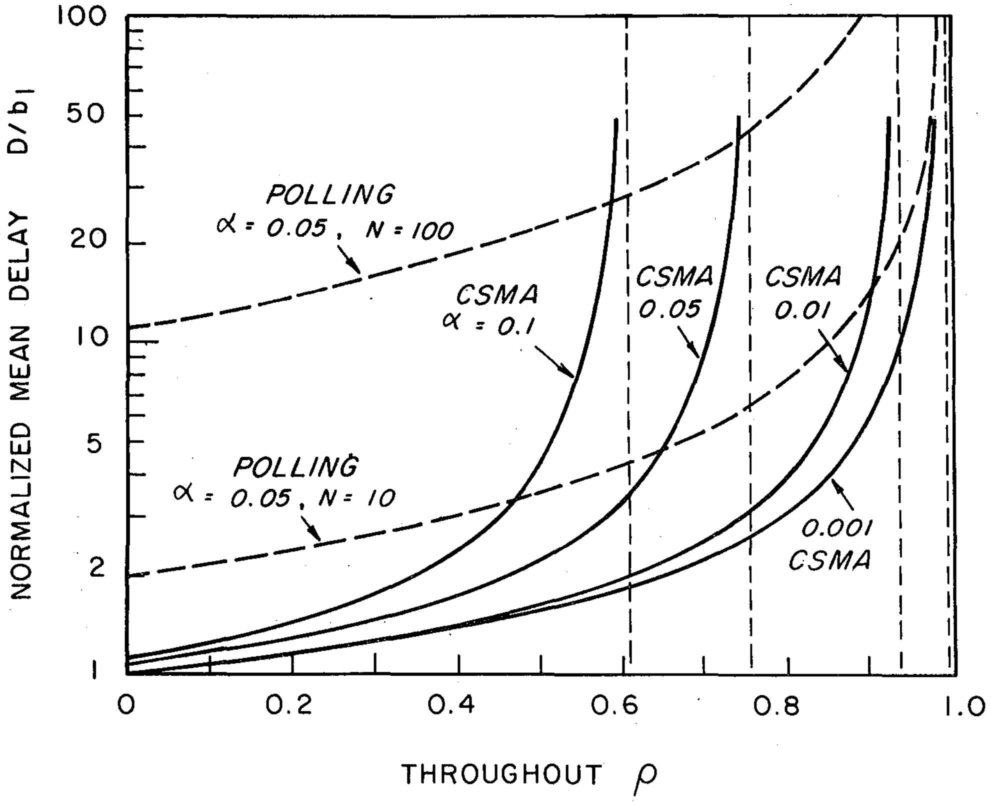


Figure 2. Delay versus throughput.

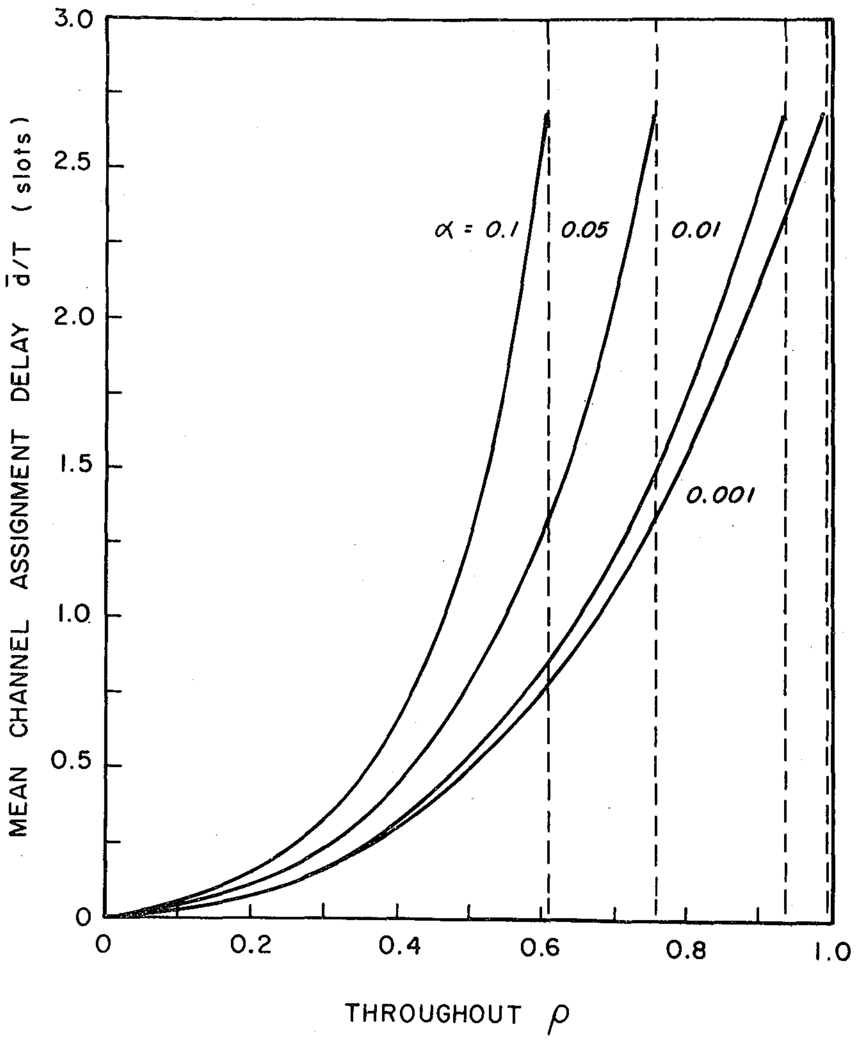


Figure 3. Channel assignment delay versus throughput.

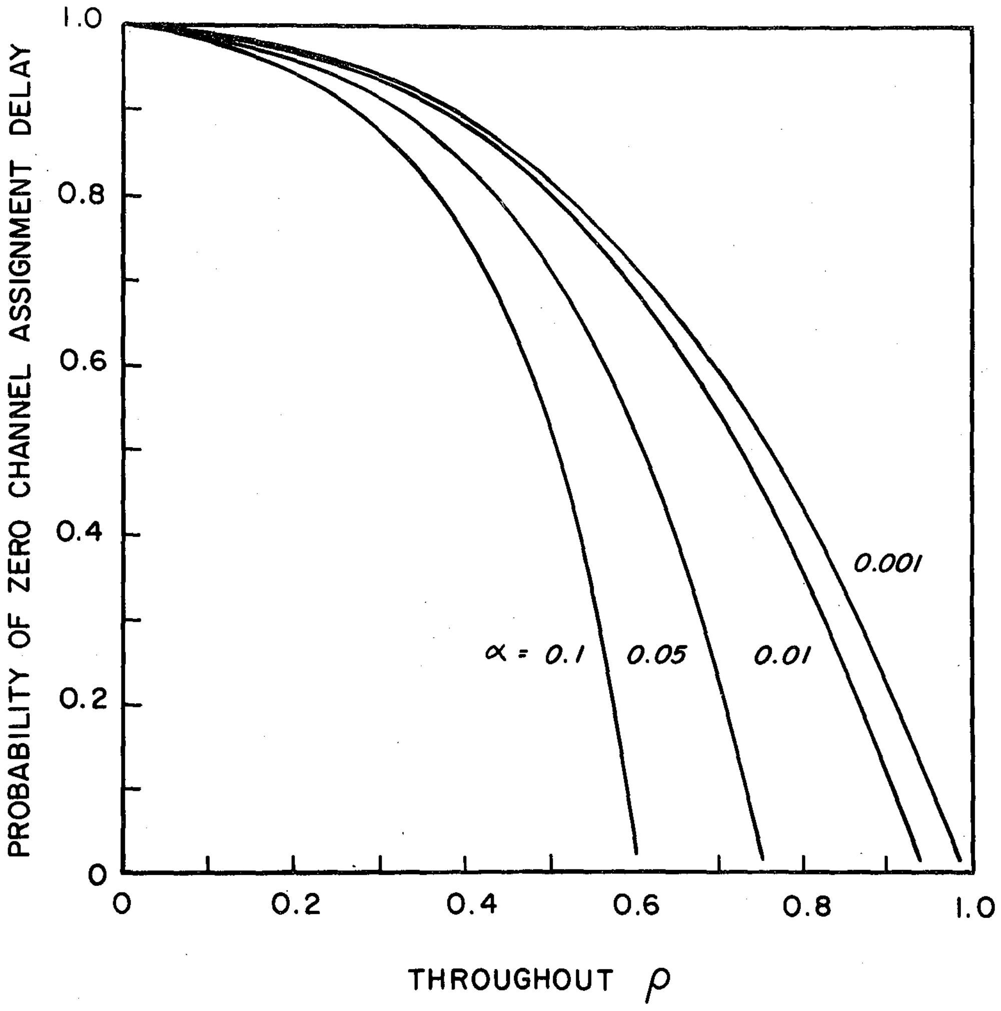


Figure 4. Probability of zero channel assignment delay versus throughput.

GLOBAL AND LOCAL MODELS FOR THE SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS

Mohamed Gouda, Donald Boyd, and William Wood

Honeywell Corporate Technology Center
Minneapolis, MN 55420

Abstract

Two models for the specification of distributed systems are presented; they are named global and local models. The global model can be used to specify the system requirements without suggesting any specific design to achieve these requirements. The local model can be used to specify some particular system designs which satisfy the given requirements. Some general verification techniques are proposed to prove theorems about the specifications in both models. We use the two models to specify a number of well-known distributed systems such as shared resource systems, schedulers, readers and writers, and the five dining philosophers. The proposed verification techniques are also applied to some of these systems.

Keywords

Specification
Requirements
Distributed Systems
Verification
Formal Modeling

1. INTRODUCTION

There has been a great interest in distributed systems in recent years (Gouda 76), (Brinch Hansen 78), (Hoare 78), and (Lamport 78). Part of this interest is due to the belief that these systems can offer high degrees of extensibility, performance, and fault tolerance (Jensen 78). However, there are still many problems concerning these systems which need to be solved before distributed systems can be realized and exploited in a practical way. One of these problems is the specification of distributed systems (Greif 75), (Gouda 76), (Boebert 79), (Laventhal 79), and (Riddle 79). In this paper, we address this problem by introducing formal models to specify distributed systems.

A distributed system consists of entities called processes which communicate only by exchanging messages. Each process has a number of local data objects which cannot be directly accessed by other processes. However, any process P can send messages to any other process Q requesting to read or to update the local variables of Q. Then, according to the internal state of process Q, these requests can be denied or honored. Thus, a process performs two kinds of operations, external operations and internal operations. The external operations consist of sending (or receiving) messages to (or from) other processes in the system. The internal operations consist of testing and updating the local variables in the process.

There are abstract machines associated with each process in the system. The abstract machines define the data types which can be used inside the process. They also define the appropriate operations which can be performed on each data types in the machine. Two (or more) processes can share the same abstract machine if the processes use the same data types which are declared by the machine. Any of the known techniques to specify abstract machines (Parnas 72), (Liskov 75), (Guttag 77), (Robinson 77), (Boyd 78a), and (Boyd 78b) can be used in conjunction with our models of a process to specify distributed systems.

In this paper, we present two models for the specification of communicating processes in distributed systems; they are called global and local models. In the global model, we assume the existence of a global controller which can read and update the internal states of all the processes in the system. This assumption leads to concise and compact specifications. However, since the global controller is not an acceptable notion in a distributed system, a global specification does not specify a solution, (i.e., a system design), it merely specifies the problem (i.e., the system requirements). In order to solve such a problem, the global controller should be replaced by subcontrollers at the system processes such that the total system behavior is preserved. The result of this replacement is a local model specification. Therefore, the global specification for a system defines the system requirements whereas a local specification for the same system defines a system design.

The global model is presented in section 2. Then verification techniques for global model specifications are discussed in section 3. Some examples of global model specifications are given in section 4. The local model is presented in section 5; and some examples of local model specifications are given in section 6.

2. THE GLOBAL MODEL

In the global model, a distributed system with K processes is specified as follows:

```
system   system name;

     process   process name 1;                            |
                                                          |
          var   list of local variables in process 1;     |
                                                          |   Data Specification
                      .                                   |      Section
                      .                                   |
                      .                                   |
     process   process name K;                            |
                                                          |
          var   list of local variables in process K;     |

     rules                                             |
                                                       |   Control Specification
          list of system transition rules              |      Section
                                                       |
end   system name.
```

Reserved words such as system, process, rules, and end are underlined. The specification consists of two sections, a data specification section and a control specification section. In the data specification section, the local variables in each process are defined using a PASCAL-like notation.

In the control specification section, a set of transition rules are defined. A transition rule has the following syntax:

$$\text{condition} \longrightarrow \text{result}$$

where both the "condition" and the "result" have the following syntax:

Simple Bool. Expr. and . . . and Simple Bool. Expr.

A simple Boolean expression is as follows:

Expression 1 'relation' Expression 2

Both "Expression 1" and "Expression 2" are based on the local variables of the system processes, and the 'relation' is any one of the following $=$, $\neq$, $\leq$, $<$. After specifying the syntax of transition rules, we discuss their semantic next.

In the global model, the global state of a distributed system is specified by the values (at that state) of all the variables in the system. Thus, the initial global state is specified by the initial values of the system variables. At the beginning, the system is at its initial global state; then its global state changes due to the "firing" of its transition rules. For a transition rule to fire at some global state, its condition must be true at that state. The firing of a transition rule consists of changing the global state such that the result of the transition rule is true at the new state.

We assume that the firing of different transition rules is mutually exclusive; i.e.,at most one transition rule can fire at a time.

There are a number of similarities between the global model and other proposed models (Keller 76) and (Bochmann 78); but there are also some differences. In particular, the global model does not have an explicit control structure for each process in the system. Instead, the transition rules in the global model describe the control structure of a "global controller"; hence, the name global model. It is assumed that the global controller can read and update the internal states of all the processes in the system.

The global controller is a virtual entity; it is not a process in the system. However, its existence makes the system specification more concise and compact. On the other hand, since the global controller is not an acceptable notion in a distributed system, a global specification does not specify a solution (i.e.,a system design); it merely specifies the problem (i.e.,the system requirements). In order to solve such a problem, we should get rid of the global controller; i.e., replace its transition rules by sets of transition rules and assign each set to some process in the system. The result is a new system model,called the local model. The local model is discussed in detail in section 5.

Now we give some examples of global specifications.

Bounded Buffer

The bounded buffer consists of three processes "producer", "consumer", and "bufprs". The "producer" has two variables, "st" (for state), and "indata" to hold the data which is to be sent to the "consumer" via the buffering process "bufprs". The producer state, referred to as "producer.st", can have one of two values "null" or "ready". If "producer.st" is "null", it means that the producer has no new data to store in the buffer. Whenever "producer.st" is "ready", it means that the content of the variable "indata" has a new value which can be copied in the buffer (provided there is an available space in it). Similarly, the "consumer" has the two variables "st" and "outdata". The buffer process "bufprs" has an array of size N to store the received data. It has also two integer variables "in" and "out",where "in" is the total number of received data items from the "producer", and "out" is the total number of data items sent to the "consumer". The global specification is as follows:

```
system  bounded buffer (N);

     process  producer;

          var  st: (null,ready) init null;  indata : real;

     process  consumer;

          var st:  (null,ready) init null;  outdata : real;

     process  bufprs:

          var  buffer:  array 0..N-1  of  real;
               in, out:  integer init o;

     rules

          producer.st = null ———→ producer.st'=ready and indata'= input;
          consumer.st = null ———→ consumer.st'=ready and output = outdata;
          in < out + N and  producer.st = ready
          ————→  in' = in+1  and  buffer'(in mod N) = indata and
                 producer.st' = null;
          out < in  and   consumer.st = ready
          ————→ out'=out+1  and  outdata'= buffer (out mod N) and
                consumer.st'=null;

end  bounded buffer.
```

Notes: (i) Because both "producer" and "consumer" have a variable named "st", we concatenate the process name and the variable name to distinguish between the two variables. (ii) There are four transition rules in this system. The first rule refers to "producer.st" in its condition, and to "producer.st'" in its result to distinguish between the value of this variable before and after the transition rule firing. (iii) The two reserved words input and output are used to imply reading from and writing into the outside world.

Shared Resource

100 users share a common resource which can be accessed by at most one user at a time. The 100 users are defined as a process array of size 100. Each of them can be in any one of three states:

"null"... means the user does not need (nor use) the resource,
"need"... means the user does need the resource,
"busy"... means the user does use the resource.

```
system  shared resource;

     processarray  user (0..99);

          var  st: (null, need, busy) int null;

     rules

          user(i).st = null ————→ user(i).st'=need;

          user(i).st = need and (forall j:0..99)(user(j).st ≠ busy)

          ————————→ user(i).st'=busy;

          user(i).st=busy ————→ user(i).st'=null

end  shared resource.
```

Notes: Each transition rule in this specification is written in terms of a free parameter "i". Since "i" is used as an index of the process array "user", its value ranges from 0 to 99. Therefore, each transition rule is equivalent to 100 different rules. For example, the first rule is equivalent to:

```
        user(0).st = null————→user(0).st' = need;
        user(1).st = null————→user(1).st' = need;
                                 ⋮
        user(99).st = null———→user(99).st' = need.
```

But instead of writing all these rules, we adopt the above short-hand notation.

3. VERIFICATION TECHNIQUES

In general, there are two classes of theorems which we may want to prove about a distributed system. A theorem in the first class has the form:

At any instant P

or P (for short)

where P is a first order predicate which contains some variables from the system specification. P is called an invariant; and the theorem is called an invariance theorem. The theorem implies that P is true in all the system states which can be reached from the initial state by any possible sequence of transition rule firing. A two-step algorithm to prove an invariant theorem (Keller 76) is as follows:

Algorithm

```
step 1: Prove that the invariant is true in the system initial
        state'
step 2: for all the transition rules in the system specification
        do Prove that if the invariant is true before the rule fires
                     then it is also true after the rule fires od;
```

From these two steps, the invariant is true at all reachable states by induction on the length of the firing sequence. An example is given later on.

Another class of theorems which may be of interest has the following form:

$$P \xrightarrow{*} Q$$

where P and Q are first order predicates which contain some variables from the system specification. This theorem means that if the system ever reaches a state S_1 where P is true, then in a finite period of time (starting from S_1) the system will reach a state S_2 where Q is true. More specifically, there is an upper bound on the number of transitions which can fire after S_1 before state S_2 is reached. The proof of such a theorem consists of finding this upper bound. These theorems are called non-starvation theorems since as we will see most non-starvation theorems can be written using this form.

Now we give some examples. Consiaer tne shared resource system in the previous section. There are two theorems which we want to prove about this system:

Mutual Exclusion

if user(x).st = busy then (forall y:0..99)(if y ≠ x then user(y).st≠busy)

Non-Starvation

$$\text{user(x).st = need} \xrightarrow{*} \text{user(x).st = busy}$$

The first theorem states that at any instant at most one user is busy using the resource. The second theorem states that if a user needs the resource, then it will get it in a finite period of time. The first theorem is from the class of invariant theorems, whereas the second one is from the class of non-starvation theorems.

To prove the invariant of the first theorem, we first show that it is true at the initial state. Then, we show that if it is true before the firing of each transition rule, then it will also be true after the rule firing.

Define S(n) to be the system state in which exactly n users are busy. Then, the initial state of the system is S(0); and the three transition rules of the system can be defined in terms of S(n) as follows:

(1) $S(n) \rightarrow S(n)$;
(2) $S(0) \rightarrow S(1)$;
(3) $n \geq 1$ and $S(n) \rightarrow S(n-1)$;

The mutual exclusion theorem can now be restated (then proved) as follows:

Mutual Exclusion Theorem

After the firing of any sequence of transition rules (the empty sequence is included), the system can either be in state S(0) or in state S(1).

Proof: The proof is by induction on the length of the firing sequence. First the theorem is true after the empty firing sequence since the initial state is S(0). Assume that the theorem is true after a firing sequence of length n, we want to show that it will be true after the (n+1)th firing. The (n+1)th firing can be of rule (1), rule (2), or rule (3). If rule (1) is fired, then $S(0) \rightarrow S(0)$, or $S(1) \rightarrow S(1)$. If rule (2) is fired, it means that the system was in state S(0), and it will become in state S(1). If rule (3) is fired, it means that the system was in state S(1), and it will become in state S(0). In all cases, the theorem is true after the (n+1)th firing. Thus, the theorem is true after any firing sequence.

To prove the non-starvation theorem, we need to show that if some user, user (x) say, is in a "need" state, then in a finite period of time his state will become "busy". Specifically, we want to show that there is an upper bound K such that at most K transition rules will fire, then the state of user(x) becomes "busy".

Actually, we cannot prove this theorem because the system specification in the previous section does permit starvation. To show this, consider the system when user (x).st = need and user (y).st = null. Starting from this state, if the transition rules in the infinite sequence (1), (2), (3), (1), (2), (3), (1)... continue to fire for user (y), then user (x) will continue to be in a "need" state forever.

To prevent starvation from the system, we add an integer "count" to each user. Initially, a user "count" has the value zero, and it is incremented each time the user state is changed from "need" to "busy". Thus, at each instant, the "count" value is the total number of times the user had an access to the shared resource. Whenever a number of users are in "need" states competing for the resource, the one with the smallest "count" will win. If there are more than one, one of them chosen arbitrarily will win. The system, after these modifications, is as follows:

```
system  shared resource without starvation;

     processarray  user (0..99);

          var   st : (null, need, busy) init null;

                count : integer  init 0;

     rules

          users(i).st = null ————→ user(i).st' = need;

          user(i).st = need  and
          (forall j:0..99)(user(j).st ≠ busy) and
          (forall j:0..99)(if user(j).st = need then user(i).count ≤
                user(j).count)
         ————→ user(i).st'=busy  and  user(i).count'=user(i).count+1;

        user(i).st = busy ————→ user(i).st'=null;

end  shared resource without starvation.
```

Now we can prove the non-starvation theorem for this system.

Non-Starvation Theorem

user (x).st = need $\xrightarrow{*}$ user (x).st = busy.

Proof: Assume user (x).st = need, we want to show that there is an upper bound K such that at most K transition rules will fire before user (x).st = busy. The worst case is when the "states" of every other user is "null", and its "count" is zero. Starting from this state, each other user can compete for the resource, get it, and prevent user (x) from becoming "busy". This can continue until the "counts" of all other users exceeds the "count" of user (x) by one. Therefore,

$$K = 99 \times 3 \times (\text{user(x).count}+1) + 99$$

where 99 is the number of other users in the system, and 3 is the number of rules that each user can fire to compete, get, and release the resource.

In this section, the verification of a "simple" shared resource system has been discussed. Our intent was to demonstrate the use of some general techniques for the verification of distributed systems using our global model. Next, we extend the discussion to more "elaborate" examples of distributed systems. In each example, we specify some distributed system using the global model, and discuss the theorems which need to be proven in the order to verify the specification.

4. EXAMPLES OF GLOBAL MODEL SPECIFICATIONS

Three examples of global specifications are presented in this section. The first example is intended to demonstrate how to use abstract data types in conjunction with the global model to specify distributed systems in terms of abstract data structures. The next two examples are intended to express the model power in specifying a variety of distributed systems.

Bounded Buffer with Abstract Data Types

A bounded buffer system is specified in section 2. Here, we specify the same system except that the buffer is declared to be of type "queue" (instead of an "array"). The data type "queue" can be defined using any technique to specify data types such as (Guttag 77) or (Boyd 78b). Assume that the following four operations are defined for the data type "queue":

length: queue → integer

add: queue x element → queue

remove: queue → queue

top: queue → element

Since the exact definitions of these operations are of little value to the discussion in this paper, we skip these definitions assuming that the reader has a reasonable idea about the meaning of these operations. These four operations can be used to specify the bounded buffer system as follows:

```
system  bounded buffer  (N);
     process producer;
          var st:  (null, ready) init null; indata:  real;

     process  consumer;
          var st:  (null, ready) init null; outdata:  real;

     process  bufprs;
          var  buffer:  queue  init  length (buffer) = 0;

     rules
          producer.st = null ------> producer.st' = ready and indata' = input;
          consumer.st = null ------> consumer.st' = ready and output = outdata;

          length (buffer) < N and  producer.st = ready
          ------> buffer' = add (buffer, indata) and producer.st' = null;

          length (buffer) > 0  and  consumer.st = ready
          ------> outdata' = top (buffer) and
                  buffer' = remove (buffer) and consumer.st' = null;

end  bounded  buffer.
```

In order to verify this system, we need to prove the following two theorems:

Invariant Theorem: $0 \leq$ length (buffer) $\leq N$

Non-Starvation Theorem: indata = X $\xrightarrow{*}$ outdata = X

The non-starvation theorem states that if the producer ever produces a value X then in a finite period of time the consumer will get it.

Readers and Writers

100 users share a common resource. A user can read or write the resource such that any number of users can read the resource simultaneously, whereas a writer needs a sole access to the resource.

```
system  readers  writers;
      process array  user  (0..99)
        var st = (null, need, busy) init null; rqst = (read, write);

      rules
        user(i).st = null ------> user(i).st' = need  and  user(i).rqst' =
            input;

        user(i).st = need  and  user(i).rqst = read  and
        (forall j:  0..99) (if user(j).rqst = write then user(j).st = null)
        ------> user(i).st' = busy;

        user(i).st = need  and  user(i). rqst = write  and
        (forall j:  0..99) (user(j).st ≠ busy)
        ------> user(i).st' = busy;

        user(i).st = busy ---> user(i).st' = null;

end  readers  writers.
```

To verify this system, we need to prove the following theorems:

Mutual Exclusion: There are two theorems to prove:

Theorem 1: If a user is reading, no user is writing; i.e.,

```
if user(x).st = busy  and  user(x).rqst = read
   then (forall y:  0..99) (if user(y).rqst = write  then
      user(y).st ≠ busy)
```

Theorem 2: if a user is writing, no other user is busy; i.e.,

```
if user(x).st = busy  and  user(x).rqst = write
   then (forall y:  0..99) (if y ≠ x  then  user(y).st ≠ busy)
```

Non-Starvation

user(x).st = need and user(x).rqst = write

$\xrightarrow{*}$ user(y).st = busy and user(y).rqst = write

This theorem states that if a user needs to write, then in a finite period of time a user (may be another one) will write. This is a weak non-starvation theorem. To make it stronger, we need to modify the specification as discussed in section 3.

Five Dining Philosophers

Five philosophers spend their lives thinking and eating. The philosophers sit at a circular table with a bowl of spaghetti in its center. The table is laid with five forks. On feeling hungry, a philosopher picks up the fork on his left and the fork on his right, eats, then puts down both forks. The system specification is as follows:

```
system dining philosophers;
    process array fork (0..4);
       var st = (putdown, pickup) init putdown;
    process array ph (0..4);
       var st: (think, hungry, eat) init think;
    rules
       ph(i).st = think------>ph(i).st' = hungry;

       ph(i).st = hungry and fork(i).st = putdown and
       fork(i+1).st = putdown------>ph(i).st' = eat and
       fork(i).st'= pickup and fork (i+1).st' = pickup;

       ph(i).st = eat------>ph(i).st' = think and
       fork(i).st' = putdown and fork(i+1).st' = putdown;

end dining philosophers.
```

5. THE LOCAL MODEL

As demonstrated by the above examples, the global model is a useful tool to specify and "easily" verify distributed system specifications. However, one of the model's problems is the lack of mechanisms to specify potential parallelism within these systems (since transition rules can only fire one at a time). On the other hand, it is this "non-parellel behavior" which simplifies the verification of distributed system specifications. In general, one needs a compromise between these two seemingly conflicting needs; i.e., introduce a scheme to specify parallelism into the model while retaining most of the features which ease verification. In this section, such a compromise is discussed.

First, we present a scheme to specify parallelism into the global model. The resulting model is called the local model. Then we show that in a "large" number of cases, proving a theorem for the local model specification (i.e., with parallelism) is equivalent to proving the same theorem for a global model specification (i.e., without parallelism).

In the local model, each transition rule belongs to one process in the system; and each process has one or more transition rules. The transition rules in one process can only fire one at a time. Parallelism is achieved when transition rules in different processes fire simultaneously. Therefore, the maximum number of transition rules which can fire simultaneously equals the number of processes in the system.

As an example, a local model specification for the bounded buffer system defined in section 2 is as follows:

```
system   bounded   buffer (N);


   process   producer;

      var   st:  (null, ready)        init  null;   indata:  real;

      rules

           producer.st = null ——————> producer. st'= ready  and
              indata' = input;

   end  producer;
```

```
  process  bufprs;
     var  buffer:  array o .. N-1  of  real;  in , out = integer init o;
     rules
          in< out + N  and
          producer.st = ready------->in'= in + 1  and
                                     buffer' (in mod N ) = indata  and
                                     producer.st' = null


          out < in      and
          consumer.st = ready------->out'= out + 1  and
                                     outdata' = buffer (out  mod N )  and
                                     consumer.st' = null
  end  bufprs;

  process  consumer;
     var  st:  (null, ready)       init  ready;  outdata:  real;
     rules
          consumer.st = null-------->consumer.st' = ready  and  output =
              outdata;
  end  consumer;
end  bounded  buffer.
```

For the sake of the local model, we assume a "discrete" view of time. The system state can be only observed at discrete instants of time t_1, t_2,... At any instant, say t_i, the system state S_i remains fixed, and no activity (i.e., transition rule firing) takes place. However, at the next instant, t_{i+1}, the system state S_{i+1}, may be different from S_i, implying that some transition rules had fired in the unobserved time period between t_i and t_{i+1}. As in the global model, a transition rule in the local model fires between a pair of observed time instants t_i and t_{i+1}, only if its condition is true at t_i. If the rule does fire between t_i and t_{i+1} then its result is true at t_{i+1}.

Because of this discrete view of time, the local model allows only an "ideal" type of parallelism. Two transition rules in different processes can either fire simultaneously (i.e., between the same pair of successive time instants), or in sequence (i.e., they fire between different pairs of successive time instants). If the two transition rules have disjoint variables then whether the two rules fire simultaneously or in sequence, the system still reaches the same global state. This means that the introduced parallelism does not introduce "new" reachable states to the system. The parallelism merely "speeds-up" the reaching to the "old" reachable states.

This property can be used to simplify theorem proving for the local model. As an example, assume it is required to prove an invariant P for some system S specified in the local model. One needs to show that P is true for all the reachable states of S. To do so, one can ignore the parallelism (implied by the local model) assuming that all the transition rules in the system only fire one at a time (i.e., global model). Then, the techniques outlined in section 3 can be used to prove P. Hence, P is true if the transition rules are fired one at a time. But, because the parallelism does not introduce new system states, P is also true in the local model. Now that we have established the importance of preventing parallelism from introducing new system states, we need some way to achieve this property. In particular, we need a set of restrictions (i.e., a discipline) to write transition rules such that this property is achieved. A discipline to write transition rules in the local model is discussed next.

In the local model, the transition rules in a process can only test (in their condition parts) and update (in their result parts) two classes of variables, namely, the process local variables and sequencers. A sequencer is a variable local to some process but it can be tested and updated by transition rules in other processes in the system. There is no limit on the number of sequencers which are defined in a process. A sequencer should satisfy the following conditions concerning its declaration, testing, updating, and its associated variables.

Sequencer Declaration

A sequencer is a variable of an enumerative type. It is declared using the reserved word seq. For example, the following statement declares a sequencer "x" which has five values:

$$\underline{\text{seq}}\ x : (x1, x2, x3, x4, x5);$$

Sequencer Testing

A sequencer can be tested in the condition part of any transition rule in the system. The test can only be of the form: seq name = seq value. Moreover, if a transition rule in one process tests for one value of a sequencer then no transition rule in any other process can test for this same value. This does not exclude the case when two (or more) transition rules in the same process test for the same value. As an example, assume that the following three transition rules (which test sequencer x) belong to the same process P:

$$x = x1 \longrightarrow v' = \underline{\text{input}}\ \underline{\text{and}}\ x = x2;$$

$$x = x2\ \underline{\text{and}}\ z = 2 \longrightarrow w' = v' + 20\ \underline{\text{and}}\ z' = z + 2;$$

$$x = x2\ \underline{\text{and}}\ z = 4 \longrightarrow x' = x4;$$

Then the following rule cannot belong to any process in the system other than P:

$$x = x1 \longrightarrow x' = x2;$$

since it tests for value x1 which is tested by some rule in P.

Sequencer Updating

A sequencer can be updated by the result part of any transition rule in the system if the rule tests the sequencer in its condition part. Because of this condition, each sequencer should have an initial value.

A sequencer update can only be of the form: seq name' = seq value. For example, the following transition rule correctly updates the above sequencer "x":

$$x = x2 \; \underline{\text{and}} \; z = 4 \longrightarrow x' = x5;$$

On the other hand, the next two rules are wrong:

$$y = 1 \; \underline{\text{and}} \; z = 4 \longrightarrow x' = x5;$$
$$x = x2 \; \underline{\text{and}} \; z = x5 \longrightarrow x' = z;$$

The first rule updates sequencer x in the result part without testing its value in the condition part. The second rule updates sequencer x using an inappropriate form.

Sequencer Associated Variables

Let x be a sequencer local to some process P. A variable v local to P is said to be associated with x if each transition rule in P which reads or updates v in its result part also tests x in its condition part. If v is associated with sequencer x, then any transition rule in the system which tests x in its condition part can test, read, or update v in its result part.

After stating the sequencer conditions, it is useful to discuss the motivations behind these conditions. As mentioned earlier, the basic motivation is to prevent the parallelism from introducing new reachable states to the system specification. Specifically, the following theorem is true:

Theorem 1

Let S be a distributed system specified in the local model. If S is at state S_1 where some transition rules (in different processes) can fire simultaneously causing S to become in state S_2, then if these rules fire in any sequence starting from S_1, S will become in S_2.

Proof: Let r_1, r_2,..., r_n be the transition rules which can fire simultaneously causing S to change its state from S_1 to S_2. To show that these rules can fire in any sequence cuasing the same state change, it is sufficient to show that no two of these rules share any variables.

In other words, it is sufficient to show that any two rules r_i and r_j do not test (in their condition parts), read or update (in their result parts) any common variables. The condition parts of r_i and r_j contain local variables and sequencers. But since they can fire simultaneously, they belong to different processes; and their local variables are different. Moreover, they can only test the same sequencer for different values; but since they can fire simultaneously and a sequencer (like any variable) can only have one value at a time, r_i and r_j must have different sequencers, if any, in their condition parts.

The result parts of r_i and r_j can have local variables, sequencers, and sequencer associated variables. Since they don't test the same sequencers in their condition parts, they can neither update the same sequencers nor read nor update the same sequencer associated variables in their result parts. Thus r_i and r_j do not have any common variables; and the final state will be the same if they fire simultaneously or if they fire in sequence. □

From the above theorem, it can be shown that proving invariants for a local model specification is equivalent to proving these invariants for the same specification assuming that transition rules in the system fire one at a time.

Theorem 2

Let P be an invariant for some distributed system specified in the local model. If P is true when the transition rules in the system fire one at a time, then P is true for the local model.

Proof: In the local model, transition rules in different processes can fire simultaneously. But from theorem 1 this parallelism does not introduce "new" reachable states. Since P is true at all the "old" reachable states, then it is also true for the local model. □

Another important motivation for defining sequencer conditions as they are defined in this section is to ease the checking of whether or not a given local specification satisfies these conditions. Actually these conditions can be verified purely on the basis of the specification syntax. Thus, the checking can occur at compile time, and it can be easily automated.

Now, a word of caution. The concept of sequencers is intended to specify synchronization in the local model specification. So it is a specification tool. It is not intended to be an implementation tool. It should not be viewed as a hint on how synchronization between communicating processes should be implemented. Our only criterion for selecting sequencers in the local model is ease of proofs.

6. EXAMPLES OF LOCAL MODEL SPECIFICATIONS

In this section, we present some examples of distributed system specifications using the local model. To compare between local and global specifications, some examples in this section are for distributed systems whose global specifications are introduced earlier.

Shared Resource:

The shared resource problem is discussed in section 2. A local model specification for the problem is as follows:

```
system    shared resource using semaphore;

     process array  user (0..99);
          seq  st:  (null, need, busy)  init  null;
          rules
               user(i).st = null ————→ user(i).st' = need;
               user(i).st = busy  and  sem(i) = one
               ————→ user(i).st' = null   and   sem(i) = zero;
     end  user;

     process  semprs;
          seq  sem:  array  0..99  of  (zero, one) init zero;
          rules
               user(i).st = need  and  (forall j: 0..99)  (sem(j) = zero)
               ————→ user(i).st' = busy  and  sem(i) = one;
     end  semprs;

 end  shared resource
```

Notes: (i) If this specification is compared with the global specification in section 2, we note that a new process "semprs" is added to the system to provide the required synchronization. (ii) The new process contains 100 semaphores (defined as a sequencer array) so that each user process can test and update its own semaphore; thus, the sequencer conditions are satisfied. (iii) The new process represents a "central" controller for the system. To show that is not a characteristic of the model but it is a characteristic of our chosen solution, we present another specification for the same problem. In this specification, there is a token which is being passed from one user process to another (Lelann 77). If a user process needs the resource, it waits until it receives the token, keeps it, then accesses the resource. When it is done, it gives the token to the next user process:

```
system   shared resource using token;

     process array  user (0..99);
          seq  token:  (act, inact)  init  user(0). token = act  and
                                    (forall j:1..99) (user(j).token=inact);
          var  st   :  (null, need, busy)  init  null;
          rules
               user(i).st = null
               ————————> user(i).st' = need;

               user(i).st = null  and  user(i). token = act  and
                                  user(i+1).token=inact
               ————————> user(i). token' = inact  and  user (i+1). token' = act;

               user(i).st = need  and  user(i). token = act
               ————————> user(i).st' = busy;

               user(i).st = busy
               ————————> user(i).st' = null;
     end  user;

end  shared  resource
```

Note: Each of the above local specifications specifies a possible solution for the same shared resource problem. The global specification of the same problem (in section 2) specifies only the solution requirements without suggesting any specific way to solve it.

Minimum Holding Scheduler

Consider a system with 100 users who share a common resource which can be accessed by, at most, one user at a time. The system has a scheduler to assign the resource to the user who will hold the resource the shortest time. Each user has a local variable called "hldtim" of type positive integer. When a user needs the resource, the value of its "hldtim" equals the expected holding time of the resource by the user. "Hldtim" is chosen to be of type positive integer so that it has a minimum value namely one time unit. The system specification is as follows:

```
system  min  hold  scheduler;

     process array  user (0..99);
          seq   st:  (null, need, busy)  init  null;
          var   hldtim: positiveinteger;
          rules
                user(i).st = null
                -------->user(i).st' = need  and  user(i) .hldtim' = input;

                user(i).st = busy  and  state(i) = inuse
                -------->user(i).st' = null  and  state'(i) = free;
     end user;

     process  scheduler;
          seq   state:  array  0..99  of  (free, inuse)  init  free;
          rules
                user(i).st = need  and
                (forall j: 0..99) (state (j) = free)  and
                (forall j: 0..99) (if  user(j).st = need  then
                user(i). hldtim ≤  user(j).hldtim)
                -------->user(i).st' = busy  and  state'(i) = inuse;
     end scheduler;

end min hold.
```

Note: The scheduler has an array of 100 binary sequencers; one sequencer for each user process. This array is introduced (instead of a single binary variable) to satisfy the sequencer condition that each process can only test a sequencer for some specific value for which no other process can test the same sequencer. Local specifications tend to increase the number of variables in the system.

7. CONCLUSIONS

We presented two formal models to specify distributed systems, a global model and a local model. The two models differ only in their abilities to specify parallelism and in their needs to specify synchronization explicitly.

In the global model, events are assumed to be effected one at a time by some global controller. Therefore, potential parallelism cannot be specified. Moreover, synchronization between conflicting events is achieved automatically; thus, no explicit synchronization mechanism or policy is needed. These characteristics make global specifications simple and straight-forward. In particular, the mechanisms and/or policies which will be introduced to the system (during the design phase) to achieve synchronization need not to be present in the system global specification. For this reason, the global model can be used to specify the system requirements without suggesting how the system should be designed or implemented.

In the local model, non-conflicting events can occur in parallel, and potentially conflicting events are serialized by the aid of "sequencers". Therefore, potential parallelism can be specified and explicit synchronization policy is required. Notice that "sequencers" can be regarded as an explicit synchronization mechanism which is built into the local model. For this reason, the local model can be used to specify different system designs which achieve the system requirements.

The two models have very similar syntax to ease the use of both models during the requirement analysis phase and during the system design phase. From our experience, both models seem to provide concise specifications for otherwise hard systems.

In the paper, we also discuss some general techniques to write and prove theorems about specifications in both models. So far, these techniques have proven very convenient to reason about distributed systems.

Acknowledgement

We are thankful to Debra Jones and Charlotte Zurn for typing this manuscript (in parallel) in such short notice.

REFERENCES

(Bochmann 78) G. V. Bochmann, Synchronization in Distributed System Modules. *Proc. 3rd Berkeley Workshop*, 1978.

(Boebert 79) W. E. Boebert, et. al. NPN: A Finite-State Specification Technique for Distributed Software. *Proc. Specifications of Reliable Software,* April, 1979, pp. 139-149.

(Boyd 78a) D. L. Boyd and A. Pizzarello, An Introduction to the WELLMADE Design Methodology, *IEEE Transactions on SE*, Vol. SE-4, No. 4, July, 1978.

(Boyd 78b) D. L. Boyd, A. Pizzarello and S. C. Vestal. *The Rational Design Methodology - Final Report,* RADC-78-208.

(Brinch Hansen 78) Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of ACM,* Vol. 21, No. 11, November, 1978, pp. 934-941.

(Gouda 76) M. G. Gouda, et al. Protocol Machines: A Concise Formal Model and its Automatic Implementation. *Proc. International Conference on Computer Communication*, August, 1976.

(Gouda 77) M. G. Gouda. *Toward a Logical Theory of Communication Protocols*. Ph.D. Thesis Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1977. Also a CCNG Tech. Report T-74, University of Waterloo.

(Guttag 77) J. V. Guttag, Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, Vol. 20, No. 6, June, 1977, pp. 396-404.

(Greif 75) I. Greif, *Semantics of Communicating Parallel Processes*. M.I.T. Project MAC TR-154, September, 1975.

(Hoare 78) C.A.R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, Vol. 21, No. 8, August, 1978, pp. 666-677.

(Jensen 78) E. D. Jensen, The Honeywell Experimental Distributed Processor - An Overview. *Computer*, January, 1978, pp. 28-39.

(Keller 76) R. M. Keller, Formal Verification of Parallel Programs. *Communications of ACM*, Vol. 19, No. 7, July, 1976, pp. 371-384.

(Lamport 78) L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Vol. 21, No. 7, July, 1978, pp. 558-565.

(Laventhal 79) M. Laventhal, Synchronization Specifications for Data Abstractions. *Proc. Specifications of Reliable Software*, April, 1979, pp. 119-125.

(Lelann 77) G. Lelann, Distributed Systems-Towards a Formal Approach. *Information Processing 77*, edited by B. Gilchrist, North-Holland Co., 1977.

REFERENCES (cont.)

(Liskov 75) B. H. Liskov, et, al, Specification Techniques for Data Abstractions. IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March, 1975, pp. 7-19.

(Parnas 72) D. L. Parnas, A Technique for Software Module Specification with Examples. Communications of the ACM, Vol. 15, No. 5, May, 1972, pp. 330-336.

(Robinson 77) L. Robinson, et. al, A formal Methodology for the Design of Operating System Software. In R. Yeh (ed.), Current Trends in Programming Methodology, Vol. I, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

(Riddle 79) W. E. Riddle, et. al, Abstract Monitor Types, Proc. Specifications of Reliable Software, April, 1979, pp. 126-138.

PROTOCOLS FOR DATING COORDINATION

Danny Cohen and Yechiam Yemini
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90291

Abstract

This paper is about the process of specifying protocols for computer communication. It uses a dating coordination protocol as an example for an interprocess communication. Since this problem has some timing constraints built into it, the resulting discussion is different than most of the more familiar protocols which do not have requirements associated with timing. Several protocols are discussed here in order to illustrate different aspects of the specification issue.

1. BACKGROUND

In the rural area of Oceanview, Kansas, people are too busy to arrange their own dates. In order to alleviate this problem a dating center (hereafter "C") was founded by the local church.

The dating center operation is generally simple. When a person (hereafter "X") is interested in a date, he writes a letter to the center, requesting a date with his sweetheart (hereafter "Y"). It is a pity that there are not many phones in this area, isn't it?

Typically a requested date is blessed unless it is found to be in conflict with the center's policy, due for example to the lack of common approach to the arts. In the lucky event that the date is blessed a time is assigned for X and Y to meet at the center. Letters are then sent to both, notifying them about the particulars of the upcoming event. Needless to say, church tradition strictly forbids X and Y from being in direct communication before their supervised meeting at the center.

Sections 2 through 7 of this note discuss a protocol for this coordination.

Most of the population of Oceanview is quite happy with this dating service. However, the Japanese community of Oceanview found it hard to take advantage of this dating service. Due to different cultural background it is very hard to find a common approach to the arts with the rest of the town people.

Therefore the local Buddhist temple decided to sponsor another dating center operating with different rules.

Section 7 discusses a different dating coordination protocol, geared to the needs of the local Japanese community.

2. THE SPECIFICATION OF THE OBJECTIVES

The objective of this protocol is to allow X to cause C to dispense the same time and place assignment for blessed dates, both to himself (X) and to the other party (Y). This operation should succeed in spite of the postal communication which in that part of the country (unlike others) may lose letters, delay them for an arbitrary amount of time (hence causing occasional "out of order" delivery) and, believe it or not, deliver several duplicates of the same letter. It is assumed that no Y ever declines to accept a blessed date.

3. THE SPECIFICATION OF THE PROTOCOL

The protocol employs the following letters:

[1] X=>C: <DATE-REQUEST>, X, Y, RX

This letter is used by X to request C to issue the time and place assignment to both X and Y. This issuance will have the effect of notifying Y that X is interested in dating her. The RX in this letter is a reference number that X assigns to this expected date.

[2] C=>X: <HEARD-YOU>, RX

This letter is sent by the center to acknowledge the reception of X's letter. It constitutes neither an approval nor a denial of the date.

[3] C=>X,Y: <BLESSED>, X, Y, T&P, RX, RC

This is an official notification of the blessed date which is sent to both parties. T&P is the specification of the time and place assigned for this date, and RC is the reference number assigned to it by C.

[4] C=>X: <DENIED>, RX, RC

This is the official denial of the date, which is sent only to X.

[5] X,Y=>C: <TNX>, RC

This is the letter that X and Y send to the center upon receiving either a <BLESSED> or a <DENIED> letter.

4. THE OPERATION OF THE X AND Y PROCESSES

State	Condition	Action	Next State
1 IDLE:	wants to date	send <DATE-REQUEST> set timers T1 and T2	2
	rec'd <BLESSED>	send <TNX>	4
2 WAIT-FOR-ACK:	T1 goes off	send <DATE-REQUEST> set timer T1	2
	T2 goes off	----	1
	rec'd <HEARD-YOU>	set timer T3	3
	rec'd <BLESSED>	send <TNX>	4
	rec'd <DENIED>	send <TNX>, expunge RA	1
3 WAIT-APPROVAL:	T3 goes off	----	1
	rec'd <BLESSED>	send <TNX>	4
	rec'd <DENIED>	send <TNX>, expunge RA	1
4 HAPPY:	date termination	----	1

Any other event is ignored. T1 is presumably very much smaller than T2.

5. THE OPERATION OF THE C PROCESS

State	Condition	Action	Next State
1 IDLE	rec'd <DATE-REQUEST>	send <HEARD-YOU>	2
2 CHECKING	date approved	send <BLESSED> to X and Y set timers T4 and T5	3
	date denied	send <DENIED> to X set timers T4 and T5	6
3 WAITXY	rec'd <TNX> from X	----	5
	rec'd <TNX> from Y	----	4
	T4 goes off	send <BLESSED> to X and Y set timer T4	3
	T5 goes off	----	1
4 WAITX	rec'd <TNX> from X	----	1
	T4 goes off	send <BLESSED> to X set timer T4	4
	T5 goes off	----	1
5 WAITY	rec'd <TNX> from Y	----	1
	T4 goes off	send <BLESSED> to Y set timer T4	5
	T5 goes off	----	1
6 WAITXX	rec'd <TNX> from X	----	1
	T4 goes off	send <DENIED> to X set timer T4	6
	T5 goes off	-----	1

Any other event is ignored. T4 is presumably very much smaller than T5.

The above is, obviously, the description of a single instance of C, dedicated to handle a specific DATE-REQUEST. It is assumed that C has a central process which identifies new requests, and creates new instances to handle them.

6. DISCUSSION

We believe that this protocol is capable of performing a good job.

However, it is obvious that the specifications of the objectives, as given in section 2, do not cover all the issues which are covered by the design and by the implementation specification of this protocol.

We suggest that the reason is that the real objectives are not fully specified. Therefore, the above protocol is an "overkill" for the specification, and simpler protocols which meet the same given specifications may be devised.

6.1 Simpler Protocols

For example, in order to meet the objectives, as specified above, only the <DATE-REQUEST> and the <BLESSED> messages are needed. Neither the <HEARD-YOU>, nor the <DENIED>, nor the <TNX> are needed. Similarly, neither the timeouts nor the retransmission are needed.

Hence, a possible simpler protocol has only the <DATE-REQUEST> and the <BLESSED> messages, without the <HEARD-YOU>, the <DENIED>, the <TNX> and any of the timeouts.

It is not hard to verify that this protocol meets the objectives as specified in section 2. Obviously it is less robust in respect to communication imperfections, but this was not specified there.

It is obvious that what we meant is to make sure that the transactions are successfully conducted, in spite of the unreliability of the supporting communication medium.

However, the term "sure" above has to be taken with a grain of salt. Obviously it is impossible to have a perfectly reliable communication on top of an unreliable medium. What if the Oceanview post office goes suddenly on strike ?!! Even though federal employees are not expected to strike, this is still a possibility.

In more precise terms, what is meant is that the success probability, of the entire transaction, should be above a certain threshold, in spite of a lower (positive) communication success probability.

The above protocol is probably a pretty good answer for this interpretation of the objectives.

We suggest that in general the problem specification should include, quantitatively, the reliability parameters and other relevant information about the environment in which the problem is embedded, like the performance of the supporting communication system, for example.

If only the increased success probability is added to the objectives, then there is even a simpler protocol which still meets the objectives.

This protocol, as the previous one, has only the <DATE-REQUEST> and the <BLESSED> messages. It does not have the <HEARD-YOU>, the <DENIED>, the <TNX> messages and any of the timeouts. It achieved the desired increase of reliability by flooding the communication system with multiple copies of each message, ad infinitum. One can prove that if the probability of a successful delivery of a message is arbitrarily small, but greater than zero, then the probability of a successful conclusion of a transaction is arbitrarily close to 100%.

6.2 Efficiency and Cost Considerations

However this protocol is not considered acceptable since some cost is associated with the transmission of messages. It results from both the communication cost, and/or processing limitations. In our story the transmission cost is paid in postal stamps, and processing limitation are reflected by the understanding that if too many copies of the first <DATE-REQUEST> reach the center, the center may never have the chance to notice another request.

Therefore, we suggest that in addition to specifying the desired performance, the cost parameters must be specified, too. One should be able to specify that he is very much interested in having a date, but that he is not willing to pay more than so many stamps for it.

However, there is an even simpler protocol which is based on the center continuously telling everyone to be always at the center, just in case someone wants to date them. This can guarantee (i.e., with probability arbitrarily close to 100%) that if your requested date is blessed (or even if it is not) then when you go to the center, your date is there. It is conceivable that some people may have some objections to this procedure. Camping on the front lawn of the center for several weeks before the data commences, is not that much fun.

The problem of missing knowledge in the protocol objectives specification causes major difficulties not only to the protocol designer community, but also to the protocol verification community.

6.3 Complete Specification

There are probably several other possible protocols which meet these objectives, and have similar flavor. All of them result from the lack of complete specification.

The missing specification includes typically the **"obvious"** details, which do not require explicit mention, but are implied from our general experience in dealing with communication protocols. They include the performance parameters, the cost parameters both for the communication and the processing resources, the cost associated with omission and commission errors, and the like.

One may argue that this type of specification does not belong to the particular problem at hand, but to the general domain of message communication, and separate the specifications into two parts, the particulars of the given problem and the generalities of the domain.

We suggest that in message communication the domain has to be **parameterized**, where the assignment of the parameters is a part of the specification of a particular problem. The model of probabilistic delivery, communication and processing costs, omission and commission errors and the like belong to the domain, but the value of these probabilities and the various costs are parameters which depends on each specific problem.

It is unfortunate that we still do not know how to completely specify the objectives of a protocols. These objectives must include the parameters of the environment, such as the supporting communication medium (below) and the expected traffic (above), the various costs associated with usage of resources such as message transmission, processing and storage, and with delays, communication errors, and the like.

It is amazing that even though we do not yet possess the ability to accurately specify protocol objectives, we have enough "engineering" experience to guide us in implementing protocols which do a remarkably good job of message communication.

The nature of these performance and cost related parameters introduce the notion of **approximations**. Protocols are not either correct or **incorrect**, but are more like many numeric problems which have a continuous spectrum of accuracy.

For some problems the objectives are such that the correctness of message delivery is more important than its efficiency. File transfer requires that each bit is reliably received, even if this implies delays. Speech communication requires efficiency and low delays, more than perfect accuracy. For speech a certain amount of errors is tolerated if this is necessary for delay considerations. Obviously, this cannot be carried to the extreme in which a zero delay is achieved by compromising (totally?) the accuracy of the signal.

7. THE JAPANESE DATING COORDINATION PROTOCOL

The Japanese community in Oceanview is much more permissive than their neighbors downtown. Direct communication between the parties is not only allowed before the date, it is even encouraged. The center role is limited to providing consultation, addresses and other matters of importance.

After choosing his sweetheart, a person writes her directly and invites her to meet him, in the temple gardens, at a certain time. Typically the recipient responds rather anxiously, and sends a letter of confirmation.

Due to old Japanese tradition one loses face if stood up for a date. Losing face, in this community, results always in the tragic act of harakiri.

When the number of these tragic acts soared, the temple leaders were able to correlate it with the low quality of the local postal service.

Without delay they set out to design a protocol which would assure the safety of all dates, thus eliminating the recurrence of these tragic consequences.

Unfortunately this task proved to be more difficult than first expected.

The reason for this difficulty is that since losing face is a serious matter, in fact a matter of life and death, the required level of safety must be 100%, not a bit less.

It turned out that no protocol could guarantee that absolute reliability, even with any finite delay. When a young mathematician managed to prove that such a protocol could not exist, the wise men at the temple were very disappointed.

For the benefit of the interested reader the proof is sketched below.

Suppose that P(N) is a protocol which under the existing conditions could guarantee a safe date, where the probability of a message to be successfully delivered is less than 100%.

The last message, the Nth one, could not carry information which is essential for the safety of the date, since its sender cannot be sure that it was received by the other party. Since it carries no essential information, it could be eliminated, and a stamp can be saved. Since P(N) is a safe protocol, so is P(N-1), which is the protocol consisting of the first N-1 messages of P(N).

Therefore a P(N-2) exists, too. So does P(N-3) and so on. Therefore, P(0) exists. This means that a safe date can be arranged without any exchange of messages.

Since the dating process is essential for the well being of the community, other communication alternatives are under study now.

8. CONCLUSIONS

Complete specification of protocols are needed for optimal implementation and for verification.

Even though we, as a community, have gathered an impressive experience in implementing protocols, our ability to specify accurately and precisely the objectives of protocols still leaves a lot to be desired.

We suggest that the specifications of message communication protocols should include the parameters of the environment, the parameters of the performance and cost constraints. The cost should include the effects of errors,. of both kinds. It should be kept in mind that absolute reliability cannot be guaranteed in environments which are less than perfect.

It is probably possible to divide the specifications into (i) the particulars of the specific problem at hand, and (ii) the generalities of the message communication domain. However, due to the diversity of this domain, (ii) may be specified only as a parameterized domain, where the specific values of its parameters are part of (i).

9. AN IMPORTANT NOTICE

Throughout the paper pursuers are referred to as belonging to the male gender, whereas females are considered always as lovely sweethearts who are always anxious to be approached. We would like to emphasize that by no means do we intend to suggest that this is a correct reflection of the roles of human beings. We are well aware that in real life the division between "pursuers" and "pursuees" does not follow the sex lines as closely as we used to pretend.

As a matter of fact, the authors of this paper are quite aware that in this day and age liberated women may play the "aggressive" role more often than their counterparts.

We are well aware of it, and regret having been born too early to enjoy it.

10. ACKNOWLEDGMENTS

We would like to thank our many colleagues who helped us putting these ideas in this form. Our many discussions helped us understand these issues in a better way. In particular we would like to thank Jon Postel of ISI and Carl Sunshine of the Rand Corporation. Special thanks to Debe for advising us about the dating issues.

6 April 1979

MULTIPLE COPY CONTROL TECHNIQUES

DISTRIBUTED CONTROL OF UPDATES IN MULTIPLE-COPY DATABASES: A TIME OPTIMAL ALGORITHM

R. J. Ramirez and N. Santoro*

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Abstract

In this paper, the problem of updating a database with multiple copies under distributed control is addressed.

An update synchronization algorithm for databases, whose copies are distributed on a store-and-forward synchronous network is presented and its complexity is analyzed. The proposed algorithm is shown to be time optimal within an additive constant for networks of arbitrary topology. The algorithm incorporates a simple priority scheme to resolve concurrent updates.

I. INTRODUCTION

When a single database is accessed by several users through a communication network, it may be advantageous to store the same data at more than one center in the network. For example, consider a database for which the expected number of "read" accesses is very large as compared to the expected number of "update" accesses. If a copy of the data is stored at each center of the network, then "read" requests can be serviced locally, reducing the operational cost and the response time of the database.

The advantage of a multiple copy distributed database are essentially based on the availability of duplicate data. Namely, this redundancy offers an increased reliability, a quicker query response, and a potential for upward scaling of database capacity [11].

The disadvantages rest on the facts that updates, originated at various centers, must be reflected in every copy, and that transmision delays, as well as the order in which updates are applied, must be taken into account to maintain internal consistency in the database.

It could be desirable to centralize the control function, i.e. to make a single center responsible to maintain the consistency and integrity of the database. In such a scheme, all other centers will request permission from the

* On leave from Instituto Scienze Informazione, University of Pisa, Corso Italia 40, 56100 Pisa, Italy.

control center to update the database and the control center will deny or grant the request on the basis of current locks. If the request is granted, then each copy is updated, and acknowledegement is sent to the control center that will then release the associated lock. For a more complete discussion and some examples of centralized control see [1, 4, 12].

An interesting alternative is represented by the distributed control scheme. In this model, the control function is distributed among all centers in the network. To make the control possible, it is necessary for each center to communicate (exchange messages) with other centers (its neighbours); and, in order to maintain the internal consistency in the database, a synchronization technique is needed. In the literature several algorithms for distributed control have been presented; they are designed to work with networks of a given topology. Namely, the centers of the network must form a sequential chain [6, 7], a daisy chain [3, 13, 14], or a star [2]; and each center must have knowledge of the network topology.

In this paper we continue the analysis of multiple copy databases with distributed control, and present a general and efficient update synchronization algorithm for networks of arbitrary topology. In order for the algorithm to work, each center needs only to know who are its neighbours, and no additional knowledge of the network topology is required. Concurrent updates are resolved by a priority mechanism that guarantees proper sequencing and avoids race conditions. In the next section, the problem is stated formally. In section III, a restricted environment is considered, a naive algorithm is described, and an improved algorithm is presented and proved to be optimal in the restricted context. In section IV we show how to modify the algorithm to work in a general environment increasing the time complexity only by a small constant, and we describe the proposed algorithm formally.

II. DEFINITIONS

Let us formally describe the framework and define some terms that will be used throughout the paper. The network is composed of n centers, each mantaining a copy of the database. Each center replies to query and update requests which are originated locally or received from some other center. At each center, messages are sent to and received from its neighbour centers. This situation can be represented using a linear graph G=(N,A), where N is a set of nodes and A a set of arcs: each node n(i) ∈ N represents a center where a copy of the database resides, and each arc a(i,j) ∈ A represents a direct communication link between n(i) and n(j) (in our application a(i,j)=a(j,i)). If a(i,j) ∈ A then n(i) and n(j) are said to be neighbours. Each center mantains a list D(i) of its neighbours,

$$D(i) = \{ n(j) \in N \mid a(i,j) \in A\}.$$

At any given time, each node can be in one of the following states

$$S = \{\text{Available, Prepared, Counting, Update}\}.$$

The model is based on the following basic assumptions:

i) Synchronization.
The clocks at each center are synchronized. Imperfect synchronization could be included in our model by using a quantity d(i,t) defined as the difference between the clock at center i and the "time" t (see [5]).

ii) Partial Reliability.
During an update, the topology of the network will not change. Partial reliability does not imply any other assumption, neither on the topology nor on the general reliability of the network.

iii) Consensus.
An update will be performed only if all centers agree on the update (see [9]).

In the next section we will analyze a "naive" algorithm for update synchronization. We will then show how to modify it in order to speed up the synchronization time, and we will prove that the resulting algorithm is time optimal.

III. RESTRICTED CONTEXT

In this section we will consider a restricted environment to simplify the presentation. In section IV we will show how to extend the result to the general case. The restricted context is as follows:

i) the database is on an acyclic network, i.e. G is a tree.

ii) at any given time there is at most one active update request.

iii) to transmit a message across any link takes a single unit of time.

The naive algorithm.

In a "naive" algorithm for the above environment, a node n(0) receives an update request originated locally. It enters state "Prepared", sends a "request" message to all its neighbors, and waits for replies from them all. Node n(0) at this time, does not have any information on the topology of the network, except which centers are its neighbors. In fact, the network topology might have changed since the last update, due to breakdown or to the reactivation of a communication link. Therefore, it is necessary for n(0) to obtain from all centers not only the consensus to update, but also some information on the topology of the network. Namely, it needs to know the radius, i.e. the time required for a message from n(0) to reach the farthermost center in the network.

Let us now continue the description of the algorithm. In a recursive fashion, node n(i), upon receipt of a "request" message from n(j), enters state "Prepared"; sets sender(i)=n(j); sends a "request" message to all its neighbors except n(j); and waits for acknowledgement from them all.

If n(i) is a leaf then, after entering state "Prepared", it sends to n(j) an "Ackowledgement" message containing a counter T(i) (in the restricted environment T(i) will be a variable initially set to zero) and waits for the "Update" signal.

In a recursive manner, node n(j) waits for "Acknowledgment" messages containing the counter T from all its neighbours (except sender(i)). Then, it sends to sender(i) an "Acknowledgement" message containing the counter T(j) defined as the last received counter T(k) incremented by one.

When n(0) has received "Acknowledgement" messages from all its neighbours, it can start the update.

In fact T(0) = max{T(i) | n(i) ∈D(0)} is exactly the radius.

The synchronization of the update proceeds as follows:

i) node n(0) enters state "Counting" and sends to all its neighbours an "update" message containing the "update vector" [5] and a new counter TP(0)=T(0)+1.

ii) in a recursive fashion, node n(i), upon receipt of an "Update" message from n(j), enters state "Counting"; saves the "update vector"; sets TP(i)=TP(j)-1; sends the update vector and TP(i) to all its neighbours except n(j), and starts the count down of TP(i). When the counter reaches zero, then node n(i) will change its state to "Update".

iii) when in state "Update", node n(i) performs the update and, when completed, it enters state "Available".

A state transition diagram is shown in Figure 1.

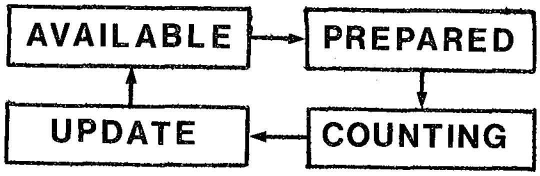


Figure 1. State transition diagram for the restricted context.

It is easy to show that all centers will count zero simultaneously, and that the time required by the naive algorithm to synchronize the network is 3*T(0). In fact, it takes T(0) steps for a "request" to reach the farthermost node; the "acknowledgement" message from that node to n(0) will also take T(0) steps to arrive; and, finally, it takes T(0) steps for all nodes to enter state "Update" simultaneously.

The improved algorithm.

We will now show how to modify the previous algorithm to reduce the synchronization time. The previous algorithm performs basically two operations: it finds the radius, and then it sends the update signal. In order to speed up the process, the above operations must be performed as simultaneously as possible. We will now describe the algorithm for the restricted environment with an example, and analyze its complexity. In section IV we will formally present the general algorithm.

Consider the graph in Figure 2(a) where index i represents node n(i).

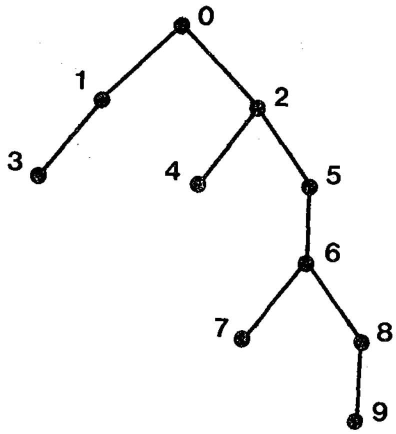


Figure 2(a).

Initially, the algorithm works as the naive method. At time t=0, n(0) receives an update request generated locally. It then enters state "Prepared" and sends a "request" message to all its neighbours, which in turn enter state "Prepared" and send a "request" message to their neighbours. This process continues recursively. Eventually, a message reaches a leaf node. In our example, at time t=2 (i.e. after two steps), both n(3) and n(4) receive the "request" message. They enter state "Prepared" and send back an "Acknowledgement" message with counter set to zero, (see Figure 2(b)).

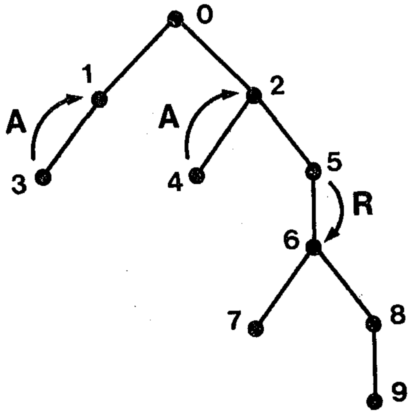


Figure 2(b).

At a bifurcation node, e.g. node n(2) in Figure 2(b), all "Acknowledgement" messages, except the last one to arrive, are destroyed. That is, only the last "Acknowledgement" and counter are considered. When the last "Acknowledgement" arrives to a node, for example n(1) at time t=3, this node sends to its "father", n(0), an "Acknowledgement" message and the counter T(1)=T(3)+1=1. At time t=4, the "Acknowledgement" message has reached n(0). In our example, the synchronization process can now start. In general, n(0) waits until the message before the last one has arrived. Upon receipt of such "Acknowledgement", n(0) sends to the only unacknowledged neighbour an "Update" message containing the "update vector" and the counter TP(0)=received_counter+1=T(1)+1=2. This situation is shown in Figure 2(c).

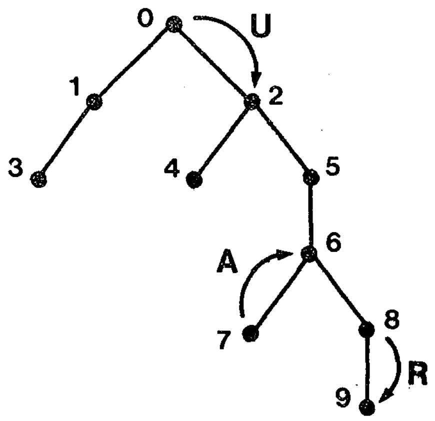


Figure 2(c).

At time t=5, node n(2) sends the "update vector" and the counter TP(2)=TP(0)+1=3 to the unacknowledged neighbour n(5). If there is more than one unacknowledged neighbour, then the node waits until all neighbours except the last have sent an acknowledgment. This process is repeated in a recursive fashion until the "Acknowledgement" message from the farthermost node and the incoming "Update" message meet in a node, as shown in figure 2(d) (actually, the two messages may "jump" over each other; this case is easily solved with a single test).

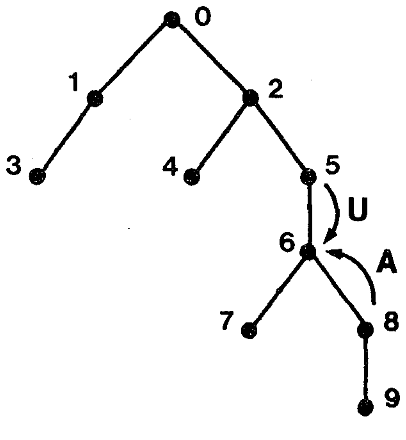


Figure 2(d).

Node n(6) now knows how distant is the farthermost node. In fact, this quantity is precisely TP(5)+1. Node n(6) enters state "Counting", sends the synchronization message and TP(6) to its neighbours, and starts the count down of TP(6). Analogously, when a node n(i) receives an update message and the counter TP(j), it will set TP(i)=TP(j)-1, send the update message to all its neighbours, enter state "Counting", and start the count down of TP(i). When the counter reaches zero, the node changes its state to "Update" and performs the required update. When the update has been completed, the node enters state "Available".

Analysis of the algorithm.

Let us now analyze the complexity of the above algorithm. We are interested in evaluating the delay between the time an applicable request* is originated and the time the update is actually performed. This delay can be expressed in terms of the number of steps needed for the originating node to reach state "Update".

In order to analyze the complexity of the algorithm, let us introduce some terminology. Given a rooted tree T, the radius r is the maximum distance from a node in T to the root, and the diameter d is the maximum distance between any two nodes in T.

When a node originates an update request, this node can be regarded as the "root" of the tree. In Figure 3, only the longest and the second longest path from the root, of length a and b respectively, are shown. Obviously r=a; it can be proved that d=a+b [10].

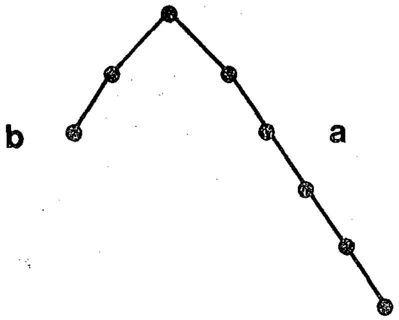


Figure 3. Radial path (a) and second longest path (b) from the root.

* An applicable request is a request that is not going to be preempted by a higher priority request. In the resticted environment every request is applicable.

In our algorithm, after the root generates the update request, it will take b steps for the "request" signal to reach the leaf in the second longest path; and it will take b steps for the "acknowledgement" signal from that node to reach the root. The "synchronization" signal (i.e. the update message in our algorithm) will then be sent along the radial path, and it will eventually meet the acknowledgement signal coming from the leaf of that path. The two signals will meet (or jump over each other) after a-b steps. At this point, the "Counting" signal is originated. Before the update can be performed, this signal must reach all nodes, including the farthermost ones. It can be shown that the farthermost nodes are not more distant from the "meeting node" than a bottom leaf in the second longest path. That is, we need other (a-b)+b steps before we can perform the update. In total, we need 2b+a-b+(a-b)+b=a+(a+b)=r+d steps. That is, the algorithm requires d + r steps to synchronize an arbitrary tree network of radius r and diameter d. We can now show that:

Proposition.

The proposed algorithm is time optimal for tree networks.

The above result follows from the fact that at least d + r steps are needed to synchronize a tree. This lower-bound has been proved for a tree of cellular automata [10]. The proof relies only on the tree-structure of the network and not on the computational power of the nodes. Therefore, it holds for our model and proves the above proposition.

IV. GENERAL ENVIRONMENT

In the above sections, we have presented a time optimal algorithm for the restricted environment. Namely, the following restrictions were made: (i) the network has a tree structure; (ii) there are no concurrent updates; and (iii) the time to transmit any message from a node to its neighbours is unitary. These assumptions were made only to simplify the description and analysis of the algorithm. In fact, a concurrent update resolution mechanism can be easily incorporated in the algorithm without increasing its complexity; more operations will be performed at each step, but the number of steps will be the same. Analogously, the algorithm can be easily modified to work on general graphs, and with different transmissions times, increasing the time complexity only by a small constant.

In next sections we will informally show how to modify the algorithm to work on the general environment and formally describe the resulting algorithm.

Collision resolution.

In this section we show how to incorporate in the algorithm a collision resolution mechanism based on priorities without increasing its time complexity.

The proposed priority scheme is as follows:

- each node is assigned a unique index (e.g. an integer); nodes do not need to know everyone else's index, but only that the indices are unique.
- the priority function Φ is available to all nodes.
- when a node is in state "Prepared", it will accept an update request with higher priority, preempting the current one; the preempted request will be saved by its originator in a queue for future processing.
- when a node is in state "Counting" or "Update", it will ignore any update request.

The priority function can be formally described as a mapping

$$\Phi : Z \times R \to Z$$

where Z is the set of positive integers, and R is the set of reals; $\Phi(i,t)$ denotes the priority of an update request originated at node n(i) at time t, and is such that:

$$\text{(i)} \quad \forall\, t' > t \quad \Phi(i,t) > \Phi(i,t')$$
$$\text{(ii)} \quad \forall\, j < i \quad \Phi(i,t) > \Phi(j,t)$$

that is, Φ is a decreasing function of the time and an increasing function of the indices. In other words, if two (or more) update requests originate at the same node, the second request can proceed only after the completion of the first update (this guarantees proper sequencing); if several update requests are originating at the same time in different nodes, the request originating at the node with highest index will be processed first (this avoids race conditions and the consequent undeterministic behaviour of the system). Let us note that because the requirement for consensus, the transmission of negative acknowledgements is not needed; the arrival of a higher-priority request will perform the same function.

General graphs.

Throughout the above discussion, we have been dealing with tree networks. However, the proposed algorithm can be used for a general network. In fact, given a network of arbitrary topology and given a starting node (i.e. a node originating an update request), we can construct a spanning tree rooted in that node and apply the algorithm to the obtained tree. Let us note that to construct a spanning tree T of a graph G is equivalent to determine for each node $n(i) \in N$ the list $L(i)$ of its neighbours in T. Obviously $L(i) \subseteq D(i)$. In order not to greatly increase the time complexity, this "tree reconstruction" must be done while executing the algorithm, and the resulting tree must be of minimum radius. This can be easily achieved in the following manner: If a node n(0) is originating the update, then it will send a "reconstruction" message to all its neighbours in the graph and wait for acknowledgment. The set L(0) will be formed by all the neighbours sending an acknowledgment. In general, a node n(i) will ignore all "reconstruction" signals, except the first, for a given update. Let n(j) be the sender of the first received reconstruction signal for an update. Then, node n(i) will send an acknowledgment to n(j); simultaneously send a "reconstruction" signal to all its neighbours in the graph, except n(j); and wait for acknowledgment. The set L(i) for the given update will be formed by n(j) and by all nodes $n(k) \in D(i)$ that have replied (always within two time steps). It can be shown that this technique constructs the tree of minimum radius [8] and increases the total time complexity by only two steps; i.e. the modified algorithm works in d+r+2 steps.

The last assumption made in the restricted environment was on the time required to transmit a message across a link. In general, the time to transmit a message x from node n(i) to node n(j) is $t(i, j, x) \neq 1$. To make the algorithm work for this general case, where $\forall\, n(j) \in L(i)\ \forall\, x$ $t(i, j, x)$ is known at node n(i), we need only to modify the counters and to take into account what kind of message we are sending or receiving. All these modifications do not involve any major change, and for simplicity are not explicitly included in the algorithm.

The algorithm.

In order to describe the algorithm including the collision resolution mechanism formally, let us review the four possible messages:

1) update request - <"R",sender, originator, time>
2) acknowledgment - <"A",sender, priority, counter>
3) synchronization - <"U",sender,update vector,priority, counter>
4) counting - <"C",sender, update vector,counter>

where the counter is analogous to a time stamp, and sender and originator are the indices of the sender node and of the originator node of the request, respectively. The algorithm is expressed in terms of which operations a node must perform, depending on its state and on the received message.

We assume that each node n(i) already knows the set L(i).

node(i) is in state "Available".

```
<"R", n(k), n(j), time>
   begin
   if n(i) = n(j) then {locally generated update request}
      begin
      compute priority P, L(i) and update vector V
      send <"R", n(i), n(i), time> to all n(p) ∈ L(i)
      vector(i) = V; prepared(i) = n(i)
      sender(i) = n(i); priority(i) = P
      if |L(i)| = 1 then {it is also a leaf}
         send <"U",n(i),vector(i),priority(i),0>to n(p)∈ L(i)
      end
   else
      begin
      compute priority P, L(i)
      prepared(i) = n(j); sender(i) =n(k); priority(i) = P
      if L(i) - n(k) = ∅ then { n(i) is a leaf}
         send <"A", n(i), priority(i), 0> to n(k)
      else
         send <"R", n(i), n(j), time> to all n(p) ∈ L(i)-n(k)
      end
   state(i) = "Prepared"
   Copy(i) = L(i) - sender(i)
   end
```

Ignore other messages

node n(i) is in state "Prepared".

```
<"A", n(k), P, C>
   begin
   if P < priority(i) then {ignore message}
   else
      begin
      Copy(i) =Copy(i) - n(k)
      if prepared(i) = n(i)then {this node isthe originator}
         if |Copy(i)| = 1 then {start synchronization}
            send <"U",n(i),vector(i),priority(i),C+1>
                  to n(p) ∈ Copy(i)
      if holding(i) then {holding synchronization message}
         begin
         if |Copy(i)| = 1 then {forward synchronization}
            send <"U",n(i),vector(i),priority(i),C+1>
                  to n(p) ∈ Copy(i)
         end
      else
         if |Copy(i)| = 0 then { send ack to sender(i)}
```

```
            send <"A", n(i), priority(i), C+1> to sender(i)
      end
   end

<"U", n(k), vector, P, C>
   begin
   if P ≠ priority(i) then {ignore message}
   else
      begin
      if |Copy(i)| = 1 then {forward synchronization}
         send <"U", n(i), vector, priority(i), C+1>
              to n(p) ∈ Copy(i)
      else
         begin
         if |Copy(i)| > 1 then {holding synchronization}
            holding(i) = true
         else
            begin
            send <"C", n(i), vector, C+1> to n(p) ∈ L(i)
            vector(i) = vector; counter(i) = C+1
            state(i) = "Counting"
            while counter(i) ≠ 0 decrement counter(i)
            state(i) = "Update"
            Perform update as described by vector(i)
            holding(i) = false
            state(i) = "Available"
            end
         end
      end
   end

<"C", n(k), vector, C>
   begin
   Copy(i) = L(i) - n(k)
   vector(i) = vector
   send <"C", n(i), vector(i), C-1> to n(p) ∈ Copy(i)
   state(i) = "Counting"
   while counter(i) ≠ 0 decrement counter(i)
   state(i) = "Update"
   Perform update as described by vector(i)
   holding(i) = false
   state(i) = "Available"
   end

<"R", n(k), n(j), time>
   begin
   Compute priority P
   if P < priority(i) then
      if n(j) = n(i) then {new locally generated
                              update temporarily rejected}
         save vector and retry later
   {priority is higher than previous request}
```

```
else
   begin
   if prepared(i) = n(i) then  {org. of old request
      old update temporarily rejected}
      save vector(i) and retry later
   begin
   if n(i) = n(j) then {locally generated update request}
      begin
      compute priority P, L(i) and update vector V
      send <"R", n(i), n(i), time> to all n(p) ∈ L(i)
      vector(i) = V; prepared(i) = n(i)
      sender(i) = n(i); priority(i) = P
      if |L(i)| = 1 then {it is also a leaf}
         send <"U", n(i), vector(i), priority(i), 0)
              to n(p) ∈ L(i)
      end
   else
      begin
      compute priority P and L(i)
      prepared(i) = n(j); sender(i)=n(k); priority(i)=P
      if L(i) - n(k) = ∅ then { n(i) is a leaf}
         send <"A", n(i), priority(i), 0> to n(k)
      else
         send <"R", n(i), n(j), time>
              to all n(p) ∈ L(i)-n(k)
      end
      Copy(i) = L(i) - sender(i)
      end
   end
end
```

node n(i) is in state "Counting".

Ignore all messages

node n(i) is in state "Update".

Ignore all messages

V. CONCLUSIONS

In this paper, an update synchronization algorithm for databases, whose copies are distributed on a store-and-forward synchronous network, has been presented and its complexity analyzed. It has been shown that the algorithm is time optimal within a small additive constant, for networks of arbitrary topology.

There are some obvious limitations in the proposed method, for example knowledge of the message transmission delays is required, and no provision for retransmission of messages is included.

On the other hand, the algorithm does not require any knowledge of the general topology of the network; therefore changes in topology can occur when there are no active update requests on the network.

Acknowledgment.

The authors would like to thank Professor F. W. Tompa for his helpful comments. The financial support of the University of Waterloo and of the Natural Sciences and Engineering Research Council Canada are also gratefully acknowledged.

REFERENCES

[1] Alsberg P.A., Belford G.G, Brunch S.R., Synchronization and deadlock, CAC document 185, University of Illinois, 1976.

[2] Chu W.W., Performance of directory systems for databases in star and distributed networks, Proc. AFIP Conf. 1976.

[3] Ellis C.A., A robust algorithm for updating duplicate databases, Proc. 2nd Berkley Workshop on Distributed Data Management and Computer Networks, 1977.

[4] Garcia-Molina H., Performance comparison of two update algorithms for distributed databases, Proc. 3rd Berkley Workshop on Distributed Data Management and Computer Networks,1978.

[5] Gelenbe E., Sevcik K., Analisys of update synchronization for multiple copy databases, Proc. 3rd Brekley Workshop on Distributed Data Management and Computer Networks,1978.

[6] Legoff H., Lelann G., Communications and synchronization tools, 1st ECI Conf., 1976.

[7] Lelann G., Distributed systems - towards a formal approach, Proc. IFIP Congress 1977.

[8] Moore E.F., The shortest path through a maze, Proc. Int. Symp. in Theory of Switching 1959.

[9] Mullery A.P., The distributed control of multiple copies of data, IBM research report RC5782, 1975.

[10] Romani F., Cellular automata synchronization, Information Sciences 10, 1976.

[11] Rothnie J.B., Goodman N., A survey of research and development in distributed database management, Proc. VLDB Conf. 1977

[12] Stonebraker M., Neuhold E., A distributed database version of INGRES, Proc. 2nd Brekeley Workshop on Distributed Data Management and Computer Networks, 1977.

[13] Thomas R.H., A solution to the update problem for multiple copy databases which uses distributed control, BBN report 3340, 1975.

[14] Thomas R.H., A majority concensus approach to concurrency control for multiple copy databases, BBN report 3733, 1977.

CONCURRENCY CONTROL IN A MULTIPLE COPY DISTRIBUTED DATABASE SYSTEM

Wen-Te K. Lin

Sperry Research Center

Abstract

The concurrency control mechanism employed by the System for Distributed Databases, SDD-1, avoids both central site control and global data locking. This paper proposes modifications to the concurrency control mechanism of SDD-1 which eliminate the need for timestamps on data items and weaken the constraints of some of the read-write protocols. These modifications reduce the amount of storage required and allow accommodation of existing databases which may make no provision for stored data item timestamps. A new protocol W is introduced which requires that write actions which participate in certain cycles in the class conflict graph be synchronized at certain sites. This protocol may reduce the degree of concurrency supported by the system. Existing SDD-1 protocols are augmented with weaker forms of these protocols which allow more flexibility in scheduling read and write actions under certain conditions. Timestamps of some actions are allowed to be changed in order to reduce the synchronization delay experienced by other actions, thereby increasing concurrency. A proof of correctness is given.

1. INTRODUCTION

Several solutions for concurrent transaction control in a multiple-copy distributed data base system have appeared in the literature (1) - (4). Most of these solutions involve various degrees of global locking, which requires a large number of intersite messages and reduces system concurrency. Some require primary sites as control centers, which may create bottlenecks in the system. One solution, presented in (4), employed by the System for Distributed Databases, SDD-1, avoids global locking and primary sites, but requires stored timestamps on all data items of the data base (or at least on all recently updated data items). This paper proposes modifications to the concurrency control mechanism of SDD-1 which eliminate the need for timestamps on data items and ease some of the synchronization protocols between read and write actions. A proof of correctness is given.

2. SUMMARY OF SDD-1 CONCURRENCY CONTROL

A transaction T is the unit of consistency and is modeled as a series of read actions followed by write actions

T = R(T,ul)...R(T,un)W(T,vl)...W(T,vm)

where ul,...,un are distinct sites at which these actions are to be executed. Similarly for vl,...,vm. Associated with each read action, say R(T,ul), is a set of data to be retrieved, called the read set of transaction T at site ul, which is also denoted by R(T,ul). Similarly W(T,ul) denotes the write set of transaction T at site ul. Each transaction T belongs to a transaction class T* which has a pre-determined set of data to be read from and written into the database at each site. These are called the read sets and write sets of the transaction class (These read-sets and write-sets are physical sets. For the purpose of this paper the concept of logical sets is not needed). The read sets and write sets of each transaction are contained in the read sets and write sets of its class respectively. We denote the read set and write set of class T* at site u by R(T*,u) and W(T*,u) respectively. R(T*,u) and W(T*,u) also denote the class of read and write actions of transaction class T* sent to site u. A class conflict graph is used to show the intersections among read sets and write sets for all transaction classes in the system. For example, in Figure 1, a transaction class is represented by nodes, one for each read and write set connected by a central node. In the figure the read set of transaction class T* intersects the write set of class S* at site u, and write sets of T* and S* intersect at site u. We draw an edge for every such intersection. We call such an edge a heterogeneous edge. An edge is also drawn between the central point, and each read set and each write set of the same class; these edges are called homogeneous edges. A path is a sequence of read nodes and write nodes (S1, S2,...,Sn) where (Si,Si+1) is either a heterogeneous edge or an adjacent pair of homogeneous edges, and no edge appears twice. If S1=Sn then the path is called a cycle. A path, or a cycle, is called non-redundant if each class appears in at most two heterogeneous edges in the path. By analyzing such conflict graphs, protocols are devised that make the system run correctly in the sense of serializability and convergence of multiple-copy data (4). Serializability means that if the system ceases to take any new transactions and lets existing ones run to completion, the final database state (which includes all external output) is the same as if

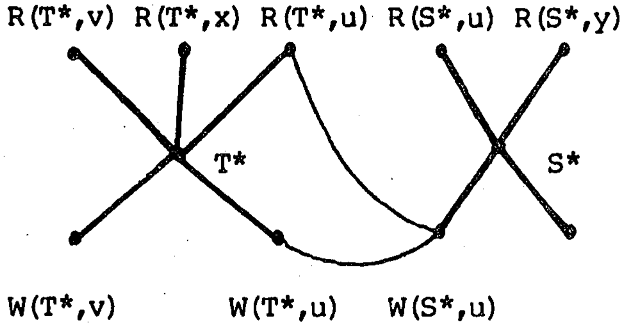


Figure 1

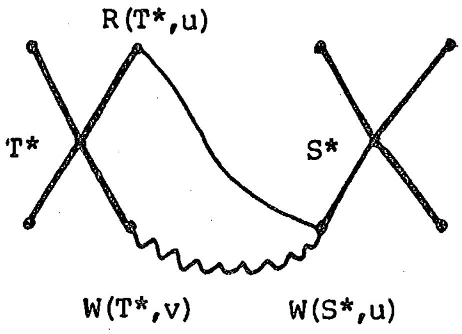


Figure 2

all transactions were run serially in some order. Convergence of multiple-copy data means that if the system ceases to take any new transactions and lets existing ones run to completion, all copies of the database will be the same.

The following assumptions are made in SDD-1:

1. There is a unique timestamp associated with each transaction. One way to ensure uniqueness is to take the originating site number as the low order digits and the local clock time as the high order digits of the timestamp.
2. Transactions are grouped into classes characterized by read sets, write sets and originating site.
3. Transactions from the same class are pipelined, i.e., actions from the same class designated for the same site are sent, received and processed in timestamp order. (This constraint can be relaxed so that if the read set and write set of a transaction class do not overlap at some site, then actions from the class designated for the site can be processed with the following three rules: (1) all read actions must be pipelined, (2) all write actions must be pipelined, (3) each read action must precede the write action of the same transaction).
4. There is a timestamp associated with each data item in the database. (This assumption will be eliminated later).

5. Write actions of a transaction are sent out only after all its read actions have been completed.

Protocols of SDD-1 as described in (4) are summarized in the following:

Definition: TM(S) denotes the timestamp of the transaction S. TM(W(S,u)) denotes the timestamp of the action W(S,u) which is initially equal to TM(S), but may be changed as discussed in later sections.

Protocol R3: Whenever R(T*,u) and W(S*,u) intersect, and there exists a non-redundant path in the conflict graph between W(S*,u) and some write set of class T* (see Figure 2), then for every pair of transactions T,S in classes T*,S* respectively, W(S,u) runs after R(T,u) (denoted by R(T,u)->W(S,u)), iff the timestamp TM(S) of S, is larger than TM(T).

Protocol R2: If read sets of T* intersect write sets of S* and Q* (S* and Q* are not necessarily distinct) at sites u and v respectively, and there is a non-redundant path between these write sets (Figure 3), then make sure that T* reads equally up-to-date data from S* and Q* at sites u and v. In other words

W(S,u)->R(T1,u) ≤ R(T2,v)->W(Q,v) implies TM(S)<TM(Q)

where T1 and T2 are from transaction class T*, T1 and T2 are not necessarily distinct, A->B means A runs before B, and A≤B means that the transaction containing action A has a timestamp which is less than or equal to the timestamp of the transaction containing action B.

Protocol R1: In R2, if $u=v$, $S^* \neq Q^*$, then make sure that T^* reads equally up-to-date data from S^* and Q^* at site u (Figure 4). (In fact this is a special case of R2). In other words

$$W(S,u) \rightarrow R(T1,u) \leq R(T2,u) \rightarrow W(Q,u) \text{ implies } TM(S) < TM(Q)$$

where T1 and T2 are from the class T^* and are not necessarily distinct.

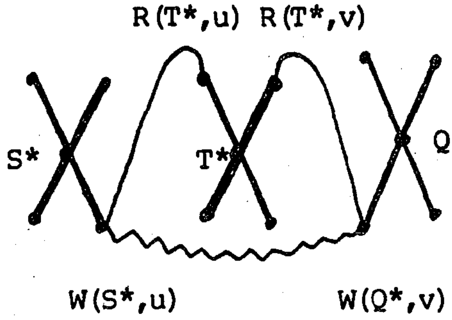


Figure 3

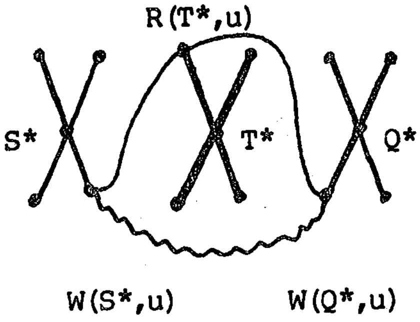


Figure 4

In implementing these protocols, SDD-1 uses a synchronization primitive for coordinating an action A (always a read action in SDD-1) with respect to a class of write actions W(S*,u) at a site u. This synchronization primitive, called SYNCH1 here, is defined as follows:

Definition: SYNCH1(A,W(S*,u),t) is a synchronization primitive which is applied at site u to the action A with respect to the write queue W(S*,u) and timestamp t. The primitive says that action A be executed if and only if actions from the queue W(S*,u) have been executed up to but not beyond timestamp t.

When synchronization primitive SYNCH1 is used, some read action may wait indefinitely because transactions in the class S* with which it synchronizes occur infrequently. In SDD-1, null-write messages are used to minimize this kind of delay.

In SDD-1, timestamps on data items are used to ensure convergence of multiple copies of data. Whenever a new data item is to overwrite an existing data item, the timestamp of the existing data item is retrieved and compared with the timestamp of the new data item. Update is carried out only if the timestamp of the new data item is larger.

For brevity, protocol p4 for handling unanticipated transactions is not described here. The results of this paper remain correct if p4 is included, however the proof of correctness becomes more tedious.

The execution symbol E is also eliminated from the transaction model, because it serves no useful purpose as far as this analysis is concerned. A more complete description may be found in (4). In the next section we add a protocol to the system which makes data item timestamps unnecessary.

3. THE WRITE PROTOCOL

The following protocol is added to the three protocols discussed in Section 2. Together they make the system run correctly without timestamps on data items. Of course a price must be paid for saving storage space. This protocol requires some synchronization of write actions at each site, which reduces concurrency among write actions. However, it does not directly increase any inter-site synchronization.

Protocol W: If write sets of T* and S* intersect at site u, i.e. $W(T^*,u) \wedge W(S^*,u) \neq 0$, and W(T*,u), W(S*,u) reside on a non-redundant cycle (including cycles involving only write actions) as shown in figure 5, then for all transactions T,S in classes T*,S* respectively, W(T,u) runs after W(S,u) if and only if TM(T)>TM(S).

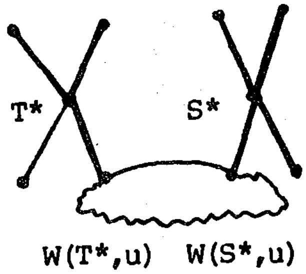


Figure 5

Implementation: Apply SYNCH1(W(T,u),W(S*,u),TM(T)) at site u for every transaction T from class T*, where TM(T) is timestamp of T. (Similar procedure must be applied to every transaction S from class S* against queue W(T*,u) at site u.)

4. PROOF OF CORRECTNESS

Correctness of a system here means the system is serializable and all copies of redundant data converge to the same values. A local log at site u is defined to be a linear sequence of all actions executed at site u which represents the actual order in which they are executed, if the system obeys the partial order constraints imposed by the assumptions and protocols discussed in the last two sections. A system log is any merge of all the local logs which preserves the the order among actions in the local logs. To prove that the system is serializable it must be shown that all the local logs are serializable and the serial orders are consistent among all the local logs. It is equivalent to prove that any system log composed of these local logs is serializable.

In the following, it will be shown that all system logs are serializable in the sense that if the following adjacent interchange rules are applied to any system log, the log can be transformed into a serial log. A serial log is a log of serial execution of those transactions in some order. The following table shows when two adjacent actions in a log cannot be switched.

R(Tl,u), R(T2,u)	Tl and T2 are from the same class (pipelining)
W(T,u), W(S,u)	The write sets of T* and S* intersect at site u. T* and S* may be the same transactions class.

R(T,u), W(S,u)
W(S,u), R(T,u)
	The read set of T* intersects the write set of of S* at site u, where S* and T* may be the same class.
	T and S are the same transaction.

The rules in the table above are less restrictive than in (4) in that E actions and augmented conflict rules are eliminated. But it is more restrictive in that any two write actions cannot, in general, be switched.

In serializing a system log, we would try to move two actions of the same transaction adjacent to each other by moving actions between them either to the left or to the right. But in doing so adjacent action symbols which belong to the same transaction must not be split up. The following lemma will be stated without proof. For a detailed proof of the lemma see [10].

Lemma T : Let L=...X(A,u)...Y(B,v)... be a subsequence (not necessarily contiguous) of a system log. Let us assume that at most two consecutive action symbols between X(A,u) and Y(B,v) can belong to the same transaction. Let us also assume that every action, (or action with one of its neighbors, if this neighbor also belongs to the same transaction), between X and Y is blocked by its (their) left and right neighbors (the blocking can be due to protocol conflict or pipelining rule). If there exists at least one action between X(A,u) and Y(B,v) in the system log L, and one of the following conditions is true, then TM(X(A,u))<TM(Y(B,v)).

(1) A*=B*

(2) A*≠B*, both X(A,u) and Y(B,v) are write actions, and there exists a non-redundant path between X(A,u) and Y(B,v) in the conflict graph other than the path shown in the log L.

Theorem 1: System logs which obey the protocols and conditions in Sections 2 and 3 are serializable.

Proof: Suppose, to the contrary, that there exists a system log L which cannot be serialized. Then there are two action symbols from the same transaction, separated by one or more action symbols from other transactions, which cannot be moved adjacent to each other, say X and Y in the following fragment of L,

$$L= \ldots X\ S1\ S2 \ldots Sn\ Y \ldots, \text{ where } X=S0, \text{ and } Y=Sn+1$$

where each one of S1,S2,...,Sn is a group of actions belonging to the same transaction, and cannot be moved to the left of X or to the right of Y. Then each Si, where $1 \leq i \leq n$, must be blocked from the left and right by some Sj and Sk, $j<i<k$. Therefore there exists a subsequence of the sequence (S1,S2,...,Sn) (not necessarily contiguous) which forms a blocking path from X to Y in the sense that each Si on this path is blocked by its left and right neighbors. Since for each Si on this path at most two action symbols of Si are needed to have Si blocked by its left and right neighbors, a blocking path from X to Y can be derived which is composed of one or two action symbols from each group Si. By lemma T (see Appendix) TM(X)<TM(Y). But since X and Y are of the same transaction, TM(X) = TM(Y), a contradiction. Therefore, the assumption that there exists a system log which is not serializable is false.

QED

5. FURTHER WEAKENING OF PROTOCOLS

Before we present the modified protocols we define a partition on the set of all read nodes and write nodes of a transaction class.

Let P(S*)={S1,S2,...,Sn} be a set consisting of all the read nodes and write nodes of the transaction class S*. We define an equivalence relation ~ on the set P(S*) as follows:

1. Si~Si for all Si in the set P(S*),
2. Si~Sj if there exists an external path between Si and Sj. By external path, we mean a path that does not include a homogeneous edge of the class S*.

The equivalence relation ~ defined above partitions the set P(S*) into disjoint blocks. We denote the block containing W(S*,u) by BLOCK(W(S*,u)).

Definition: A write node W(S*,u) satisfies condition (a), if BLOCK(W(S*,u)) consists only of the node W(S*,u).

Definition: A write node W(S*,u) satisfies condition (b), if the read node R(S*,u) exists, and BLOCK(W(S*,u)) does not include any read node R(S*,v) where v is not equal to u.

5.1 Protocol R3a and R3b

Under certain conditions, protocols R3, R2, R1, and W are more restrictive than necessary. The following protocols are relaxed versions of protocol R3. Protocol R3a applies when W(S*,u) in figure 2 satisfies condition (a); protocol R3b applies when W(S*,u) in figure 2 satisfies condition (b).

Protocol R3a: If W(S*,u) in figure 2 satisfies condition (a), then the two rules as defined below must be followed.

(1) For every transaction S from class S*, the timestamp TM(W(S,u)) can be changed. But for any pair of transactions S1,S2 from class S*, TM(S2)>TM(S1) if and only if TM(W(S2,u))>TM(W(S1,u)).

(2) For every pair of transactions T,S in classes T*,S* respectively, W(S,u) runs after R(T,u) if and only if TM(W(S,u))>TM(R(T,u)).

Implementation: Attach the read condition (S*,TM(T)) to R(T,u). When R(T,u) arrives at site u, apply synchronization primitive SYNCH2(R(T,u),W(S*,u),TM(T)) or SYNCH3(R(T,u),W(S*,u),TM(T)) as described in the following two definitions.

Definition : SYNCH2(R(T,u),W(S*,u),t) is a synchronization primitive which is applied at site u to the read action R(T,u) with respect to the queue W(S*,u) and the timestamp t.
If the last write action executed from queue W(S*,u) is W(S1,u) with timestamp TM(S1) when R(T,u) arrives at site u, and if t>TM(S1), then instead of waiting for actions from queue W(S*,u) to be processed up to but not beyond timestamp t as in SYNCH1, site u can proceed to execute R(T,u) immediately. But site u must also add (t-TM(S1)) to the timestamp of every action from class W(S*,u) not yet executed. Or site u can choose a time t' anywhere between TM(S1) and t (inclusive) and execute R(T,u) only after actions from class W(S*,u) have been executed up to timestamp t'. But site u must add (t-t') to the timestamp of every action from class W(S*,u) not yet executed. Of course if t<TM(S1), then R(T,u) will be rejected. In adding time to the timestamps of write actions, care should be taken to ensure that the new timestamps are unique, and that the new timestamps will not become larger and larger which may delay the execution of these write actions indefinitely. The only purpose of adding (t-TM(S1)) to the timestamp of every action from class W(S*,u) not yet executed is to ensure that these write actions have timestamps larger than t, and that their order is preserved. There are implementations other than simply adding (t-TM(S1)) to the timestamps, which achieve these two effects. Synchronization primitive SYNCH3, which follows, is one such implementation.

Definition: SYNCH3(R(T,u),W(S*,u),t) is a synchronization primitive which is applied at site u to the read action R(T,u) with respect to the queue W(S*,u) and the timestamp t.
At site u, associate with the queue W(S*,u) two timestamp variables LAST(W(S*,u)) and NEXT(W(S*,u)) which are initially set equal to 0. The variable LAST(W(S*,u)) stores the timestamp of the last write action executed from the queue W(S*,u). The variable NEXT(W(S*,u)) is used to compute the timestamp of the next write action to be executed from the queue W(S*,u). Let W(S,u) be the earliest pending write action, if one exists, from the queue. Define its (modified) timestamp TM(W(S,u)) as follows:

$$TM(W(S,u)) = Max(TM(S), NEXT(W(S^*,u))+1) ,$$

where one should be added to the local clock portion of the timestamp. Whenever a write action W(S,u) is executed, LAST(W(S*,u)) and NEXT(W(S*,u)) are set equal to TM(W(S,u)). When the read action R(T,u) arrives at site u the following occurs. If LAST(W(S*,u))<t then R(T,u) is executed immediately and NEXT(W(S*,u)) is set equal to Max(t,NEXT(W(S*,u))). Otherwise R(T,u) is rejected.

Alternatively, if LAST(W(S*,u))<t then site u chooses a timestamp T1 such that LAST(W(S*,u))≤T1<t. Site u then delays execution of R(T,u) until actions from class W(S*,u) have been executed up to but not beyond timestamp T1. On execution of R(T,u), NEXT(W(S*,u)) is set equal to Max(t,NEXT(W(S*,u))).

Protocol R3b: If W(S*,u) in figure 2 satisfies condition (b), then protocol R3a, augmented with the following rule, must be followed.

(1) For every transaction S from class S*, all the actions in BLOCK(W(S,u)) must have the same time stamp.

Implementation: Attach the read condition (S*,TM(T)) to R(T,u). When R(T,u) arrives at site u, apply the synchronization rule SYNCH4(R(T,u),R(S*,u),W(S*,u),TM(T)), defined below.

Definition : SYNCH4(R(T,u),R(S*,u),W(S*,u),t) is a synchronization primitive which applies to the read action R(T,u) with respect to the queues R(S*,u) and W(S*,u), and the timestamp t.
At site u, associate with the queue R(S*,u) of actions from class S*, two timestamp variables LAST(R(S*,u)) and NEXT(R(S*,u)), and associate with the queue W(S*,u) one timestamp variable LAST(W(S*,u)) all of which are initially set equal to 0. Let R(S,u) be the earliest pending read action, if one exists, from the R(S*,u) queue. Let us define the timestamp NEW(R(S,u)) as follows:

$$NEW(R(S,u)) = Max(TM(S), NEXT(R(S^*,u))+1) .$$

If R(S*,u) is a member of BLOCK(W(S*,u)) then TM(R(S,u)) must be changed to NEW(R(S,u)). Otherwise TM(R(S,u)) stays unchanged.

Whenever the read action R(S,u) is executed, LAST(R(S*,u)) and NEXT(R(S*,u)) are set equal to NEW(R(S,u)), no matter whether TM(R(S,u)) is changed or not, and site u must send a read-completion message together with the timestamp NEW(R(S,u)) to the originating site of action R(S,u). The originating site must use this timestamp to timestamp all the write actions in BLOCK(W(S,u)). When a write action W(S,u) is executed at site u, LAST(W(S*,u)) is set equal to TM(W(S,u)). Notice that at all times LAST(W(S*,u)) ≤ LAST(R(S*,u)).

When a read action R(T,u) with read condition (S*,t) arrives at site u the following occurs.
(1) If t<LAST(W(S*,u)) then R(T,u) would be rejected.
(2) If t<LAST(R(S*,u)) and t>LAST(W(S*,u)) then R(T,u) must wait until actions from the W(S*,u) queue have been executed up to but not beyond timestamp t.
(3a) if t>LAST(R(S*,u)) and t>LAST(W(S*,u)), then site u must wait until actions from queue W(S*,u) have been executed up to but not beyond timestamp LAST(R(S*,u)) before it executes R(T,u). After execution of R(T,u), NEXT(R(S*,u)) is set equal to MAX(t,NEXT(R(S*,u))). Or,
(3b) Site u can choose a timestamp t', where LAST(R(S*,u))≤t'≤t, and wait until actions from both queues R(S*,u) and W(S*,u) have been executed up to but not beyond timestamp t' before executing R(T,u). After execution of R(T,u), NEXT(R(S*,u)) is set equal to MAX(t,NEXT(R(S*,u))).

5.2 Protocol R2a,R2ab and R2b

Similarly, under certain conditions protocol R2 can be relaxed. The following protocols are the relaxed variations of protocol R2.

Protocol R2a: If W(S*,u) and W(Q*,v) in figure 3 both satisfy condition (a) then the two rules as defined below must be followed.

(1) For every transaction S from class S*, the timestamp TM(W(S,u)) can be modified. But for every pair of transactions S1,S2 from class S* TM(S2)>TM(S1) if and only if TM(W(S2,u))>TM(W(S1,u)). (Similarly for class Q*.)

(2) For every transaction S from class S*, Q from class Q*, T1 and T2 from class T*, if W(S,u) runs before R(T1,u), TM(R(T1,u))≤TM(R(T2,v)), and R(T2,v) runs before W(Q,v) then TM(W(S,u))<TM(W(Q,v)). In other words
W(S,u)->R(T1,u)≤R(T2,v)->W(Q,v)
implies TM(W(S,u))<TM(W(Q,v)).

Implementation: Attach read condition (S*,t) and (Q*,t) to R(T,u) and R(T,v) respectively, where t can be arbitrarily chosen by the originating site. Then apply SYNCH3(R(T,u),W(S*,u),t) and SYNCH3(R(T,v),W(Q*,v),t).

Protocol R2ab: If W(S*,u) and W(Q*,v) of figure 3 satisfy condition (a) and (b) respectively, then protocol R2a, augmented with the following rule, must be followed.

(1) For every transaction Q from class Q*, all the actions in BLOCK(W(Q,v)) must have the same timestamp.

Implementation: Attach read condition (S*,t) and (Q*,t) to R(T,u) and R(T,v) respectively, where t can be arbitrarily chosen by the originating site. Then apply SYNCH3(R(T,u),W(S*,u),t) and SYNCH4(R(T,v),R(Q*,v),W(Q*,v),t).

Protocol R2b: If both W(S*,u) and W(Q*,v) of figure 3 satisfy condition (b), then protocol R2ab, augmented with the following rule, must be followed.

(1) For every transaction S from class S*, all the actions in BLOCK(W(S,u)) must have the same timestamp.

Implementation: Attach read condition (S*,t) and (Q*,t) to R(T,u) and R(T,v) respectively, where t can be arbitrarily chosen by the originating site. Then apply SYNCH4(R(T,u),R(S*,u),W(S*,u),t) and SYNCH4(R(T,v),R(Q*,v),W(Q*,v),t).

5.3 Protocol R1a,R1ab, and R1b

Similarly, protocol R1 can be relaxed under conditions as discussed in the prevous section.

Protocol R1a: If W(S*,u) and W(Q*,u) in figure 4 satisfy condition (a), then the two rules as defined below must be followed.

(1) For every transaction S from class S*, the timestamp TM(W(S,u)) can be modified. But for every pair of transactions S1,S2 from class S* TM(S2)>TM(S1) if and only if TM(W(S2,u))>TM(W(S1,u)). (Similarly for Class Q*.)

(2) For every transaction S from class S*, Q from class Q*, T1 and T2 from class T*,
W(S,u)->R(T1,u)≤R(T2,u)->W(Q,u)
implies TM(W(S,u))<TM(W(Q,u)).

Implementation: Attach read conditions (S*,t) and (Q*,t) to R(T,u), where t will be chosen at site u. Let the timestamp variables associated with W(S*,u) and W(Q*,u) as mentioned in synchronization rule SYNCH3 be LAST(W(S*,u)) and LAST(W(Q*,u)) respectively. Site u then chooses a timestamp for t such that t≥Max(LAST(W(S*,u)),LAST(W(Q*,u))), and applies SYNCH3(R(T,u),W(S*,u),t) and SYNCH3(R(T,u),W(Q*,u),t).

Protocol R1ab: If W(S*,u) and W(Q*,u) satisfy condition (a) and (b) respectively, then protocol R1a, augmented with the following rule, must be followed.

(1) For every transaction Q from class Q*, all the actions in BLOCK(W(Q,v)) must have the same timestamp.

Implementation: Site u chooses a timestamp for t as discussed in the prevous implementation. It then applies SYNCH3(R(T,u),W(S*,u),t), and SYNCH4(R(T,u),R(Q*,u),W(Q*,u),t).

Protocol Rlb: If W(S*,u) and W(Q*,u) both satisfy condition (b), then protocol Rlab, augmented with the following rule, must be followed.

(1) For every transaction S from class S*, all the actions in BLOCK(W(S,u)) must have the same timestamp.

Implementation: Choose a timestamp for t as discussed in protocol Rla. Then apply SYNCH4(R(T,u),R(S*,u),W(S*,u),t) and SYNCH4(R(T,u),R(Q*,u),W(Q*,u),t).

5.4 Protocol Wa and Wb

Protocol W can be relaxed under certain conditions too.

protocol Wa: Assuming that W(T*,u) and W(S*,u) intersect and both reside on a non-redundant cycle as shown in figure 5, and that W(S*,u) satisfies condition (a), then the two rules as defined below must be followed.

(1) For every transaction S from class S*, the timestamp TM(W(S,u)) can be changed. But for any pair of transactions S1,S2 from class S*, TM(S2)>TM(S1) if and only if TM(W(S2,u))>TM(W(S1,u)).

(2) For every transction T from class T* and S from S*, W(T,u) runs after W(S,u) if and only if TM(W(T,u))>TM(W(S,u)).

Implementation: Apply SYNCH3(W(T,u),W(S*,u),TM(T)) to every transaction T from class T*.

Protocol Wb: If W(T*,u) and W(S*,u) intersect and both reside on a non-redundant cycle, and W(S*,u) satisfies condition (b), then protocol Wa, augmented with the following rule, must be followed.

(1) For every transaction S from class S*, all the actions in BLOCK(W(S,u)) must have the same timestamp.

Implementation: Apply SYNCH4(R(T,u),R(S*,u),W(S*,u),TM(T)) to every transaction T from class T*.

5.5 Summary of Protocols

The following table summarizes all the protocols discussed in this paper. An "x" in the table means that the protocol uses the corresponding synchronization primitive.

	R3	R3a	R3b	R2	R2a	R2ab	R2b	R1	R1a	R1ab	R1b	W	Wa	Wb
SYNCH1	x			x				x				x		
SYNCH3		x			x	x			x	x			x	
SYNCH4			x			x	x			x	x			x

6. PROOF OF CORRECTNESS

In the implementations of the new protocols described in Section 5, timestamps of some read actions and write actions can be changed. Therefore the assertion that any two actions from the same transaction have the same timestamp as asserted in the proof of theorem 1 and lemma T in section 4 is no longer true. But this assertion is needed in the proof of theorem 1 and lemma T only when two actions from the same transaction are involved in a cycle. But for any two actions from the same transaction involved in a cycle, either both of their timestamps have been changed to the same value (if condition(b) is true), or both of their timestamps have not been changed at all (if condition(b) is not true). Therefore, the proof of theorem 1 and lemma T is still correct for these new protocols.

7. CONCLUSION

A new protocol is introduced to eliminate the need for timestamps on data items. This protocol reduces the concurrency of write actions in each site. The degree of loss of concurrency depends on the conflict graph structure of each application. For some applications, for example, those in which changes to the data base do not occur frequently or do not have to be processed immediately, the saving of storage space may outweigh the loss of concurrency.

Existing read-write protocols are weakened under certain conditions to allow some read and write actions to wait less. These weaker protocols not only allow more flexible scheduling of some read and write actions, but also reduce or even eliminate the requirement for null-write messages, and yet improve system performance at the same time. Preliminary results from a simulation study have confirmed this. A proof of correctness is also given.

ACKNOWLEDGEMENT

The author would like to thank Dr. Murray Edelberg for many fruitful discussions.

REFERENCES

1. R. H. Thomas, "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control", BBN Report No. 3340, Bolt Beranek and Newman, Inc., July, 1975.
2. P. A. Alsberg, "Research in Network Data Management and Resource Sharing: Final Research Report", CAC Document no. 210, Center for Advanced Computation, Univ. of Illinois at Urbana-Champaign. Sept. 30, 1976.
3. D. J. Rosenkrantz, et al., "A System Level Concurrency Control for Distributed Database Systems", 1977 Berkeley Workshop on Distributed Data Management and Computer Network, Berkeley, May 1977.
4. P. A. Bernstein, et al., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case)". Technical Report, CCA-77-09, Computer Corporation of America, Cambridge, MA., Dec. 1977.
5. J. Rothnie, et al., "Analysis of Serializability In SDD-1: A System for Distributed Databases", COMPSAC 77, Nov. 1977.
6. J. Rothnie, et al., "The Redundant Update Methodology of SDD-1: A System for Distributed Databases", COMPSAC 77., Nov. 1977.
7. K. P. Eswaran, et al., "The Notions of Consistency and Predicate Locks in a Database System", CACM, Nov. 1976.
8. C. H. Lee, et al., "Distributed Control Schemas for Multiple-Copied File Access in a Network Environment", COMPSAC 77, p. 722, Nov., 1977.
9. C. A. Ellis, "A Robust Algorithm For Updating Duplicate Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Network, May 1977.
10. Wen-Te K. Lin, "Concurrency Control in a Multiple Copy Distributed Database System", Research Paper RP#79-20, Sperry Research Center, June 1979

A New Concurrency Control Algorithm for Distributed Database Systems

Toshimi Minoura

Computer Systems Laboratory
Stanford University
Stanford, California

ABSTRACT

A new concurrency control algorithm for distributed database systems that spatially extends the idea of "exclusive/share locks" is presented. The new algorithm, extended true-copy token algorithm, combines a locking mechanism and a "true-copy token" mechanism. "True-copy tokens" handle partitioned data that cannot be handled efficiently by locks alone.

1. INTRODUCTION

A distributed database system is one of the hottest issues among many theorists and practitioners. The system must provide an integrated interface to its users by hiding partition and duplication of some data. Furthermore, although transactions are processed concurrently, their effects on the system and the users must be as if they were processed in sequence. Without reasonable concurrency, most distributed database systems will be impractical.

The concurrency control problem in a distributed database system has been studied by many researchers [BADA-78, BERN-78, ELLI-77, GARC-78, GELE-78, GRAP-76, LELA-78, ROSE-78, STON-79, THOM-78]. However, a satisfactory solution is yet to come. In this paper we present still another algorithm that we hope gives some new insights.

In section 2 we briefly introduce a formal model of a distributed database system. Following [LAMP-78], an execution history of transactions is defined as a partial ordering on action events, so we do not assume the existence of the totally ordered global time. Also two operational consistency conditions used so far in the literature are discussed.

In section 3 an "extended true-copy token" algorithm is presented. A "true-copy token" is used to designate a "true-copy" that provides the current "logical component" value when a logical component is represented by multiple "physical components". The algorithm efficiently supports "multiple migrating localities". A new concept "effective global time" is introduced in section 4, and its usefulness is shown in the correctness proof of the extended true-copy token algorithm.

In section 5 we briefly discuss the new algorithm and some related algorithms. Although no discussion about the concurrency control problem for a distributed database system is complete without discussing the resiliency problem, this is not addressed in this paper.

2. DISTRIBUTED DATABASE SYSTEM MODEL

We assume that a distributed database system DDBS = {X, Y, Z, ...} consists of a set of logical components, each of which can be assigned a value independently. A logical component X = {x1, x2, x3, ...} is represented by a set of one or more duplicated physical components that are supposed to assume the same value except for transitional periods during update operations.

A site H of a distributed database system is a subset of the set of all physical components in the system, i.e., $H \leq (X + Y + Z + \ldots)$. Every two sites must be disjoint, and the union of all sites is the set of all physical components.

Here the terms "logical" and "physical" are used to indicate only a relative degree of abstraction. "Physical" does not mean direct implementation by hardware; a "physical component" may be a "logical component" at another level of abstraction.

A transaction T = {A, B , ...} is a set of actions. An action is a group of operations that we find convenient to treat as a single group. Operations in the action can be interpreted in two ways: "logical" or "physical." A logical operation is considered to access logical components, and a physical operation is considered to access physical components. More specifically, a logical operation read(X) is equivalent to a physical operation read(xi), any i, and a logical operation write(Y) is equivalent to a set of physical operations { write(y1), write(y2), ... }. In the sequel, we assume that operations are "physical" unless we say otherwise. A write operation to a physical component at a remote site is informally called an update.

An action is executed on a single site. Actions belonging to the same transaction, however, may be executed on different sites; a transaction may migrate around different sites. A transaction can even spawn multiple actions that operate concurrently. Thus concurrent processing of actions belonging to the same transaction as well as to different transactions may occur. What constitutes a single action may vary according to the system designer's discretion as long as the previous constraints are observed. For example, update operations of the same content to duplicated physical components at different sites must belong to different actions, but different update operations to different components at the same site can be grouped into a single action when these operations belong to the same transaction.

Unusual notations used in this paper: "+" for set union; "<" for set inclusion; and "/" for negation.

The execution of an action A is characterized by the occurrences of its initiation event "a" and termination event "$\underline{a}$", which we will call *action events*. We define the partial ordering on the set of action events following [LAMP-78]:

Definition. An *execution history* "<<" of a set of transactions is an irreflexive partial ordering on the set of action events caused by the execution of these transactions. For two events a and b, a<<b iff

1. Events a and b have taken place at the same site, and event a preceded event b;
2. Event a is the sending of a message and event b is the receipt of the same message at another site; and
3. The pair (a,b) is in the transitive closure of the ordering obtained by the above two rules, i.e., a<<c and c<<b for some action event c.

A write operation, especially an update to a duplicated component, may be *redundant* because the value written by it is overwritten by another write operation without being read by any action. These redundant write operations can be omitted. By properly ignoring redundant write operations, the inter-site traffic can be reduced, thus efficiency of the system operation can be enhanced.

A *consistent execution history* of transactions is one in which the system and the users see the database state as if the transactions were processed sequentially. A concurrency control algorithm is consistent iff any execution history realized by the algorithm is consistent. [PAPA-77] has given the minimum condition for consistent transaction processing, which we call consistency condition C1, and has shown that the consistency test of an execution history is NP-complete. Other authors [ESWA-76, MINO-78, SCHL-78, STEA-76] have used a stricter condition that allows a polynomial-time consistency test of an execution history. The latter condition, which we call consistency condition C2, is sufficient but not necessary under the same premise with [PAPA-77, BERN-78].

In this paper we use informal arguments, but more formal treatment can be found in [MINO-79]. We can prove the following statements about consistency conditions C1 and C2:

1. For any execution history of transactions, if C2 is true, then C1 is true; and
2. C1 is equivalent to C2 if the range of A is a subset of the domain of A for all actions A.

The fact that consistency condition C2 is necessary and sufficient when a "read-set" is a subset of a "write-set" for any action, has been observed in [ESWA-76, STEA-76].

3. EXTENDED TRUE-COPY TOKEN ALGORITHM

In this section we present a concurrency control algorithm that spatially extends the notion of exclusive/share locks. "Primary sites" [STON-79, GRAP-76] and a "circulating token" have been used as consistency control mechanisms for duplicated data. In [LELA-78], a "circulating token" is used for issuing "tickets"; a similar technique can be used to designate a "true copy", a version of data contained in a physical component whose value is current. An "extended true-copy token" mechanism is a generalization of these ideas; it uses "exclusive-copy tokens" and "share-copy tokens" that designate "migrating primary sites". Locking is performed over these migrating primary sites.

Two types of copies, namely "share" and "exclusive" copies, are important in the following discussion. A _true-copy indicator_ Ix as well as lock Lx is associated with each physical component x, and Ix can assume one of the three states, namely, "void", "share-copy" and "exclusive-copy". Although an update operation to a duplicated physical component is formally an action that belongs to some transaction, we may, in some sense, consider that it is carried out by the system. A transaction needs to lock only one copy of the duplicated physical components that it "directly" accesses. A physical component whose true-copy indicator state is either "share-copy" or "exclusive-copy" is informally called a _true copy_, i.e., either an _exclusive copy_ or a _share copy_. A "void" copy is in a transient state and whose content cannot be trusted. To visualize the transfer of access permission rights by the mechanism described below, we assume that a true copy possesses a _true-copy token_, i.e., either an _exclusive-copy token_ or a _share-copy token_. Also two types of locks, namely "share" and "exclusive" locks, are assumed in the following discussion.

We do not explicitly state the algorithm that implements these mechanisms. However, it can be easily constructed observing the following rules. We call such an algorithm an _extended true-copy token algorithm_ A1. Note that the following rules do not assume the existence of global time.

Rules.

1. At the point of system creation there exists one exclusive copy for each logical component. An exclusive-copy token can be transferred to another physical component. When the token transfer occurs, all updates made so far to the new physical component should precede or accompany the token transfer. Updates to the new physical component must follow the logical execution order.

2. An exclusive copy can become a share copy. A share copy can spawn multiple copies of itself.

3. When an exclusive copy is required, all share copies must shrink into a single share copy that may become an exclusive copy.

4. A transaction can set an exclusive lock of a physical component x only when the state of Ix is "exclusive-copy". An exclusive copy cannot be revoked until the exclusive lock is released.

5. A transaction can set a share lock only when the state of Ix is "share-copy". A share copy cannot be revoked until all share locks on it are released.

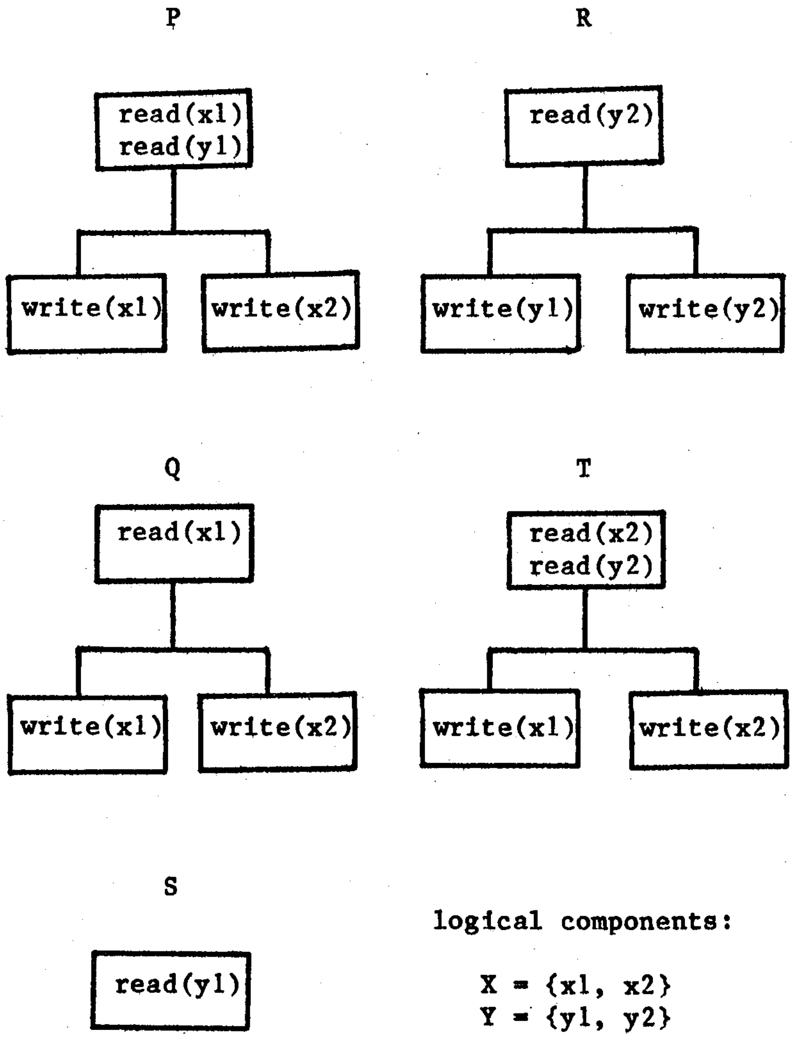
6. Locking on true copies by a transaction must be "two-phase".

In a sense, true-copy tokens are used to realize logical components, and locking is done over these logical components; true-copy tokens handle duplicated copies that cannot be efficiently handled by a locking mechanism alone. Although two-phase locking (refer to [ESWA-76] for two-phase locking) is used, it is not a complete locking; not all accesses to physical components are done with the physical components being locked. Update operations to the physical components at remote sites are performed without locking; true-copy tokens are used to properly sequence these update operations.

Update operations can be performed in the right order either by carrying the latest value of a logical component with the true-copy token or by letting an exclusive copy token issue sequence numbers that are unique relative to the logical component and performing updates according to these sequence numbers. Note that updates originate only from an exclusive copy, and that they can be uniquely ordered by these methods. Redundant updates can be discarded to reduce the inter-site traffic as we mentioned in section 2; if two write operations occur to the same physical component at a remote site while an exclusive-copy token is held at some site, the preceding write operation is redundant.

Fig. 2 diagramatically shows which combination of transactions shown in Fig. 1 can be processed concurrently. "Active" means that by using local data a transaction can be executed except for remote updates.

In Fig. 2(a), transaction P can proceed because x1 is an exclusive copy, and y1 is a share copy. Note that P makes read/write accesses to logical component X and a read-only access to logical component Y. The update to x2 by P can be discarded because it is overwritten by the update by transaction Q; it is redundant.



logical components:

X = {x1, x2}
Y = {y1, y2}

sites:

H = {x1, y1}
I = {x2, y2}

Fig. 1. Transactions.

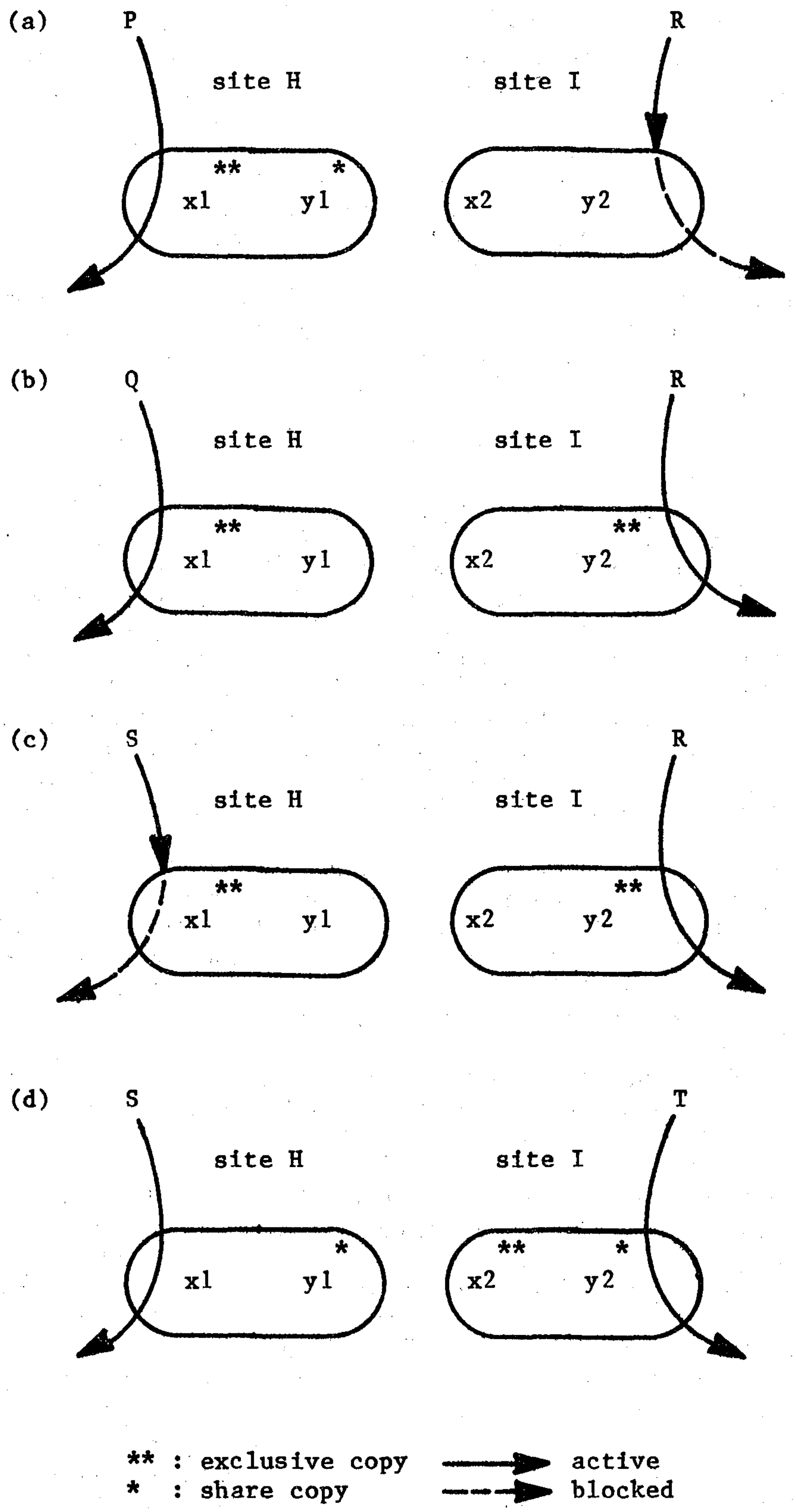


Fig. 2. Extended true-copy token algorithm.

Also in Fig. 2(a), transaction R tries to make read/write accesses to logical component Y; however, physical component y2 is a share copy and not an exclusive copy, so R cannot exclusively lock y2 and is blocked.

Once P is completed at site H and transaction Q starts its execution using only an exclusive copy x1, a share copy token of y1 can be released and y2 can become an exclusive copy; R can proceed. In Fig. 2(b), both Q and R are running concurrently. The update to x2 made by Q must be sent to site I before x2 becomes an exclusive copy and is accessed by T; in general, only the last update made to an exclusive copy needs to be sent to the other sites.

In Fig. 2(d) two share copies, y1 and y2, exist in the system at the same time, and both transactions S and T are active.

A transaction is two-phase locked iff no lock requests are released before all lock requests become active. An immediate consequence of the two-phase locking is that all lock requests of a transaction are active at some point during the execution of the transaction. We define a binary relation "<<p" on a set of transactions that use two-phase locking. R <<p S iff Rp << Sp, where Rp and Sp are the times when all lock requests are active in transactions R and S, respectively. Note that Rp and Sp are not action events; we have extended the definition "<<" to cover them. Note that both "<<" and "<<p" are acyclic.

Now we shall prove the correctness of the extended true-copy token algorithm by showing that any execution history realizable by the algorithm satisfy consistency condition C2; see [ESWA-76, MINO-78, MINO-79, SCHL-78] for consistency condition C2.

<u>Theorem 3.1</u>. Concurrency control algorithm A1 is consistent.

<u>Proof</u>. We show that any execution history realizable by the concurrency control algorithm A1 satisfies consistency condition C2.

Assume that action A of transaction R and action B of transaction S conflict over physical component xk. First, if xk is locked by both transactions R and S, accesses to xk by actions A and B are made in the same order with "<<p" on R and S. Second, if xk is accessed by action A without locking but by B with locking, A's access must be an update operation. When an update by A is completed before xk becomes lockable and is accessed by B, i.e., A precedes B, R must have made a write access to some exclusive copy xi (xi /= xk) performing the same write operation with the update by A. Then S can exclusively or share lock xk only after the exclusive lock on xi is released by R, hence Rp<<Sp. When an update by A occurs after action B, R can make an access to some exclusive copy xi (xi /= xk) only after the true-copy token is released by xk; Sp<<Rp. In both cases, A and B are executed in the same order as R and S are ordered by "<<p".

Finally, if xk is accessed by both actions A and B without locking, these accesses are update operations, and the ordering mechanism of update operations guarantees that updates are done in the same order with "<<p" on R and S.

We have shown that conflicting operations are performed in the same order with "<<p" on the transactions to which they belong; consistency condition C2 is satisfied because "<<p" is acyclic.

Q.E.D.

4. EFFECTIVE GLOBAL TIME

In the previous section we have directly proved that algorithm A1 maintains consistency condition C2. From a system structuring standpoint; however, it is more desirable to presuppose that a logical component itself can assume a value. We define the value of a logical component as follows:

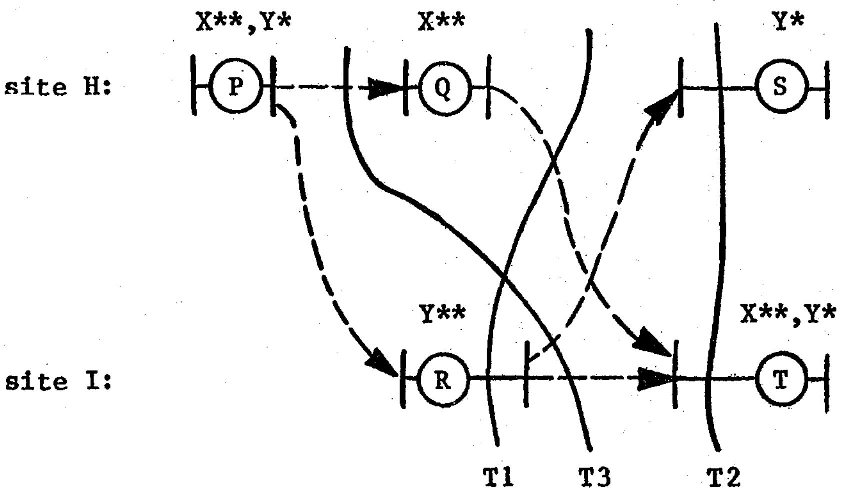
Definition. The value of a logical component is specified by the value of the physical component xi that is either an "exclusive copy" or a "share copy".

If we can globally pinpoint time, we can assert that the value of each logical component is uniquely defined; there is at most one exclusive copy, and when there are multiple share copies, their contents are the same. However, our formalism does not allow the use of global time.

Fortunately, we can define "effective global time" that is totally ordered as far as a realizable execution history of transactions is concerned.

Definition (effective global time). An effective global time for a given execution history is defined as a "slice" of an execution history of transactions. A slice of the execution history is a subset E1 of the set E of action events, such that for all action event a in E1 and action event b in E - E1, b << a does not hold.

The above definition is not intuitive, so we informally represent an effective global time as a dividing line in the graph of an execution history as shown in Fig. 3, which gives a possible execution history realizable by the algorithm A1 for the transactions shown in Fig. 1. For example, the effective global time T1 is the set of events to the left of the line labeled T1.



** : exclusive copy
* : share copy

Fig. 3. Effective global times.

<u>Definition</u> <u>(":<<")</u>. We define the ordering ":<<" on the set of effective global times as follows: For two effective global times T1 and T2, T1 :<< T2 iff T1 < T2, i.e., the effective global time ordering is equivalent to the set inclusion relation.

In Fig. 3, we have T1 :<< T2. Some effective global times are incomparable as T1 and T3 in Fig. 3. Fortunately, however, we have the following lemma for a realizable execution history.

<u>Lemma</u> <u>4.1</u>. A realizable set of effective global times is totally ordered.

<u>Proof</u>. In a realizable execution history, an action event that once took place cannot be revoked; the effective global time monotonically increases.

Q.E.D.

For example, in Fig. 3 once the effective global time T1 is reached there is no way to reach the effective global time T3; the effective global time T2 can be reached after T1.

Now we shall prove the correctness of the extended true-copy token algorithm by showing that in effect logical components are accessed under two-phase locking.

<u>Lemma</u> <u>4.2</u>. In algorithm A1 we can assume that logical components are accessed, i.e., accesses are made only to the physical components whose contents are equal to the values of their respective logical components.

Proof. Accesses are made only to an exclusive copy or a share copy that defines the value of the logical component.

Q.E.D.

Lemma 4.3. A logical component value is uniquely defined at any effective global time when it is accessed.

Proof. We prove that at any given effective global time there exists at most either one exclusive copy or multiple share copies of the same content for each logical component. First, assume that we have one exclusive copy xi. Another exclusive or share copy xj can exists only after xi ceases to be an exclusive copy or before xi becomes an exclusive copy. In the first case xj can be a true copy only after a true-copy token is transferred from xi to xj, and in the second case xi can be an exclusive copy only after a true-copy token is transferred from xj to xi. Therefore in either case time precedence can be established, and xi and xj cannot coexist at the same effective global time; if we have one exclusive copy, we cannot have another exclusive copy or share copy.

Second, when a share copy creates another share copy their contents are the same. Furthermore, the values of these share copies will not change until all share copies are revoked and a single exclusive copy is created. Therefore multiple share copies for any logical component contain the same value at any effective global time.

Consequently a logical component value is uniquely defined at any given effective global time.

Q.E.D.

At global time T2 in Fig. 3, for example, y1 accessed by transaction S and y2 accessed by transaction T have the same content. Also notice that at any realizable global time at most one exclusive copy exists for each logical component.

Lemma 4.4. In algorithm A1, two-phase locking is realized over logical components; more precisely, at any given effective global time write-write and read-write mutual exclusions are realized over logical components.

Proof. Assume that some logical component has been exclusively locked by some transaction; some physical component belonging to the logical component must possess the exclusive-copy token, and it must have been locked by that transaction. Then there cannot be any other true copies, and the only true copy is exclusively locked; therefore, other transactions cannot access the logical component, i.e., write-write and read-write mutual exclusions are realized over the logical component.

Q.E.D.

The correct operation of algorithm A1 can be concluded from Lemmas 4.1 - 4.4. Although two-phase locking was specified as part of the rules for the algorithm A1, it is not mandatory;

other types of consistent locking may be used.

In the extended true-copy token algorithm, transactions are blocked in two ways: trying to lock a physical component that has a true-copy token and waiting for a physical component, which the transaction wants to access, to get a true-copy token.

Theorem 4.5. If a locking mechanism over logical components does not cause deadlocks, an extended true-copy token algorithm can be designed so that it may not introduce deadlocks.

Proof. Although we do not describe the details, we can show that the "token transfer" mechanism for realizing logical components can be designed so that it can not introduce deadlocks. The fact that the locking mechanism over logical components does not cause deadlocks means that there are no deadlocks as long as we can establish logical components (true copies). In establishing logical components, special care must be taken so that the different physical components belonging to the same logical component do not block each other by each getting a subset of the share-copy tokens; this problem can be resolved by assigning a priority to the physical components.

Q.E.D.

5. DISCUSSIONS

The merits of the extended true-copy token algorithm can be summarized as follows:

1. It is intuitive and has a simple structure.

2. Multiple copies of a file are supported while it is used for read-only purposes; also it can be updated by revoking multiple copies.

3. An exclusive copy may migrate among different sites.

Multiple share copies are useful for a file that is mostly used in a read-only mode at many sites but needs to be updated occasionally at some site, e.g., a directory, a timetable, etc.

A migrating exclusive copy of a file is useful when more than one site actively use it, e.g., an airline seat reservation table for a flight from San Francisco to Tokyo; the inter-site traffic may be reduced by swapping the file at some interval.

One way to measure the capability of a concurrency control algorithm is to see how various "localities" are supported. We do not give a precise definition of the "localities", but it roughly means a set of physical components that must be directly accessed to execute a transaction. A smaller locality is preferable to larger one. "Primary site" mechanism supports "multiple static localities" of transaction processing, but fails to support "migrating localities". "Circulating token" mechanism supports a "single migrating locality", but fails to support "multiple localities". Most of other mechanisms currently proposed do not support small localities well. The extended true-copy token algorithm is intended to support "multiple migrating localities".

The level of concurrency realized by the algorithms in [BERN-78, LAMP-78] that use timestamps can be shown to be equivalent to having only one exclusive-copy token in the system. The similarity between a timestamp algorithm and a "single token" algorithm can be understood if we assume that the site whose local clock is the slowest has the token; a token transfer is made in a disguised form by sending a message whose timestamp is ahead of a local clock of some other site. This is further discussed in [MINO-79].

6. SUMMARY

A distributed database system with possible partitioned and duplicated data has been formalized with the consequence that the same operational consistency conditions for a centralized system were applicable for a distributed database system.

A new algorithm, extended true-copy token algorithm, was presented. The new algorithm supports either multiple read-only copies or a single read/write copy for each logical component without violating the consistency condition. In its correctness proof we introduced the new concept of effective global time.

We hope that the ideas developed in this paper will help in the design and analysis of better algorithms. Making a concurrency control algorithm resilient is essential in a practical environment. We hope to report on the resilient extended true-copy token algorithm later.

ACKNOWLEDGEMENTS

The author is much indebted to Hector Garcia-Molina for carefully reading the manuscript and making many valuable corrections and suggestions. The author wishes to thank Sandy Briggs for painstakingly correcting his English. Clarence Ellis and Gio Wiederhold also contributed many important suggestions. Keith Marzullo and Tim Gonsalvez also helped the author much with his English.

This work was partially supported by the Air Force Office of Scientific Research under contract No. F49620-77-C-0045.

REFERENCES

[BADA-78] Badal, D. and Popek, G. A proposal for distributed concurrency control for partially redundant distributed data base systems. Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978, pp. 273-285.

[BERN-78] Bernstein, P., Rothnie, J., Goodman, N. and Papadimitriou, C. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). IEEE Tr. on Software Engineering SE-4, 3 (May 1978), 154-168.

[BERN-79] Bernstein, P., Shipman, D., and Wong, W. Formal aspects of serializability in database concurrency control. IEEE Tr. on Software Engineering SE-5, 3 (May 1979), 203-216

[ELLI-77] Ellis, C. A robust algorithm for updating duplicate databases. Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, pp. 146-158.

[ESWA-76] Eswaran, K., Gray, J., Lorie, R. and Traiger, I. The notions of consistency and predicate locks in a database system. CACM 19,11 (Nov. 1976), 624-633.

[GARC-78] Garcia-Molina, H. Performance comparison of update algorithms for distributed databases. Computer Science Department, Stanford University, Oct. 1978.

[GELE-78] Gelenbe, E. and Sevcik, K. Analysis of update synchronization for multiple copy data-bases. Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978, pp. 69-90.

[GRAP-76] Grapa, E. Characterization of a distributed data base system. UIUCDCS-R-76-831, Dept. of CS, U. of Ill., Oct. 1976.

[LAMP-78] Lamport, L. Time, clocks and the ordering of events in a distributed system. CACM 21, 7 (July 1978), 558-565.

[LELA-78] Le Lann, G. Algorithms for distributed data-sharing systems which use tickets. Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978, pp. 259-272.

[MINO-78] Minoura, T. Maximally concurrent transaction processing. Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978, pp. 206-214.

[MINO-79] Minoura, T. Analysis of concurrency control mechanisms for distributed database systems. Unpublished. 1979.

[PAPA-77] Papadimitriou, C., Bernstein, P. and Rothnie, J. Some computational problems related to database concurrency control. Proc. Conf. Theoretical Comp. Sci., Waterloo, Canada, Aug. 1977, pp. 275-282.

[ROSE-78] Rosenkrantz, D., Stearns, R. and Lewis, P. System level concurrency control for distributed database systems. ACM Tr. on Database Systems 3, 2 (June 1978), 178-198.

[SCHL-78] Schlageter, G. Process synchronization in database systems. ACM Tr. on Database Systems 3, 3 (Sept. 1978), 248-271.

[STEA-76] Stearns, R., Lewis, P. and Rosenkrantz, D. Concurrency control for database systems. IEEE Symp. on Foundations of Comp. Sci., Oct. 1976, pp. 19-32.

[STON-79] Stonebraker, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Tr. on Software Engineering SE-5, 3 (May 1979), 188-194.

[THOM-78] Thomas, R. A solution to the concurrency control problem for multiple copy data bases. IEEE COMPCON 78, Feb. 1978, pp. 56-62.

NETWORK RESOURCE ALLOCATION

SYNCHRONIZATION OF DISTRIBUTED SIMULATION USING BROADCAST ALGORITHMS

J. Kent Peacock, Eric Manning, and J. W. Wong

Department of Computer Science and
Computer Communications Networks Group
University of Waterloo
Waterloo, Ontario N2L 3G1

Abstract

Simulation, particularly of networks of queues, is an application with a high degree of inherent parallelism, and is of considerable practical interest. We continue the analysis of synchronization methods for distributed simulation, defined by the taxonomy in our previous paper. Specifically, we develop algorithms for time-driven simulation using a network of processors. For most of the synchronization methods considered, each node k of an n-node network simulation cannot proceed directly with its part of a simulation. Rather, it must compute some function $B_k(v_1, v_2, \ldots , v_n)$, where v_i is some value which must be obtained from node i. The value of v_i at each node changes as the simulation progresses, and must be broadcast to every other node for the recomputation of the B-functions. In some cases, it is advantageous to compute the B-function in a distributed manner. Broadcast algorithms for such distributed computation are presented. Since the performance of a broadcast algorithm depends on the properties of the inter-process communication facility, we characterize some particular cases and give algorithms for each of them.

INTRODUCTION

Simulation is a widely used technique for system performance evaluation. The conventional approach to simulation is to develop a simulation program for a model, and then execute this program in a centralized computer system. This approach has led to the development of packages such as CSMP (7) for continuous simulation, and GPSS (10) and SIMSCRIPT (11) for discrete simulation.

The recent development of low-cost microprocessors has suggested an alternative approach to simulation. In this approach the simulated system is decomposed into components, and these components are simulated in a distributed manner over a network of processors. This approach is particularly attractive for the simulation of queueing network models (17) because of the inherent parallelism typically found in these models, and of their wide-spread application to computer systems and communication networks. Such parallelism can be exploited in the decomposition to give a potentially more cost-effective method of simulation. The distributed approach, however, requires the proper synchronization of the components before the simulation can be carried out correctly.

In our previous paper (16), a taxonomy which characterizes the different simulation methods was described. A slightly modified version of this taxonomy is shown in Figure 1. At the first level, we distinguish whether there is one or a network of processors available. With a network of processors, the simulation is decomposed into components and distributed over the processors. No such decomposition is assumed in the case of one processor only. The next level deals with the event-driven or time-driven nature of simulation. In event-driven simulation, the changes in system state are simulated when an event occurs, and the sequence of the simulation time (which corresponds to the se-

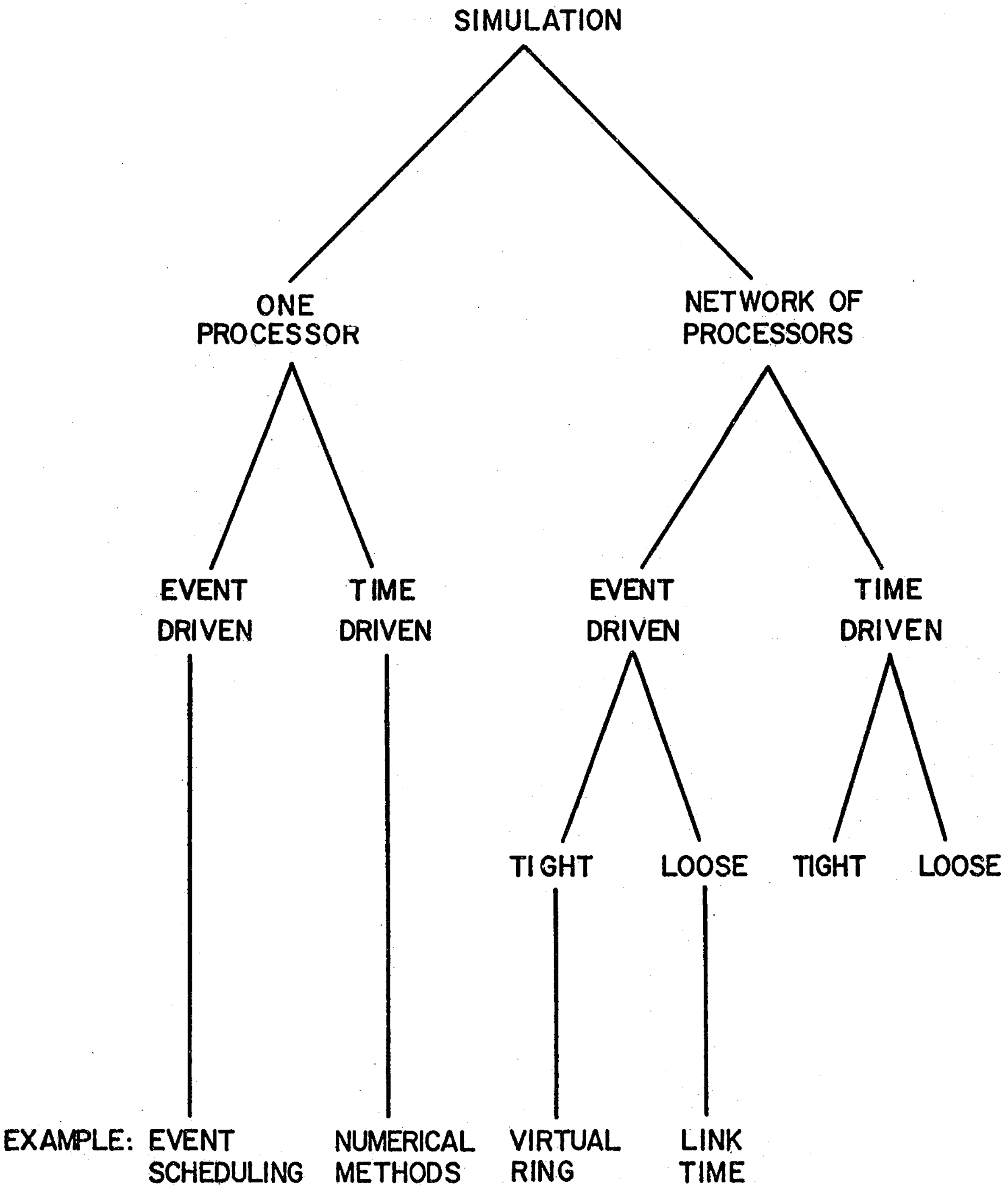


Figure 1. Taxonomy Tree

quence of event times) is monotonically non-decreasing. In time-driven simulation, the simulation time is incremented by a fixed amount which defines a simulation interval. All of the changes in system state in the present interval are simulated before advancing the simulation time to the next interval.

For the case of a network of processors, we also have a third level, depending on the value of simulation time at each component. The method is *tight* if the value of simulation time is the same for all components at each instant of real time. On the other hand, a *loose* method allows different components to have different values of simulation time at a given instant of real time. Loose simulation methods thus allow more exploitation of parallelism.

Algorithms for event-driven simulation with a network of processors have been developed by the authors (16), by Chandy et al. (2,3,4), and by Bryant (1). Examples are the virtual ring algorithm for tight event-driven and the link-time algorithm for loose event-driven simulation. In this paper, we consider the time-driven methods and present algorithms for the synchronization of the components. A fundamental feature of these algorithms is that a component (or a central controller) must broadcast a signal to every other component to indicate the end of a simulation interval. This broadcast feature is also observed in distributed algorithms for event-driven simulation, as well as other applications, such as distributed data bases.

We thus consider a class of algorithms called *broadcast algorithms* which are suitable for distributed simulation using a network of processors. A recent paper by Dalal and Metcalfe (8) has dealt with the broadcast of packets throughout a packet-switching network, where the topology of message passing is fixed according to the network structure. (By *message-passing topology*, we mean the structure *chosen* for messages to follow in a broadcast from a source to all other nodes of the network.) We shall be more interested in exploring cases in which our network allows any message-passing topology, and we shall look for topologies which give the minimum time to complete a broadcast.

For convenience, we will base our discussion on the simulation of a queueing network model with n nodes, where each node corresponds to a component in our decomposition. The general form of our broadcast algorithms requires that each node k must maintain some function $B_k(v_1,v_2,\ldots,v_n)$ where v_i is a value obtained from node i. The values of the v_i's change as the simulation progresses, and must be broadcast to every other node for the recomputation of the B-functions. Of particular interest is the case that a node k broadcast a request for the computation, in a distributed manner, of B_k. Algorithms for such a distributed computation are presented. The performance of these algorithms under three types of communication facilities are investigated.

LOOSE TIME-DRIVEN METHODS

In distributed simulation using the loose time-driven approach, simulation time advances by a fixed quantum size q. Whenever conditions permit, a node simulates its component over the time interval from s to s+q (which we call a tick), and then advances its simulation time to s+q. In the case of a queueing network model, these conditions are met when the node's immediate predecessors have all advanced their simulation time to s. In this section, we outline two algorithms for the loose time-driven method.

Centralized Algorithm

The centralized algorithm for loose time-driven simulation makes use of the interconnection graph of the simulated system. Each time a node finishes a tick, it sends an "advance" message to the synchronizer, which increments a clock for that node. For each immediate successor of this node, the algorithm checks if all of its immediate predecessors have advanced at least as far as the new time. If so, the synchronizer sends a message to the successor node telling it to start the next tick, including the minimum of its predecessors' times in the message. This node may then simulate up to that time. This algorithm has the desirable property that it takes the minimum number of messages to do the synchronization, that is, a maximum of two messages per tick per node.

Distributed Algorithm

The distributed algorithm for loose event-driven simulation bears a very strong resemblance to the link time algorithm for loose event-driven simulation (16). The main difference is that the link time is defined as the simulation time at the source node of an empty link, rather than as a lower bound on the next arrival, and it gets incremented by one quantum after each tick.

TIGHT TIME-DRIVEN METHODS

Synchronization of tight time-driven methods requires a method of determining when all nodes have completed the tick from s to s+q, and a means to inform all of the nodes that they should start simulating the next tick.

We expect that the cost-effectiveness of this method is heavily dependent on the distribution of processing requirements among the components of the simulation. Let p_i be the processor time per tick required at component i. If there is a k such that $p_k >> p_j$ for all $j \neq k$ then we would expect this time to dominate the time required per quantum interval. If each processor contains only one component, then we would also expect that all processors except the one containing component k would be idle most of the time. If we want to make maximum use of processors used or minimize the number of processors required, we may want to assign more than one component to each processor. If each p_i is known and constant, this is a bin packing problem with the capacity of the bins set to the maximum p_i. The solution to this problem gives optimal performance at the lowest cost, neglecting the overhead required for synchronization.

Centralized Synchronization Algorithm

This synchronization algorithm consists of a central process which keeps track of which components have finished the current tick, and which tells the components when to start the next tick. We note that with n components in the simulation, we require n messages for all the components to signal when done, and n messages to notify them that the next tick should be started, for a total of 2n messages per tick.

In designing an algorithm to perform this synchronization, we want to make the overhead per tick as low as possible. Since the processing time required per tick is dominated by the component with the largest p_i, we should inform that component first that time should be advanced. Extending this, we adopt the approach that the components are notified in the reverse order in which they signal the completion of the previous tick. This approach is based on the assumption that the p_i's are correlated, that is, p_i for time s+q to s+2q is likely to be approximately equal to p_i for time s to s+q.

We can analyze the performance of this algorithm relative to the p_i's by considering r_i, the number of nodes which are sent timer-advance messages before node i. Then the time for a tick is going to be at least $\max_i(p_i+(r_i+2)m)$, where m is the time for a message transfer. This could be larger due to the fact that only one message may be received at a time, so that messages arriving at the same time will suffer additional queueing delays.

Distributed Polling Algorithm

This algorithm performs a function similar to the centralized algorithm presented above, but does not keep track of the order of completion for each tick. It works in two phases: one phase for keeping track of which components have finished the current tick, and the other for notifying all components that the next tick may be started.

In the first phase, a message containing an n-bit field and a count of the number of nodes yet to complete the current tick is passed from node to node. The i'th bit of the n-bit field is 1 if the i'th component has finished the current tick and 0 otherwise. When the message arrives at component i, the component waits until processing of the current tick is complete, and then turns on the i'th bit and decrements the counter. If the counter is now zero, it is changed to n and the second phase is entered. Otherwise, the message is sent to a component whose entry in the bit vector is 0.

In the second phase, a message containing just a counter is sent around a virtual ring. When a node receives the message, the counter is decremented, and if it is not now zero, the message is passed onto the next node in the ring. Once a node has passed on the message, it starts processing the next tick. If the counter is 0, then it has returned to the node which initiated phase two, and this node starts off a phase one message with the bit vector set to 0's and the counter set to N.

Both this and the centralized algorithm are required to broadcast a "start next tick" message to all nodes. In the centralized case, the central controller sends the message to every node, whereas in the distributed case, the message passes around a virtual ring. Also, the virtual ring algorithm for the case of tight event-driven simulation given in our previous paper (16) broadcasts a new next event time to all nodes using a virtual ring. Neither of these message-passing topologies was chosen because of any virtue other than simplicity. It seems likely that there exist other ways of passing the messages to all the nodes which would offer better performance, and so we have developed and investigated some other algorithms for performing this broadcast. These are presented in the next section, which deals with broadcasting in a more general way.

BROADCAST ALGORITHMS

We consider the general case that node k of an n-node network simulator maintains some function $B_k(v_1, v_2, \ldots, v_n)$, where v_i is some value which must be obtained from node i. The value of this function is used to determine whether node k may proceed with its portion of the simulation or not. The value of v_i at each node changes as the simulation progresses, and hence so does the value of each function B_k. If each node maintains a copy of each v_i, then a change at node j from v_j to v_j' requires only that v_j' be broadcast to all other nodes in order for the new value B_k' to be computed. On the other hand, if the nodes do not maintain copies of the v_i's, then a change in v_j could in general require that every v_i be broadcast, since the B_k functions would have to be recomputed from scratch. So, there is a classical time/space tradeoff to be made here.

Of special interest to this study are B-functions of the form v_1 op v_2 op ... op v_n, where op is a commutative, associative operator. In this case, one could distribute the computation of the B_k among the entire set of nodes. Instead of broadcasting changes in v_j, the node would broadcast a request to compute the function B_k. We thus have two separate kinds of broadcast to consider: simple broadcast where every node eventually gets a message containing the new v_j', and broadcast with reply where each node receiving a broadcast from node k computes some sub-expression of B_k and replies with the result. The broadcast with reply has the important property that it can be used to implement broadcast with positive acknowledgment, where the B function is simply the logical function "all nodes have received the message".

There is no requirement that all of the B-functions be the same, but this is an interesting sub-case. Tight event-driven simulation uses the function $M = B_k = \min_i(net_i)$, and net_i is the next event time at node i. Besides having the same function at all nodes, the minimum is also a commutative and associative binary operator, so that we could let a designated node initiate a broadcast with reply to compute M in a distributed fashion. Once that node received the reply with the value of M, it would broadcast it to the others. Nodes for which $net_i = M$ would then be able to proceed with their parts of the simulation.

Some Broadcast Algorithms

We now consider algorithms to accomplish a broadcast and relate their performance to properties of the inter-process communication facility. For the moment, we assume that the inter-process communication is such that message delays between any two processes are constant and identical. We also consider the simple case in which only one broadcast is active at a time.

The particular algorithm which takes the minimum time depends heavily on the amount of interference there is between messages in the message transmission network. With heavy interference, as when all processes are assigned to a single processor, the minimum time to complete a broadcast with reply is proportional to the number of messages. On the other hand, with low interference, the number of messages is less important and the topology of the message passing dominates the minimum time. We consider the following cases: 1) complete interference, where only one message can be in transmission at a time; 2) interference at each node, where any number of messages are in transmission, but only one message per node can be sent or received at a time; and 3) broadcast facility, where one node at a time is allowed to send the same message to all others.

Complete Interference. An example of complete interference is the assignment of all processes to a single processor. The minimum time solution to this problem is to pass the request around a virtual ring. Upon reaching the last node, it is sent back to the source.

To argue that this solution takes the minimum time, we first note that the time is proportional to the total number of messages required to inform all nodes of the request and to collect the replies. Hence, minimizing the number of messages is equivalent to minimizing the time. Since each node besides the source must receive at least one request, and send at least one reply, and the source must send at least one request, and receive at least one reply, n is a lower bound on the total number of messages required, which the virtual ring meets. The essential feature of the ring which makes this possible, is that the request to a node's successor is also that node's reply, so the two functions are combined into one message. To see that this is the only structure which achieves this lower bound, we consider a topology in which one node sends the request and reply separately. Since every other node sends at least one message, the sum of messages sent is at least n+1. Therefore, to use only n messages, each node must receive and send only one message. The only topology for which this is possible is a ring.

No Interference Between Nodes. An example of this type of facility, where there is no interference between nodes to send to different destinations is a fixed time-division multiplexed (TDM) bus. This case is treated by having every node which has received the broadcast send messages to nodes which haven't. For example, the source (say node 0) informs node 1. Then nodes 0 and 1 inform nodes 2 and 3, giving 4 nodes which now have received the broadcast. Then nodes 0 through 3 inform nodes 4 through 7, and so on. Thus we see that at each stage, we double the number of nodes which have received the broadcast. After p message-passing time units, the structure of the tree produced outlines a subset of the edges of a p-dimensional hypercube. This is illustrated in Figure 2, which shows how a tree with 8 nodes can be mapped onto a cube. This approach is optimal because no other approach can broadcast the message to more nodes at each stage.

If this is a broadcast with reply, then when all of the nodes have received the message, the reply phase begins. The replies are returned in the opposite direction to the broadcast messages. A node does not reply until all of the nodes to which it sent a message have replied. Thus, leaves in the tree reply immediately with the v_i requested. As each reply is received at an intermediate node, the result of the function for the subtree of that node is accumulated, and when all replies have been received and processed, the result is sent to the requestor of the node. The tree of nodes which haven't replied therefore halves in size at each

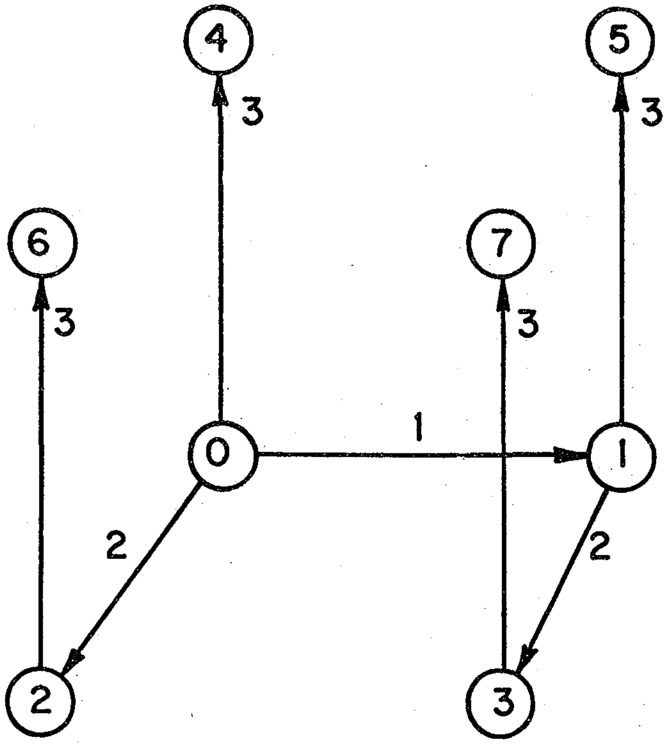


Figure 2. Hypercube Broadcast Representation

stage, just as it doubled its size during the request phase, until eventually only node 0 remains. We see that this takes a total of 2n-2 messages, and a time of 2 ceil($\log_2 n$) to complete the computation of the function, where ceil(x) denotes the smallest integer $\geq$ x. Note that for n $\leq$ 6, 2 ceil($\log_2$ n) $\geq$ n, so that the virtual ring solution is at least as good for these values, and we choose to use it since its implementation is simpler.

This particular reply scheme falls into a class of so-called "echo algorithms" studied by Chang (6). We notice that this approach to the reply is not optimal since it does not take advantage of the fact that sub-expressions of B_k can be computed as the broadcast is propagated in the forward direction. If this is done, the number of sub-expressions left to be merged at the end of the broadcast phase is the number of leaves in the tree, rather than the number of nodes. It should be possible in most cases to combine the smaller number of sub-expressions in less time.

We have assumed thus far that sending and receiving are synchronized, that is, node a in the process of sending a message to node b cannot start to send another message until the first has been received at node b. It will be useful to define the characteristics of message passing more formally as follows. Suppose that node a starts to send a message to node b at t_1 and can start another send at t_2, and that node b starts to receive the message at t_3 and finishes receiving it at t_4. We can now define $S = t_2 - t_1$ as the send time and $D = t_4 - t_1$ as the delay time. Thus far, we have considered only the case in which $S = D =$ constant. We will next generalize this to the case in which $S \neq D$, where S and D are still constants. For simplicity, we only consider the broadcast without reply case in the remainder of this section.

It will be useful in the subsequent discussion to use the broadcast tree, which we now define. An example of such a tree for $D = S$ is shown in Figure 3. All nodes on a horizontal line receive the broadcast from the same node, and that node is connected vertically to the left end of the horizontal line. Also, the horizontal direction is calibrated to represent the time at which a node receives the broadcast.

We first consider the case for which $D > S$. An example of this is the use of a non-blocking send primitive (15), where the sender is free to start another message without waiting for completion of those in transit. For ease of analysis, we let $D = kS$, k a positive integer, and give an example of a broadcast tree in Figure 4 for $k = 3$. Let N(i) be the number of nodes which have received a broadcast at time $t = iS$. With our optimal strategy, there will thus be N(i) nodes starting to send a message at iS. These messages

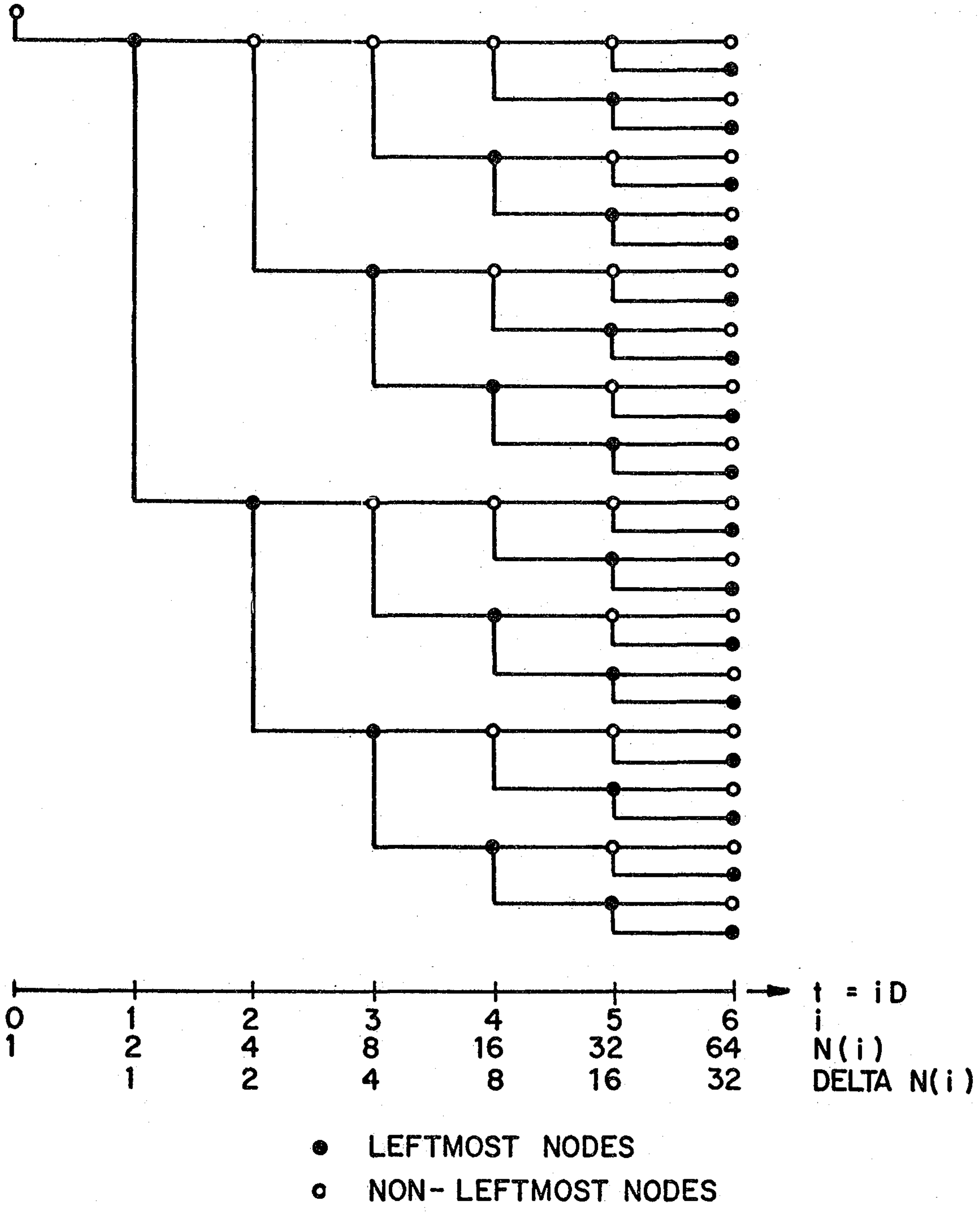
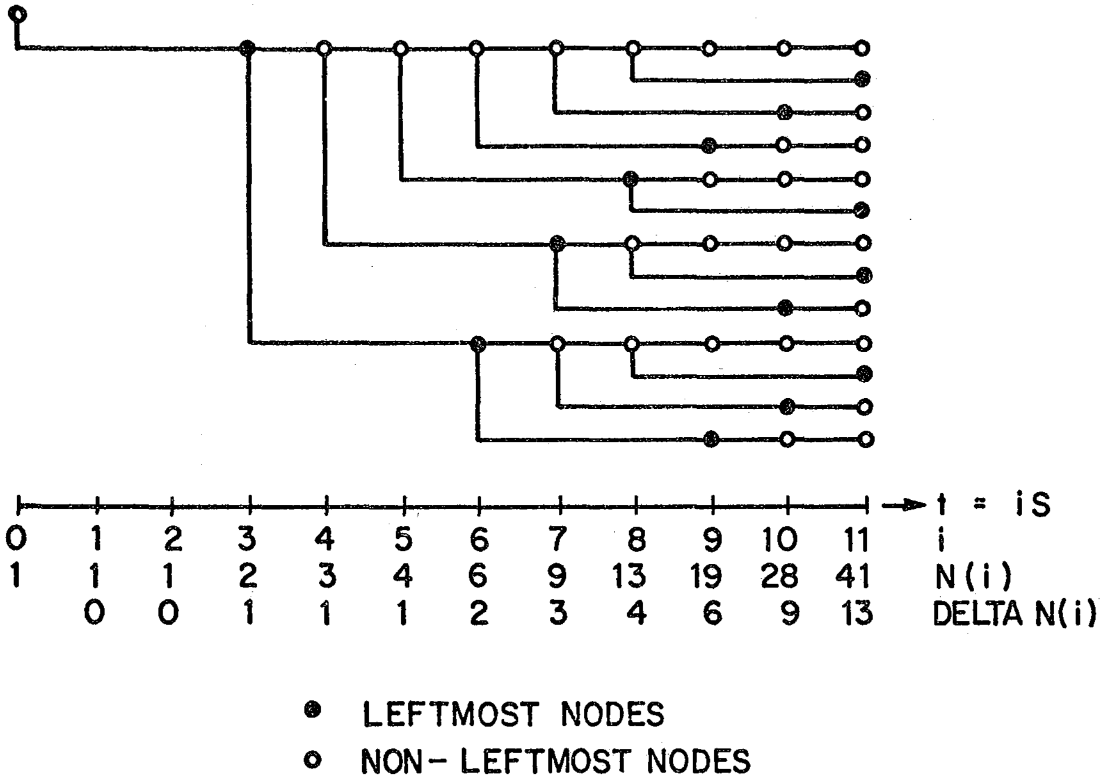


Figure 3. Broadcast Tree for D = S

are not received until time iS + D = (i + k)S. Since we send to destinations that do not know of the broadcast, N(i) new nodes are informed at time (i + k)S, and so we can write, replacing i by i-k, N(i) = N(i-1) + N(i-k). Since the first message from the originator of the broadcast is received at time kS, we have the initial conditions, N(i) = 1 for 0 $\leq$ i $\leq$ k-1.



i	0	1	2	3	4	5	6	7	8	9	10	11
N(i)	1	1	1	2	3	4	6	9	13	19	28	41
DELTA N(i)		0	0	1	1	1	2	3	4	6	9	13

Figure 4. Broadcast Tree for D = 3S

To find the time taken for a broadcast to n nodes, we must find the minimum i such that n $\leq$ N(i). We observe that for k = 1, this gives the same result as we obtained from our previous analysis of this case, namely that N(i) = 2^i and hence the broadcast time is ceil($\log_2$ n). Also, for k $\geq$ n-1, the originating node sends messages to all of the n-1 remaining nodes.

The case $D < S$ occurs when blocking primitives (15) are used, where the sender is not re-activated until an acknowledgment is obtained from the receiver. To ease the analysis, we consider the sub-case in which $S = kD$, k a positive integer. We define $N(i)$ to be the maximum number of nodes that can be informed of the broadcast at time iD.

In this case, $N(i)$ is the number of nodes in the broadcast tree at time iD. It will be more convenient to study the behaviour of $\Delta N(i) = N(i) - N(i-1)$ rather than $N(i)$, since $\Delta N(i)$ is the number of nodes which receive broadcast messages at time iD. It will be helpful to visualize a broadcast tree, as shown in Figure 5 for the case $k = 3$. The messages received at time $t = iD$ will be of two types: those leftmost on a horizontal branch of a broadcast tree, and those not leftmost. The number of messages received which are not leftmost is the same as the number received at time $t - S = (i - k)D$. This follows from the fact that every node which sends a message that is received at time $t - S$ also sends a message which is received at time t. The number of messages received which are leftmost is the same as the number of nodes which received the broadcast at time $t - D$ because each of these nodes immediately sent a message which is received at time t. We can thus write $\Delta N(i) = \Delta N(i-1) + \Delta N(i-k)$. Here we have the initial conditions $\Delta N(i) = 1$ for $1 \leq i \leq k$ and $N(0) = 1$. We notice that the recurrence for the $\Delta N(i)$'s is the same recurrence as we found for the case $D > S$. In this case, however, if $k \geq n-1$, the optimum solution is a virtual ring for broadcast reply, and a virtual ring without the last edge for the simple broadcast.

We notice that we can also write the recurrence for the case $D > S$ in the form $\Delta N(i) = \Delta N(i-1) + \Delta N(i-k)$, with suitable initial conditions. This suggests that there may be a general recurrence relation which covers both cases. Let $t = iq$, where q is a quantum size and let $D = jq$ and $S =$

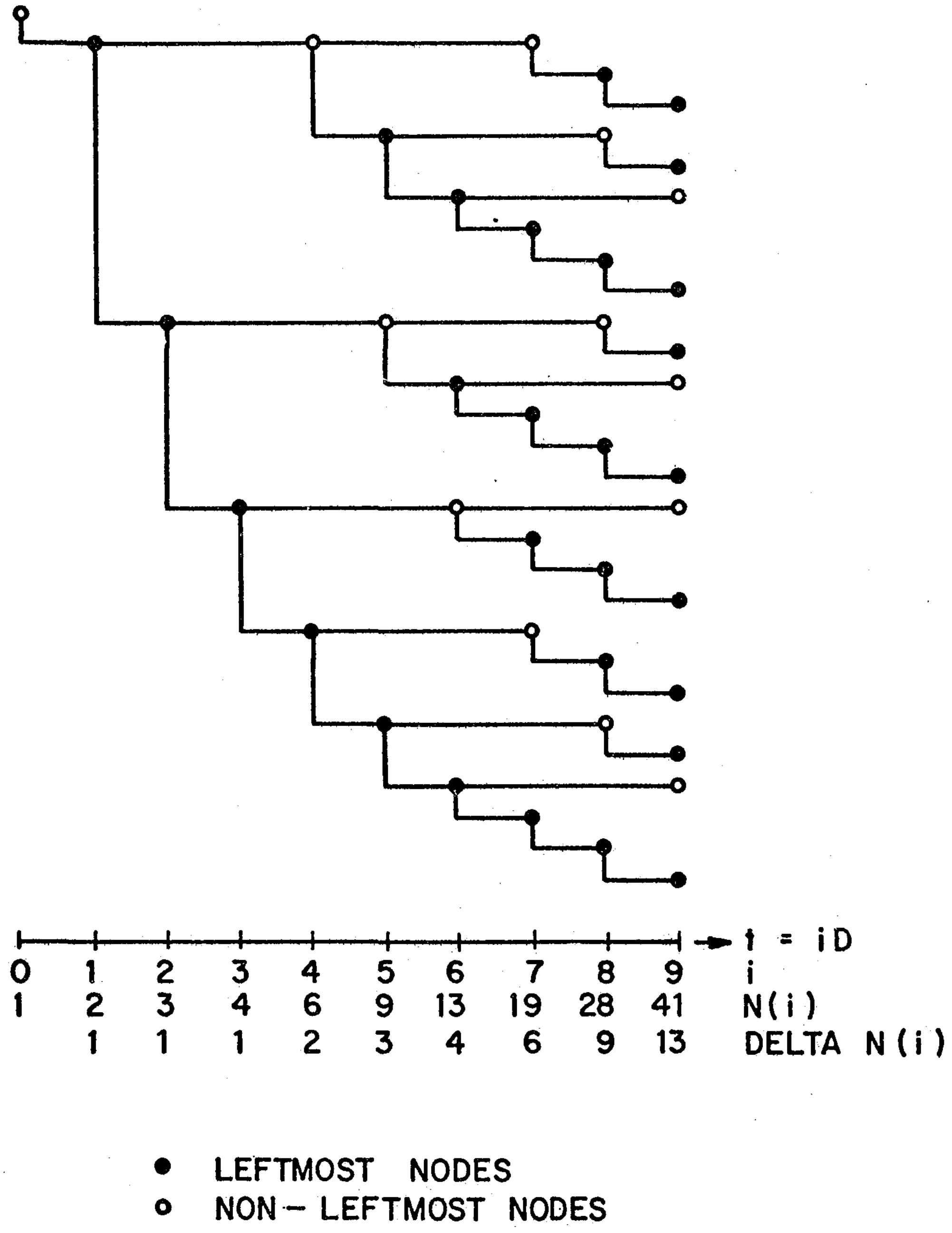


Figure 5. Broadcast Tree for S = 3D

kq. Then, by an argument similar to the one for $D < S$, we obtain the recurrence $\Delta N(i) = \Delta N(i-j) + \Delta N(i-k)$. Since the first message sent arrives at time D, we have the initial conditions $\Delta N(i) = 0$ for $i < j$, and $\Delta N(j) = N(0) = 1$.

It is useful to consider the generating function for the $\Delta N(i)$ sequence. Using a derivation similar to that in Knuth (13) for the Fibonacci sequence, we obtain the generating function $G(z) = z^j / (1 - z^j - z^k)$. One could obtain a closed-form solution for $\Delta N(i)$ by finding the roots of the denominator of $G(z)$, obtaining the partial fraction expansion, and inverting the result. (In order to minimize the order of the denominator, it is best to choose q so that the greatest common divisor of j and k is 1.) It is rather interesting that the denominator of $G(z)$ is symmetric in j and k, since this means that interchanging D and S results in a $\Delta N(i)$ which is the same except for a shift along the i-axis of $j - k$.

Hardware Broadcast Facility. The final case is one in which there is some broadcast facility available which allows a node to send the same message to all other nodes. An example of a facility which has the potential to perform this type of broadcast is Ethernet (14), even though it could also be considered to fall in the class of total interference. In this case, the source node sends its request to all nodes at once, but cannot receive their replies all at once, so we could use the same tree structure as the previous case, during the reply phase only. If the broadcast takes time T, then the total time to do a broadcast with reply for the case $S = D$ is $T + \text{ceil}(\log_2 n)D$.

IMPLICATIONS AND FUTURE WORK

The results of the previous section have given us the capability to generate a broadcast tree which provides broadcasting in the minimum time, provided that the S and D values can be determined. However, the analysis of these broadcast trees assumes that only one request is active at a time. If this is not the case, then queueing delays caused by competing requests will tend to make the message switching mechanism behave as one with higher interference, so that the choice of optimum message passing topology becomes unclear.

An approach to this problem which we intend to investigate is the inclusion of queueing delay into the value of D. In general, the larger the number of broadcasts active at any one time, the larger is the queueing delay, and the larger will be the D value used. On the other hand, if the simulation is processor-bound, it may be desirable to restrict processor time for broadcasts, so that messages are not sent at the maximum rate possible. This corresponds to an effective increase in S.

The analysis of the case with no interference is the most general result of the previous section. We note that the total interference case is approximated by $S >> D$, and that the hardware broadcast case is approximated by $S << D$. It will be useful to explore the ways in which S and D can be traded off against one another, and then look for the best values.

The assumption of constant S and D between all node pairs of a network may not be realistic for some systems. Thus, the extension of these results to cases in which the D and S values are not the same for all nodes, and in which they are not constant is worthwhile.

Finally, the generating function for the general recurrence, $G(z) = z^k / (1 - z^j - z^k)$, requires further investigation. The denominator of G(z) has one real root between 0 and 1, as can be readily seen from the fact that G(0) = 1 and G(1) = -1. It is our conjecture that this root dominates the asymptotic behaviour of $\Delta N(i)$, which, if true, would allow us to write $\Delta N(i)$ approximately as Cr^i for large i. Here, r = 1/r', where r' is the value of the root between 0 and 1.

REFERENCES

1. Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems", MIT/LCS/TR-188, Massachusetts Institute of Technology, Cambridge, Massachusetts, (Nov 1977).

2. Chandy, K.M., Holmes, V., and Misra, J., "Distributed Simulation of Networks," Technical Report 81, Department of Computer Sciences, University of Texas at Austin, Texas 78712, also submitted to Computer Networks.

3. Chandy, K.M. and Misra, J., "A Non-Trivial Example of Concurrent Processing: Distributed Simulation," Technical Report 82, Department of Computer Sciences, University of Texas at Austin, Texas 78712, also in Proceedings COMPSAC, Chicago, Nov. 16-18, 1978.

4. Chandy, K.M. and Misra, J., "Specification, Synthesis, Verification, and Performance Analysis of Distributed Programs; A Case Study: Distributed Simulation," Technical Report 86, Department of Computer Sciences, University of Texas at Austin, Texas 78712.

5. Chang, E., and Roberts, R., "An Improved Algorithm for

Decentralized Extrema-Finding in Circular Configurations of Processes", CACM Vol 22 No 8, pp 281-283, (May 1979).

6. Chang, E., "Echo Algorithms: Depth Parallel Operations on General Graphs", submitted to SiComp.

7. Continuous System Modelling Program, IBM Document No GH20-0367-4, (Jan 1972).

8. Dalal, Yogen K., and Metcalfe, Robert M., "Reverse Path Forwarding of Broadcast Packets", CACM, Vol 21-12, (Dec 1978).

9. Emshoff, J.R., and Sisson, R.L., *Design and Use of Computer Simulation Models*, Macmillan, New York, (1970).

10. IBM, *General Purpose Simulation System System/360 User's Manual*, GH 20-0326, White Plains, N.Y., (1970).

11. Kiviat, P.J., Villanueva, R., Markowitz, H.M., *Simscript II.5 Programming Language*, Consolidated Analysis Centers Inc., Los Angeles, California, (1973).

12. Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Toronto, (1978).

13. Knuth, Donald E., *Fundamental Algorithms* The Art of Computer Programming, Vol 1, Second Ed., pp 78-83, Addison-Wesley, Reading, Mass., (1973).

14. Metcalfe, Robert M., and Boggs, David R., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM, Vol 19 No 7, (July 1976).

15. Meisner, N.L., "Process Management and Communication Facilities for Distributed Operating Systems", Master's Thesis, University of Waterloo, Waterloo, Ontario (1979).

16. Peacock, J.K, Wong, J.W., and Manning, Eric, "Distributed Simulation using a Network of Micro-Processors", Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks, (Aug 1978), and Computer Networks, Vol 3, No 1, pp 44-56, (Feb 1979).

17. ACM Computing Surveys, Special Issue on Queueing Network Models of Computer System Performance, (Sept 1978).

ACKNOWLEDGMENT

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada.

THE UPDATING PROTOCOL OF THE ARPANET'S NEW ROUTING ALGORITHM: A CASE STUDY IN MAINTAINING IDENTICAL COPIES OF A CHANGING DISTRIBUTED DATA BASE

Eric C. Rosen

Bolt Beranek and Newman Inc.

Abstract

In May 1978 a new routing algorithm was installed in the ARPANET. In this algorithm, each network node makes an independent routing decision, based on information about delays throughout the network. The delay on a particular line is measured at the nodes attached to that line, and disseminated to the rest of the network in the form of a "routing update." This paper discusses one aspect of the routing algorithm, viz. its updating protocol (i.e. the protocol used to disseminate the updates). The problem of devising a good updating protocol is shown to be a problem in the management of a distributed data base. The requirements which any such protocol must meet in order to be satisfactory are presented and discussed. The protocol is then developed so as to meet these requirements. Other possible protocols are discussed, and shown not to meet the requirements.

1. INTRODUCTION

The design of distributed adaptive routing algorithms for packet-switching computer networks gives rise to many and varied problems. In this paper we discuss one such problem, as well as the solution we devised as part of the design of a new routing algorithm for the ARPANET. (This new algorithm, described in [1], became operational in May 1979.) The problem arises from the fact that although each packet switch (node) in the network must make an independent decision on how to route packets, the data base it needs to make these decisions is a distributed data base. That is, each node has direct access to only a small portion of the data base; to gain access to the rest, the nodes must communicate with each other. The messages used to transmit the routing data base information are known as "routing updates." Choosing a good routing update protocol is a problem in distributed data base management; it is this problem that will be discussed here.

2. THE PROBLEM

In distributed routing, each node runs an independent "shortest-path computation" which maps certain state information about the network into a set of routes from the given node to each other node. A routing algorithm may be said to be adaptive insofar as the chosen routes adapt systematically to changes in this state information. If one wants to have routing which adapts only to changes in the network's topology, then the only state information which is necessary is the up/down status of each network line. If, on the other hand, one wants the routing to adapt to changes in packet delay, then the necessary state information is the delay over each network line; this is the approach adopted in the ARPANET. This state information is gathered by a measurement process which runs in each node. The state of a particular line, however, can be directly measured only by the node that transmits over that line; there is no way for a node to directly measure the delay over a line to which it is not connected. Nevertheless, if each node is to make an independent routing decision, each node must know the delay over each network line. This is what gives rise to the distributed data base problem. In order for each node to perform an independent shortest-path computation, each node must have access to a data base which consists of the delay over each network line. Since each node is able to measure the delay on only a few lines, the data base is distributed throughout the network.

There are two possible approaches to solving the problem of the distributed data base. One way is to distribute the shortest-path computation itself so that each piece of the computation has direct access to the part of the data base that it needs. This is the approach taken by the ARPANET's old routing scheme. The alternative approach is to develop a quick and reliable updating protocol for transmitting changes in the data base to all nodes in the network. This makes the entire distributed data base locally available to each node. This approach, adopted by the ARPANET's new routing algorithm, is the one that shall be discussed here.

3. REQUIREMENTS OF THE UPDATING PROTOCOL

Protocols for handling process-process communication abound in computer networks, and one might think that devising an updating protocol for routing offers no special problems not found generally in the design of such protocols. This is not the case. The role of routing in the network places special requirements on the updating protocol. If each node is to maintain its own copy of the entire data base, and if each node's routing decisions are to be made entirely on the basis of its own copy of the data base, then it is essential to ensure that identical copies of the data base are kept at all nodes. If this constraint is not met, then the nodes may make conflicting routing decisions, causing a major network failure. For example, suppose A and B are neighboring nodes, and D is a third node elsewhere in the network. If A were ever to decide that traffic destined for D should be routed via B, while B decides that traffic for D should be routed via A, neither A nor B would be capable of delivering traffic to D; traffic would loop endlessly between A and B. For the shortest-path computation used in the ARPANET's *new* routing scheme, this situation can be shown to be impossible, *if all the nodes have identical copies of the data base.* If they do not have identical copies of the data base, however, then there is no assurance that the routing scheme will be able to deliver packets to their destinations. It must be understood though that the data base is constantly changing. Whenever the average delay over a line changes, or when a line goes down or comes up, there is a change in the value of one of the entries in the distributed data base. This change must be made known to all nodes quickly if the routing algorithm is to continue to operate correctly. Our problem is to devise a protocol which ensures, to the greatest degree possible, that all nodes maintain an identical copy of the data base, even though it is under continual change. The requirements of such a protocol are the following:

1. *Reliability.* The protocol must ensure delivery of all updates to all nodes. The ordinary data transfer protocol of the ARPANET is not sufficiently reliable. Data packets in the ARPANET can be lost due to node crashes, network partitions, or severe congestion. Loss of data packets under these (admittedly low probability) circumstances is unfortunate, but it does not have any *globally* deleterious effect. Loss of a routing update, on the other hand, will result in inconsistent copies of the routing data base, possibly crippling routing and bringing down the network. A more reliable protocol must be found.

2. *Quickness.* Since updates cannot make their way across the network instantaneously, there will always be some interval of time after a new update is generated when the copies of the data base throughout the network are not identical. Our goal is to keep this interval as small as possible. Note that when an ordinary data packet travels slowly, the only bad effect is that some user sees a long delay. When routing updates travel slowly, however, all users can suffer.

3. Priority. Whenever routing updates contend with other packets for the same resources (such as buffer space, line bandwidth, or processor bandwidth), the updates must be given priority. To put this point another way, the flow of updates must not be slowed down when the network is heavily loaded.

4. Sequential delivery. If two updates contain information about the same line, then the updates must be processed at all nodes in the order in which they are generated. If different nodes process these updates in different orders, inconsistent copies of the data base are sure to result. Note, however, that as long as the updates contain complete information, so that later updates obsolete earlier ones, it is not necessary to have guaranteed sequential delivery. When later updates arrive before earlier ones, the later ones can be processed immediately, and the earlier ones simply discarded when they arrive. A policy of guaranteed sequential delivery would delay the processing of the later update until the earlier one arrives, thereby defeating the requirement of quickness.

5. Efficiency. The routing updates should not place such a great demand on network resources that the routing scheme does more harm than good.

4. THE UPDATING PROTOCOL DEVELOPED

Some of these requirements were easy to handle within the structure of the ARPANET nodes. Others were more difficult, and required the development of new protocols unlike anything previously found in the ARPANET. One of the easy ones was priority. The ARPANET already had a priority queueing structure which could be easily adapted to allow highest priority to routing updates. To handle efficiency considerations, we made the update messages small and infrequently generated. Each update message from a given node contains information on all the lines emanating from that node, rather than on just one line. The update packets themselves are quite small, about 176 bits on the average. Furthermore, each node is constrained to generate updates only infrequently. Changes in delay on a line cannot cause generation of updates more often than once every 10 seconds. Additional updates can be generated if a line goes down or comes up. However, when a line goes down it cannot come back up for at least 60 seconds, so there is no need to worry about excessive updating due to line failures. An important consequence of these features of the update generation process is that there is no need to exert flow control on the routing updates. They simply cannot be generated frequently enough to give rise to the sort of problems which flow control is needed to prevent.

For speed and reliability we decided to use a transmission procedure known as flooding. Each update from a given node carries a sequence number which identifies it uniquely. When a node generates an update, it sends a copy of the update to each of its neighbors. Whenever a node receives an update which it has not seen before, it sends a copy to each neighbor, except the one from which it it was received. When a node receives an update which it has seen before (or which was generated prior to one it has seen before), the update is discarded. The two most salient aspects of the transmission procedure are:

1. The transmission of routing update messages is in no way dependent on the performance of the routing algorithm. Even if some problem arises with the routing algorithm, transmission of routing updates is not affected. This independence is an important reliability measure. It is also important in ensuring quick transmission. In effect, it establishes a fixed routing policy for transmission of routing updates, allowing the updates to bypass many of the normal packet-forwarding procedures. This means that the forwarding of updates can be done at the highest priority level, with negligible processing delay.

2. Each node receives a copy of each update from each of its neighbors. This ensures that updates cannot be lost due to node failures or network partitions.

It must be understood, though, that transmission across the network consists of a sequence of point-to-point transmissions, or hops. Flooding assures speed and reliability only insofar as the individual point-to-point transmissions are quick and reliable. Packets transmitted from one node to another need to be protected against the following problems:

1. Line errors. Bursts of noise on the telephone line connecting two nodes can result in a packet's failing to be received correctly.

2. Buffer shortage at the receiving end. Exhaustion of the receiver's buffer pool can cause it to miss a packet.

3. Processor overload at the receiving end. On occasion, the nodes have been observed to have such a heavy processing load that they sometimes miss

packets because they cannot process their interrupts fast enough.

So we need some sort of reliable transmission protocol, whereby updates get retransmitted until they are known to have been correctly received at all nodes. One possibility would be to have every node receiving an update send an acknowledgement to the source of the update. If the source does not receive acknowledgements from all other nodes, it retransmits the update, either flooding it again or sending it directly to the node (or nodes) which did not receive it the first time. However, reliable transmission protocols based on retransmissons from the source tend to be cumbersome, slow, and inefficient. Such protocols may be suitable for transmission of ordinary data, but <u>not</u> for transmission of routing updates. The only alternative is to have a protocol that requires each update to be acknowledged over each line on which it is transmitted, and retransmitted on a line-by-line basis whenever necessary. The network does have such a reliable point-point transmission protocol, known as the IMP-IMP protocol, which it uses in the transmission of ordinary data packets. This protocol divides each line (in each direction) into eight logical channels. Each logical channel can have only one packet in flight at a time. Once a packet has been transmitted on a logical channel, no further packets can be transmitted on that channel until the first one has been acknowledged. A packet which is not acknowledged within a certain period of time is retransmitted, and the retransmissions will continue periodically until an acknowledgement is finally received. This protocol, whatever its merits for data packets, is unsuitable for routing updates. It has reliability, but not quickness. The problem is that the IMP-IMP protocol can block transmission on a line, even if the line is idle. This will happen if all eight logical channels are filled with packets awaiting acknowledgement. While the acknowledgements are being awaited, the line may be idle, yet no additional transmissions are possible, since there are no empty logical channels. (A similar point could be made against other link transmission protocols, such as HDLC.) We do not, however, want to delay the transmission of a routing update merely because all eight logical channels are in use by data packets.

One way to alleviate the problem is to add additional logical channels to be used only for routing updates. For instance, if there are NN nodes, one could add NN logical channels. A routing update would be sent on the channel which corresponds to its source node. Then routing updates would compete for logical channel space only with other routing updates which originate from the same node. This does not totally alleviate the problem of blocking transmission on an idle line, since it is possible for several updates from the same node to arrive in rapid succession, in which case their transmission would be unnecessarily slowed. However, the routing updates have three special properties which distinguish them from ordinary data packets and enable the logical channel protocol to be significantly simplified:

1. If a routing update from node A is transmitted from node B to node C, and then a later update from node A is received at node B before the prior update is acknowledged, we no longer care whether the prior update is correctly received at node C or not. Since the later update obsoletes the prior one, there is no reason to continue retransmitting the prior one, and all resources can be devoted to the transmission of the later one.

2. Each routing update carries a sequence number which (together with the number of its source node) can be used to identify it uniquely on each link over which it is transmitted. (The assignment of sequence numbers to routing updates will be discussed later.) Ordinarily, data packets do not carry any identifier which the link transmission protocol can use. Therefore, the link transmission protocol must maintain its own set of identifiers to assign to packets. If the set of identifiers is small, then only a small number of packets can be in flight at once. The IMP-IMP protocol maintains only eight identifiers (one for each logical channel) for ordinary packets, which is why only eight packets can be in flight at once. If, however, the link transmission protocol for routing updates identifies the updates by their sequence numbers, and if the sequence numbers are 6 bits long (as in the ARPANET), up to 64 updates from each node could be in flight on a link at any time.

3. Ordinary data packets must be kept buffered while awaiting acknowledgement. One reason why only a few data packets can be in flight at once is that each in-flight packet uses a buffer, and the ARPANET is short on buffer space. Routing updates, however, need not be kept buffered while awaiting acknowledgement. When a routing update is received at a node, the information it contains is copied into the node's copy of the data base. If the update has to be retransmitted, it can be re-created from the information in the node's data base tables. Hence there is no need to keep the update packet buffered, and buffering shortage does not constrain the number of updates in flight at once.

The second property eliminates the problem of blocking transmission on an idle line. The link transmission protocol for routing updates need never delay transmission when the line is idle, since updates sent on the same logical channel are not competing for sequence number space. The first property implies that in some cases, it is not even necessary to wait for an acknowledgement. The third property implies that there is not a large buffering cost in having many routing updates in flight at once on a single line. Therefore, we have adopted the following protocol. On (each direction) of each network line there is a separate logical channel for each node which may be the source of a routing update (i.e. for each node.) After an update is transmitted on a line, it is retransmitted periodically over that line, until one of the following two events occurs:

a) It is acknowledged by the node at the other end of that line.

b) A later update from the same source node is received over any line.

This protocol meets the desiderata of quickness and reliability.

We have now shown how to meet all the requirements except one -- sequential delivery. One possible way to ensure sequential delivery would be to refrain from generating a new update until the previous update is known to have been received by all nodes. While procedures for doing this do exist, they tend to be slow and unresponsive. Therefore, we have chosen to ensure sequential delivery by the use of sequence numbers. Every time a node generates a new update, it assigns it a sequence number one greater than that assigned to its previous update. The other nodes in the network use this sequence number to determine which of two updates (from the same source node) is the more recently generated. However, the use of sequence numbers introduces a new protocol problem, that of keeping sequence numbers synchronized.

Suppose node A receives an update from node B with sequence number 7. At some later time, node A receives an update from node B with sequence number 6. Furthermore, no update from B arrives at A in the interim. Ordinarily, this would imply that node A has received the updates out of order. Update number 6 has already been obsoleted by by update number 7, and should just be discarded. However, suppose that node B had crashed after sending update 7. When it comes back up, it may not remember that its last update was numbered 7; it may start its numbering over again. In that case, update 6 may actually be more recent than update 7, and node A will do the wrong thing. A similar problem can arise as a result of network partitions. Suppose that a series of line failures partitions the network, so that there is no path between node A and node B. While the partition lasts, node B continues to generate updates, giving each one a higher sequence number than the last. Node A, however, cannot receive these updates. Since the sequence numbers must be represented in a finite number of bits, they will eventually wrap around. Suppose that node B's sequence numbers wrap around several times during the partition, and that when the partition ends, the next update sent by B is numbered 6. Again, node A will make the wrong decision.

It is clear that when communication is re-established between two nodes that have been temporarily unable to communicate, some explicit procedure must be invoked to enable those two nodes to get their sequence numbers re-synchronized, so that each knows what sequence number to expect next from the other. Most protocols that depend on sequence numbers use a handshake procedure to synchronize their sequence numbers at the beginning of a communication. However, this is not suitable for our purposes. Since every node generates updates which must go to all other nodes, there would have to be a handshake between each pair of nodes. In a 100-node network, this is 10,000 handshakes. Clearly, it would be desirable to find a synchronization procedure which is more efficient.

It may be thought that the routing data base itself contains enough information to enable all nodes to re-synchronize their update sequence numbers after a partition, without any explicit handshake procedure. After all, the routing computation enables each node to know whether another node is reachable (i.e. whether a path to the other node exists) or not. When a node becomes unreachable, all updates from it can be ignored. When it becomes reachable again, the next update received from it can be accepted, no matter what its sequence number is. This automatically resynchronizes the sequence numbers.

Although this scheme is superficially attractive, it has serious difficulties, as would any scheme which requires the nodes to selectively ignore some updates. Recall that if there is any long-term discrepancy in the data bases maintained by the nodes, the routing calculation may result in the formation of routing loops which can make the network useless. The proposed scheme enables such discrepancies to exist after a partition ends. Suppose (for concreteness) that the network is partitioned East-West. When the partition ends, the Eastern nodes will initially appear unreachable to the Western nodes, and vice versa. Then updates will begin to flow across the East-West boundary. Eventually, all nodes will have processed updates from all other nodes, and they will all see each other as reachable again. The problem arises though because Western nodes cannot begin to process updates from Eastern nodes as soon as they become reachable. Rather, they must wait until the Eastern nodes appear reachable, according to the routing computation. Nodes in the East do not appear to be reachable to nodes in the West as soon as they actually become reachable; the Eastern nodes appear to be reachable when updates from the East get processed by the Western nodes. The order in which nodes start to appear reachable depends on the order in which updates are processed. But as updates flow from East to West, different Western nodes will process the updates in different orders, and at different rates. An eastern node that appears reachable to one Western node at time t may not appear reachable to another Western node until some later time t', if various updates from the east reach the Western nodes in different orders. This means that if E is an Eastern node, and W1 and W2 are Western nodes, there may be some time interval during which W1 can accept updates from E, while W2 must ignore them. If W2 ignores an update, it does not forward it. Therefore W2 (and all nodes beyond it) have no chance to get an update from E until some later time, when E generates another update.

The result is that it may be a very long time before updates from E are able to reach all the Western nodes (even though they are able to reach some Western nodes in a very short time). During this time, the nodes' copies of the data base are inconsistent.

It must be understood that the problem is not merely that it will take a long time to re-integrate the two segments after a partition. Rather, when a partition ends, incorrect routing patterns may form which affect communication even between nodes in the same segment. For example, two Eastern nodes which are communicating with each other may begin routing their traffic to each other via a series of Western nodes. But if the Western nodes hold inconsistent information about the Eastern nodes, the traffic may never get through. As a result, some nodes which were able to communicate during the partition may not be able to communicate after it ends.

The source of the problem with the proposed scheme is that some nodes are forced to ignore certain updates while others are not. It is dangerous to ignore updates selectively. Unless all nodes ignore the same updates at the same time, their copies of the data base may not be identical. One way to avoid this problem is to develop a scheme which allows all nodes to process all updates they receive, as soon as a partition ends:

> Let zero serve as a canonical lowest sequence number. No update packet ever carres a sequence number of zero. However, when a node A is determined by a node B to be unreachable, B acts as if the sequence number of A's most recent update were zero. Then when B next receives an update from A, the new update is automatically accepted as a recent update, and processed normally.

The intent of this scheme is that when a partition ends and updates begin to flow again between the segments, they can be accepted and processed as soon as they are received. There is no need to wait until a node appears reachable before its updates can be accepted. However, this scheme has a different sort of problem which is just as serious.

Suppose again that the network is partitioned East-West. Let M be an Eastern node which is on the border of the partition. Let A, B, and C be three other Eastern nodes which are connected in a triangle, and let W be a Western node. Let m be an update from M which reports the partition. That is, the other Eastern nodes detect the partition as a result of processing m. (Presumably m reports that the line between M and its Western neighbor M' has gone down.) Let w be an update from W which reached the Eastern segment of the network just before partition, and let s(w) be its sequence number. Now it is certainly possible that m gets to A before w does, and that w actually follows m around the triangle. As update m travels around the triangle, IMPs A, B, and C will determine that W is unreachable; henceforth they will act as if W's last update had had sequence number zero. Almost immediately thereafter, update w will be received. Since zero is the canonical lowest sequence number, s(w) > 0, so even though w was generated before the partition, it looks like a recent update. It is accepted and forwarded. However, the next time A, B, or C does a routing computation, they will again determine that W is unreachable, and again begin to act as if its most recent sequence number were zero. Once they do this, they no longer "remember" that they have seen w before. When w comes around the loop again, it again looks like a recent update (s(w) > 0), and is again accepted and forwarded. There is nothing to stop this process, which may continue indefinitely. In fact, w may still be traveling around when the partition ends. Once the partition is over, W will eventually send out another update, w'. This may result in w and w' being in the network at the same time. If the partition lasted long enough for the sequence numbers to wrap around, then it is meaningless to compare s(w') with s(w). As a result, the nodes may incorrectly believe w to be more recent than w', and routing will be based on very old and out-of-date information. Depending on the exact values of s(w) and s(w'), this problem may persist for a very long time, causing extremely bad performance throughout the whole network (for instance, if w' reports that one of W's lines has gone down, lots of traffic may be routed to a non-existent line).

We see from this that it is not enough to allow all updates to be processed as soon after a partition as they are received. In addition, we must be able to ensure that if the partition has lasted long enough for sequence number wrap-around to have occurred, then no pre-partition updates are still extant. One way of ensuring that updates do not stay around the system too long is simply to time them out. When the last received update from a given node becomes "too old", the next update from that node should be automatically accepted as the more recent, no matter what sequence number it has. This eliminates the problem of an old update circulating in the network for an unlimited amount of time. In the example above, by the time the partition ended, w would be "too old", so w' would be automatically accepted as more recent when it is received.

The most accurate way to determine the age of an update would be to maintain a globally synchronized clock. Each update packet would carry the time of its creation at its source, as well as a sequence number. Then each node would know exactly how long ago an update was generated, subject to the resolution of the clock and possible inaccuracies of synchronization. Use of a globally synchronized clock has several problems, however. One problem is simply the difficulty of maintaining synchronization. But the most serious problem is that of re-synchronizing after a partition. When a partition forms, there is no way of ensuring that the clocks in the two segments stay synchronized. If, when the partition ends, updates flow between the two segments before re-synchronization is established, the results are unpredictable. So not only must there be a method of re-establishing sync, there must also be some way for the nodes to determine whether or not sync has been re-established, so they know whether or not it is safe to pass on updates. While such methods can no doubt be developed, they would add a significant amount of complexity to the scheme. It is worthwhile therefore to develop a means of determining the age of an update which does not require globally synchronized clocks.

Suppose node A transmits update a which is received at node B. At any given time, the age of update a (from the point of view of B) can be divided into two components - transit time and holding time. Transit time is the time it took the update to travel from A to B. Holding time is the time since it arrived at B. If we may assume that, within a certain amount of time after an update is initially created, its holding time at any given node will be very much larger than its transit time to that node, then we may neglect the transit time, and equate the update's age to its holding time. But the holding time can be computed from a purely local clock. No global synchronization is necessary at all.

In the ARPANET, cross-network transit times are generally on the order of 100 milliseconds. Within a minute, say, after any update is created, its holding time at any node would always dominate its transit time to that node by so much that the transit time could be neglected. There is only one exceptional case. If an update has to be retransmitted many times, it may age significantly in transit. If A has held an update for 59 seconds, retransmitting many times before B finally acknowledges it, we do not want B to have to wait yet another minute before regarding the update as too old. The time the update was held at A must be figured in.

This problem is resolved by having the update packets carry around some indication of their age. When an update is first generated, its age is zero, and a zero is stored in its age field. When an update is received, its age field is stored, and periodically incremented. When a packet is re-transmitted, the current stored value of the age field is placed in the packet. Since we know how often any node can generate updates, and we know how many bits the sequence number is to be stored in, we can compute the minimum time needed for the sequence numbers to wrap around. Once an update has been held for so long that the sequence numbers from its source node may have wrapped around, it is regarded as "too old", and the next update received from that source is considered to be the more recently generated, no matter what its sequence number is. This will only work if the minimum time to wrap around is much greater than the transit time, but that is easy to ensure.

Similarly, if a node fails, it must be held off the network for enough time to allow its last update to become "too old". Once that happens, its first new update will be accepted as the most recent, independent of sequence numbers.

5. THE UPDATING PROTOCOL SUMMARIZED

In this section, we summarize the updating protocol developed in the previous section. Each update packet has a header in which are stored its age, its sequence number, and the identification number of its source node. The sequence number is assigned by the source node at the time the update packet is created, and is one greater than the sequence number of the update packet previously generated by that source node (modulo n, of course, where n is the maximum sequence number). In the case of the first update packet generated by a node, any sequence number may be assigned. An update packet is given an age of zero at the time of its creation. The source node then transmits the update packet to each of its neighbors. The update packet is retransmitted periodically to a given neighbor until that neighbor acknowledges it, or until a new update packet is created which obsoletes the first one.

When a node receives an update packet from one of its neighbors (which may or may not be the original source of the update), the node sends an acknowledgement to the neighbor. The source node identifier and the update sequence number are used to identify the acknowledged packet uniquely. Then the receiving node must check to see whether any update packet from the same source node has been previously received. If not, the age and sequence number of the update are stored. (The stored value of the age will be incremented periodically, until it reaches some maximum value, after which the update will be considered to be "too old".) The update is sent to each neighbor except the one from which it was received. It is retransmitted periodically to a given neighbor until it is either acknowledged by that neighbor, or it becomes "too old", or a more recent update packet from the same source node is received. When an update needs to be retransmitted, it is re-created from tabled information; it is not kept in a packet buffer. Note in particular that when the update is re-created, its age field is copied from the tabled age field. Since the tabled age field is incremented periodically, the age field carried by a retransmitted update packet is not generally the same as the age field carried by the original copy of that update packet.

If a received update packet is not the first from a particular source node, a determination must be made as to whether it was generated more recently than the update previously received from that source node. (Of course, the neighbor which transmitted the packet must be sent an acknowledgement, whatever the outcome of this determination.) If the stored value of the age field (which corresponds to the previous update) is at its maximum value, the previous update is too old, and the current one is considered to be the more recently generated one. If the stored value of the age field is not at its maximum value, the current update's sequence number is compared with the sequence number of the previous update (i.e. with the tabled sequence number) to see which update is the more recently generated. If the current update was not more recently generated than the one previously received (or if it is a duplicate of it), it is simply discarded. Otherwise it is forwarded to all the neighbors except the one from which it was received, as described in the previous paragraph. Its sequence number and age are stored, replacing those of the previous update.

The parameters of this algorithm must be chosen so that it is impossible for the sequence numbers to wrap around in less time than it takes for an update to reach its maximum age. This ensures that the most recently generated update will always be correctly chosen, even in the case of network partition.

When a node fails, it must not be allowed to restart until enough time has elapsed so that any extant updates that originated from that node will have reached maximum age. This ensures that the first update generated by that node after restart will always be considered more recent than any previous updates, regardless of sequence numbers.

6. CONCLUSION

The problem of designing a protocol for transmission of routing updates is an example of a problem in the management of a distributed data base. This sort of problem is different from the problems for which communications protocols have traditionally been designed, and it leads to a protocol which is significantly different from any of the ARPANET's internal protocols. How the issues and solutions discussed here may be applied to the management of distributed data bases in other applications is a question still to be addressed.

7. ACKNOWLEDGEMENTS

Significant contributions to the work described here were made by John McQuillan, Ira Richer, and Paul Santos.

REFERENCES

1. J. M. McQuillan, I. Richer, E. C. Rosen, "The New Routing Algorithm for the ARPANET", in preparation.

The NIC Name Server--A Datagram Based Information Utility

J. R. Pickens, E. J. Feinler, and J. E. Mathis

SRI International

Abstract

In this paper a new method for distributing and updating host name/address information in large computer networks is described. The technique uses datagrams to provide a simple transaction-based query/response service. A provisional service is being provided by the Arpanet Network Information Center (NIC) and is used by mobile packet radio terminals, as well as by several Arpanet DEC-10 hosts. Extensions to the service are suggested that would expand the query functionality to allow more flexible query formats as well as queries for service addresses. Several architectural approaches with potential for expansion into a distributed internet environment are proposed. This technique may be utilized in support of other distributed applications such as user identification and group distribution for computer based mail.

INTRODUCTION

In large computer networks, such as the Arpanet [1], network-wide standard host-addressing information must be maintained and distributed. To fulfill that need, the Arpanet Network Information Center (NIC) at SRI International has maintained, administered, and distributed the host addressing data base to Arpanet hosts since 1972 [2].

The most common method for maintaining an up-to-date copy on each host is to bulk-transfer the entire data base to each host individually. This technique satisfies most host addressing needs on the Arpanet today. However, some hosts maintain neither a complete nor a current copy of the data base because of limited memory capacity and infrequent processing of updates. In addition, with the expansion of the Arpanet into the internet environment [3, 4], a strong need has arisen for new techniques to augment the distribution of name/address information.

One method currently being investigated is the dynamic distribution of host-address information via a transaction-based, inquiry-response process called the Name Server [5, 6]. To support this investigation, the NIC has developed a provisional Name Server that is compatible with a level of service specified in the Defense Advanced Research Projects Agency (DARPA) Internet experiment [5]. The basic Name Server is described in this paper and a set of additional functions that such a service might be expected to support is proposed.

The discussion is structured as follows: Section 1 describes the

NIC Name Server and how it is derived from the NIC data base service. Section 2 describes extensions of the name server which allow a richer syntax and queries for service addresses. Section 3 discusses architectural issues, and presents some preliminary thoughts on how to evolve from the current centralized, hierarchic service to a distributed Name Server service.

THE NIC NAME SERVER

A Name Server service has been installed on SRI-KA, an Arpanet DEC-10. Inquiry-response access is via the Internet Name Server protocol [5], which in turn employs the DARPA Internet Protocol, IP4 [7].

To demonstrate the service a simple interactive facility is provided to format user input into name server requests--e.g. a query of the form !ARPANET!FOO-TENEX returns an address such as "10 2 0 9" (NET=10, HOST=2, LOGICALHOST=0, IMP=9; for details of host address formats see [8]).

User access to the name server has been implemented on several Arpanet DEC-10 TENEX and Packet Radio Network LSI-11 Terminal Interface Unit (TIU) hosts [9, 10]. While the TENEX program serves primarily as a demonstration vehicle, the LSI-11 program provides a valuable augmentation of the TIU's host table. A typical scenario is that when the packet radio TIU is initiated or initialized, it contains only a minimal host table, with the addresses of a few candidate name servers. The user queries the name server with a simple manual query process, and then uses the address obtained to open a TELNET connection to the desired host.

The information to support the name server originates from the NIC's Arpanet host address data base. An optimized version of this data base is presented to the name server upon program initiation as an input file.

NAME SERVER ISSUES

The basic name server provides a simple address-binding service [5]. In response to a datagram query [7, 11], the name server returns either an address, a list of similar names if a unique match is not found, or an error indication. Several useful additional functions can be envisioned for the name server such as service queries and broader access to host-related information.

Similar Names

An important issue to be resolved is that of the interpretation given to the "similar names" response. A standard definition should be given and not be left to implementors' whims.

If the "similar names" response is used primarily to provide helpful information to a human-interface process, then it may be useful to model the behavior of the name server on the behavior of other known processes that present host name information on demand. An example of this is a common implementation of virtual terminal access on the Arpanet, User TELNET [12], in which three different functions occur:

1. Upon termination of host name input (e.g. <CR>), the user is warned only with an audible alarm if the name used is not unique. If the name is unique, the name is completed, and the requested operation is initiated.

2. In response to <ESC>, either the name will be completed if unique or the user will be warned with an audible alarm if the name is not unique.

3. Only in response to "?" will a list of similar names be printed. "Similar names", in this case, means all names that begin with the same character string. The list is alphabetized.

In support of this style of user interface, it may be appropriate to return the "similar names" response only when requested. Two ways to achieve this might be either to set an option bit or to use "?" to force the similar names response.

Query Syntax

A second issue in the provision of name server service is that of query syntax. The basic level of service previously described allows only a few query functions. With more intelligent name servers, complex queries can be supported.

The current Internet name server requires two fields in the query string--a network name field and a host name field. If the network field is non-existent, a local network query is assumed.

Since network independent queries are desirable, an expanded query functionality must be specified. One way this might be done is to add to the notion of "missing field", which means "local", the notion of a special character like "*", which means "all".

The semantic range of queries afforded by adopting this convention is listed below (Note: ~ is used to mean "null". If both network and host fields are null the representation is "~ ~". "N" means "network" and "H" means "host"):

~	~	local net, local host (validity check?)
~	*	local net, all hosts
~	H	local net, named host
*	~	all nets, local host (inverse search)
*	*	all nets, all hosts (probably prohibited)
*	H	all nets, named host
N	~	named net, local host (inverse search)
N	*	named net, all hosts
N	H	named net, named host

By combining the on-demand similar-names function, "all" and "local", and by allowing "*" to be prefixed or appended to the query string as a wild card character, one can query as follows:

~	SRI*?	All hosts named SRI* on local net
*	SRI*?	All hosts named SRI* on all nets
*	*UNIX*?	All hosts named *UNIX* on all nets

Service Queries

It has been suggested that the name server be generalized into a binding function [13, 14]. In this context, allowing service queries is a very useful extension. One application of this service, that exists within the Packet Radio Project at SRI, is the need to find the addresses of Hosts that support the LoaderServer service--a service that allows packet radio TIUs to receive executable programs via downline loading.

Service querying, unlike host names querying, requires a multiple response capability. The querying process would, upon receiving multiple service descriptors, attempt to establish access to each service, one at a time, until successful.

Service descriptors consist of an official name, a list of alias names, and a network-dependent address. In the case of Arpanet Internet services, this address field would consist of the host address(32 bits), port(32 bits), and protocol number(8 bits). For Arpanet NCP services, the address would consist of a host address(24 bits) and a socket(32 bits).

Syntactically, service queries can be derived from host queries by the addition of a service name field, as below:

NET HOST SERVICE

A network-independent service query, for example, can be represented as:

* * SERVICE

Name Server Options

The concept of options has been introduced in the discussion of the "similar names" function. Another group of options may be used to specify the format of the reply. At one extreme is the compact, binary, style such as exists in the basic level of service. At the other extreme is an expanded, textual, style such as is represented by a NIC host table record with official and alias names included. Options can be envisioned that specify:

- Binary versus text format
- Inclusion of each field in the reply
- Inclusion of official name, per field, in the reply
- Inclusion of alias names, per field, in the reply
- Inclusion of other miscellaneous information, such as operating system, machine type, access restrictions, and so on.

Other options can be envisioned that specify the scope of the search, such as "find SERVER hosts only". An alternate form for specifying formats might be to settle on several standard ones, and allow an option to select among them.

Certainly, not all name servers can support all such options, and not all options are equally useful. Thus, the proposed list will be expanded or contracted to fit the actual needs of processes using the name server service.

MORE DATA Protocol

It should be noted that some of the proposed name server extensions have the potential for generating more than a single datagram's worth of reply, since the current DARPA Internet Protocol limits the size which all networks must support to 576 octets [15]. In spite of this, the size of such replies need not require a full-blown stream protocol. Several alternatives exist:

1. Disallow options that imply large replies.
2. Truncate the packet for large replies.
3. Ignore the recommended maximum datagram size.
4. Utilize an alternate base protocol for such requests.
5. Develop a MORE DATA protocol.

If alternative 1 is chosen, the potential for overflow exists, even with the basic level of service. Alternative 2 implies unpredictable behavior to the user of the name server service. Alternative 3 reduces the availability of the service. Alternative 4 is certainly possible, but may be overkill.

Alternative 5 appears to be a reasonable solution and could be implemented very simply. The name server could return, as part of the reply, a code of the following form:

```
+------+---------+
| MORE | ID_NEXT |
+------+---------+
```

where ID_NEXT is a name-server-chosen quantity that allows the name server to find the next block of reply, the next time it is queried. This quantity may be an internal pointer, a block number, or whatever the name server chooses. Follow-on queries may be implemented by recomputing the entire original query and discarding output until the ID_NEXT block is reached, or by efficiently storing the entire reply in a cache, fragmented into blocks (with appropriate decay algorithms).

Dynamic Updates

In the previous discussion, the host name data base was assumed to have been a static or slowly changing entity with an administrative and manual update authority. This model reflects most of the network needs in the Arpanet and Internet communities. However, dynamic automated updating of the host table may be needed in the future, especially in mobile environments such as the Packet Radio Network.

In a closed user group community (such as a local network of mutually trusting hosts), dynamic updating becomes simply a technical question concerning packet formats. In wider communities, a mechanism to authenticate the change request must be developed; however, the authentication issue is outside the scope of this paper.

ARCHITECTURE

The Name Server concept is invaluable for allowing hosts with incomplete knowledge of the network address space to obtain full access to network services. Whether for reasons of insufficient kernel space or of dynamically changing environments, the need for the service is little questioned. However, significant design issues revolve around the methods for providing the service and for administering and updating the data base.

In the current NIC Name Server, the service is centralized, and is supported by a data base administered by a single authority. In the long range, other architectures are possible that present more flexible ways to distribute host information within and between networks and administrative entities. These present opportunities for dynamic, automated, approaches to the maintenance and sharing of data--particularly host name data.

From an evolutionary point of view, initial Name Server services will likely exist as a centralized service, possibly with one large

Name Server that has knowledge of multiple networks. From this beginning, an expansion in two orthogonal directions can be envisioned.

- In the direction of internal distribution, the name server can be partitioned into multiple, cooperating processes on separate hosts. The data base can be replicated exactly or managed as a distributed data base.
- In the direction of administrative distribution, multiple autonomous name servers may exist that exchange data in an appropriate fashion, on a per network or other administrative basis.

For hosts with small host tables, caching might be used, whereby local, temporary copies would be maintained of subsets of the addressing data base. Such copies may be obtained either by remembering previous queries made of name servers, or by receiving automatic distributions of data from name servers. For mobile hosts, in which even the home network is unknown, it would be possible to maintain a very sparse table with the addresses of only a few name servers.

For hosts lacking even the knowledge of name server addresses, a very primitive name server function can be provided by all network hosts that would allow real name servers to be located; e.g., a query of the form "* * RealNameServer" addressed to an arbitrarily selected host could elicit the address of a real Name Server.

Finally, the possibility exists for multiple name servers to communicate dynamically in attempting to resolve queries. If, for example, a name server on the Arpanet receives a query for a host on the Packet Radio Network, then the Arpanet name server can conceivably query the Packet Radio Network Name Server to resolve the reply.

FUTURE WORK

The techniques discussed in this paper for providing dynamic access to and maintenance of host address information are generally applicable to other information-providing services. Two possibilities currently under investigation at SRI include user identification servers [16] and time servers, which offer people/address and time-stamp information, respectively, as datagram driven information utilities.

Further work is needed to refine the implementation of the name server and its using query processes. Two items in particular are direct system calls into the NIC data base management system for general access to host information and process-level query interfaces for using processes. The design issues for efficient implementation are complex and involve some trade-offs. The most obvious trade-off is volume of network traffic versus "freshness" of information. The local host table handler can either send a message to the name server for every address request, or it can use some type of local cache to remember frequently requested names. SRI is exploring both the process-level interface for the LSI-11 TIU and the problems of local host table management in small machines operating in a dynamic environment.

Information services such as the Name Server are integral components of distributed systems and applications. However, the effective utilization of such services is a relatively unstudied and unexplored area. One area in which SRI plans to study their impact on distributed architectures is in computer based mail applications. Information utilities in this environment can range from the support of simple mailbox address queries to complex address list management needed for inter-organizational and group resolution.

CONCLUSION

A provisional Name Server service, based on the NIC host address data base was described in this paper. In addition, a collection of design ideas for an internet Name Server service has been presented.

Work is continuing on the refinement of the NIC name server service. New requirements and opportunities for functional distribution are being investigated. Many questions have been raised in exploring an expansion of the existing service. Such an expansion is expected to result in more useful support of internet (and intranet) capability.

ACKNOWLEDGMENTS

This work was supported by the Defense Communications Agency, Contract DCA200-C-641, and the Defense Advanced Research Projects Agency, Contract MDA903-78-C-0126. Special thanks go to Ms. Holly Nelson for providing the programming support for the provisional service and to members of the DARPA Internet Working Group, especially Dr. Jon Postel, for specification of a detailed internet Name Server protocol for the basic level of service.

REFERENCES

1. L. G. Roberts and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," in Proceedings of SJCC, pp. 543-549 (AFIP, 1970).
2. M. D. Kudlick and E. J. Feinler, Host Names On-line, RFC 608, Stanford Research Institute, Menlo Park, California (January 1974).
3. V. G. Cerf and R. E. Kahn, "A Protocol for Packet Network Interconnection," IEEE Transactions on Communication Technology, Vol. COM-22, pp. 637-641 (1974).
4. V. G. Cerf and P. T. Kirstein, "Issues in Packet-Network Interconnection," Proc. IEEE, Vol. 66, No. 11, pp. 1386-1408 (November 1978).
5. J. Postel, Internet Name Server, IEN 89, Information Sciences Institute, Univ. of Southern Calif., Marina Del Rey, California, May 1979.
6. J. R. Pickens, E. J. Feinler and J. E. Mathis, An Experimental Network Information Center Name Server (NICNAME), IEN 103, SRI

International, Menlo Park, California (May 1979).

7. J. Postel, Internet Protocol, IEN 81, Information Sciences Institute, Univ. of Southern Calif., Marina Del Rey, California (February 1979).

8. J. Postel, Address Mappings, IEN 91, Information Sciences Institute, Univ. of Southern Calif., Marina Del Rey, California (May 1979).

9. R. E. Kahn, "The Organization of Computer Resources into a Packet Radio Network," IEEE Transactions on Communications, Vol. COM-25, No. 1, pp. 169-178 (January 1977).

10. R. E. Kahn, S. A. Gronemeyer, J. Burchfiel, and R. C. Kunzelman, "Advances in Packet Radio Technology," Proc. IEEE, Vol. 66, No. 11, pp. 1468-1496 (November 1978).

11. J. Postel, User Datagram Protocol, IEN 88, Information Sciences Institute, Univ. of Southern Calif., Marina Del Rey, California (May 1979).

12. E. Leavitt, TENEX USER'S GUIDE, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts.

13. R. Bressler, A Proposed Experiment With a Message Switching Protocol (section on Information Operator), pp. 26-31, RFC 333, Bolt Beranek and Newman Inc, Cambridge, Mass. (May 15, 1972).

14. Y. Dalal, Internet Meeting, January 24,25 1979, (Group Discussion).

15. J. Postel, Internet Meeting Notes - 25,26 January 1979, pp. 12, IEN 76, Information Sciences Institute, Univ. of Southern Calif., Marina Del Rey, California (February 1979).

16. E. J. Feinler, "The Identification Data Base in a Networking Environment," in NTC '77 Conference Record, pp. 21:3 (IEEE, 1977).

A PROTOCOL FOR BUFFER SPACE NEGOTIATION

Dan Nessett

University of California, Lawrence Livermore Laboratory
Livermore, California 94550

Abstract

There are at least two ways to manage the buffer memory of a communications node. One technique views the buffer as a single resource that is to be reserved and released as a unit for a particular communication transaction. A more common approach treats the node's buffer space as a collection of resources (e.g., bytes, words, packet slots) capable of being allocated among multiple concurrent conversations. To achieve buffer space multiplexing, some sort of negotiation for buffer space must take place between source and sink nodes before a transaction can commence.

Results are presented which indicate that, for an application involving a CSMA broadcast network, buffer space multiplexing offers better performance than buffer reservation. To achieve this improvement, a simple protocol is presented that features flow-control information traveling both from source to sink as well as from sink to source. It is argued that this bidirectionality allows the sink to more effectively allocate buffer space among its active communication paths.

INTRODUCTION

Imagine for a moment a bright young engineer who has been assigned the task of designing a computer communications network. Being conscientious as well as bright, this engineer first looks at the current networking literature for some backround into computer communications techniques. Almost immediately two philosophies of computer network implementation present themselves. The first supports a resource-reservation viewpoint whereby a network user requests the use of various network facilities, is granted use of those facilities (generally at some time later than when they were requested), and finally, when they are no longer needed, releases those resources for use by others. Opposed to this philosophy is the resource-multiplexing school. It supports the viewpoint that user requests for network resources should be managed in such a way that the user is not logically aware that those resources are being used by anyone else. For example, those supporting the resource-reservation approach would argue for connection-oriented or virtual circuit communications primatives, which give the user the responsibility for bringing up and tearing down logical connections. On the other hand, those supporting the resource-multiplexing philosophy would generally argue for message-oriented communications primatives, which relieve the user from such responsibilities [1, 2, 3, 4, 5].

If our engineer is not only bright and conscientious, but also something of an historian, this rivalry will call to mind earlier confrontations. All during the late 1960's a debate raged about whether or not demand paging or prepaging (sometimes referred to as swapping) provided the better performance in paged virtual memory systems. Even earlier, there were debates about which CPU scheduling algorithm should be used (e.g., First-In/First-Out, Round-Robin, Prioritized Round-Robin, Multilevel Priority) and about whether or not deadlocks could be prevented if a job could demand its resources dynamically.

Unfortunately for this young engineer, the necessary analysis of resource-reservation versus resource-multiplexing in computer networks does not yet exist. This paper concentrates on one problem where either resource-reservation or resource-multiplexing may be used, that of message buffer management. It attempts to answer the question of whether message buffer space should be reserved and released as a single resource or should be multiplexed over a number of concurrent conversations.

The research reported here was motivated by an earlier simulation study [6] that examined a networking facility in which meassage buffer reservation was employed. Results of that study suggested that buffer multiplexing could be utilized to enhance the facility's performance. Hence, a project was begun to compare the old simulation results with results to be obtained from a new simulation of the facility, modified to employ message buffer multiplexing. It soon became clear that there were a number of decisions to be made concerning how the message buffer was to be multiplexed. In particular, the handling of flow control and the management of buffer space emerged as key issues. The first of these became a topic of research in itself, and lead to the formulation of a flow control protocol in which requests for remote buffer space are negotiated. The second issue remains an area for future investigation. This paper, therefore, not only reports the results obtained by comparing buffer reservation with buffer multiplexing, but also introduces a protocol for buffer space negotiation which, it will be argued, possesses certain advantages over more traditional flow control protocols.

MESSAGE BUFFER MANAGEMENT

Consider the following situation. A set of computing systems are connected together by a data switch. This data switch is constructed in such a way that data is not lost while in transit between two connected computer systems nor is it possible that data will be duplicated in transit and the duplicates delivered as if they were independent. Furthermore, data is guaranteed to arrive at a destination in the same order in which it is sent. Such a facility will be called a *lossless, sequencing, nonduplicating data switch* (Fig. 1). The switch is implemented in such a way that a computer system connected to it is capable of communicating with any other connected system. How these logical connections are managed (i.e., by connection-oriented or message-oriented techniques) is immaterial to the following discussion.

Each system possesses buffer space (which will be called the message buffer) where data is staged before going out onto the switch and after arriving from the switch. The question arises how message buffer space should be managed. The remainder of this paper attempts to answer that question.

MESSAGE BUFFER RESERVATION

One way to manage the message buffer is to reserve its use on a conversation by conversation basis. That is, when system α wishes to deliver data to system β, it first requests the use of the two message buffers. Once the message buffers have been obtained (i.e., α

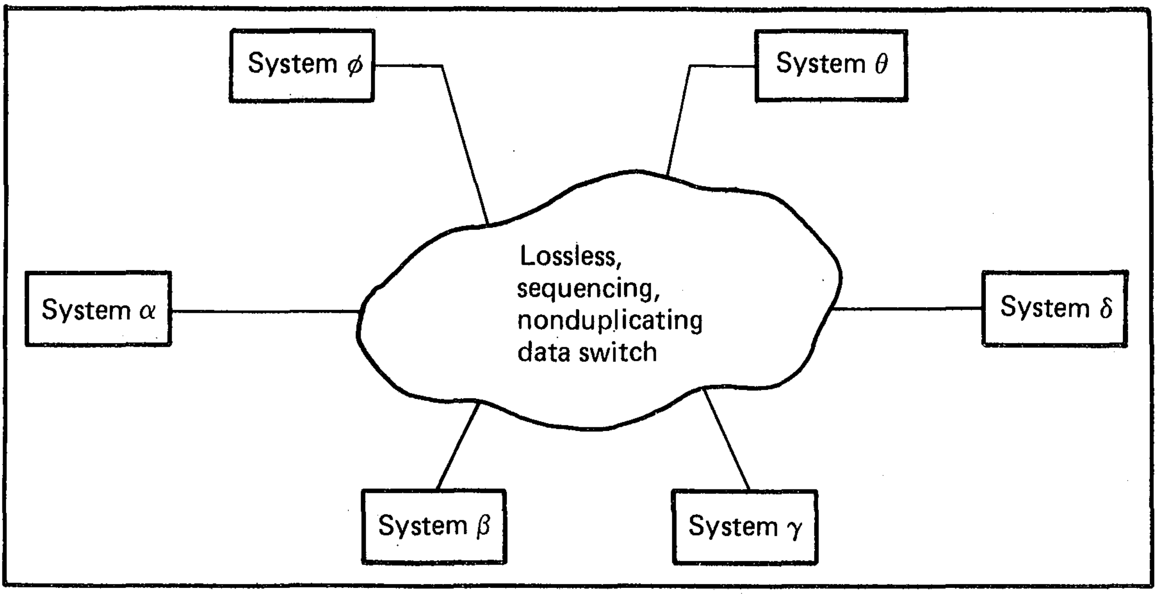


Fig. 1. Six computing systems connected by a lossless, sequencing, nonduplicating data switch.

successfully *reserves* them), no other system will be granted their use until α releases them. Such a management scheme views the message buffer as a single resource incapable of being shared by many systems concurrently.

As stated, in order for data to pass from α to β, the message buffer of both α and β must be reserved. To acheive this, a reservation protocol must be provided so that α can request β to reserve its message buffer. The details of this reservation protocol need not concern us here. The major pertinent characteristic of this protocol is its ability to (eventually) effect the reservation of two message buffers--one on the *source* machine and one on the *sink* machine.

Such a management scheme has one obvious defect. During the time when a system has reserved the two message buffers, those message buffers cannot be used for any other purpose *even though they are not being totally utilized.* For example, suppose both α and δ wish to transfer data to β. Also suppose α is capable of delivering data to the data switch 10 times faster than δ (e.g., its connection runs at 50 Mbps, while the connection of δ runs at 5 Mbps). Furthermore, suppose β can accept data from the switch as fast as α can supply it (i.e., it also has a 50 Mbps connection). Under these circumstances, whenever δ and β are conversing, their effective connection runs at the lower data rate (e.g., 5 Mbps). If β can dispose of the data flowing through its message buffer at a faster rate than δ can supply it, the message buffer of β will not be in use 100% of the time. Even though the message buffer of β is not in use, it is reseved and not available for some other conversation (e.g., for an α to β transfer). This results in the underutilization of β (which is forced for part of the time to operate at computing system δ's communication rates).

MESSAGE BUFFER MULTIPLEXING: WINDOWED FLOW CONTROL

One way to remedy the underutilization effect of the message buffer reservation technique is to allow multiple conversations to use a single message buffer concurrently. That is, each time a system wishes to transfer data from itself to a destination system, some *portion* of both participating message buffers are allocated for that data transfer. This allows another system to also obtain space in the message buffer of either system (Fig. 2).

As with the buffer reservation technique, some protocol must be provided so that space in both message buffers can be allocated. A common approach is to use *windowed flow control* [7]. In this scheme, the source system obtains from the sink system a *window* specifying how much data the sink is likely to accept. Each time data is sent from the source to the sink, the sink returns an *ack* value informing the source how much of that data was accepted. In addition a new window value is by a particular source. Second, it does not know how big this parcel will be when it does arrive (although it does know that it should be smaller than the window that it sent to that source). Confronted with this lack of information, the sink must effectively guess what these two items might be for each source and then allocate its buffer space accordingly. Needless to say, if the sink is not good at guessing or if the sources are in general unpredictable, this buffer allocation technique can result in a great deal of inefficiency.

For example, suppose the sink guesses that source α will be soon sending a large parcel of data and that source β will probably not be sending any data soon. Furthermore, suppose the sink guesses that when β does send some data it will be a small parcel. According to its guess, the sink allocates a large amount of space to α and a small amount of space to β (Fig. 4a). It informs both α and β of their respective window sizes and waits for data to arrive. Now suppose the sink guessed badly, i.e., source α has no data to send, while source β has a great deal of data it wishes to deliver to the sink. In this case, source β will begin sending its data segmented in the small parcels dictated by the small window size it has received from the sink (Fig. 4b). This forces source β to use small parcel sizes and detrimentally affects data switch performance in two ways. First of all, the lossless, sequencing, nonduplicating data switch is very likely a combination of switching hardware and communications software. Each parcel of data transmitted through it incurs a fixed amount of bandwidth overhead due to various low-level header and "packaging" information that accompanies the parcel. This implies that the effective data switch bandwidth decreases as the number of parcels into which a data unit is segmented increases. Second, extra delays are incurred, since the source must wait for a windowing "reply" to be received from the sink before the next parcel can be sent. Both of these factors motivate both source and sink to keep the window size for that conversation as large as possible.

In our example, the sink will begin to receive many small parcels from source β. After a certain amount of time, the sink will probably suspect that its guess of expected traffic was quite bad. In reaction to that suspicion, the sink can do one of two things. It can simply continue giving source β a small window size and live with the resulting inefficiency, or it can renege on its window commitment to source α and allocate some of that space to the source β data transfer (Fig. 4c). If, after the sink reneges, source α sends a large parcel of data to the sink, some or all of that parcel will not be accepted by the sink since buffer space is no longer available for it (Fig. 4d). Thus, reneging on a window can itself cause inefficiencies to occur. Needless to say, more complicated thrashing-like situations can occur where the sink is forced to continually renege on its source windows.

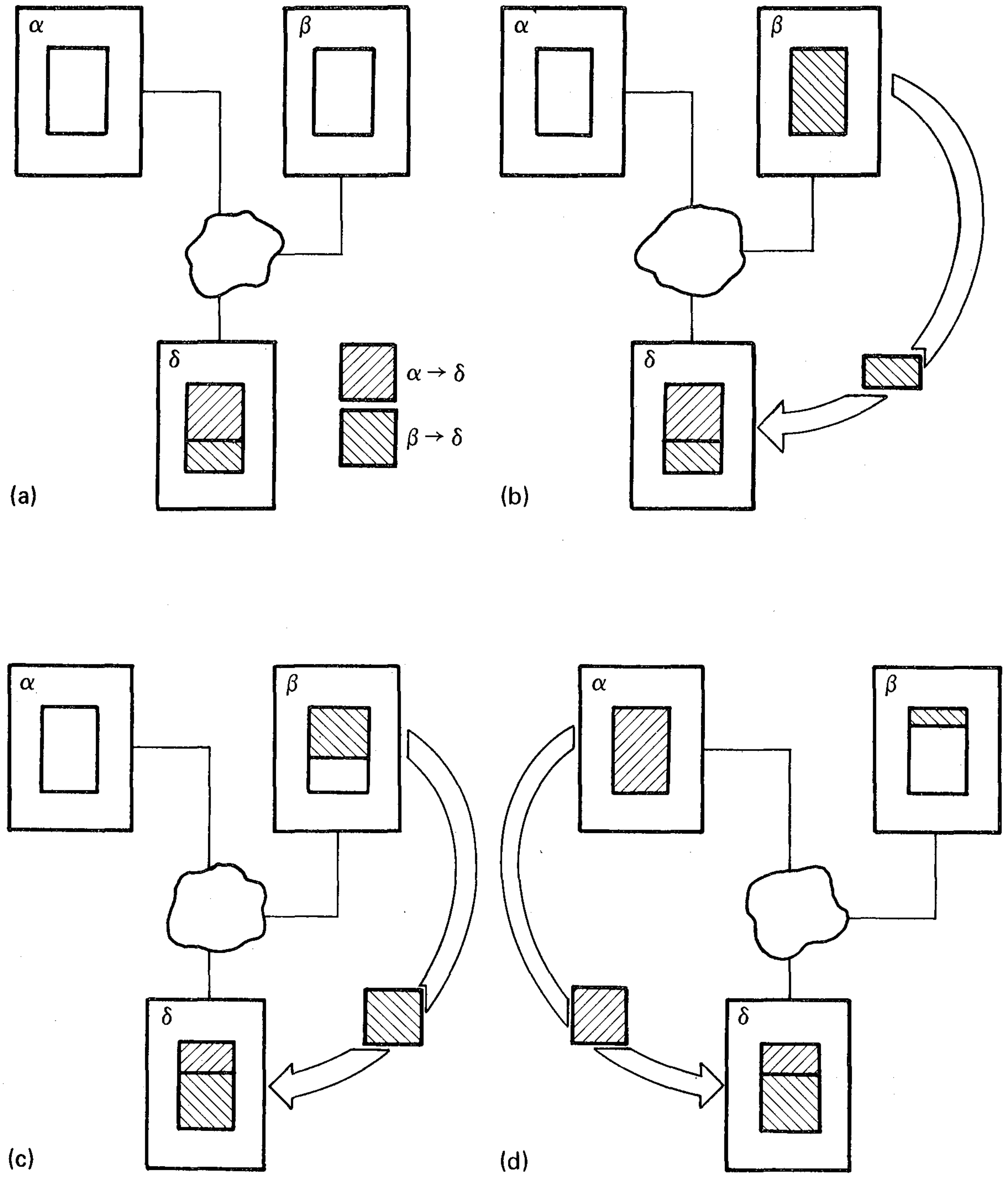


Fig. 4. Potential inefficiencies using windowed flow control. (a) Sink δ initially assigns a large buffer space and window size to α, and a small space and window size to β. (b) Source β begins sending its data in correspondingly small parcels. If β has much data to send, inefficiencies result. (c) Sink δ reneges on α's large window. It assigns more buffer space to β, which can now send larger parcels. (d) If α tries to send a parcel as large as its original window, buffer space is no longer available. Thus, new inefficiencies arise.

MESSAGE BUFFER MULTIPLEXING: GIMME-GIVEYA

The disadvantage of windowed flow control discussed above can be summarized in the following way. Control information flows from the sink to the source in the form of a "damping" factor called the window. However, no control information flows from the source to the sink. If all systems were continuously attempting to transmit data to every other system, this deficiency would cause no problems. When the traffic over the data switch is bursty, however, such *forward* control information is desirable. The GIMME-GIVEYA (rhymes with Jimmy-Olivia) protocol provides this forward control path. It is first described in its "pure" form [9] and then presented in a more efficient "piggyback" form, which distinctly resembles windowed flow control.

The GIMME-GIVEYA protocol is based on the concept of *buffer space negotiation*. The initiator of a data transfer, the source, requests space in the message buffer of the sink. Sometime after the request, the sink grants space in its buffer, notifying the source of the space size. The space allocated will be less than or equal to the space requested. The source then sends data to the sink in an amount equal to the space granted. When the source has more data to send (which will be immediately, if the space granted by the sink was less than the space requested by the source), the above sequence is repeated.

The protocol used to implement this negotiation is equipped with three buffer-management messages, which correspond to the three stages of the negotiation. When the source wishes to inform the sink of a buffer space request, it issues a GIMME message, which contains the size of the space requested. The sink responds with a GIVEYA message, which contains the size of the space granted. The source then sends a DATA message to complete the buffer space transaction. An example of a GIMME-GIVEYA message exchange is shown in Fig. 5.

Pure GIMME-GIVEYA suffers from an overabundance of messages transmitted for each parcel of data delivered. While windowed flow control delivers a data parcel for every two messages transmitted (the ACK/WINDOW message and the DATA message), pure GIMME-GIVEYA requires three (a GIMME, a GIVEYA and a DATA message). To reduce this traffic, the GIMME message for the *next* buffer space request can be *piggybacked* onto the DATA message of the current transaction. Doing this reduces the messages per data parcel to two (Fig. 6). Notice that piggyback GIMME-GIVEYA includes an ACK message (upon which the GIVEYA is piggybacked), which allows the sink to accept less than the total amount of data encapsulated in the DATA message. While this addition allows the sink to renege on promised message buffer space (something that should be avoided if at all possible), it in no way encourages that practice. In fact, the forward control path was added so that reneging on buffer space promises could be avoided. There are situations, however, when reneging may still be necessary (e.g., a source system crash occuring after space has been allocated by the sink, tying up buffer space wastefully). As can be seen by comparing Figs. 3 and 6,

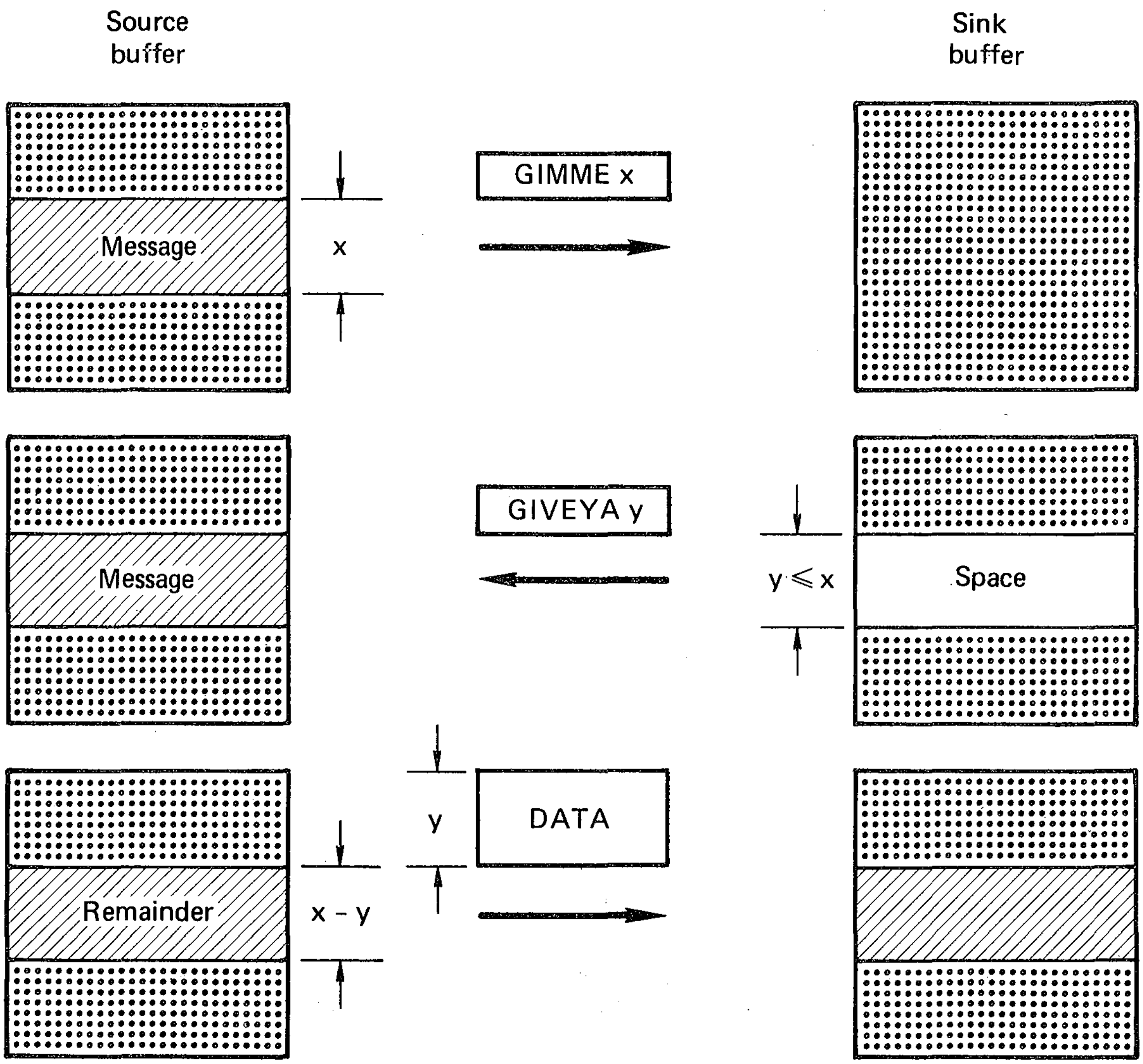


Fig. 5. Typical message exchange for the "pure" GIMME-GIVEYA protocol.

piggyback GIMME-GIVEYA is very similar to windowed flow control. The main difference between the two is the accompanying GIMME message piggybacked on each DATA message. This GIMME information provides the forward control path which has been mentioned previously. This similarity remains even if data messages are "pipelined." In that case, a GIMME message would be piggybacked onto a DATA message only if it is necessary to request more sink buffer space.

While piggyback GIMME-GIVEYA is more efficient than pure GIMME-GIVEYA, it requires a seperate beginning and ending sequence to respectively gain an initial space allocation and inform the sink that no more message buffer space is needed. Figure 7a illustrates the sequence of messages which are exchanged to initially gain sink message buffer space. The source first sends an unattached GIMME message, to which the sink responds with an unattached GIVEYA message. The source then sends a DATA message piggybacking its next GIMME message.

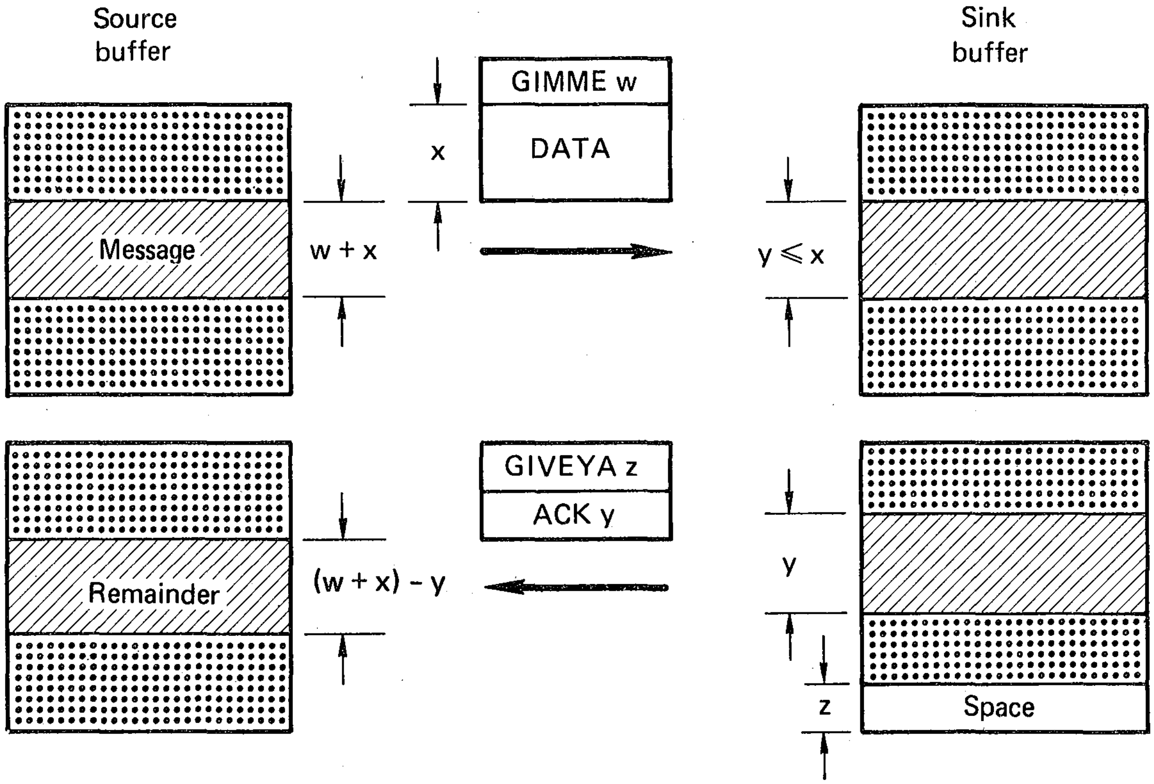


Fig. 6. Typical message exchange for the piggybacked GIMME-GIVEYA protocol.

Piggyback GIMME-GIVEYA can now proceed until the source no longer has any data to send to the sink. When this condition obtains (Fig. 7b), the source piggybacks a GIMME 0 message on its last DATA message (informing the sink that it desires no more message buffer space). The sink responds by acknowledging receipt of all or part of the data parcel and acknowledges the GIMME 0 message by piggybacking a GIVEYA 0 message upon the ACK. At this point, the source must initiate a new, unattached GIMME-GIVEYA sequence (Fig. 7a) when it wishes to transmit more data to the sink.

APPLICATIONS OF THE GIMME-GIVEYA PROTOCOL

It has been argued that whenever an entity (e.g., a system, a process) wishes to multiplex its buffer space among several conversations and whenever those conversations involve bursty traffic, then some sort of buffer space negotiation is needed to achieve efficient utilization of that buffer space. The GIMME-GIVEYA protocol provides a framework for negotiation. However, this framework, which includes forward and reverse control paths, must be used in conjunction with a suitable buffer allocation strategy in order to actually *obtain* efficient buffer space utilization. If the allocation strategy makes no use of the forward or reverse control information, the advantages of the GIMME-GIVEYA protocol become vacuous. Another way of thinking about this is to see the GIMME-GIVEYA protocol as offering a *mechanism* to

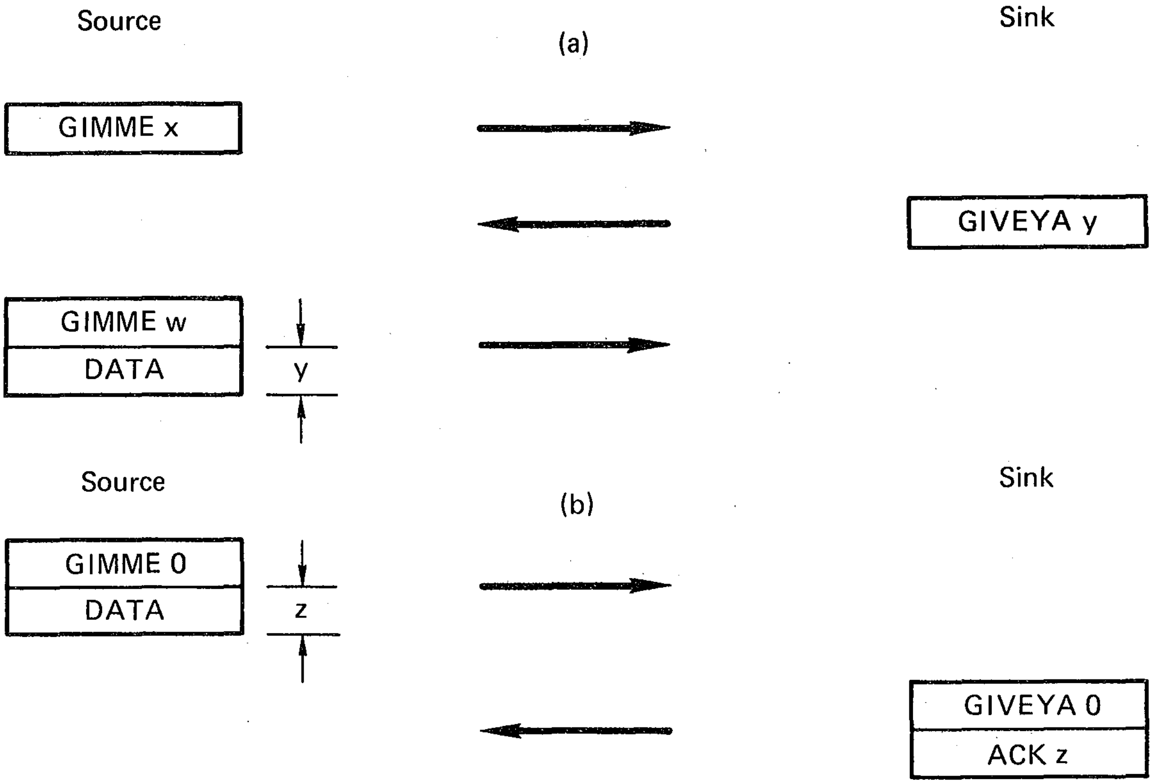


Fig. 7. Separate beginning and ending sequences of piggybacked GIMME-GIVEYA. (a) The initial exchange. (b) Informing the sink that no more buffer space is needed.

achieve efficient buffer space utilization, while the buffer allocation strategy implements a *policy* which may or may not allow such efficient utilization to be achieved.

For example, if a process converses with only one other process or if a process statically allocates its buffer space for each communication path, there is no advantage for it to use the GIMME-GIVEYA protocol. If, however, a process is attempting to dynamically multiplex its buffer space among multiple conversations, or if a system is using a common buffer area for the purposes of end-to-end communications, GIMME-GIVEYA allied with a suitable buffer allocation strategy can be of real benefit. The possibility of using the GIMME-GIVEYA mechanism in end-to-end protocols and to aid in the control of congestion in store-and-forward networks is under active consideration.

Application of the GIMME-GIVEYA protocol should be beneficial whenever the two conditions stated above (i.e., a desire to multiplex buffer space, and bursty traffic) obtain. Different applications may require different buffer allocation strategies, but the mechanism of buffer space negotiation should remain constant. For this reason, investigation of suitable buffer allocation strategies will also be a major research effort in the future.

COMPARING BUFFER RESERVATION AND BUFFER MULTIPLEXING: A SIMULATION STUDY

A simulation study was undertaken to compare message buffer reservation with message buffer multiplexing. The study focused on a commercially available procuct [10] which implements a data switch via a CSMA (Carrier Sense Multiple Access) broadcast bus. Every host system is connected to the bus by one or more communication *adapters*, which handle all bus contention details. The underlying capacity of the broadcast bus is 50 Mbps. An adapter is equipped with buffer storage where message data is staged before being transmitted. Message transmission proceeds by the host indicating to its adapter that it wishes to transmit data, the adapter reserving both its own and the destination adapter's buffer memory, the data being transferred from source to destination host via the two interfacing adapters, and then the source adapter releasing both its own and the destination adapter's buffer. A more detailed exposition of the protocols and algorithms used by adapters to effect buffer reservation and data movement can be found in Donnelley and Yeh [6]. The simulation results from their work on the buffer-reservation approach are used here for comparing buffer reservation with buffer multiplexing.

The simulation of Donnelley and Yeh was modified to implement a buffer-multiplexing approach utilizing the pure GIMME-GIVEYA buffer-management protocol. The bus contention logic as well as the report and graphics generation sections were kept intact. The possibility of developing a mathematical model of either the buffer-reservation or GIMME-GIVEYA protocol was rejected due to the complexity of the analysis. The only way a mathematically tractable analysis could be carried out would be to apply unrealistic and distorting simplifications to the problem. This did not seem advisable.

Throughput Comparison

The first configuration that was studied consists of three hosts, each interfaced to the broadcast bus through its own adapter. Each adapter was equipped with 8K (K = 1024) bytes of buffer memory. The first buffer-allocation scheme studied allocates buffer space in exponentially decreasing sizes. That is, one buffer is one-half as large as the total buffer memory, another is one-fourth as large, etc. This strategy will be called *exponential buffer allocation*. For a three-host configuration this scheme partitions buffer memory into four blocks: one 4K in length, another 2K, and the final two 1K. Since each adapter can only have four "paths" active at one time (two paths carrying traffic *from* the adapter to the other two, and two paths carrying traffic *to* the adapter from the other two), this buffer-allocation strategy prevents anomalous conditions from occuring, such as store-and-forward lockup [11]. The assignment of these blocks to each path is dynamic, according to the following rules. A path is allocated the smallest block large enough to satisfy the GIMME request. If no block larger than or equal to the GIMME request is available, the largest one available is allocated.

It should be emphasized that partitioning memory into blocks whose lengths are a power of 2 can result in very large memory sizes for only a moderate number of adapters. That is, since adding an adapter to the data switch implies either decreasing the size of the smallest buffer space or increasing the minimum buffer size to be twice as large as the previous minimum, adapter buffer size will eventually grow exponentially with the number of adapters added to the data switch. In a practical situation, with a more realistic number of data switch connections, such a constraint would be intolerable and so some other buffer-allocation scheme would be necessary. Results given below show how performance can be significantly affected by the buffer-allocation strategy employed, especially at high loads.

When comparing the simulation results of GIMME-GIVEYA and the buffer-reservation protocol, one must consider the accuracy of the underlying model. The simulation for the buffer-reservation protocol was based on an existing implementation of that protocol, and so its results should be realistic. Message length, time to process messages, etc., were carefully chosen to accurately reflect the operation of the existing system. Since no implementation of GIMME-GIVEYA currently exists, it was not possible to achieve this level of accuracy in its simulation. Even though the parameters which should affect the protocol's performance were carefully chosen to be realistic, the results presented below should be viewed with some caution.

Figure 8 presents the major result of this paper. The horizontal axis represents the load placed upon the data switch,* and the vertical axis shows the throughput obtained. Load is measured in terms of the average waiting time between host requests for data transfer. The arrival process waits an exponentially distributed time, requests the transfer of data (which may require multiple DATA messages), and, after sucessful transmission, cycles back to wait another exponentially distributed time. For a detailed explanation of the arrival process, see Donnelley and Yeh [6]. As can be seen from Fig. 8a, buffer multiplexing offers a 30% to 50% increase in throughput performance over buffer reservation, for a message size of 8K bytes. The performance results presented for the buffer-reservation approach are the best results obtained by Donnelley and Yeh using a deadlock prevention scheme suggested by Shoshani. At every load, buffer multiplexing offers improved performance, but the most dramatic improvement occurs at high loads. For a message size of 12 bytes (Fig. 8b), buffer multiplexing achieves 10 times the throughput (5 Mbps versus 0.5 Mbps) of buffer reservation. The more dramatic improvement for small message sizes is probably the result of a 12 byte message fitting into any of the (4K, 2K, or 1K) message buffers. This decreases the number of DATA messages

*For details on measuring load, see [7].

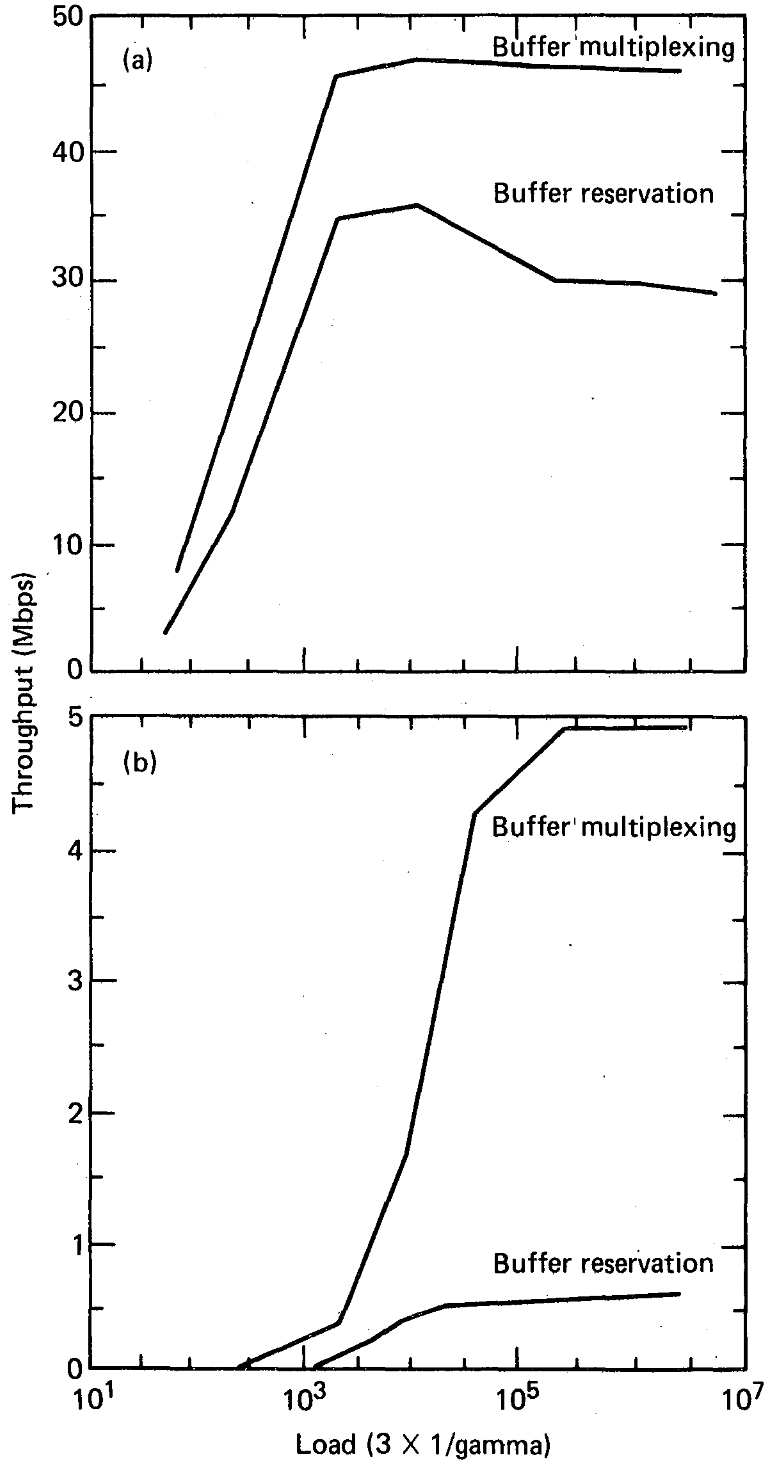


Fig. 8. Throughput as a function of load placed upon a data switch. Buffer multiplexing is compared with buffer reservation (a) for 8K-byte messages and (b) for 12-byte messages. For a detailed explanation of how load is measured, see [7].

transmitted per host message and thereby reduces the data switch overhead, as discussed in the section on windowed flow control. These results demonstrate the advantages of buffer multiplexing over buffer reservation, at least for the CSMA broadcast faciltiy studied.

It is fairly difficult to provide a detailed analysis of all the factors contributing to the improved performance of buffer multiplexing over buffer reservation. However, the improvement is probably caused in a large part by the adverse effect on throughput of the reservation-release activity centeral to any buffer reservation technique, but missing from the buffer multiplexing scheme. It is not that the reservation class messages (e.g., "reserve your buffer", "my buffer is already reserved", "release the reservation on your buffer") themselves consume a large amount of bandwidth, but the time spent by each system managing the reservation protocol, and in particular the time spent taking remedial action if a desired buffer is already reserved (e.g., reporting failure to the host, setting up a new transfer), is time that could be spent transmitting data. Since the reservation scheme also needs some sort of flow control protocol to ensure that the data delivery rate matches the receiver consumption rate, the extra reserve-release mechanisms of the buffer reservation approach tend to decrease its transport efficiency.

Figure 9 shows the performance obtained by buffer multiplexing for various message sizes. As can be seen, the maximum throughput for

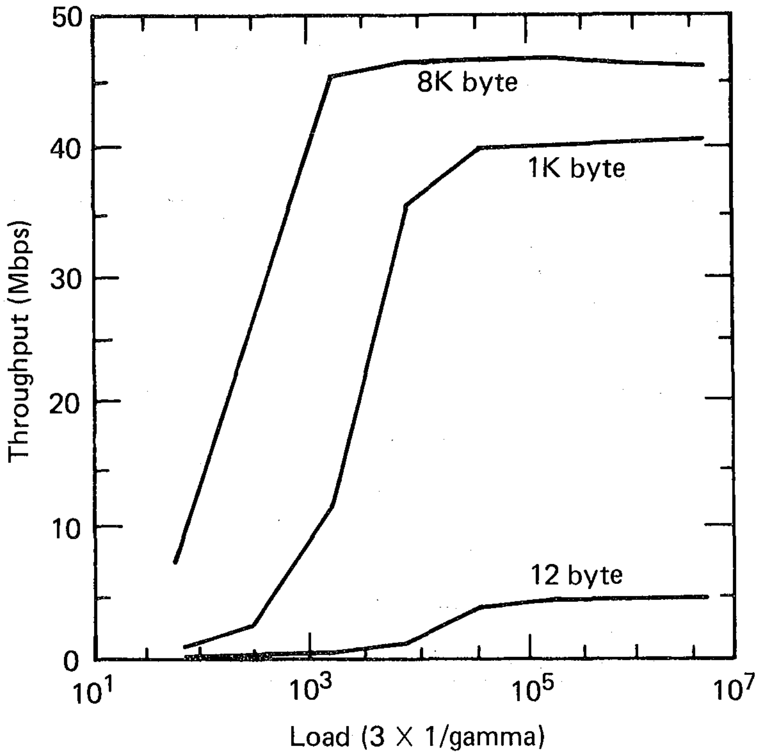


Fig. 9. Throughput as a function of load placed upon a data switch when buffer multiplexing is used. Results are shown for three different message lengths.

message sizes of 12, 1K, and 8K bytes was respectively 5 Mbps, 40 Mbps, and 47 Mbps. This family of curves illustrates the stability of buffer multiplexing as the load increases. The performance of buffer reservation (Fig. 8a), on the other hand, suffers a slight dip as load increases from medium to high.

Slow Path Interference

To further illustrate the advantages of buffer multiplexing over buffer reservation, the following experiment was performed. Consider a configuration consisting of three hosts connected to the broadcast bus by three adapters (Fig. 10). Furthermore, suppose hosts α and β are connected to their adapters by 50 Mbps channels, while host δ is connected to its adapter by a 5 Mbps channel. Two paths will be allowed to be active, the $\alpha \rightarrow \beta$ path and the $\delta \rightarrow \beta$ path. The experiment is designed to determine if the slow $\delta \rightarrow \beta$ path would interfere with the faster $\alpha \rightarrow \beta$ path. Figure 11 shows the results of the experiment when buffer reservation is used. At high loads the faster path's throughput is "pulled down" by the slower path. This occurs because β's buffer is being reserved by δ and then held while an 8K-byte message is transferred from host δ to host β. Since the buffer-reservation scheme employs double buffering in adapters, only 4K bytes are transferred from adapter to adapter at one time. This causes the buffer memory of adapter β to be reserved while the second 4K bytes of data is being transferred from host δ to adapter δ (at a rate of 5 Mbps). When the load increases, the resulting underutilization of adapter β's buffer causes the throughput of path $\alpha \rightarrow \beta$ to degrade. Note that this effect is not actually caused by the double buffering scheme employed. Even if

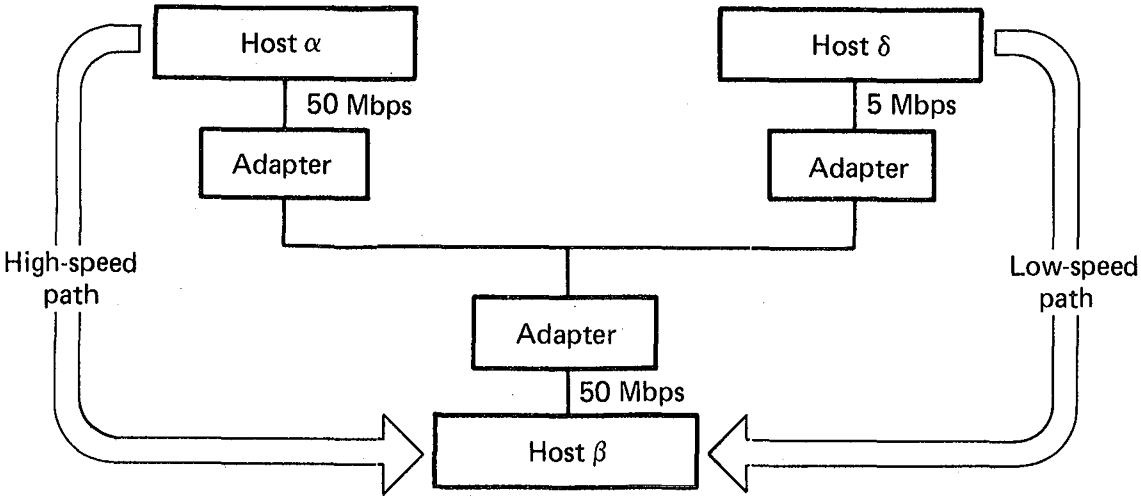


Fig. 10. Configuration for investigating whether the use of a slow path ($\delta \rightarrow \beta$) between hosts will interfere with the use of a faster path ($\alpha \rightarrow \beta$).

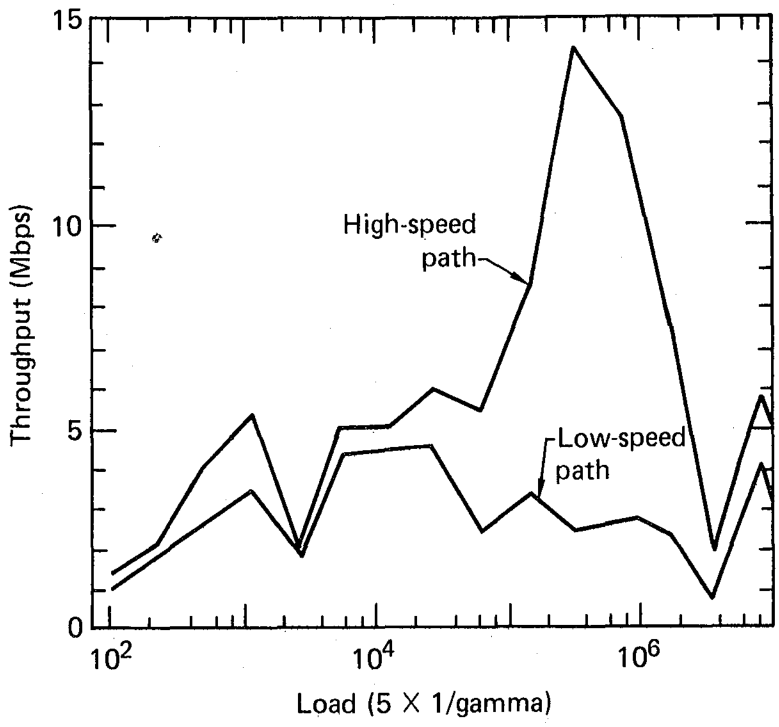


Fig. 11. Throughput as a function of data switch load when buffer reservation is used for the configuration in Fig. 10.

the complete 8K bytes of adapter buffer space were available for such a transfer, the same "pull down" effect would occur for host messages greater than 8K bytes.

The path throughput curves for buffer multiplexing are presented in Fig. 12. Notice that path $\alpha \rightarrow \beta$ throughput is not degraded by path $\delta \rightarrow \beta$ activity. Throughput for path $\alpha \rightarrow \beta$ increases until it reaches a plateau of around 20 Mbps. This is still less than the underlying channel capacity of 50 Mbps because of the overhead of the GIMME-GIVEYA messages, the computation time in the adapters for buffer management, bus contention, and the fact that adapter α must share buffer space with adapter β (i.e., it does not obtain all 8K bytes of adapter δ's buffer). The jump in $\alpha \rightarrow \beta$ throughput and slight dip in the $\delta \rightarrow \beta$ throughput at the highest load is caused by a buffer capture effect of the allocation strategy. That is, when a message transfer is completed, the buffer space allocated to it is released back into the free buffer pool. When two paths are simultaneously active, one has a 4K-byte buffer allocated to it, while the other has a 2K-byte buffer allocated. At low loads the $\alpha \rightarrow \beta$ and $\delta \rightarrow \beta$ paths get approximately equal use of the 4K-byte buffer space because message interarrival time is fairly long. At the highest load, however, message interarrival time has almost gone to 0, which results in the 4K-byte buffer being reallocated to the $\alpha \rightarrow \beta$ path almost as soon as it was released by it. This behavior causes "buffer capture" of the 4K-byte buffer by the $\alpha \rightarrow \beta$ path.

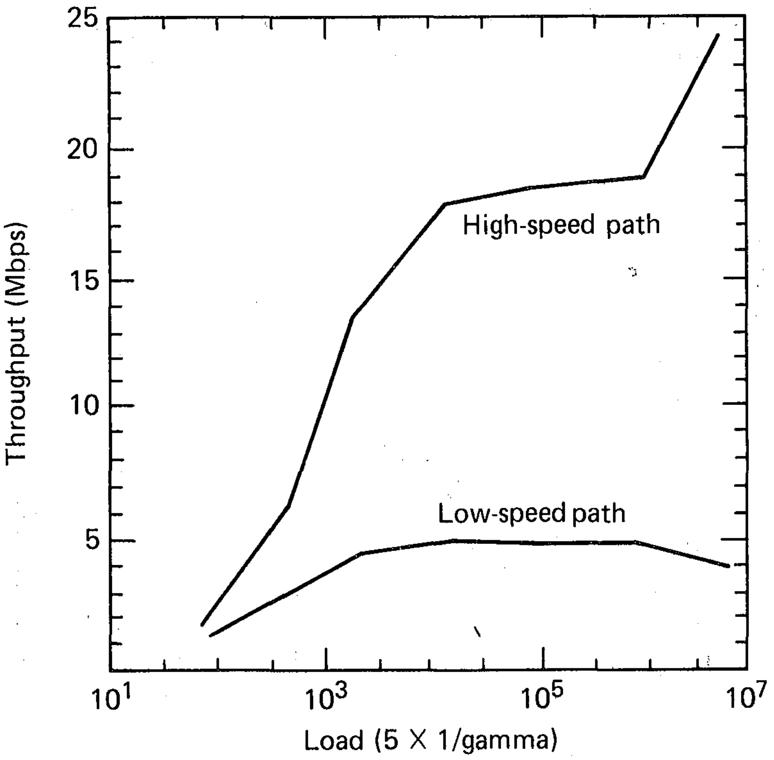


Fig. 12. Throughput as a function of data switch load when buffer multiplexing is used for the configuration in Fig. 10.

Effect of Allocation Scheme

To investigate the sensitivity of buffer-multiplexing performance to the buffer allocation scheme employed, two changes were made in the simulation. First of all, the number of adapters connected to the bus was increased from three to five. This was done so that the results being obtained did not depend on some idiosyncrasy of a three-host/adapter configuration. Second, a different buffer allocation scheme was implemented so that its performance could be compared to the exponential buffer allocation scheme. The second allocation technique assigns constant buffer size to each possible path into and out of an adapter. Since there are four other adapters to which or from which and adapter can be transferring data, eight constant size buffers of 8K bytes each were assigned. This resulted in an adapter buffer size of 64K bytes.

Figure 13 presents the results of the buffer allocation scheme comparison. Notice that at high loads the throughput of the exponential allocation scheme drops slightly. This decline results from the scheme's poor utilization of additional adapter buffer space. A 64K-byte buffer is divided into one 32K-byte buffer space, one 16K-byte buffer space, etc. The 32K-byte buffer space is only being utilized by

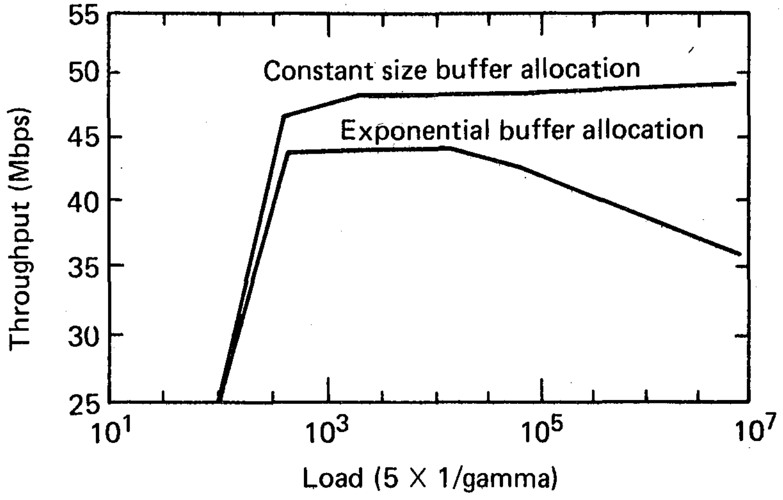


Fig. 13. Throughput as a function of data switch load for two different buffer allocation schemes: constant size and exponential.

25%, since the message size being simulated was 8K-bytes. On the other hand, each smaller buffer (say a 512-byte buffer) is not big enough to hold a complete host message, so that transmission overhead was increased. Constant size buffers, on the other hand, are very suitable to the traffic pattern simulated. Throughput for this scheme remained constant as load increased from medium to high.

The results comparing buffer allocation schemes only demonstrate the sensitivity of data switch performance to the chosen buffer allocation scheme. No claim is being made that constant size buffer allocation is superior to exponential buffer allocation. It just so happens that for constant message sizes, constant size buffer allocation provides superior characteristics (a not too suprising result). It is the author's opinion that neither of these buffer allocation schemes show very robust performance for all types of traffic characteristics.

SUMMARY

GIMME-GIVEYA, a protocol for buffer space negotiation, has been introduced. Arguements have been given as to why it is preferable to the more traditional windowed flow control mechanism. The GIMME-GIVEYA protocol has been used to demonstrate the superiority of buffer multiplexing over buffer reservation, at least for the CSMA broadcast bus system studied. There seems to be no reason to doubt that these results have more general application.

ACKNOWLEDGEMENTS

The work reported in this paper was conducted as part of the Local Network Research Project at the University of California's Lawrence Livermore Laboratory. Four other members and former members of that project, Jed Donnelley, Bruce Watson, Richard Watson, and Jeffrey Yeh, made many valuable comments, suggestions, and criticisms, both during the conduct of the research and during the writing of this paper. Bruce Watson deserves special mention for producing the results presented in Fig. 11. I would like to express my appreciation to them for their help. 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.

REFERENCES

1. Walden, D. C., "A system for Interprocess Communication in a Resource Sharing Network," CACM, 15 (April 1972), 221-230.

2. Fletcher, J. G. and Watson, R. W., "Mechanisms for a Reliable Timer-Based Protocol," Computer Networks, 2 (Sept./Oct. 1978), 271-290.

3. Watson, R. W. and Fletcher, J. G., "A Protocol Structure for Network Operating System Services," to appear.

4. Watson, R. W., "Protocol Design Issues for Local Computer Networks: Illustrated for a Backend Storage Network," to appear.

5. Pouzin, L., "Virtual Circuits vs. Datagrams--Technical and Political Problems," Proc. of AFIPS NCC (1976), pp. 483-494.

6. Donnelley, J. E. and Yeh, J. W., "Interaction Between Protocol Levels in a Prioritized CSMA Broadcast Network." Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks (August 1978), pp. 123-143, reprinted in Computer Networks, 3 (Feburary 1979), 9-23.

7. Pouzin, L., "Flow Control in Data Networks--Methods and Tools," Proc. Third Int. Conf. on Computer Communications (Toronto, August 1976), pp. 467-474.

8. Geissler, A., et. al., "Free Buffer Allocation--An Investigation by Simulation," Computer Networks, 2 (July 1978), 191-208.

9. Nessett, D. M., "Protocols for Buffer Space Allocation in CSMA Broadcast Networks with Intelligent Interfaces," Proc. Third Conference on Local Networking (University of Minnesota, October 1978).

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

11. Kahn, R. E. and Crowther, W. R., "Flow Control in a Resource-Sharing Computer Network," IEEE Trans. on Communications COM-20, 3 (1972), 539-546.

IMPLEMENTATION OF DISTRIBUTED SYSTEMS — II

LABELED SLOT MULTIPLEXING: A TECHNIQUE FOR A HIGH SPEED, FIBER OPTIC BASED, LOOP NETWORK

Sheldon Blauman

TRW Communications Group
Torrance, California

Abstract

A high speed, fiber optic based, ring structured, local computer network is described. The TTL based prototype system operates at a line rate of 20 mbps. The interface logic has been specified to allow implementation in the faster ECL components, which could operate at line rates to 200 mbps. The loop interface mates the high speed fiber optic channel to its relatively slow computer elements through a technique called Labeled Slot Multiplexing (LSM). The byte multiplexed LSM loop is non-hierarchial and asynchronous, requiring no host computer or line supervisor. Agents on the loop contend for space non-destructively, placing byte packets on the line only when space is available. Time slots on the loop are not pre-determined, and packets may be inserted whenever space exists. Address recognition is implemented at the line level, and provides for both functional and physical addresses. Up to 63 devices may share the loop, with a potential for 192 logical functions. The loop interface is modular, separated into two logical/physical packages, a line interface and a processor interface. The line interface contains only the logic which must function at line rates, with the slower, byte oriented operations performed in the less expensive processor interface.

INTRODUCTION

Fiber optics, promising high bandwidths and low error rates with excellent noise immunity (1,2,3), offers an almost ideal communications medium for computer to computer links, and makes possible new approaches to local network design. The primary challenge facing a designer is the efficient utilization of the medium, which has a much greater bandwidth potential than most local networks require, or can use. TRW's Communications Group's research organization has been experimenting with the application of fiber optics to local computer networks.

The first step in applying fiber optics is deciding on a network topology. Two considerations help to make that decision: the high speed of fiber optic links, and their greater interface cost over purely electronic communication lines. A loop, or ring structure can utilize a high speed communications medium, and requires a minimum number of interfaces while providing a path between all its agents (4).

A basic problem in applying fiber optics to a loop network is that the medium has much greater bandwidth potential than the computing elements it is to couple. If the optical channel is run at the relatively slow speed of its processors there is little justification, other than noise immunity, for the more expensive fiber optics over conventional communication channels. If the channel is run at a bandwidth greater than that of its agents, there must be a means of gearing the agents to the speed of the loop, which may be in the hundreds of megabits/sec. range (5), requiring very high speed ECL logic. This effectively eliminates most existing loop protocol designs (6,7,8,9), which all appear to require interface units too expensive to implement in high speed components. The need for costly ECL logic at higher bandwidths necessitates a new approach to loop protocols; one which minimizes logic at the line level, and in particular does not require large, high speed buffers. A network of that type has been developed at the University of Cambridge, in England, which proves a simple system may make effective use of a high speed medium (10). The following design, independently arrived at, is similar in concept but very different in approach.

Before evolving the network design, we will outline our requirements for a local distributed computer system. It is desireable that there be no supervisory agent in the network, eliminating that potential single point of failure. Address recognition at the line interface level is essential to high speed operation, and automatic recognition of logical addresses as well as a device's physical address would eliminate the need for routing tables or schemes. A global address capability is also a desired feature, allowing a single message to be sent to multiple logical destinations. Because we are dealing with a reliable and a high speed communications medium, segmented messages are not necessarily required for efficient line utilization, few errors being expected, and when occuring, retransmission being very rapid. This prompts the requirement for a continuous message transmission capability. One additional feature, which becomes feasible with high speed communication channels, is the ability for an agent to accept its own messages, allowing a computer to issue a request for a function and then accepting the request itself if no other agents are capable of processing that function.

Incorporating all the above mentioned features into a local network both aids in system operation and simplifies the software requirements. The software simplicity is of particular importance if micro-computers are to be the agents in the network. We will now proceed with the generation of a design which will achieve all these desired capabilities.

LINE PROTOCOL

The requirement for mimimum line level logic, and in particular short line buffers, preordains some form of line multiplexing. In returning to basic communications engineering, a possible protocol for a high speed loop of this design is Time Division Multiplexing (TDM), perhaps with byte length slots. TDM requires a marker, which each agent counts from to find its assigned slot, or logical channel. Two problems are that a supervisor is required to generate the marker and clock the loop, and that each slot requires a reserved location, even when unused. The requirement for a supervisor is the most disturbing of the problems, since it presents a single point of failure for the network, which otherwise could be non-hierarchical. Of course any of the agents on the loop is also a potential single point of failure, since the loop is broken at each agent, fiber optics not lending themselves to large numbers of passive taps. The individual agent problem can be solved through a normally closed optical switch, which is only open when the associated agent is alive and well; switches of this type have been built at TRW. However the supervisor cannot be bypassed, and must be functional for the loop to operate, making failsoft operation difficult.

An advantage of fiber optics is that generically it provides an almost open ended bandwidth capability, currently limited to rates approaching 200 mbps/km (5,11), but with potential for much higher rates. It seems reasonable to utilize some of this abundance of resource as overhead, if doing so can solve some of the problems inherent in TDM. The primary need for the supervisor is for the generation of the marker which each agent uses to locate its slot. An alternative method is to label each slot with the identity, or channel number, of its owner. An agent listening on a channel can then watch for its channel I.D., rather than count to a slot from a marker. A transmitting agent places its labeled data on the line as the data becomes available and whenever space exists on the line. As in TDM, the packet, including both data and label, must be removed or replaced with a new packet by the originator after one loop cycle. In addition to eliminating the marker requirement, and hence the supervisor, this technique should allow better line utilization by not reserving channel space and not fixing the transmission rate to that of the slowest agent. The disadvantage of this scheme is it requires a high overhead for the label relative to the data, particularly if the data is only 8 bits in length. However the bandwidth of the fiber optic link can always be increased to compensate, just by improving the components. This allows a designer to freely determine the correct bandwidth for a system, knowing that it can be achieved by properly specifying the electooptic interface. The real bandwidth limitation is the electonics, and not the line itself.

LINE INTERFACE

The labeling technique described above defines a method for identifying data on the line, but does not specify how a message may be directed to a destination. Since the network is a loop, and a packet passes completely around before being removed by the originator, all agents will see every packet. This allows messages to be addressed functionally, without knowlege of the the physical location of a requested resource. Of course physical addressing is also required, since it is necessary to transmit the response specifically to the requesting agent. Transmitting a message on a byte by byte basis requires potential receivers to identify the destination at the very start of the message, so they might determine if they should capture subsequent packets from that source. This formulates the first rule of our message protocol: messages must begin with their destination, which may be either a logical or a physical address. To simplify the protocol, the destination field is limited to a single data field, in this case one byte, allowing 255 destination addresses. Since some of these must be physical addresses it is necessary to define the number of agents permitted on the loop. For our network purposes, 63 is a reasonable number of nodes, allowing 192 logical functions in the system. This fixes the source I.D. portion of the packet label at 6 bits. To distinquish the initial destination, or start of message (STX) packet, from a subsequent data packet, a flag is required, adding one bit to the label. It is also desireable to acknowlege packets which have been accepted and copied, which adds one more control bit, for a total label size of 8 bits. In addition to the label and data fields, start/stop bits are required for synchronization, giving a total packet size of 18 bits (longer start/stop "fields" may be required at higher line speeds).

With the packet format described, it is possible to define the line interface logic. To be effective, this message protocol must be processable at the line level in high speed logic. To facilitate this, each packet is fully buffered at the line interface, introducing a one packet delay, during which time all line operations are performed. The most difficult feature to implement is the ability to recognize an acceptable, logically addressed request at line speeds. This problem is solved through the use of a high speed 1 x 256 bit RAM chip, with each bit location corresponding to a destination address. Each function processable by an agent is flagged by a bit set in the high speed RAM by the agent's software. There is also one bit set to indicate the physical address of the agent, since some messages, particularly responses, must be sent to a specific physical location. In addition, a physical address register is required, which is compared to the label field to determine which packets have been transmitted by that agent and must be removed. The line interface is designed to look for acceptable packets independently of identifying its own transmissions, allowing it to accept its own messages. This eliminates the need for the agent's operating system to support two separate transaction routing schemes, simplifying the overhead software, an important consideration in memory limited microprocessors.

The logic design of the line interface has been kept simple in order to minimize hardware and to allow operation at line speeds. The optical receiver is always in communication with its upline agent to maintain syncronization. This eliminates the requirement for a long startup header. When data is seen on the line, a clock is derived from the signal and passed with the data to the receiver logic, which collects the incoming packet in a serial to parallel shift register. After the full packet is received, the information is passed to a receiver operating latch and to an output shift register. The receiver immediately checks to determine if the packet was originated at its node. If it was, the output is inhibited to prevent retransmission of the packet, removing it from the loop. If not, retransmission is initiated while the receiver performs subsequent checks on the captured information. The transmit clock is derived from a local oscillator and is not bit synchronized with the input derived clock.

If the receiver is not in the process of receiving a message, that is, not currently listening for a specific "channel", it checks for the presence of the new message flag. If set, the data byte contains the destination for the message. The receiver decodes the destination as the address to its RAM, and checks the bit at that address to determine if it is set. If the bit is set, and the packet has not been acknowleged by some upline agent, that node is an eligible receiver. The receiver sets the acknowledgment bit in the packet in the output latch, sets an internal busy flag for itself, and traps the address of the originator, or channel I.D., in a register to be compared with the label on all subsequent packets. One exception to the above sequence is in the checking of the acknowledgment flag. A class of functions is specified as global, and when a destination address falls within that range, the presence of the ack bit is ignored in the decision to accept the message. This allows all eligible receivers to accept a broadcast message concurrently.

Once a receiver has accepted a packet, a virtual link has been established between it and the originator. The receiver then watches all subsequent packets passing on the loop for those with a matching label field. As the packets from the prescribed source are found, they are acknowledged and the data byte captured and relayed to the processor interface. The sequence continues until the receiver sees a packet from that source with the message control flag set, as in the first packet of the message. This second occurance of the control flag in conjunction with a reserved code indicates the end of transmission, or ETX. This event causes the receiver to notify the processor interface, which sets a status and interrupts the processor. However, the line interface remains busy until signaled by the interrupt processor it may proceed to search for a new message. The ETX packet looks identical to a start of message (STX) packet, except that destination address zero is reserved as an ETX code, which no other receiver will confuse as a valid starting address.

PROCESSOR INTERFACE

The loop interface is not designed for direct computer connection; it requires a separate interface (ie: controller) between it and the processor. This minimizes the amount of hardware which must be redesigned to mate other processors to the network. The processor interface is byte oriented, operating at the DMA rate of the processor, and not at line speed. This allows a design utilizing much lower speed logic than that in the line interface. The processor interface is full duplex, with separate input and output sections, each with its own DMA channel. When eligible for a new message, the input section has a preallocated empty buffer available for immediate access. When the line interface locates a message and begins to trap and relay the data, the processor interface DMA's each byte into the assigned buffer. As bytes are received and stored, an address register is incremented, and a preset buffer byte count decremented. When the byte count goes to zero, the processor is interrupted and a new buffer requested. The sequence continues until an ETX occurs. The ETX occasions an interrupt for message completion processing, however it does not automatically reset the line and processor interfaces as not busy, both of which wait until made available for new messages by the interrupt software.

The input portion of the interrupt code has three basic functions: supplying new buffers, assigning messages to their processing routines, and determining resource availability. The latter function, though software oriented, is pertinent to the description of the loop architecture. Upon completion of an input sequence, the interrupt handler chains the newly received message to its destination program, and if the message is a functional request, determines if there is capacity for more transactions of that function type. If the resources are available, the interrupt processor instructs the interface hardware to return to non-busy mode, watching for new eligible messages. If the resources for that function have been saturated, the interrupt handler clears the the bit in the line interface memory associated with that function before making the receiver available for new messages.

The output section of the interrupt processor operates in a similar, though complementary fashion. When called by a program to transmit a message, the system presets processor interface registers with the address and length of the first buffer, then sets a control register to indicate a new message is available. If the message is fully contained in the one buffer, the control word also indicates that the last byte should cause the ETX bit to be set. The writing of the control byte starts up the output logic of the processor interface hardware.

The processor interface hardware formats the first packet of the message with the flag bit set, passes it to the line interface, then waits for the first packet to return before issuing the subsequent bytes of the message. If the first packet returns un-acked, the output logic issues an interrupt to notify the software a receiver was not found. The software will decide on subsequent action, which will generally be to reissue the request some number of times, assuming potential receivers are momentarily busy. After the first packet has been accepted, the bytes will be DMA'd from the processor memory at a programmably set rate calculated to not overrun the receiver's DMA capablity. If any of the returned packets are not acked, the processor interface will set an error flag and issue an interrupt to force a retry of the entire message.

The logic in the output section of the line interface is simple. If the output latch, or register, is clear, and no data is in the input register, the output section gates the packet into its output latch, knowing it has sufficient time to transmit the entire packet, even if the receiver immediately gets incoming data. This provides a nondestructive contention scheme, which allows packets to be sent as frequently as possible, yet not interfere with other messages occupying the loop at the same time.

ERROR HANDLING

Possible transmit errors are non-acked returned packets or an incorrect number of returned packets. The hardware detects "ACK" errors and keeps a returned packet count for software detection of lost packets. Possible receive errors are incompleted transmissions (underflow) and inability to process the next packet in time (overflow). The hardware sets an overflow flag when that condition occurs, and the software monitors receive cycles to determine if an underflow has occured. However there is no way to detect garbaged or orphaned packets continually recirculating on the loop without the addition of a line monitor, which is prohibited under our original design criteria. This is an intentional deficiency, since few line errors are expected in a fiber optic network; however, the condition must be provided for in some fashion, because it can happen.

A simple and effective, though somewhat brusque technique, is to periodically halt the loop for a long enough time to completely clear it. Unfortunately this will also abort any valid messages on the line at the time; however, both transmitters and receivers will detect the flush generated errors and reset themselves. This loop shutdown capability is under software control, and when invoked inhibits the transmitter from initiating or retransmitting any packets.

Because a low error rate is an inherent and justifying feature of fiber optics, loop flushes may be performed periodically on a relatively infrequent basis of minutes or even hours, with the option for an operator initiated flush. Properly implemented, this capability should cause little if any system disruption; however, it is necessary to guard against gradual loop degradation from accumulating garbage.

Though somewhat basic, the technique appears an excellent alternative to a loop monitor or supervisor, particularly in the case of a low error rate medium.

CONCLUSION

This completes the description of the fiber optic loop interface. For ease of development the first prototype version is implemented in lower speed TTL logic, functioning at 20 mbps, which should be adequate for evaluation purposes. However, the logic is designed for bandwidths to 200 mbps, utilizing high speed ECL components at the line interface level. The prototype processors are Motorola 6800 based, and the processor interface exists as a one card controller on the bus. The line interface exists as a separate package, linked to the processor interface through a cable. In theory, only the processor interface board would have to be redesigned to allow adaption of other processors to the loop.

Because this paper systematically evolved the design of the loop interface from a hardware viewpoint, little attention has been paid to justifying the approach from a software viewpoint. However, all hardware features were specified with an operating system structure in mind, and a good percentage of the code had been generated before the final design stage of the hardware was complete, resulting in some design changes to arrive at a fully integrated hardware/software package.

SUMMARY

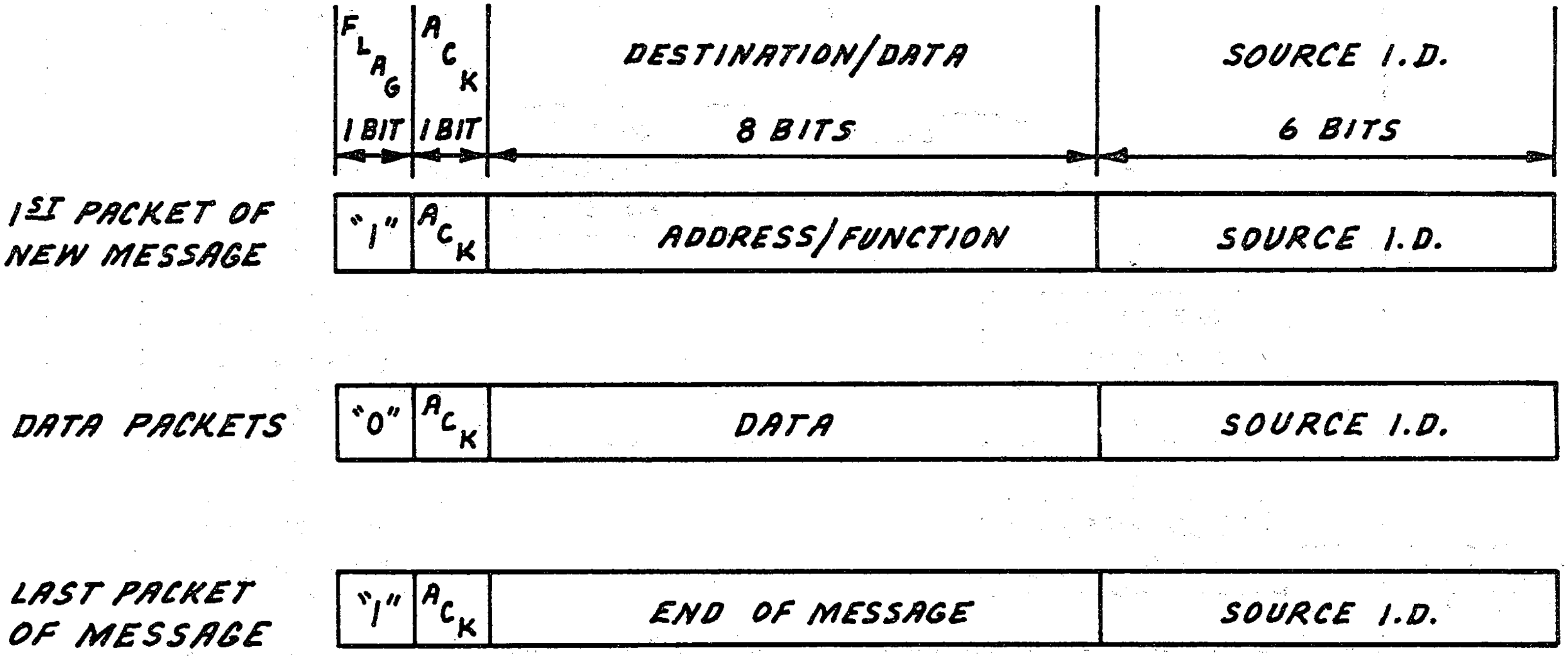
A distributed local computer network has been described which utilizes fiber optics as its communication channel. The line and message protocols both depend upon, and take advantage of, the high bandwidths and low error rates provided by the fiber optic medium. A labeled slot multiplexing scheme is used, each slot containing a data byte and label. The label on each packet identifies its originator, and includes an STX/ETX flag and an acknowledgment bit. The originator of a packet watches for its return, and removes it from the loop, the acknowledgment bit indicating its acceptance by an eligible receiver. The first byte of a message, indicated by the presence of the STX bit, must contain the destination, which may be logical or physical. The 8 bit byte allows 255 addressess, 63 of which are reserved as physical unit numbers. Destination zero is reserved as an ETX flag, and in conjunction with the presence of the ETX bit in the label, indicates the end of a transmission to an active receiver. The line interface logic includes a 1x256 RAM chip which provides the table of acceptable destinations at any processor. This allows the dynamic detection and acknowledgment of messages processable by that receiver.

The above structure is intended to both provide a high speed, non-hierarchical loop and to minimize operating system software requirements, an important feature for micro-processor based networks. Messages proceed around the loop with only a one packet delay at each agent, and with no requirment for active software intervention. The functional destination capability eliminates the need for routing tables, and a subset of functions designated as global allows a single message to be directed to multiple destinations. These features allow concurrency and redundancy to be provided with relatively little software effort. The message protocol requirements are simple, only specifying that each message begins with an 8 bit destination code and ends with a zero byte. Though byte oriented, the protocol is not ASCII structured nor bit patterned, allowing any data format to be used. The prototype system depends on a software generated and decoded CRC for error detection. However, this capability could be implemented in hardware in the processor interface logic.

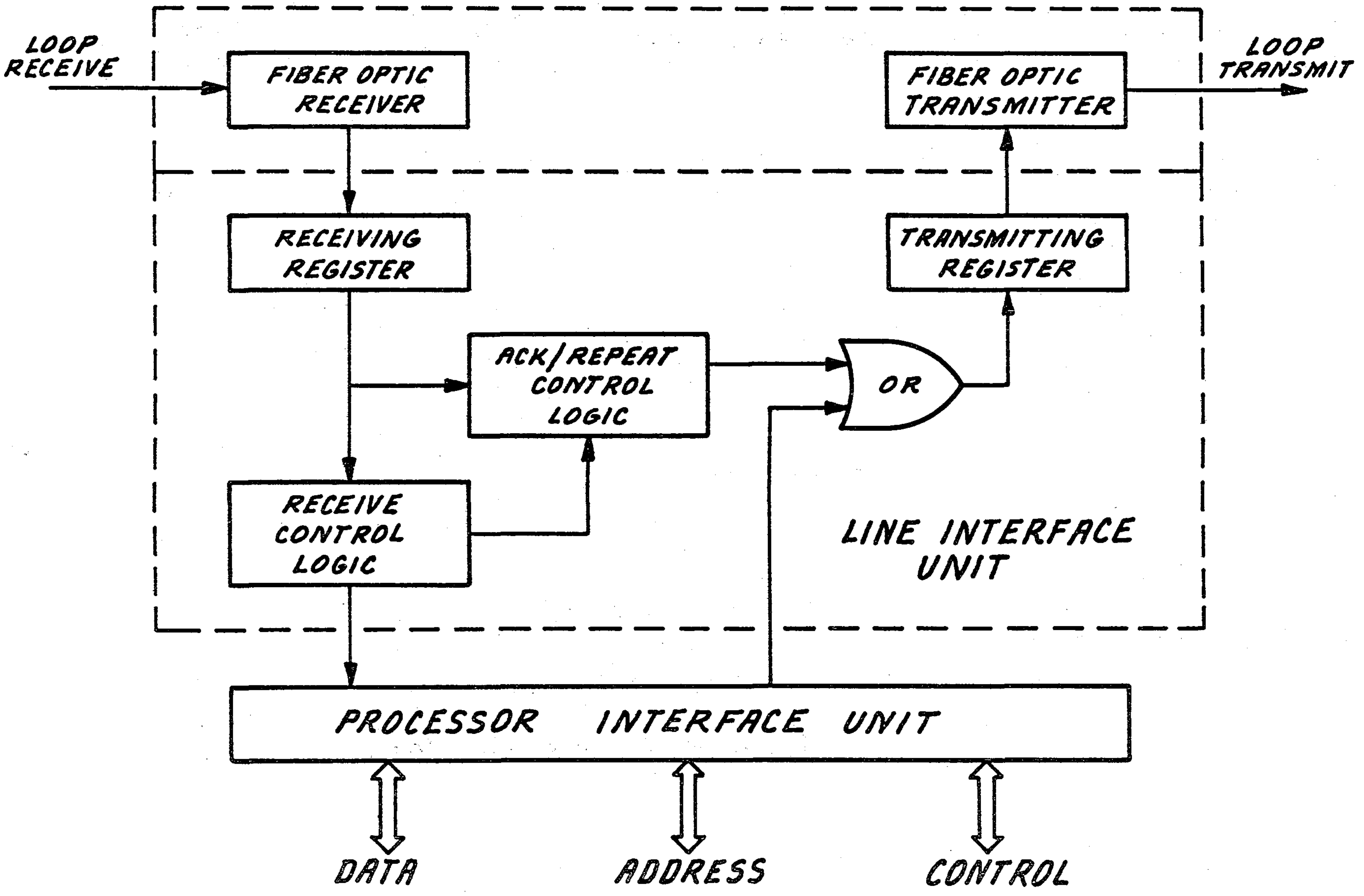
Though the prototype system utilizes micro-computers as its processing agents, a mini-computer LSM network would be even more practical, better utilizing the bandwidth potential of the fiber optic loop. Looking ahead, the next generation of 16 bit micro-processors appear to be an excellent candidate for agents on an LSM loop. With the protocol modified to allow 16 bits of data, this combination could prove to be an extremely potent network computer, the loosely coupled components functioning as one entity, but more failsafe and more easily expanded than a single mainframe computer.

LOOP PACKET FORMATS

Each packet propogates once around the loop where it is then removed by the originating agent. The "ACK" bit is controlled by the receiving agent(s) upon error free reception.



LOOP INTERFACE BLOCK DIAGRAM



REFERENCES

1) J.R. Jones and D.F. Hemmings, "Optical Fiber T-Carrier Transmission Systems", National Telecommunications Conference, Birmingham, Alabama, December, 1976.

2) J.H. Mullins, "A Bell System Optical Fiber System - Chicago Installation", National Telecommunications Conference, Los Angeles, Calif., December, 1977.

3) R.E. Epworth, "ITT 140 Mbit/s Optical Fibre System", National Telecommunications Conference, Los Angeles, Calif., Dec., 1977.

4) J.R. Pierce, "How Far Can Data Loops Go?", Transactions on Communications", IEEE, Vol. COM-20, No. 3, June, 1972.

5) I.Jacobs, "Telecommunication Applications of Fiber Optics", National Telecommunications Conference, Los Angeles, Calif., Dec., 1977.

6) D.J. Farber, et al., "The Distributed Computing System", Proceedings of the Seventh Annual IEEE Computer Society International Conference, Feb., 1973

7) A.G. Fraser, "SPIDER - A Data Communications Experiment", Computing Science Technical Report No. 23, Bell Laboratories, Dec., 1974.

8) J. Labetoulle, E. Manning, and R. Peebles, "Analysis and Simulation of a Homogeneous Computer Network", Technical Report E-59, Computer Communications Networks Group, University of Waterloo, Feb., 1977.

9) C.C. Reames and M.T. Liu, "A Loop Network for Simultaneous Transmission of Variable - Length Messages", Proceedings 2nd Annual Symposium on Computer Architecture, Houston, Texas, Jan., 1975.

10) M.V. Wilkes, "Communication Using a Digital Ring", PACNET Conference, Sendai, Japan, August, 1975.

11) M. Tanaka, I. Ikushima, M. Maeda, and K. Nagano, "New Kind of Noise Performance in Optical PCM 200 Mb/s Transmission Systems", National Telecommunications Conference, Birmingham, Alabama, Dec., 1978.

A DISTRIBUTED FILE MANAGER FOR THE TRW EXPERIMENTAL DEVELOPMENT SYSTEM

Scott Danforth
TRW R2/1170
1 Space Park
Redondo Beach, CA 90278

Abstract

Since 1976, the Computer Engineering Section at TRW has been using Concurrent Pascal [1] in its multiple-minicomputer Signal Processing Facility for research into the software engineering of special purpose locally distributed systems. In such systems, the particular operating system support required at any processor can be quite specific, removing the necessity for use of typical vendor-supplied general purpose operating systems. With the aid of appropriate Concurrent Pascal code segments, complete operating system environments can be easily constructed which exactly match local processing requirements. Intercomputer link drivers, file systems, graphics packages, and performance monitors are examples of typical services. To these, we have added a simple distributed file manager (DFM) which maintains three possible levels of access control for a file distributed redundantly over any number of machines.

The purpose of this paper is to discuss the particular services or application code interfaces into the DFM which were decided upon, the operation of the manager and its representation as a collection of Concurrent Pascal system types, and our method of monitoring its performance.

1.0 INTRODUCTION

Our work has been conducted with an emphasis on the beneficial effects of Concurrent Pascal on distributed systems software engineering for multiprocessor local network architectures. We believe that two barriers to rapid and reliable construction of such distributed systems are the use of vendor-supplied operating systems, which are usually designed to support general purpose timesharing on a single processor, and the use of vendor-supplied languages which do not provide the level of software structure necessary for simple and effective solutions to problems of parallel execution. We don't claim that Concurrent Pascal, or any other language designed to support concurrent programming (e.g. Modula, Ada, etc.) is "the" answer to low risk implementation of distributed systems. However, we do feel it is important to break the vendor-supplied OS "habit" and these languages provide an important and workable means of doing so.

Using Concurrent Pascal (or other appropriate languages), application-tailored operating systems can be created and their performance studied and monitored in the space of weeks. Simulation of data distribution protocols, for instance, may be unnecessary when actual implementation takes no more time.

In our search for an expedient software development base, a Concurrent Pascal operating system was developed starting from Per Brinch Hansen's original SOLO system [2]. The resultant Experimental Development System (EDS) has now been further modified with the addition of a Distributed File Manager (DFM). While the DFM is primarily meant to be available for incorporation into special-purpose operating systems supporting distributed processing applications, inserting it into EDS provided insight into the interface between a distributed processing capability and applications code, and allowed us to easily test and monitor DFM performance.

The DFM discussed in this paper is oriented toward maintaining three possible levels of access control for a single redundantly distributed file. Distributed database protocols are often presented in the context of updates, and our particular choice of access levels is an attempt to generalize these results in a straightforward fashion to include less expensive types of control for read operations. The three levels of access control correspond to the following possible characterizations of the local file copy: latest copy; consistent copy; and possibly inconsistent copy. Updates may be performed only on a latest copy, but reads may be performed under any level of DFM control. Responsibility for requesting the appropriate level is left to the application.

Extensions to this basic capability for handling a single distributed file readily suggest themselves. However, since this file is essentially a shared array of disk pages made available to application code on any number of machines, a useful environment for distributed processing applications is made available.

The following presentation is structured in a top-down manner. First an overview of EDS is given in order to introduce system access graphs and provide a general idea of the software environment into which the DFM was inserted. This is followed by a discussion of the application code interfaces to the Distributed File Manager and the services which were made available. The structure of the DFM in terms of Concurrent Pascal system types, and the rationale behind its operation are detailed, after which we discuss performance monitoring. We conclude with a few words concerning the utility of the DFM.

2.0 THE OPERATING SYSTEM

Our Experimental Development System (EDS) is written in Concurrent Pascal. In addition to the usual Pascal data types such as integer, array, record, etc., Concurrent Pascal provides the additional system types of "process" and "monitor." These are provided in order to facilitate the explicit high-level expression of a multiple process environment, and the means by which processes may communicate with each other. If a Concurrent Pascal process wishes to do so, it may load and execute a Sequential Pascal program which has been previously compiled. This is equivalent to what happens in a timesharing system when a process associated with a remote terminal responds to a command by loading and executing a user program.

The basic structure of EDS is three processes (input, job, and output), plus monitors to allow the job process to communicate with the input and output processes. Through these monitors, the job process first makes its I/O requirements known to the input and output processes, and then subsequently accepts or sends blocks of information as they are made available. Depending on the communicated requirements of the job process, the I/O processes load and execute appropriate sequential programs. Unlike the I/O processes, which are essentially dependent on the job process for information determining their course of action, the job process prompts the user by loading and executing a sequential program specifically designed as a user interface. Operating system services made available to a sequential program include the ability to request that a different sequential program be loaded and run, thus the user interface program reads a command line from the console, parses it it determine the user's requirements, and then requests that the appropriate program be run. When the required program has run to completion, the user interface is continued (with the completion status of the requested program made known to it), and the cycle repeats itself. Figure 1 is an access graph or diagram of the important EDS system components.

The simplicity of EDS is one of its most valuable features, allowing it to be rapidly modified to suit its users' requirements. Such simplicity provides an excellent basis for distributed processing experiments, and in general helps to blur the distinction between application and operating system code. Though we feel that this is an asset when creating specially tailored systems (in which the operating system "is" the application code to a great degree), it should be noted that Concurrent Pascal can be, and has been used to create larger and more sophisticated operating systems. Three worthy of mention are the Interactive Graphics Operating System of John Barr [3], the MUSIC Multi-User system of Klaus-Peter Lohr [4], and the Capabilities Operating System (LINUS) of Mike Ball [5].

Typical services provided by the EDS processes which execute sequential code (we have already mentioned the ability to run sequential programs) include a set of file access services - open, close, get, put, etc. - and a set I/O functions - readregister, writeregister, awaitinterrupt, etc. - which are useful for writing device drivers in Pascal [6]. To these services, an additional set has now been made available for experimentation. They provide the ability to manipulate a locally disk-resident array of pages which are shared redundantly (updates are broadcast and incorporated into external file copies) with cooperating job process programs on other machines. These newly provided services will now be disucssed.

3.0 DISTRIBUTED FILE SERVICES

The EDS file system is built around a disk-resident catalog of named files and their attributes. One attribute of a file is the disk address of its page map. Opening a file involves a search of the catalog for a file's attributes, followed by reading it's page map into memory. Subsequent file I/O requests (get,put) are then processed by referencing this page map and performing the appropriate disk access.

The DFM performs its local operations in a similar manner, making use of the catalog to locate a file's page map, etc., with the restriction that only one file may be accessed through it. Pages may be read from this file at any time, Using Concurrent Pascal (or other appropriate languages), application-tailored operating systems can be created and their performance studied and monitored in the space of weeks. Simulation of data distribution protocols, for instance, may be unnecessary when actual implementation takes no more time.

In our search for an expedient software development base, a Concurrent Pascal operating system was developed starting from Per Brinch Hansen's original SOLO system [2]. The resultant Experimental Development System (EDS) has now been further modified with the addition of a Distributed File Manager (DFM). While the DFM is primarily meant to be available for incorporation into special-purpose operating systems supporting distributed processing applications, inserting it into EDS provided insight into the interface between a distributed processing capability and applications code, and allowed us to easily test and monitor DFM performance.

The DFM discussed in this paper is oriented toward maintaining three possible levels of access control for a single redundantly distributed file. Distributed database protocols are often presented in the context of updates, and our particular choice of access levels is an attempt to generalize these results in a straightforward fashion to include less expensive types of control for read operations. The three levels of access control correspond to the following possible characterizations of the local file copy: latest copy; consistent copy; and possibly inconsistent copy. Updates may be performed only on a latest copy, but reads may be performed under any level of DFM control. Responsibility for requesting the appropriate level is left to the application.

Extensions to this basic capability for handling a single distributed file readily suggest themselves. However, since this file is essentially a shared array of disk pages made available to application code on any number of machines, a useful environment for distributed processing applications is made available.

The following presentation is structured in a top-down manner. First an overview of EDS is given in order to introduce system access graphs and provide a general idea of the software environment into which the DFM was inserted. This is followed by a discussion of the application code interfaces to the Distributed File Manager and the services which were made available. The structure of the DFM in terms of Concurrent Pascal system types, and the rationale behind its operation are detailed, after which we discuss performance monitoring. We conclude with a few words concerning the utility of the DFM.

2.0 THE OPERATING SYSTEM

Our Experimental Development System (EDS) is written in Concurrent Pascal. In addition to the usual Pascal data types such as integer, array, record, etc., Concurrent Pascal provides the additional system types of "process" and "monitor." These are provided in order to facilitate the explicit high-level expression of a multiple process environment, and the means by which processes may communicate with each other. If a Concurrent Pascal process wishes to do so, it may load and execute a Sequential Pascal program which has been previously compiled. This is equivalent to what happens in a timesharing system when a process associated with a remote terminal responds to a command by loading and executing a user program.

The basic structure of EDS is three processes (input, job, and output), plus monitors to allow the job process to communicate with the input and output processes. Through these monitors, the job process first makes its I/O requirements known to the input and output processes, and then subsequently accepts or sends blocks of information as they are made available. Depending on the communicated requirements of the job process, the I/O processes load and execute appropriate sequential programs. Unlike the I/O processes, which are essentially dependent on the job process for information determining their course of action, the job process prompts the user by loading and executing a sequential program specifically designed as a user interface. Operating system services made available to a sequential program include the ability to request that a different sequential program be loaded and run, thus the user interface program reads a command line from the console, parses it it determine the user's requirements, and then requests that the appropriate program be run. When the required program has run to completion, the user interface is continued (with the completion status of the requested program made known to it), and the cycle repeats itself. Figure 1 is an access graph or diagram of the important EDS system components.

The simplicity of EDS is one of its most valuable features, allowing it to be rapidly modified to suit its users' requirements. Such simplicity provides an excellent basis for distributed processing experiments, and in general helps to blur the distinction between application and operating system code. Though we feel that this is an asset when creating specially tailored systems (in which the operating system "is" the application code to a great degree), it should be noted that Concurrent Pascal can be, and has been used to create larger and more sophisticated operating systems. Three worthy of mention are the Interactive Graphics Operating System of John Barr [3], the MUSIC Multi-User system of Klaus-Peter Lohr [4], and the Capabilities Operating System (LINUS) of Mike Ball [5].

Typical services provided by the EDS processes which execute sequential code (we have already mentioned the ability to run sequential programs) include a set of file access services - open, close, get, put, etc. - and a set I/O functions - readregister, writeregister, awaitinterrupt, etc. - which are useful for writing device drivers in Pascal [6]. To these services, an additional set has now been made available for experimentation. They provide the ability to manipulate a locally disk-resident array of pages which are shared redundantly (updates are broadcast and incorporated into external file copies) with cooperating job process programs on other machines. These newly provided services will now be disucssed.

3.0 DISTRIBUTED FILE SERVICES

The EDS file system is built around a disk-resident catalog of named files and their attributes. One attribute of a file is the disk address of its page map. Opening a file involves a search of the catalog for a file's attributes, followed by reading it's page map into memory. Subsequent file I/O requests (get,put) are then processed by referencing this page map and performing the appropriate disk access.

The DFM performs its local operations in a similar manner, making use of the catalog to locate a file's page map, etc., with the restriction that only one file may be accessed through it. Pages may be read from this file at any time,

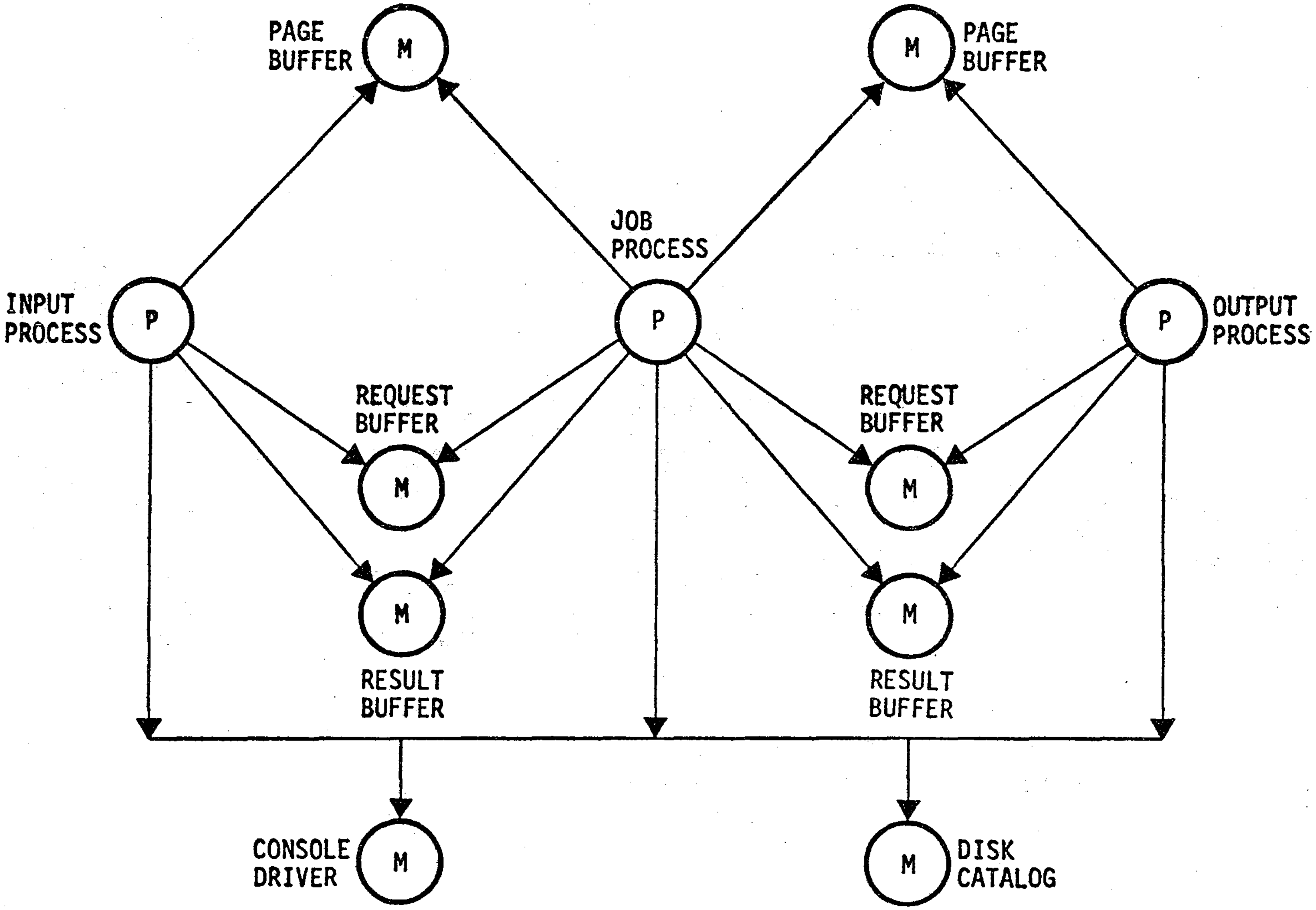


FIGURE 1. BASIC EDS SYSTEM COMPONENTS

and the application code has the option of requesting that the DFM maintain inter-page consistency during local file access. This is done by allowing the DFM to exercise a certain amount of access control.

Updates may only be performed on what is called the "latest" copy of the file, access to which is globally exclusive. This file copy contains all previous updates, thus the term "latest."

A non-exclusive "consistent" copy is also available for read-only operations, and this consistent copy will reflect all or none of the updates performed on any "latest" copy. In practice, this copy will initially be the latest version of the file, with new updates from other nodes temporarily locked out. If another node is performing updates on the "latest" copy at the time a "consistent" copy is requested, the DFM will wait until these updates are finished before locking the local copy and continuing the program which requested a "consistent" copy. The use of the word "consistent" here is based on the assumption that the file is always left in a consistent state upon completion of updates on any "latest" copy. Since access to consistent copies is not globally exclusive, efficiency of file access is made possible for code which doesn't require update privileges.

Programs which don't require inter-page consistency need not request a consistent copy, in which case reads will be performed with no file locking in effect, and without delay even if updates are being currently performed. Applications which allow pre-allocation of information to particular file pages would be capable of exploiting this. An example would be a mailbox application.

Actual file IO is supported with the usual page update and read services. An additional facility whereby an accessing program may wait until one of a set of pages is updated before performing a read is also available.

The separate DFM service calls will now be listed, and their meanings discussed. The semantics of these DFM service calls are a separate concern from the underlying manager protocols which implement them, details of which are disucssed in a later section. For instance, while the particular manager we used employs distributed control, there is no reason why the same services might not be provided through the use of a centralized control protocol.

3.1 DFM Entry Procedures

The following represents the application code interface to the DFM. For those unfamiliar with Pascal-like code, the keyword "var" is used to give a called procedure the ability to modify or assign a value to a passed parameter. In the following, the passed parameter "OK" is used for the purpose of allowing the DFM to explicitly refuse particular requests for service. For example, a PUT_COPY would be refused (and the value "False" returned in the variable "OK") if the DFM was not maintaining a latest copy at the time of the request.

```
type
   PAGE = array[1..512] of char;
   ID   = array[1..12 ] of char;
   FILE_CURRENCY = (POSSIBLY_INCONSISTENT,
                    CONSISTENT,
                    LATEST);
```

```
NODE = 1..10;
NODE_SET = set of NODE;
WAITPAGE = 1..100;
WAITPAGE_SET = set of WAITPAGE;

procedure entry CONNECT(RECEIVE_SET: NODE_SET;
                        FILE_NAME: ID;
                        var OK: boolean);

procedure entry DISCONNECT;

procedure entry REQUEST_COPY(LEVEL: FILE_CURRENCY;
                             var OK: boolean);

procedure entry RELEASE_COPY;

procedure entry PUT_COPY(PAGENO: integer;
                         BLOCK: PAGE;
                         var OK: boolean);

procedure entry GET_COPY(PAGENO: integer;
                         var BLOCK: PAGE);

procedure entry GET_NEXT(TRIGGER_SET: WAITPAGE_SET;
                         var PAGENO: integer;
                         var BLOCK: PAGE);
```

3.2 DFM Entry Semantics

The services available through the above interface are now described.

3.2.1 Connect

This service and its counterpart (Disconnect) represent an area not usually mentioned in conjunction with DFM protocols. This may be due to the assumption that initialization is a minor detail, but we found Connect to be a very interesting problem, requiring its own special protocol (i.e. aggreement among the DFM's) for a solution.

We decided to allow the local application program to specify the nodes from which external updates will be received (as well as the file on which they should be processed) in order to avoid building this information into the DFM (creating potential inhomogeneity in the managers), and to avoid the possibility of unexpected updates to a file not explicitly connected via a local request. This last possibility might not represent any great danger to a system for which necessary synchronization is achieved in some other way, but we nevertheless chose to enforce synchronization as part of the Connect service.

Stated briefly, a local Connect service will complete as soon as all nodes from which updates will be accepted have themselves requested a Connect service. This approach has some interesting implications (not all favorable) and was chosen with an eye on code complexity. If all nodes specify the same receive_set in the Connect request, then the usual redundant copy database results. Disjoint partitioning of the network nodes is also possible. Unfortunately, a non-disjoint partitioning would be vulnerable to lost updates (updates generated by a successfully connected node before another node issues a connect request specifying the desire to accept updates from the first node). This problem appears to occur outside the range of expected DFM usage, but it points out the fact (as if we needed to be reminded) that possibilities for interesting and possibly erroneous DFM protocol design decisions exist aside from those associated directly with update and access control.

3.2.2 Disconnect

This is the counterpart to Connect. A Disconnect service will complete as soon as all nodes from which updates can be accepted have themselves requested a Disconnect service.

3.2.3 Request_Copy

This service represents our attempt to generalize global mutual exclusion update protocols to include less expensive types of access for read operations in a straightforward and easily implemented fashion. In the discussion that follows, assume that each network node has performed a Connect service specifying all network nodes as the receive_set. Then each node has one local disk file (referred to as the local file copy) on which reads and distributed updates will be processed. All calls to Request_Copy instruct the DFM to maintain a certain level of control on local and external access to this file until a corresponding Release_Copy service is executed. In the interim, any number of permitted accesses of the local file copy may be performed.

External accesses are always updates (in a redundant copy database) and local accesses may be either updates or reads. The three possible levels of access control for the local file copy are:

1. external updates allowed, local updates not allowed, and local reads allowed

2. external updates not allowed (but queued for later inclusion in the local file copy), local updates not allowed, and local reads allowed

3. external updates not allowed (none can arrive due to the globally mutually exclusive nature of the protocol by which this level of access is granted), local updates allowed, and local reads allowed

In the first case, no guarantee concerning the consistency of the local copy can be made by the DFM since a related stream of external updates may only be partially completed at the time a series of local read operations are performed. In the second case, the DFM is able to guarantee consistency of the local file copy because no updates to this copy can occur, and because the Request_Copy service for this level of access will not complete until the local copy reaches a consistent state. Consistency is assumed to occur at the completion of a stream of related updates, which is signalled by an updating node when it performs a Release_Copy service. Only in the third case may updates be created for inclusion in the local copy and distribution to "connected" external file copies. Use of a global mutual exclusion update protocol (and waiting for the completion of locally queued external updates before local access is allowed) guarantees seriality of update operations.

As mentioned previously, three possible characterizations of the local copy corresponding to these access levels are: possibly inconsistent; consistent; and latest. The default condition is the possibly inconsistent state.

3.2.4 Release_Copy

This is the counterpart to Request_Copy. The result of performing this service is to return a local file copy to the possibly inconsistent state, as well as to signal the end of related updates if the local copy was previously a latest copy. If the local copy was previously a consistent copy, external updates (if any) from nodes having the latest copy will then be processed. In all cases external updates are processed in the order in which they are originally performed on a local copy.

3.2.5 Put_Copy

This service is the means by which an update of "connected" file copies is performed. Assuming that the local file copy of the node requesting this service is the latest copy, the following actions are taken. First the update is broadcast to external nodes. The update protocol we use employs a ring network for this purpose. This serves to balance the load of distributing updates to all copies. The update is passed from one node to the next, with each DFM examining the update to determine its origin. If the origin is contained in the receive_set for a node, then the update is queued for processing at that node. Each node in the network sees the update in this way, until the update returns to its origin. Here the update is removed from the ring. As soon the update is originally released to travel about the ring, the update of the local copy at the originating node is performed. Subsequent requests for the Put_Copy service are handled as soon as they are issued, without waiting for the return of previous updates from around the ring.

The particular intercomputer link chosen determines the data transmission protocol which is used. In our case, a fairly sophisticated link is used which implements CRC checking at the hardware level. The software is designed to recover from hardware-detected transmission errors by retransmitting the message. Since both sender and receiver are notified by the hardware when transmission errors occur, this is fairly easy to do. This is one area in which local networks can differ drastically from those more geographically dispersed.

3.2.6 Get_Copy

This service is the means by which a read access of the local file copy is performed. This service is available at any level of access control.

3.2.7 Get_Next

This service was included to allow an application to receive an externally generated page update as soon as it arrives locally. In making the Get_Next service request, an application specifies which pages it is interested in, and the next arriving update for any of these pages causes the call to complete. The number of the page which was updated is returned along with the new page contents.

A request for a consistent copy at this point (as soon as the return from Net_Next is effected) would complete as soon as all of the related updates which included the "trigger" are processed on the local file copy, and would lock out further sets of updates from some other (or the same) external node. This might be important if separate sets of updates could arrive faster than they can be processed by the application.

4.0 DISTRIBUTED FILE MANAGER IMPLEMENTATION

Various aspects of the DFM implemention are now discussed.

4.1 DFM Structure

The system components which make up the DFM are detailed in the access graph of Figure 2. A list of these system types, and a short discussion of their respective functions will now be given. The parameters in the following type definitions designate the system components which are available for use by a component of the type being defined. For instance, a component of type RINGOUTPROCESS can make calls to a component designated as ROB (which is of type RINGOUTBUFFER).

```
type RINGOUTBUFFER = monitor;
```

This buffer holds messages destined for transmission to other nodes, via the ring-network structure used for inter-machine communication. DRMANAGER calls it in order to create ring messages, and RINGOUTPROCESS calls it in order to receive them.

```
type RINGOUTPROCESS = process(ROB:  RINGOUTBUFFER);
```

This process contains the PCL-11 communication link transmission driver, and implements the ring structure. The process accesses ROB to get a message for the ring, transmits the message to the "next" ring node, then cycles back to get and transmit the next message, etc.

```
type EXTUPDATEBUFFER = monitor;
```

This buffer holds messages and updates associated with external nodes from which updates can be received. DRMANAGER writes to it, and EXTUPDATEPROCESS reads from it. When the local file copy is locked, this is where external updates queue up for ultimate delivery when a local Release_Copy is performed.

```
type FILEMANAGER = monitor;
```

This monitor performs file I/O, and contains file locks for use in maintaining required levels of file protection. It is accessed by DRMANAGER to request locking, EXTUPDATEPROCESS to perform external updates, and the job process to perform local file manipulations.

```
type EXTUPDATEPROCESS = process (EUB:  EXTUPDATEBUFFER;
                                  FM:  FILEMANAGER);
```

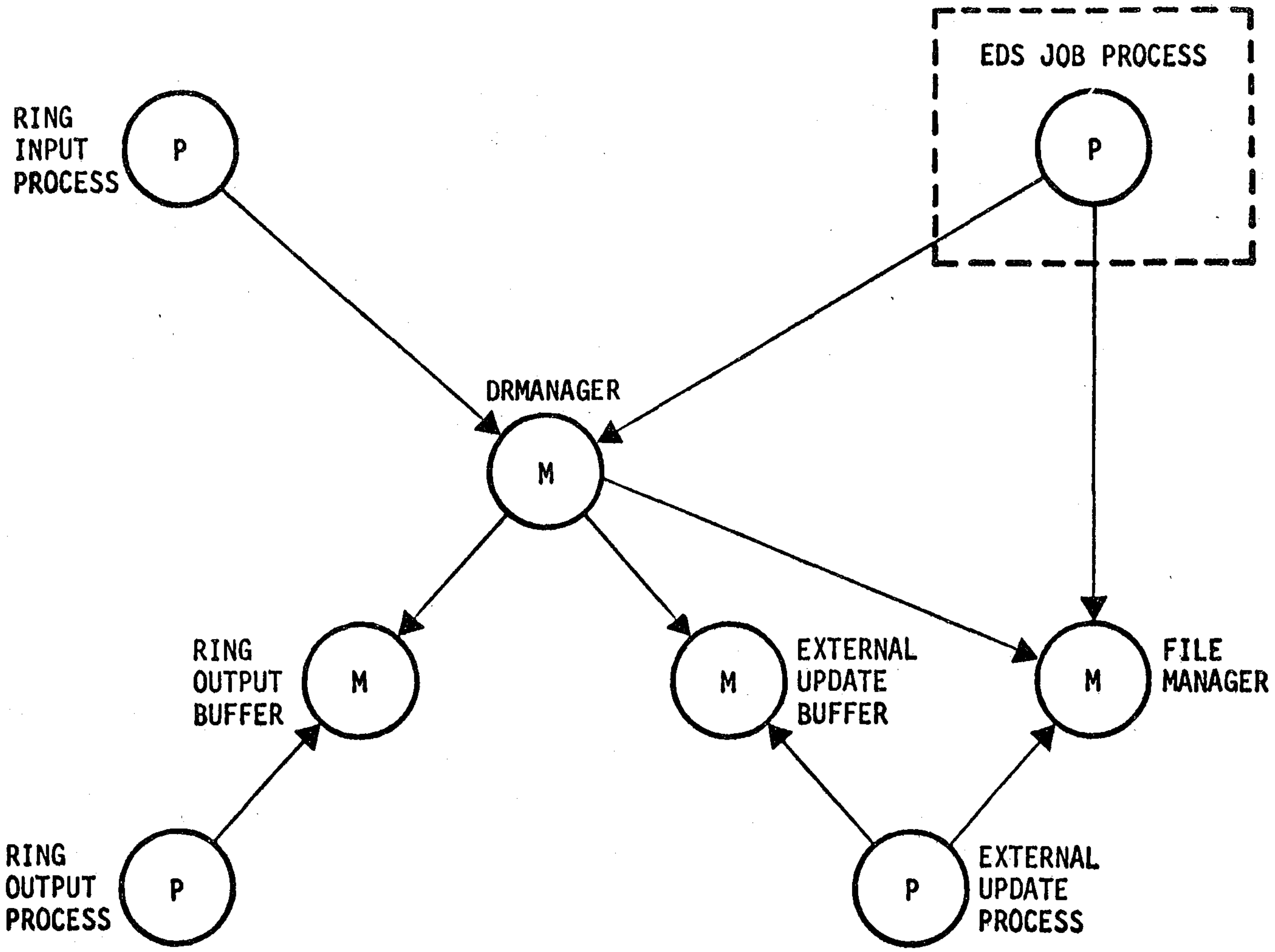


FIGURE 2. DFM SYSTEM COMPONENTS

This process reads updates posted by DRMANAGER to EUB, performs appropriate calls to FM, and then cycles back to get another transaction from EUB, etc.

```
type DRMANAGER = monitor (FM:  FILEMANAGER;
                          ROB:  RINGOUTBUFFER;
                          EUB:  EXTUPDATEBUFFER);
```

This monitor mediates requests for DFM services from the local job process, and from external managers at other nodes whose requirements are pulled off the ring by RINGINPROCESS. When appropriate, it sends messages to the ring via ROB, relays external updates via EUB, and sets locks by calling FM. DRMANAGER is called by the local job process, responding to service requests, and RINGINPROCESS, responding to ring arrivals.

```
type RINGINPROCESS = process(DRM:  DRMANAGER);
```

This process contains the PCL-11 communication link receiver driver. It accesses the PCL-11 bus to receive messages addressed to this node, calls DRM to deliver them, then cycles back to receive the next ring message, etc.

4.2 DRMANAGER Operation

While the collection of system types listed above are responsible for performing collectively as the Distributed File Manager, the DRMANAGER monitor contains the basic intelligence related to the protocol whose purpose is to provide global mutual exclusion of access to latest copies, and perform distribution of updates. Aside from the particular services we chose to make available to applications programs, it is the operation of DRMANAGER which may be of greatest interest to others. In what follows, we will provide a short overview of this operation.

The DRMANAGER Connect protocol has already been discussed. A major influence on its development was our decision to use a distributed control update protocol. If we had used a centralized manager, Connect would have been more straightforward. A primary motivation for this work was the question of DFM services however, and we decided to make use of a distributed control protocol which we previously implemented following a description given by Ellis [7]. As discussed in the concluding remarks, a centralized control protocol is now being investigated as well.

Our paper [8] examines Ellis' protocol in some detail. His protocol is elegant and requires few ring messages, but only allows one transaction on a file copy (each transaction is implicitly a Request_Copy and a Release_Copy as well). Bringing a database from one consistent state to another usually involves a series of related file transactions, so we extended Ellis' update protocol by requiring explicit request and release of the file copy, between which any number of file transactions might occur. The resulting protocol is used by DRMANAGER in order to control access to what we have termed the latest copy, and distributes updates which are processed on it.

Access control and maintenance of what we have termed a consistent copy was motivated by the desire to allow a series of related reads (for example, an index traversal) without requiring the use of the latest copy, access to which is mutually exclusive. Providing a consistent copy proved to be as interesting as the update protocol, and the use of a concurrent programming language significantly aided our visualization of the problem. Requests for a consistent copy involve no ring traffic since they are handled locally in the FILEMANAGER monitor by waiting for the end of the present set of updates (which is signalled by a release message), and then locking out further updates until the consistent copy is locally released via Release_Copy.

4.2.1 The Distributed Control Update Protocol

The underlying idea behind the DRMANAGER update protocol is the manner in which "simultaneous" requests for a latest copy are handled. Each node has a unique priority, and messages are sent on a ring-network communication structure in order of increasing node priority. A request for the latest copy travels over the ring and is examined by each local DRMANAGER before being relayed on. If such a request arrives at a node of higher priority than that of its origin, and the higher priority node itself has a request in transit (this is the meaning of simultaneous in this context) then the lower priority request is delayed and saved. Subsequently, the higher priority node will receive its request when it returns around the ring. This return indicates that it's request for the latest copy is granted.

After performing the necessary transactions on its file copy (with page updates being appropriately broadcast as they occur), the higher priority node performs a Release_Copy. A release message is then sent around the ring which signifies the end of updates to this particular latest copy, and indicates whether or not a lower priority request has been saved. This message is followed immediately by the saved lower priority request, if any. The saved lower priority request, if any, then continues around the ring to ultimately return to its origin.

An additional constraint is used to limit the number of requests which have to be saved (i.e. removed from the ring pending grant and release of the latest copy) at any node: once a request is relayed, the relaying node may itself make no requests until a release message indicating no following saved requests is received. With this restriction, no node need ever save more than one request; without it, the maximum would be dependent on the total number of nodes in the network. The protocol has the nice property that no request is ever denied (thus retries are not necessary), and all requests are ultimately granted. Since the node priorities are used only to break ties, each node is served fairly.

4.3 Hardware Configuration

The actual hardware that is used includes a DEC PCL-11 (Parallel Communications Link) intercomputer link which connects four PDP-11 machines and one VAX-11/780. The PCL is a multi-dropped TDM Bus (time division multiplexed - i.e. transmission time is time-sliced by node, allowing one 16 bit parallel data transfer per slice). Although this link supports a totally connected network, it is used here to implement a ring structure, in accordance with the requirements of the distributed control update protocol.

One of the strong points of the PCL link is that is supports reconfiguration of a network in case of node failure. The DFM presently implemented does not make use of this capability, however, and is vulnerable to node failure. An interesting and effective use of the PCL would be to implement a "wheel" structured network in order to support distribution of updates via a ring structure (to balance update broadcast load) and to provide for centralized control of access to the latest copy (an efficient method for global access control).

5.0 PERFORMANCE MONITORING

In order to monitor the DFM and observe the overhead associated with its various operations, we dedicate a separate machine to the function of driving a graphics display device (in this case a Tektronix 4014) to report this information. At this machine, a process reads data posted to it via inter-computer link, and sends this information to a buffer. Another process reads the contents of this buffer at a set frequency, and makes calls into a display utility implementing a graphics capability on the Tektronix device in order to display bar graphs for each machine which is reporting.

At the reporting machines a reporting process is included which reads performance information posted to a buffer by the job process as it makes calls to the DFM, and then sends this information via inter-computer link to the display machine. Each call by the job process to the reporting buffer increments a count representing the number of times a particular DFM call has been made. This count is reset to zero each time the process in charge of relaying this information to the display machine calls to receive it. Since this is done once a second, the information which is ultimately displayed is the number of various DFM calls made per second. The particular information we display is (for each reporting machine): request of latest copy; request of consistent copy; updates, reads, and read_nexts.

This admittedly simple-minded scheme allows us to easily determine the performance of the DFM by writing a simple application program which does nothing but make use of DFM services. The overhead associated with requesting the latest copy without contention is summarized as follows:

# network nodes	overhead (in msecs)
1	31
2	70
3	109
4	145

These results show the expected dependency on the number of machines in the network. This dependency could have been simulated, and an estimate of the respective overheads obtained, but the above values reflect actual run-time system performance - a fact we feel would be important when performaing tradeoff analysis for proposed systems.

Since requests for a consistent file copy are handled locally, this overhead is independent of the number of participating nodes. Consistent requests were observed to take about 5 milliseconds.

These measurements were made using an interpreter based implementation of the Concurrent Pascal language [2], running on PDP-11/40, 11/45, 11/60, and VAX-11/780 machines.

6.0 CONCLUDING REMARKS

The Distributed File Manager implemented appears to supply a basic and useful capability for the control and manipulation of distributed information. For instance, typical real-time databases often present the problem of simultaneous updating and reading by numerous and loosely related tasks. The DFM supports the partitioning of such activity onto separate processors. Implementation of data pipeline schemes is another possibility.

As always, the application should determine the particular software support which is required, and with Concurrent Pascal one may confidently advocate this approach. The total implementation time required for the DFM, for instance, was three weeks. This supports the contention that Concurrent Pascal is an effective programming tool for distributed systems. Furthermore, it lends additional credence to the ideal of the "tailored" or special purpose operating system; for although the DFM may not exactly fit the requirements of a particular application, extensions or added capabilities are no great problem. Multiple files, file partitioning onto various processors, keeping lists of changed pages, maintaining files in primary storage instead of on disk, utilization of shared memory - these are all easily done, and with a minimum of "artificial" (i.e. vendor supplied) constraint and difficulty.

Due to the dependency of latest-copy request overhead on the number of machines in the network, one might wonder why a centralized control protocol was not chosen. Garcia-Molina has presented results which seem to convincingly indicate the superiority of centralized control protocols over those utilizing distributed control [9].

In our environment, one of the greatest problems associated with special purpose distributed systems is their high risk. For this reason, considerations of code complexity are fairly important to us. Garcia-Molina claims centralized control is easier to implement, but we believe that consideration of fallback in the event of machine failure may turn the tables with respect to code complexity. (In any case, DRMANAGER required only about 200 lines of code to implement all of the services discussed.) In order to handle node failures in the distributed control case, we see a recovery capability essentially arising out of a no-lost-message constraint (with certain embellishments) which would require a small additional amount of code at each node. In the centralized control case, complete centralized manager code must exist at each node (as opposed to only one if node failures are not handled) to recover from arbitrary node failures, and switchover must not only worry about lost messages, but also must have been preceded by posting system-wide status information to some delegated secondary. At switchover, a new secondary must be chosen and a complete status summary posted to it as well.

Because of such details, we believe that fault tolerant distributed control may actually be less complex than centralized alternatives. The DFM discussed in this paper is presently being extended to provide immunity to node failure, and since we are also implementing a centralized control protocol, a more realistic comparison of the relative benefits of the two approaches should soon be available.

REFERENCES

1. Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE TSE 1,2 (June 1975), pp. 199-207

2. Brinch Hansen, P., "The Architecture of Concurrent Programs," Prentice-Hall, 1977

3. Barr, J., "A Methodology for the Design of Interactive Graphics Operating Systems," PhD Thesis, UCLA 1978

4. Graf, N., Ketschmar, H., Lohr, K., and Morawetz, B., "How to Design and Implement Small Timesharing Systems using Concurrent Pascal," Software Practice and Experience, 9,1 (Jan 1979) pp. 17-24

5. Ball, M., "The LINUS Capability Based Operating System Kernel," Distribution Tape, NOSC, San Diego, CA.

6. Heimbigner, D., "Writing Device Drivers in Concurrent Pascal," ACM SIGOPS Review 12,4 Oct. 1978, pp. 16-33

7. Ellis, C., "Consistency and Correctness of Duplicate Database Systems," Proceedings of the Sixth Symposium on Operating Systems, ACM SIGOPS Volume II, Nov. 1977

8. Danforth, S., "Implementation of a Distributed Resource Manager in Concurrent Pascal," paper submitted to First International Conference on Distributed Computing

9. Garcia-Molina, H., "Performance Comparison of Two Update Algorithms for Distributed Databases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978, pp. 108-119

NETWORK SUPPORT FOR A DISTRIBUTED DATA BASE SYSTEM[1]

Lawrence A. Rowe and Kenneth P. Birman
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

COCANET is a local computer network being developed, in part, to support distributed data base system research. A multidestination, or multicast, protocol is provided to satisfy the communication requirements of the INGRES distributed data base system. These requirements include sending messages to a dynamically varying subset of processes. Efficient implementation of the multicast protocol in a local broadcast network is described. In addition, internetwork support of the protocol is discussed.

COCANET extends a conventional UNIX[2] programming environment across multiple processors by supporting transparent resource sharing and message-oriented interprocess communication mechanisms.

Keywords: Local computer networks, multidestination addressing, multicast protocols, distributed data base systems, resource sharing.

1. INTRODUCTION

COCANET is a local computer network designed and currently being implemented at U.C. Berkeley. It was developed to support research on distributed data base systems and to provide shared access to resources available on different machines in the Department of Electrical Engineering and Computer Science (e.g., a high-resolution graphics output device and the ARPANET).

The computers which will be connected to the prototype network are DEC PDP-11's (an 11/70 and two VAX's) running the UNIX operating system [Ritchie 78]. The physical architecture of the prototype is a ring using local network interfaces (LNI) developed at U.C. Irvine [Mockapetris 77]. The UNIX programming environment has been extended across the different hosts so that users connected to one host can transparently access resources located on the other hosts. The network software is organized to allow local network hardware other than LNI's to be used.

COCANET supports a conventional process-to-process communication protocol and a multicast, or multidestination addressing, protocol. While the multicast protocol was primarily motivated by the communication requirements of the INGRES distributed data base system [Epstein 79, Stonebraker 77], it can also be used for other distributed applications (e.g., data base machines [Stonebraker 79a]).

[1] The first author was supported in part by AFOSR Grant 78-3596 and the second author was supported in part by DOE contract W-7405-ENG-48

[2] UNIX is a trademark of Bell Laboratories.

Most previous research on distributed data base systems has not directly addressed the problem of network communication protocols [Rothnie 77]. Research on computer networks, on the other hand, has addressed some of the communication issues [Dalal 78, Farber 73, McQuillan 78], but not all of them (e.g., sending a message to a dynamically varying subset of processes). Moreover, broadcast (send to all hosts) and multicast protocols are not always supported at the application program interface (e.g., see the ARPANET).

This paper describes the communication requirements of distributed INGRES, a multicast protocol designed to meet those requirements, and an efficient implementation of the protocol in a local broadcast network. The paper does not address the problem of security in a distributed environment (the protection mechanisms implemented in the prototype are described in [Birman 79]).

In the next section, the communication requirements of distributed INGRES are illustrated by examining several sample queries. Section 3 describes the COCANET UNIX interprocess communication protocols. The LNI implementation of the multicast protocol and it's extension across network boundaries is presented in section 4.

2. DISTRIBUTED INGRES

This section presents an overview of distributed INGRES, examples of queries which motivated the multicast protocol, and the communication architecture of the system.

Figure 1 shows the logical organization of distributed INGRES [Stonebraker 77]. A user is connected to a master INGRES which spawns slave INGRES's at sites where a fragment of the data base exists. Master INGRES processes user queries by sending commands to the slaves which access the local data and then send the results to other slaves or to the master. A query optimization algorithm produces a sequence of commands to solve the query [Epstein 78]. In addition, master INGRES h ındles updates, crash recovery, and concurrency control [Stonebraker 79b].

A trivial personnel data base is used to illustrate the commands that are sent between INGRES processes. Assume that the employee and department relations

```
EMP(emp#,name,dept#,salary,...)
DEPT(dept#,name,floor,...)
```

are distributed according to the criteria

```
distribute EMP at
    site-1 where EMP.dept# = 93 or EMP.dept# = 122,
    site-2 where EMP.dept# = 47,
    site-3 where EMP.dept# > 0

distribute DEPT at
    site-1 where 10 < DEPT.dept# and DEPT.dept# < 100,
    site-3
```

Distribution criteria restrict a tuple to one unique site [Ries 78]. The distribute commands place tuples of employees in departments 93 and 122 at site-1, department 47 at site-2, and all other departments at site-3 (assuming that department numbers are positive). Department tuples are placed at site-1 for departments 11 through 99 and at site-3 for all other departments. Thus, employee information is stored at three physical locations while department information is stored at two.

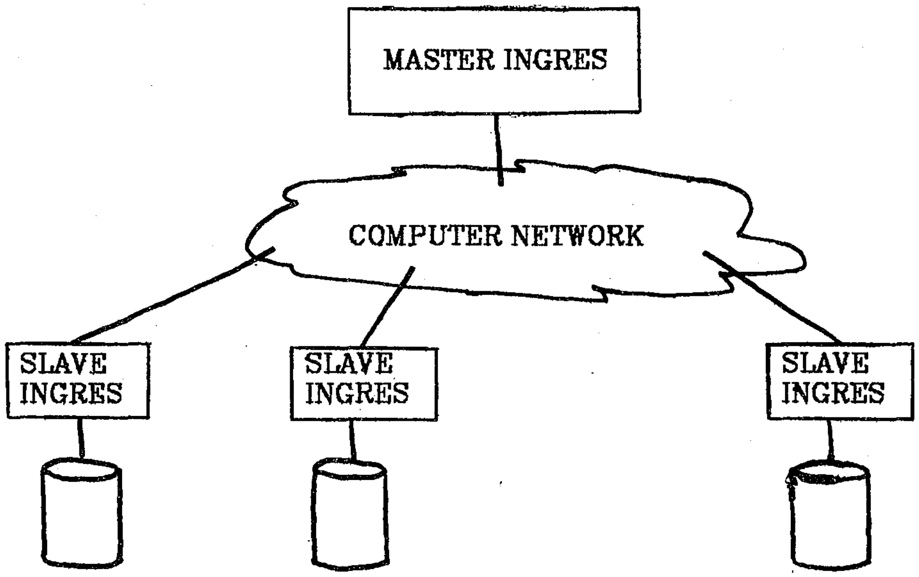


Figure 1. Logical Organization of Distributed INGRES

The first example query is

```
retrieve (EMP.name)
   where EMP.salary > 30K
```

This query can be processed by sending the query to each site that has a fragment of the EMP relation. Each site executes the query on its local fragment. The result relations are sent back to master INGRES which collects the responses and passes them on to the user. Thus, master INGRES must be able to broadcast to slave INGRES's and to receive process-to-process messages from each slave. Notice that if the query had been on the DEPT relation only a subset of slaves (e.g., sites 1 and 3) would have to receive the message. COCANET supports a multicast protocol that allows messages to be sent to arbitrary subsets of a set of processes.

The second example illustrates another use of the subset addressing feature. To reduce the number of sites to which a command must be sent, master INGRES uses the distribution criteria to identify what subset of sites might have relevant data. For example, suppose the query is

```
retrieve (EMP.name,DEPT.floor)
   where EMP.dept# = DEPT.dept# and EMP.dept# = 93
```

which lists names and locations for employees in department 93. Because the department tuple and employee tuples for that department are stored only at site-1 (see the distribution criteria above), the query need be sent only to that site.

A slave INGRES needs to be initiated at a particular site only after the first query that needs data at that site is processed. A user who submits ad hoc queries interactively may access only a small part of the data base during a terminal session. For example, assume that the data in the sample data base is distributed over 20 sites and that the terminal user only requests information about employees in one or two departments. If the queries can be solved without ever accessing data stored at the other sites, slave INGRES's do not have to be initiated at those sites. On the other hand, after several queries, the user might request information from a site which previously had not been queried. The slave could then be initiated and the query processed. To support this feature, the multicast protocol must be able to add processes dynamically to a connection.[3]

The last example shows the use of multidestination file transfer by slaves to send relations to other slaves. It also shows how complex queries which involve several steps are processed. The query

```
retrieve (EMP.name,DEPT.name)
    where EMP.dept# = DEPT.dept# and EMP.salary > 30K
```

lists the employees and the department they work in if they make more than $30K. One strategy to process the request follows. First, at each site which has a fragment of the EMP relation run the restriction

```
retrieve into TEMP(EMP.name,EMP.dept#)
    where EMP.salary > 30K
```

Second, build a complete copy of the TEMP relation at each site which has a fragment of the DEPT relation. In this example, site 1 sends a copy of it's TEMP fragment to site 3, site 2 sends a copy of it's TEMP fragment to sites 1 and 3, and site 3 sends a copy of it's TEMP fragment to site 1. Third, at each site which has a DEPT fragment perform the join

```
retrieve (TEMP.name,DEPT.name)
    where TEMP.dept# = DEPT.dept#
```

Note that the TEMP relation in this join query is the union of all TEMP fragments. The results are then sent to master INGRES. Because TEMP is a complete copy of all qualifying employee tuples, combining the result of running this query at each DEPT fragment gives the answer to the original query.

These operations are synchronized by master INGRES which initiates each step after acknowledgments are received that indicate successful completion of the previous step. Slaves must be able to send data to several other slaves (e.g., site 2 sent data to sites 1 and 3). When a slave must send data to other slaves, a receptor process is initiated at each destination site, a connection is opened to just those receptors, and the data is transmitted to all of them.

Distributed INGRES does not require guaranteed delivery on multicast transmissions, i.e., an explicit acknowledgment from each destination site that the message was received. The high-level distributed INGRES protocol has an acknowledgment mechanism built in to the pattern of communication (e.g., send a command, receive response that the command was completed and possibly data, send the next command, and so forth). Because the application is a data base system, the high-level protocol also has an elaborate crash recovery mechanism which is invoked if an expected response is not received after a reasonable delay. Consequently, whether communication to one or more slave processes is interrupted by a transmission failure or by a hardware or software failure at the destination site, the data base system will recover.

[3] Processes might also be dynamically removed from a connection. This capability can be supported but no proposal has been made to use it.

In summary, a multicast protocol is needed which allows messages to be sent to multiple destinations and to dynamically varying subsets of those destinations. Moreover, it must be possible to add new processes to the set of destinations. COCANET supports such a protocol as described in the remainder of the paper.

3. INTERPROCESS COMMUNICATION

Conventional UNIX does not support an IPC protocol that can be easily extended to a network environment [Chesson 75, Sunshine 77]. Among other problems, the UNIX IPC protocol neither allows a process to wait for a message from more than one process at a time nor allows two unrelated processes to communicate (e.g., a process may wish to communicate with a mail daemon which is always executing).

COCANET UNIX supports two message-oriented IPC protocols to solve these two problems. The first allows two processes to send messages to each other (called *process- to- process*). The second allows one process to send messages to several processes (called *multicast*). These protocols can be used to implement distributed applications. Besides distributed INGRES, they are used to extend all conventional UNIX file operations to a network file system. Consequently, existing programs can access remote files without modification. The remainder of this section describes how these protocols are used. A complete description of their integration into UNIX will be available in [Birman 79].

COCANET UNIX introduces *connections* between processes over which messages can be sent. To the processes, a connection looks similar to a conventional UNIX pipe [Ritchie 78]. A simple example will assist in the explanation. Suppose two processes A and B on different hosts are communicating (assume for the moment the processes and the connection between them already exists). *Send* and *receive* operations on the connection provide a full-duplex, message-oriented communication link, that is, a bi-directional sequence of messages with the sender's identity encoded in each message. Because more than one process can *send* on a connection, a process can receive messages from many sources and can wait for a message from one of several processes. A *status* operation is also provided so that a process can test whether a message exists or if the connection is in an error state.

To understand how connections between processes are created, the network file system and network connection name space must be explained. The file system in UNIX is organized as a tree structure. "/" refers to the directory at the root of the tree so that "/f" refers to the file or directory named f. Under COCANET, the file system of each host includes pseudo-directories which contain addressing information needed to communicate with hosts on the network. For example, in the file system on host-1, "/host-2" refers to the root of the file system on host-2.

Connections are referenced by the pair <destination host, connection name>. Suppose process A was on host-1, process B was on host-2, and the connection name was AtoB in the example above. Process A refers to the connection as "/host-2=AtoB" and process B refers to it as "/host-1=AtoB". /host-1 and /host-2 are the file system pseudo-directories on hosts 2 and 1, respectively, that specify the destination host. AtoB is a name in the connection name space maintained by the network software.

Now suppose that process A existed on host-1 and that it wanted to initiate B and to open a connection between the two processes. The steps necessary to accomplish this are as follows. First, A creates a process-to-process connection "/host-2=AtoB" using the operation *create*. The *create* operation makes the connection name AtoB defined from /host-1 (where A is located) known in host 2. Second, A initiates the process B on host 2 using the conventional UNIX *exec* operation on a remote file (e.g., "/host-2/B"). B is passed the connection reference "/host-1=AtoB" as a program argument. Process B opens the connection reference which has been passed using the *open* operation. Now, messages can be sent back and forth over the connection. The state of the system after each operation is shown in figure 2. Connections are closed by calling the *close* operation.

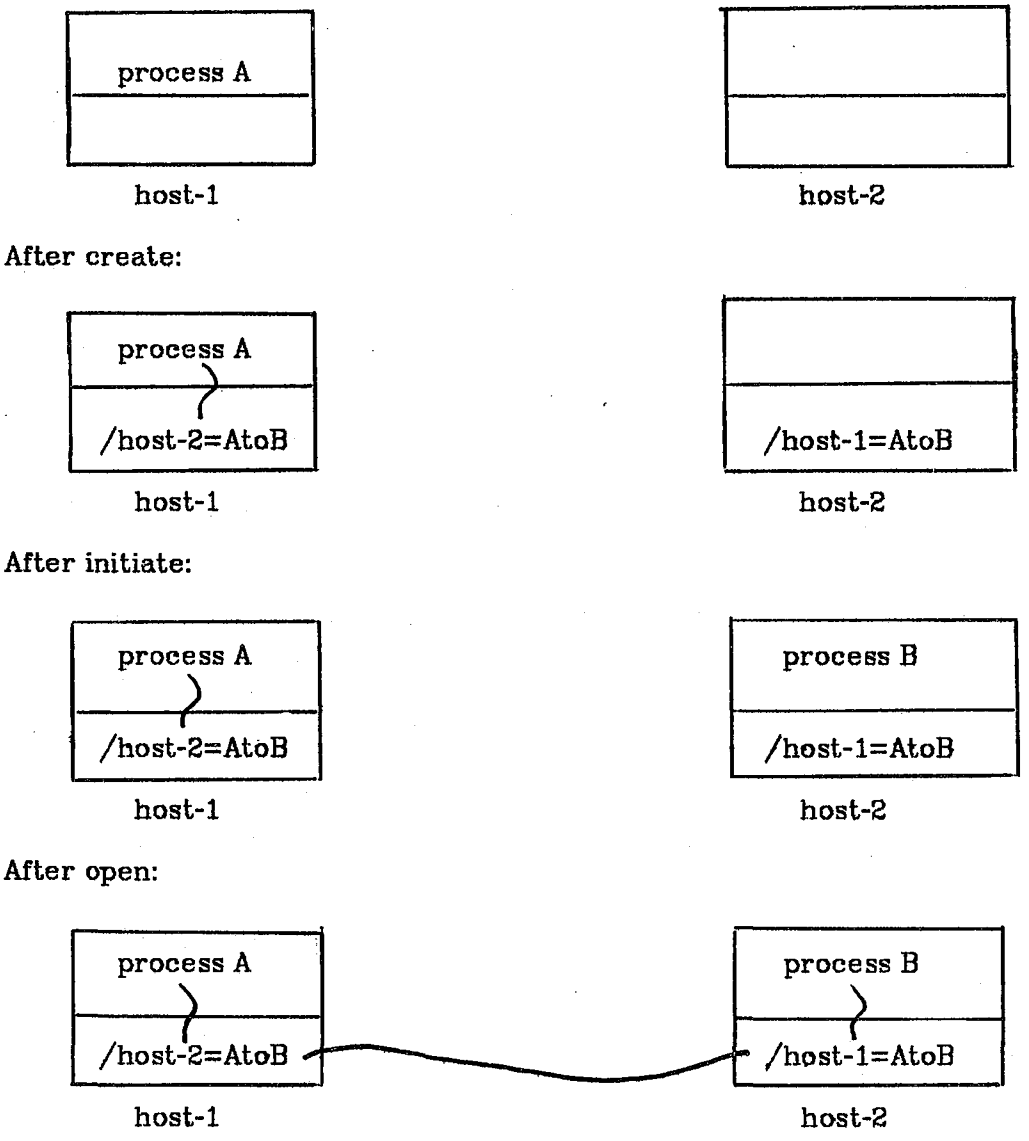


Figure 2. Connection Establishment

Multicast connections are established in the same way except that a *multiCreate* operation is called which accepts a list of destinations rather than just one. After a destination process opens a multicast connection, messages can be received but not sent on that connection by the destination process. Multicast connections are inherently one-way.

Most applications which use a multicast connection will also open a process-to-process connection back to the multicast source. The reference for the process-to-process connection can be passed to the destination process either as a program argument or in a message sent over the multicast connection.

Operations are also provided to add a new destination process to an existing multicast connection (*addMultiDest*) and to change the subset of processes which are to receive messages sent on the multicast connection (*changeMultiSubset*).

4. MULTICAST PROTOCOL IMPLEMENTATION

This section describes the implementation of the multicast protocol. The message-oriented IPC protocols described in the previous section which constitute the user programming environment are called the *network access protocols*. These protocols are implemented within a particular network by a lower level protocol called an *intranet protocol*. The first subsection describes an intranet implementation of the multicast protocol in a local network composed of LNI's. The second subsection discusses some hardware changes to improve the reliability of an LNI-based network. The last subsection describes how the multicast protocol can be extended across network boundaries.

Given a multicast connection between a source process and n destination processes, how can a message be sent to a subset of those destinations, say, $m < n$ destination processes? One approach is to send m separate messages, one to each destination in the subset. In a local network this approach is impractical for moderate sized m because the communication overhead is high and the elapsed time to send that many messages can be long.

Another approach is to send one message that will be received by all n destinations. Some indication of which m destinations are actually to receive the message can be encoded into the message header (e.g., a variable length address field or a fixed length bit string). Then, either network software or each destination process can determine whether the message should be accepted or discarded. This method can be implemented in a broadcast network such as DCS [Farber 73] or Ethernet [Metcalfe 76] assuming that some form of destination address matching is performed in the network interface. This approach is practical in those applications where m / n approaches 1.

On the other hand, if m / n is small, as might be the case in a distributed data base system, other factors may cause this approach to be impractical. First, $n - m$ hosts will receive a message which is not addressed to one of their processes. These hosts must service an interrupt and scan the message header to determine if the message should be discarded (the scan operation will probably not be in the interrupt routine which means the network control program must process the message). Second, as the number of hosts which are to receive a multicast increases, the probability that some hosts will fail to receive it increases. Thus, the message will have to be retransmitted.

A better approach to sending a message to a subset of a multicast connection is to send one message in such a way that only hosts with one of the processes in the subset actually receive the message. The associative address matching capability of the LNI can be used to implement this approach. A fixed length bit string representation of the multiple destination addresses is encoded into the addresses of multicast channels.[4] The LNI address matching can then be used to determine whether a particular message is addressed to a process in the attached host.

4.1. Intranet Multicast Implementation

The LNI provides very flexible message addressing in the hardware. Each interface has a table of addresses to which it will respond.[5] The destination address in a message is associatively matched with the entries in the name table as the message passes the interface. If the message address matches an entry in the table, the message is copied into the attached host. Two masks, one in the message and one stored with each name table entry, can restrict the matching to arbitrary subfields of the addresses. If either mask bit is set, the corresponding bits in the addresses need not match. Several examples are given in figure 3. The current version of the LNI has 32 bit addresses and 16 name table entries. These apparent hardware limitations are discussed in more detail below.

Figure 4 shows the format of multicast addresses. The OHN and CHAN fields, taken together, uniquely determine the multicast channel. Each multicast destination corresponds to a bit in the SUB field. The destination's name table mask restricts the SUB field comparison to the assigned bit. For example, suppose the multicast channel and destinations were as depicted in figure 5 (only the SUB field is shown and it is limited to 6 bits for simplicity). To send a message to destinations at sites 2 and 4, it is addressed to 010100. Only those sites will match the address (the mask in the message itself is set to zero). Thus, only the sites which are supposed to receive the message do receive it.

Name Table

entry	addresses	masks
1	00111010	00000000
2	11010100	00000011
3	11000010	00000101
4	00011000	00000000

Incoming Message

dest. address	dest. mask	remarks
00111010	00000000	matches entry 1
11010011	00000000	does not match
11000111	00000000	matches entry 3
00001000	00010111	matches entry 4

Figure 3. Address Matching Examples

[4] A channel is the intranet analog of a connection.

[5] For historical reasons, this table is called a *name table*.

Field	Size (bits)	Use
TYPE	2	Specifies type of message. 0 broadcast 1 process-to-process 2 multicast 3 unassigned
OHN	6	Originating host number on this network.
CHAN	8	Data transmission channel number. OHN and CHAN fields together uniquely determine the multicast channel in this network.
SUB	16	Selects subset of processes on this network to receive the message.

Figure 4. Multicast Channel Address Format

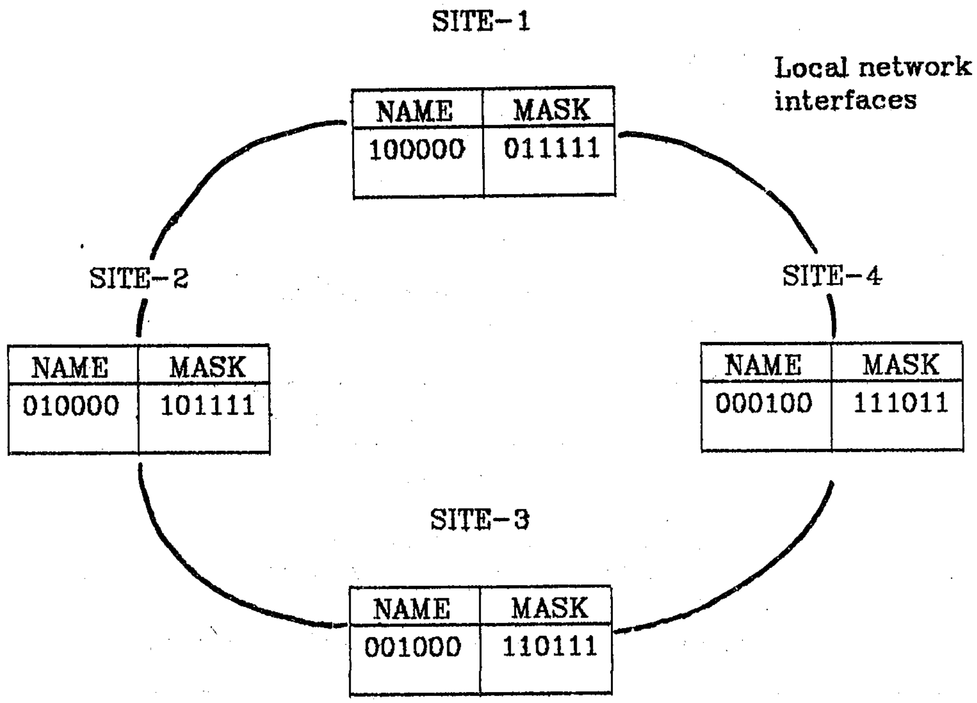


Figure 5. Subset Addressing Example

New destination processes can be added dynamically to a multicast channel by assigning the next available bit in the SUB field to the process and entering the name in the destination processes' host LNI. These actions are taken in response to the operations *addMultiDest* and *open* called, respectively, by the multicast source and destination processes.

The LNI and the implementation described in this subsection appear to have four limitations:

1. an LNI name table is limited to 16 entries,

2. a host is limited to 256 (2^8) open multicast channels which it originated,

3. a local network is limited to 64 hosts, and

4. a multicast channel is limited to 16 destination processes. [6]

These limitations are discussed in the following paragraphs.

The prototype LNI's have been implemented in TTL medium scale integration. The design of the interfaces was biased towards a large scale integration

[6] A channel is limited to 15 destinations if the sequence value is moved into the name as suggested in the next subsection.

(LSI) implementation and allows variation in the number of bits per address and the number of entries in an LNI name table [Mockapetris 78]. Consequently, the limited number of name table entries holds only for the interfaces we have ordered.[7] In a reasonable sized network an LNI implemented in LSI will have several hundred name table entries. The second and third limitations are caused by the 4 octet (8 bit bytes) address size. These limitations can be relaxed by enlarging the address by several octets (e.g., to 6 or 8 octets).

The last limitation, the number of destination processes on a multicast channel, can be avoided by using more than one channel to implement a multicast connection, i.e., one multicast connection may have several intranet channels. To send a message may require that two or more copies of the message be transmitted around the ring.[8] Another approach to avoid this limit is to have longer addresses. While this approach works, it is not too practical because the number of destinations increases only by one for each additional address bit. Nevertheless, for some applications this approach may be optimal. A completely different approach to the problem is to allow variable length addresses and to have the network interface recognize the more complex format. The point is that several low-level implementations exist and it remains to be seen which will be cost effective for different applications.

4.2. LNI Hardware Improvements

This subsection discusses two LNI hardware changes that would improve network reliability in the areas of multicast channel message sequencing and network robustness in the presence of failures.

To guarantee that the most recently received message is not a copy of the previous message, a one bit sequence value is used. The network software in each host anticipates the sequence value of the next message to be received. If a transmission failure occurs, the source host retransmits the message with the same sequence value. By comparing the sequence value in the message with the sequence value in the last correctly received message on that channel, a destination host can determine if the message is a copy of the last one received. If so, it is discarded. Otherwise, the sequence value is updated and the message is passed to the destination process.

This technique works as long as the sequence values for the destinations remain synchronized. However, because each host does not receive every message on a multicast channel, the values must be synchronized prior to changing the addressed subset. For this reason, before any subset change, a sequence bit synchronize message is sent to the old subset to clear the sequence value (i.e., set to zero). Thus, the sequence value on the first transmission is guaranteed to be cleared.

Another problem with multicast message sequencing arises because hosts which are temporarily busy may not accept messages [Rowe 75]. The LNI associates with each message as it is transmitted around the ring two status bits, called the match and accept bits. If a message is copied into a host a one is ORed into the accept bit. If the destination address was matched but the message could not be copied, a one is ORed into the match bit. These status bits indicate to the transmitting host whether the message was received by all destinations. The possible status bit settings and their meanings are summarized in figure 6.

[7] Three devices have been ordered. We chose 16 entries because of cost considerations and because the first experiments with COCANET will be small.

[8] Dynamic assignment of destinations to channels may reduce the number of transmissions required if some subsets account for a high volume of traffic.

MATCH	ACCEPT	MEANING
0	0	The message was addressed to a non-existent process; no LNI recognized the message.
0	1	The message was transmitted to one or more processes; at least one LNI copied the message.
1	0	The message was recognized by one or more processes, but no LNI could copy the message.
1	1	The message was transmitted to at least one process, but at least one LNI recognized the message but could not copy it.

Figure 6. Status Bits Results

Suppose that a multicast message is to be sent to several sites and that the first time the message is sent the status bits returned are match-accept. In other words, some sites received it and some did not. If subsequent retransmission also results in match-accept status, the transmitting host is unable to determine whether all sites have received the message. For example, a site which received the first transmission could fail to receive the second, and a site which failed to receive the first transmission may have successfully received the second. This problem results from the fact that two bits are being used to describe the status of multiple destinations.

This problem can be eliminated by moving the sequence value into the address in the LNI name table. When a message is accepted, the sequence value in the name table is flipped. Now, if the message must be retransmitted, it will not match and consequently will not be accepted by hosts which successfully received the first transmission. For multicast messages this means that only those sites which did not accept the first transmission will match the retransmission. Thus, the match bit can be interpreted as indicating continued failure to copy the message into some addressed host(s).

Although this solution can be implemented with the current LNI hardware, it may be impractical because the sequence bit must be flipped by software. Redesign of the LNI, however, could lead to efficient support of the multicast sequencing mechanism.

The second problem with the LNI hardware is that all addresses in a name table are unusable if a host crashes. As a message passes through the interface the address will be matched, a copy will be attempted (if appropriate), and the match and accept bits will be set regardless of whether the host operating system is functioning normally. Some mechanism is needed to clear the name table either from another host on the ring, from a wire center [Saltzer 79], or from monitoring hardware at the failed host [Kunzelman 78].

4.3. Internet Support of a Multicast Protocol

This subsection describes one way to support the multicast protocol across several networks. A hierarchical address space for interconnected networks is assumed, e.g., hosts have addresses of the form "NET:HOST". It should be noted that distributed INGRES must know about the network topology because the time required to send messages between the various processes (from the master to the individual slaves, between slaves, and from individual slaves to the master) influences the query optimization algorithm.

A multicast message is sent to destinations on a foreign network by sending one copy of the message to a gateway process that is responsible for forwarding the message to all destinations which can be reached through the gateway. To illustrate this idea, consider the networks in figure 7. Source process S in a COCANET wants to send a message to processes D1 and D2 in the same network and processes D3 and D4 in another network. One copy of the message is sent to P which forwards the message to D3 and D4 using whatever multicast implementation is best in the foreign network. Subset addressing can be supported either by encoding destinations in a message header or by having P maintain an active destination list.

Messages propagate across several networks in the same way. One copy of the message is sent to a gateway which sends it to the local destinations and to the next gateway(s). If any gateway process is unable to deliver the message to

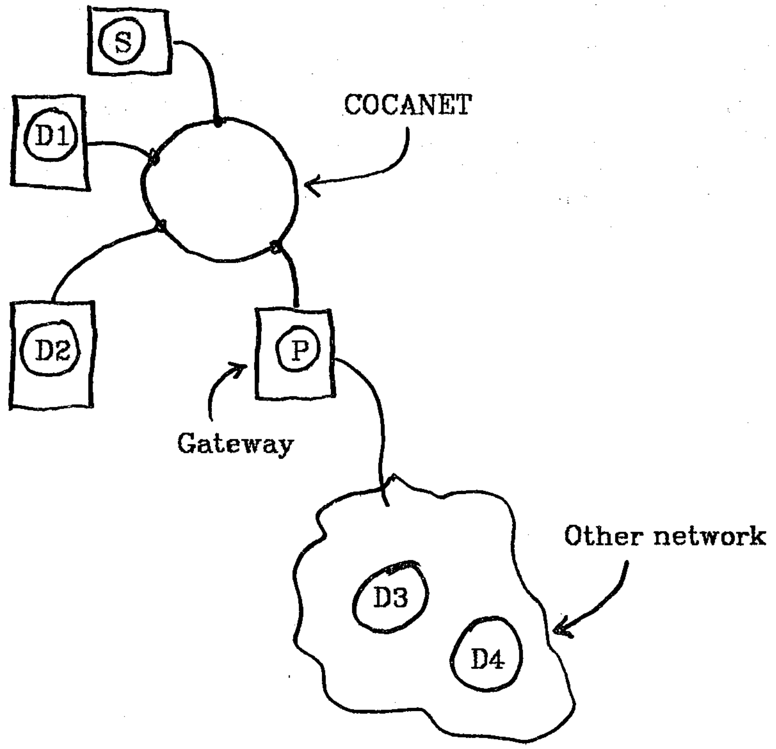


Figure 7. Internetwork Multicast Example

a foreign destination, a control message is sent back to the source host which causes the multicast connection status to be set to an error state. Because the multicast protocol does not guarantee delivery, this action should be taken only when a catastrophic failure occurs. No explicit flow-control mechanism is assumed other than that provided to control transmission through the gateway.

5. SUMMARY

This paper describes the design of a multicast communication protocol for a distributed data base system. The protocol includes a novel subset addressing capability and the ability to add new processes to a connection which is already open. An efficient implementation of the protocol in a local computer network composed of LNI hardware was also presented. In addition to the multicast protocol, the UNIX programming environment has been extended across multiple hosts to support general resource sharing.

The current status of the prototype COCANET is that the network software has been debugged in a single machine environment which simulates the sending and receiving of messages. The prototype should be fully operational shortly after we receive the network interfaces.

ACKNOWLEDGMENTS

Paul Mockapetris proposed the idea of moving the multicast channel sequence value into the LNI name table described in section 4. We also want to acknowledge the contributions made by our colleagues on the INGRES project, notably Bob Epstein, Mike Stonebraker, and Mike Ubell. We want to thank the anonymous referees whose comments on an earlier draft of this paper were very helpful. Finally, we want to thank Dennis Hall whose support made this project possible.

REFERENCES

[Birman 79] K. Birman. COCANET: a resource sharing network. M.S. Th., Dept. Elec. Eng. and Comp. Sci., U.C. Berk., forthcoming.

[Chesson 75] G. Chesson. The network UNIX system. *Proc. 5th Symp. Op. Sys. Prin.*, Op. Sys. Rev., vol 9, no. 5 (special issue), pp. 60-66.

[Dalal 78] Y. Dalal. Reverse path forwarding of broadcast packets. *Comm. of the ACM*, vol. 21, no. 12 (Dec. 1978), pp. 1040-1048.

[Epstein 78] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. *Proc. 1978 SIGMOD Conf.* (May 1978), pp. 169-180.

[Epstein 79] R. Epstein. Query processing techniques for distributed relational database systems. Ph.D. Diss., Dept. Elec. Eng. and Comp. Sci., U.C. Berk., forthcoming.

[Farber 73] D. Farber, et. al. The distributed computing system. *Proc. 7th Ann. IEEE Comp. Soc. Int. Conf.* (Feb. 1973), pp. 31-34.

[Kunzelman 78] R. Kunzelman, et. al. Progress report on packet radio experimental network. Quarterly Tech. Rep. 14, SRI Proj. 6933 (Nov. 1978).

[Metcalfe 76] R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Comm. of the ACM*, vol. 19, no. 7 (Jul. 1976), pp. 395-404.

[McQuillan 78} J. McQuillan. Enhanced message addressing capabilities for computer networks. *Proc. IEEE*, vol. 66, no. 11 (Nov. 1978), pp. 1517-1527.

[Mockapetris 77] P. Mockapetris, M. Lyle, and D. Farber. On the design of local network interfaces. *Proc. 1977 IFIP Cong.*, vol. 77 (Aug. 1977), pp. 427-430.

[Mockapetris 78] P. Mockapetris. Design considerations and implementation of the ARPA LNI name table. Tech. Rep. 92, Dept. Info. and Comp. Sci., U.C. Irvine (revised May 1978).

[Ries 78] D. Ries and R. Epstein. Evaluation of distributed criteria for distributed data base systems. U.C. Berk., Elec. Res. Lab., Memo. M78/22 (May 1978).

[Ritchie 78] D. Ritchie and K. Thompson. The UNIX timesharing system. *Bell Sys. Tech. J.*, vol. 57, no. 6, part 2 (July-Aug. 1978), pp. 1905-1930.

[Rothnie 77] J. Rothnie, and N. Goodman. A survey of research and development in distributed database systems. *Proc. 3rd Int. Conf. Very Large Data Bases* (Oct. 1977).

[Rowe 75]. L. Rowe. The distributed computing operating system. Tech. Rep. 66, Dept. Info. and Comp. Sci., U.C. Irvine (Jun. 1975).

[Saltzer 79] J. Saltzer and K. Pogran. A star-shaped ring network. To appear in *Proc. Local Area Comm. Network Symp.*, MITRE Corp and Nat. Bur. of Stand. (May 1979). Also available as Network Imp. Note 4, MIT Lab. Comp. Sci. (Feb. 1979).

[Stonebraker 77] M. Stonebraker, and E. Neuhold. A distributed data base version of INGRES. *Proc. 1977 Berk. Workshop on Dist. Data Man. and Comp. Net.* (May 1977).

[Stonebraker 79a] M. Stonebraker. MUFFIN: A distributed data base machine. Unpublished paper.

[Stonebraker 79b] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Tran. on Soft. Eng.*, vol. SE-5, no. 3 (May 1979), pp. 188-194.

[Sunshine 77] C. Sunshine. Interprocess communication extensions for the UNIX operating system: I design considerations. R-2064/1AF, Rand Corp. (Jun. 1977).

TRANSACTION PROCESSING IN THE DISTRIBUTED DBMS-POREL

U. Fauser
E.J. Neuhold

Institut für Informatik
University of Stuttgart
Azenbergstr. 12
D-7000 Stuttgart 1
Fed. Rep. of Germany

Abstract

We first give a short description of the architecture of the DDBMS-POREL which we are currently implementing at the University of Stuttgart. Afterwards we study in some detail the transaction processing control, the Execution Monitor, which at runtime is responsible for the administration and control of the transaction flow in the network. Each transaction in the DBMS is associated with a specific Execution Monitor which completely supervises its processing and initiates and executes the necessary synchronisation, resource handling and recovery preparation.

1. INTRODUCTION

POREL is a decision support and data base management system based on a distributed relational data base designed to run on a heterogeneous network of minicomputers. POREL ignores the possible existence of other local data base systems and makes no attempt to integrate them into the distributed data base.

Therefore a totally new system had to be designed including user interfaces, languages, synchronization aspects, network features, authorization mechanisms, integrity features and the data base machines themselves.

The word "heterogeneous" in our system therefore stems from the underlying computer network which contains different hardware and also different operating systems. But upon this inhomogeneity a homogeneous DB system is built up.

We achieve acceptable independency from the base systems by choosing the high level language PASCAL as one of the implementation tools. However, PASCAL on one machine is not PASCAL on another. We therefore take an existing, easily portable, PASCAL system and install it on every new machine in our network. Of course some efforts must be put in for rewriting the code generation phase and the runtime routines, but in this way we have access to the same language interface everywhere. The still necessary assembly written routines are kept simple and small in number as they have to be redone on every new machine category. Through this approach the user sees only a single data base system. In such a unified and integrated system he does not need to know where data are kept or programs are executed. That is the DDBMS presents itself like a centralized system.

POREL supports three interfaces for the data base user:

- A Relational Data Base Language - RDBL - a non procedural, algebra oriented interactive language for data definition, manipulation and control (see[10]: 78/5);
- A Host Language (PASCAL or FORTRAN) with RDBL as data language, whereby RDBL has been extended with a cursor concept for navigating in a tuple at a time logic through relations similar to the solution which is found in SEQUEL 2 (see[10]: 78/8).
- A problem solving decision support system which provides the user with an environment adapted to his application area (see [15]).

A detailed discussion of these interfaces is beyond the scope of this paper and not necessary for an understanding. Actually according to the single system philosophy, the chosen interfaces are quite independent from a distributed environment, they could also be found in a centralized data base. For further study the interested reader is referred to the indicated references.

The whole data base system is based on a set of relations stored on the different nodes. If necessary even single relations may be distributed over the network, however only horizontal splitting is supported.

As the long distance communication is much slower than local access and communications costs are quite significant, the effectiveness of a distributed system directly depends on the amount of information interchange. This network traffic carefully has to be minimized ([2], [6], [8]).

To improve data access in retrieval queries, copies of relations or parts of it may be kept all looking identical for the user. But care has to be taken with update sensitive data as update traffic is increased and networkwide consistency is more complicated to achieve.

In our solution one of the copies is selected as original and always used for the updates. Other copies may be behind in various stages (delayed update). A mechanism based on version numbers keeps track of the actual state. This eventually may lead to a situation where, when a copy is to be used, it is easier to replace the copy by the original than to effect all outstanding updates.

All this operating information as well as the relation and database descriptions are kept in several system catalogues (see [5], [10]: 78/4).

The catalogue organization has been adapted to the partitioning of the analysis phase into two parts by providing for a network independent (NUA) phase a so called short catalogue and for the network oriented phase (NOA) the long catalogue. For the NUA the (short) catalogue is copied onto each site, giving fast local access to avoid a possible data transmission bottleneck in such a system. In the case of changes in the catalogue this redundancy will cause considerable overhead, but the data contained in the short catalogue are chosen to be of rather static nature. The long catalogue information for the NOA phase is associated with the corresponding data locations, i.e. stored wherever a part of the described data is located (original or copy) and therefore not always locally available.

2. POREL SYSTEM ARCHITECTURE - AN OVERVIEW

Before we start to explain the functions of the Execution Monitor in detail we give a short introduction into the different components of POREL and illustrate their function.

Figure 1 shows a process oriented representation of the architecture, as it would be seen by a user with deeper knowledge of what is going on below him. The figure does not show the multi-user, multi-process environment which actually exists. That is, more than one dialog process, more than one network independent analyzer etc. may run in parallel at the same time. However, each node in the network has only one execution monitor, catalogue manager, scheduler and lockhandler.

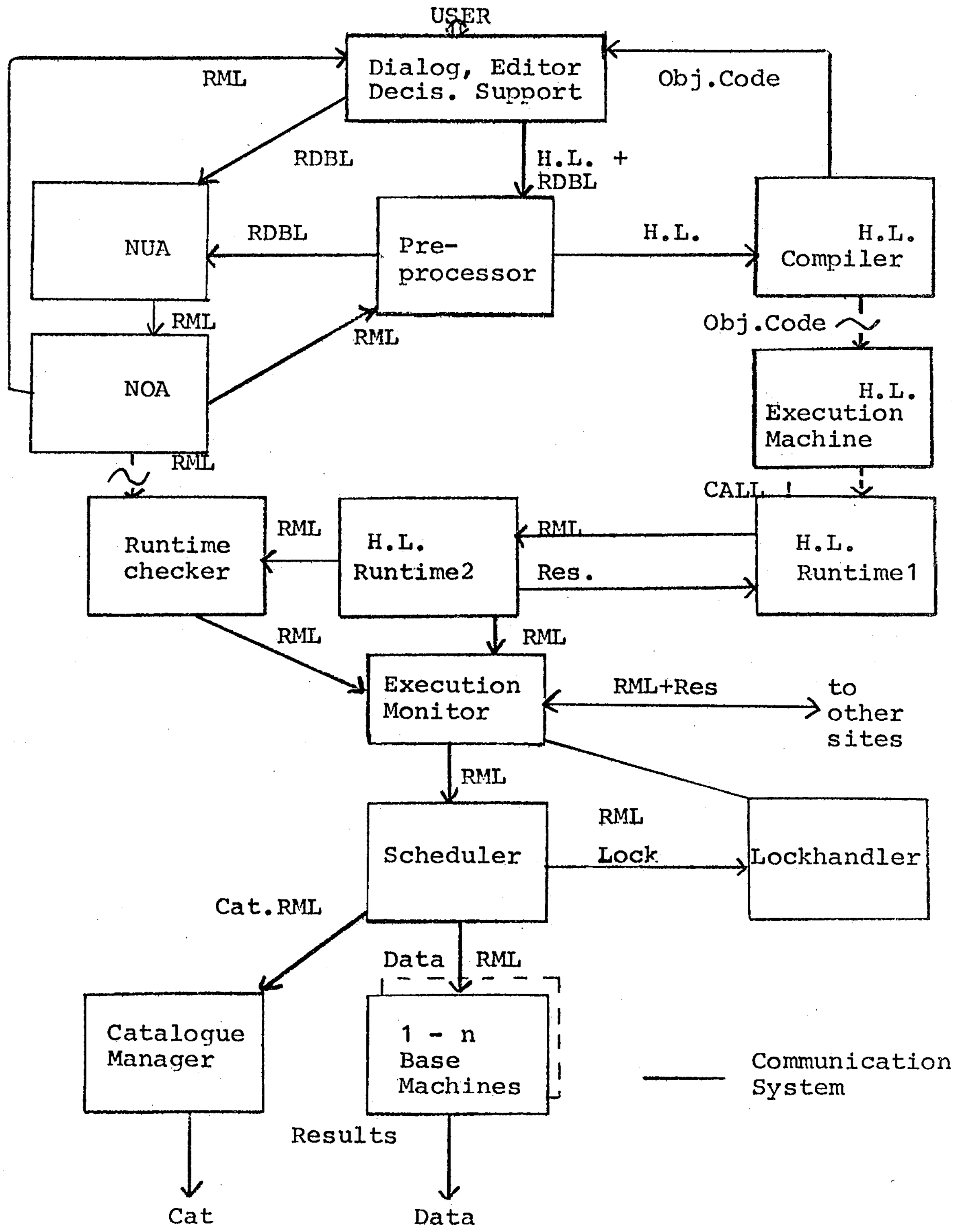


Figure 1: The Process Structure of POREL

Now let us explain some important modules (for details see [3], [7], [10]).

The command and dialog level accepts all inputs from the user, gives helping hints in case of troubles, routes information to the chosen interfaces and formats the results on the user display.

The Network Independent Analysis NUA (see 10 : 10/7) is in the conventional sense a compiling system which gets as input all RDBL source language statements from the user interfaces (RDBL stand alone, host language, decision support). It has to do all work which can be done without knowledge of the underlying distributed system, i.e. all information is locally available (short catalogues) and compiling actions cause no data communication between sites.

NUA's duties are:

- Syntactical and semantical analysis
- Generation of code in the relational machine language (RML) without regard to network or distribution aspects (RML is an algebra oriented nonprocedural data base language)
- Code optimization
- Insertion of integrity constraints and trigger functions (see [16])
- Analysis for possibly parallel executable parts in a transaction
- Separation of necessary access rights for later authorization checking
- Insertion of synchronization and timing control
- Delivery of program status and user statistical information

The Network Oriented Analysis (NOA) (see [10]: 78/10,6) level belongs logically to the compile time actions, but was separated from the NUA as it covers now the network and distribution aspects of the DBMS and therefore needs information possibly distributed in the network, so causing some system data transfer traffic.

NOA's duties are:

- Looking for the network locations of data used by the transaction
- Collecting the long catalogue information for each used relation and user authorization data
- Checking authorization statically
- Doing more necessary breaking up and parallelization of object code, inserting more timing, locking and synchronization control commands

- Searching for the optimal execution place by minimizing data transfer.

The Runtime Checker (NUA2 and NOA2) fulfills duties which are taken from the NUA as well as from NOA. It effects its control only in case of a time delay between translation and execution (not compile-and-go mode). In this case the generated code and information has to be compared against the actual state of the DB for any possible changes affecting the correct execution of the code.

The Execution Monitor (see [10]: 78/11) is responsible for the timing control, the synchronization and administration during the execution phase of the different parts of a transaction. It also has to provide necessary information for recovery purposes. It is the communication partner for all other monitors in the network.

The monitor's duties are:

- Distribution of the transaction parts according to their destination in the network
- Timing control by interpreting the connecting WAIT-statements
- Reacting to incoming control messages giving the state of an execution, error conditions, rejected requests etc.
- Global lock control with voluntary preemption in case of a possible deadlock situation
- Providing global recovery information
- Starting recovery analysis in erroneous situations
- Communicating with other monitors

The Scheduler distinguishes the different requests given to it and routes them to the appropriate modules:
lock request to the lockhandler
catalogue request to the catalogue manager
normal user request to one of the n base machines the scheduler has under its control.
System internal requests are thereby treated with higher priority to guarantee faster execution.

The lockhandler controls lock information for all local data and grants or rejects incoming requests.

The Catalogue Manager is responsible for maintaining all system catalogues (short and long form) in a consistent state and for providing all required information for the system modules.

The Base Machines (see 10 : 78/12,9, 12) are the most important parts in the system as they represent the local data management system. They process all requested manipulations on the local relations, provide results for the user and status messages for system modules and keep local recovery information and transaction reset facilities. A base machine has no data access to the network, all involved data for the executed partial transaction have to reside on the local node.

The Communication System (see [10]: 78/13) has to satisfy all process communication requests in the whole database, independently on whether they are local or global. It has to transmit data and control information, do the necessary conversions of data representation and it has to supervise the successful and errorfree execution of the communication protocols. In case of an error situation it has to give control back to the requesting process. The communication system uses the planned X.25 network of the German PTT and some higher protocol levels which have been worked out in cooperation with the PIX group. However until X.25 actually becomes available we use a terminal simulation technique for interconnecting our computers.

3. THE EXECUTION MONITOR-STRUCTURE AND ORGANIZATION

The user transaction on its way through NUA, NOA and/or the host language system was up to the moment of arrival at the Execution Monitor only analyzed locally at the user's node. Of course some networkwide information (system catalogues) had already to be used to satisfy the needs of some of these modules, but no execution took place.

When a transaction enters the monitor its runtime phase starts. This monitor, the controlling monitor, now is responsible during the total lifetime of the transaction for timing control and other organizational events connected with this transaction. Lifetime of a transaction means all processing from entering the monitor module until the final actions following the END-TRANSACTION statement are executed.

To perform this duty the monitor takes a central place in the system architecture where the actual processing activity starts. He governs all distribution aspects. That is, it distributes with the aid of the interprocess communication system the transaction parts in the network. and supervises their execution.

The distribution processing is based on input data coming from different sources. Most of the processing requests originate from the NOA phase of the local node in form of transaction parts. Other requests are sent from monitors of remote sites. These requests are administrated by the local monitor during their local lifetime, but still the remote monitor supervising the original transaction has overall control.

A second equally importand input stream is represented by the status control messages coming from other monitors as well as from the local base machines for guiding the organizational supervision of the transaction processing. To allow this control all transaction input to the monitor is organized in two parts: one part containing monitor control statements and a second part containing object code for the base level. This part is not affected by monitor actions, whereas the monitor control statements will be processed by the monitor directly.

Those transaction parts coming from the local NOA are distributed in the network according to the control information which has been determined by NOA as an optimal execution strategy. Those transaction parts coming from remote monitors are immediately routed to the local base machine level.

The different parts of a transaction have to be processed in a predefined sequence to get the requested results. For that reason all transaction parts have WAIT-instructions in the control part to identify these synchronization order. The monitor evaluates the WAIT conditions and realizes through them the proper execution sequencing.

Only if all WAIT conditions of a transaction part are fulfilled the executable code is routed to its destination. If the receiver happens to be a remote monitor then the WAIT instruction is substituted by an empty WAIT as its execution can start immediately at the destination site as all timing constraints have already been removed.

In the WAIT instruction the timing interconnection between transaction parts is established by specifying a number of transaction part names whose termination must be awaited before the execution of RML code belonging to that WAIT part can be started.

Example	WAIT	PT11	PT8
	WAIT	PT11	PT9
	WAIT	PT11	PT10

This means that the actual transaction part named PT11 cannot be released by the monitor for execution before the end signals of the parts PT8, PT9 and PT10 have been acknowledged at the controlling monitor.

Each transaction is a closed unit of consistency and integrity and it begins normally with a transaction part without WAIT conditions, indicating that the initialization part can be started immediately. In the initialization part, according to the transaction building rules, the used (sub-)relations and the access modes have to be requested. To control and synchronize resource usage we have chosen a preclaiming strategy to guarantee simple reset handling in case of threatened deadlocks. Therefore all data (subrelations) used in the transaction are requested before execution begins. This happens in the LOCK-phase which is always incorporated in the first part of a transaction. Its correct execution is controlled by the monitor. All single requests in the lockphase are sent directly to the local base level or via the remote monitors to the lockhandlers of the affected nodes. The addressed lockhandler checks the request and depending on already existing locks sends back a message to the originating monitor: positive if the request can be granted, negative if the requested lock mode is not compatible with locks already granted for other users.

Even if only one of the lock requests is rejected, the monitor has to release voluntarily all already granted requests to avoid possible deadlocks in the network as the result of cyclic waiting requests. Releasing of granted locks can still take place without special precautions because the actual transaction execution has not yet started. Data manipulations can only start after all requested data locks are granted.

Nevertheless a transaction may still have to wait very long, without actually being deadlocked, until all requests are granted, especially if it requires many data, and consequently, in a heavy multiuser environment, always one or more requests are rejected. We are aware of this problem which can be solved by introducing a priority schedule or ordering of lock requests but in the first version of POREL this aspect has not been included. The monitor simply repeats rejected lock phases from time to time until the time all requests are satisfied. In a system not too heavily loaded (experimental phase) this strategy should present no actual problem.

UNLOCK statements in the closing control component of a transaction cause no further coordination, they are just routed to the responsible lockhandlers to free the data.

Transaction parts without LOCK/UNLOCK statements are within their synchronization restrictions not further controlled by the monitor.

The state of all transaction parts is stored in an internal control state table of the monitor, as it has to know the execution status of each part to guarantee correct execution. This internal table provides additional information for status requests, for routine error controls (e.g. time out failures) as well as for the initialization of recovery mechanisms in erroneous situations.

Some requests (catalogue handling) are transferred to the monitor as priority messages by setting the priority flag in the routines of the communication system. In the monitor's input queue those requests are therefore processed first. But during processing of a non priority request an incoming priority message does not deactivate this activity i.e. there is no interrupt facility, it has to wait until the input queue is searched for new demands.

4. STATUS AND CONTROL MESSAGES

The following group of messages for the monitor contains status and control information coming from the various base machines, either local or remote. They are the result of state changing actions or illegal requests and are used for updating the monitors' internal tables.

a) Termination Message

The processing of a transaction part in the base level has successfully ended. If it concerns the last part of a transaction, i.e. the whole transaction has now terminated, all information in the internal tables is erased, later status requests result in "transaction unknown". A later reset of such a terminated transaction by the monitor is not possible. Only transaction recovery during lifetime is supported by the monitor. More generalized recovery mechanisms using checkpoints and log information have to be used for such purposes.

If merely an internal transaction part is terminated, the state description in the internal table is changed to FINISHED.

b) Zero Message

This identifies the processed transaction part as meaningless, i.e. it has had no effects in the base level. This can occur in case that one cannot determine in advance where to actually process a request because of insufficient information. The NOA therefore has to distribute the transaction on all nodes where data it could possibly need are stored. Example: An employee relation is distributed according to the department membership; all departments have their own subrelation.

```
DISTRIBUTE   PERSON

        SUBSET ADMINISTRATION   ON   K1
        WHERE DEPT = 3
        SUBSET FOOD             ON   K2
        WHERE DEPT = 2
                  .
                  .
                  .
```

A user then formulates a request not containing the distribution criterion.

```
SELECT PLACE, ADDRESS FROM PERSON
       WHERE NAME = 'HENRY MEYER'
```

The NOA cannot determine where to process this request and must create parallel requests searching all subrelations of PERSON in the network.

Normally only one of these requests ends up successfully. The others are for this request meaningless, i.e. they end up with a zero message which is treated like a termination message. It actually affects only recovery situations where such requests need not to be recovered or repeated.

c) Additional Message

Means that a transaction part has terminated, but during processing has realized that due to changes in data fields belonging to the distribution criterion the actual result tuples no longer can stay on the execution node. The executing base machine has no possibility to effect control on other nodes directly, it can only collect the affected data and give them back to the monitor with some transaction code prepared by an invocation of proper NOA components.

For the monitor this indicates the end of the active partial transaction but the internal tables cannot yet be searched for the next fulfilled WAIT-statement. Instead the additional code has to be placed in between and sent to the proper processing node(s). Only after the execution of these additional transaction parts has taken place the monitor can continue its normal sequence of executing the transaction. The processing of other transactions however is not affected.

d) Error Message

A transaction part has terminated in the base machine in an erroneous state (e.g. a violation of integrity constraints). Now the whole transaction has to be reset and the monitor searches in its tables for all associated parts and transfers them to the recovery module which does the actual work. The monitor forgets this transaction and sends an error message to the user.

e) LOCK/UNLOCK message

These are lockhandler ready messages following data lock or release requests. In case of a LOCK it tells whether the request was granted or rejected, in case of UNLOCK it is merely an acknowledgement that the data is released again.

There may be some status messages not destined for local monitor, these are only routed to the addressed monitors. That is we have decided that only monitors should communicate control information between different sites (for result handling see section 6.1).

For control purpose the system modules or the user may ask for the processing status of their transaction. The controlling monitor then returns as result that static information contained in his internal tables, i.e. it is not checking whether the transaction processing base machine is still alive or has failed. That is the duty of a recovery module.

5. DATA STRUCTURES IN THE MONITOR

We distinguish between incoming and outgoing data and the internal tables for keeping track of data flow and processing states in the network.

5.1 Incoming Data

After the modifications to the RML statements made by the NOA, which adds all network oriented aspects, the transaction is given to the monitor for further handling.

The internal form of a transaction is essentially represented by a sequence of quadruples together with additional tables and program control information.

Another table, the transaction structure description, provides hints which object code belongs to which partial transaction, where to process (SEND) this code, how the timing constraints look like (WAIT), and which data is to be acquired or released (LOCK/UNLOCK). The monitor is only interested in this structure description from which it gains its control information. The object code tables are not further investigated here.

The transaction structure description consists of the following elements:

a) SEND PLACE FROM TO

SEND is the code indicating where (PLACE) the code for a part transaction has to be processed. FROM-TO points to the RML object code tables.

b) WAIT OWNNAM WAITNAM

This instruction specifies that transaction part OWNNAM cannot be started before transaction part WAITNAM has signalled its termination. If more than one transaction part has to be waited for several WAIT statements have to be issued.

c) LOCK MODE SRID NODEID

For each subrelation SRID which is used in the transaction a lock request must exist indicating the necessary MODE (WRITE, weak READ, strong READ) and the location of the data in NODEID. As we have chosen a distributed preclaiming strategy all lock requests have to be stated before the actual transaction starts.

d) UNLOCK MODE SRID NODEID

This instruction releases subrelations at transaction termination. The MODE parameter is redundant and serves only for control purposes.

The transaction structure description therefore looks like:

Transaction start

```
SEND    ....
(WAIT)  ....
  .
  .
  .
LOCK    ....
  .
  .
  .
```

where to process
a WAIT statement only, if the transaction is embedded in a program consisting of several transactions which have to be processed sequentially.
for each used subrelation a lock request

Internal transaction part

```
SEND    ....
WAIT    ....
  .
  .
  .
```

for each predecessing part transaction a WAIT statement.
LOCK's cannot and may not occur (preclaiming)

Transaction end

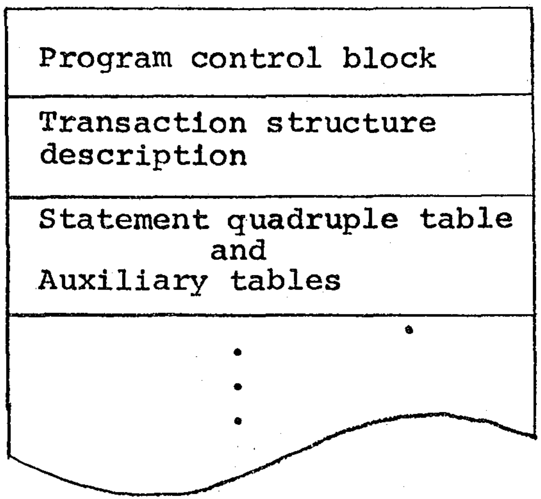
```
SEND    ....
WAIT    ....
  .
  .
  .
UNLOCK  ....
  .
  .
  .
```

corresponding to unlock

As long as all rules for constructing transactions are obeyed the monitor accepts processing requests in form of transaction parts from the host language interface, up to whole programs consisting of several transactions to be executed in sequential order.

Together with the structure description the monitor gets the program control block (PCB) containing program- and user oriented administrative data: user name, input location, data, time, message flow history, file names, message kind, etc. (see also section 6.1).

Figure 2 shows a possible storage map


Program control block
Transaction structure description
Statement quadruple table and Auxiliary tables
. . .

Comprising transaction parts forming up to several transactions.

Code for transaction part 1

transaction part 2

Depending on the size of such a block it is stored directly in the message buffer of the communication system or as files on external devices. But the monitor accesses only the PCB and the structure description and therefore a good strategy is to store the object code information on files and the monitor information in memory.

5.2 Outgoing data

They are firstly LOCK/UNLOCK statements destined for the responsible lockhandlers guarding the requested data and on the other hand complete transaction parts, e.g. all tables for transaction part 1 and the PCB, which are sent to the executing base machine possibly via other monitors. The structure description, except the LOCK/UNLOCK statements, remains in the monitor area.

5.3 Internal data

For the control of transaction parts which are to be processed, are in processing, or are already finished the monitor needs an internal table to keep track of the status changes.

Origin	Stor	User	PTA	Entr	Proc	Proctime		State	Wait
	Loc		name	time	loc	start	term		

Meaning of the fields:

Origin — is the processname which transmitted the request to the monitor, i.e. the immediate predecessor in the message flow history.

Stor loc — Where is the actual program code stored.

User — User's process name to which results and error messages are sent.

PTA name — Name of this transaction part for evaluation of WAIT's, status requests and recovery purposes

Entr time — Time of entry into the monitor.

Proc loc — Where is the code executed.

Proc time — Start: Time of activating the partial transaction on the executing node
Term: Time of receiving an end or error message.

State — Processing state of transaction part
Possible contents:

- NEW — transaction part is in monitor area but not yet started because WAIT is still invalid
- IN WORK — just under execution
- FINISHED — processing terminated
- BAD — processing cancelled in an error situation
- LOCK — just doing the lock phase
- DELAY:TIME — processing reset due to failed lock requests

WAIT/LOCK — WAIT condition, which must be fulfilled before activation. It is represented as a pointer to the structure description where also LOCK/UNLOCK information is available.

To control the lock phase the monitor keeps for each transaction the essential information in a lock table.

Transaction	Lockinfo	Nodeid	Mark

Mark indicates here whether the subrelations (lockinfo) are granted or rejected.

A similar table is used for storing already evaluated WAIT conditions.

transaction	true WAIT cond

All transaction bound information is kept till the end of processing. It is erased when the transaction has successfully terminated or is given to the recovery module in error situations.

6. ADDITIONAL REMARKS

6.1 Communication flow

In our first approach all information flow in the network was supposed to be controlled by the monitor. But this presented a serious bottleneck and we changed the approach and made the monitor responsible for only transaction control and status/control messages. All other information not destinated to these purposes are exchanged immediately between the involved processes (e.g. results of catalogue requests, query results and error messages to user or system modules).

Direct communication means that only the communication system is engaged in the transfer, the monitor is freed from that work.

For this reason all processes in the network must be uniquely identified. In order not to loose control totally such immediate interprocess information flow has to be successfully finished before the acting process delivers any depending status message to the monitor. For example, if in a transaction part an output of results to the user is made then the base machine may not send a termination message to the monitor before the result transfer is finished, i.e. has been acknowledged by the user process.

In order to always have all information necessary for communication control it must be added to all messages as a communication control block, and it is contained in the PCB. This control block is maintained by those system modules which are involved in the handling of that message on its way from origin to final destination. For the communication system these control blocks act like normal data and are not treated specially.

Necessary contents of the communication control block:

ORIG — Original sender, i.e. the process which started the message, normally a user I/O process.

ACSEN — Actual sender, the process which is currently concerned with handling the message

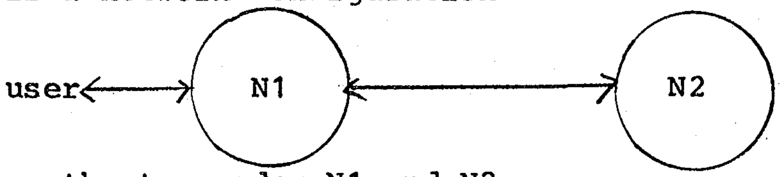
RMON — Monitor responsible for transaction control = monitor on user's input node

MKIND — Kind of transmitted message
e.g. termination
error
processing request
status request
etc.

DEST Final destination place, only necessary if a direct communication is not possible and intermediary services are required (e.g. monitor)

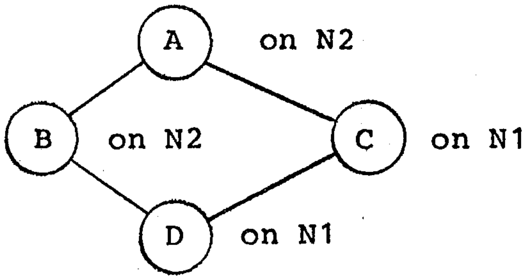
Example : Figure 4 illustrates the different actions during a transaction execution. (The line numbers in the figure will be used as reference points in the explanation below.)

Given is a network configuration incorporating the two nodes N1 and N2.



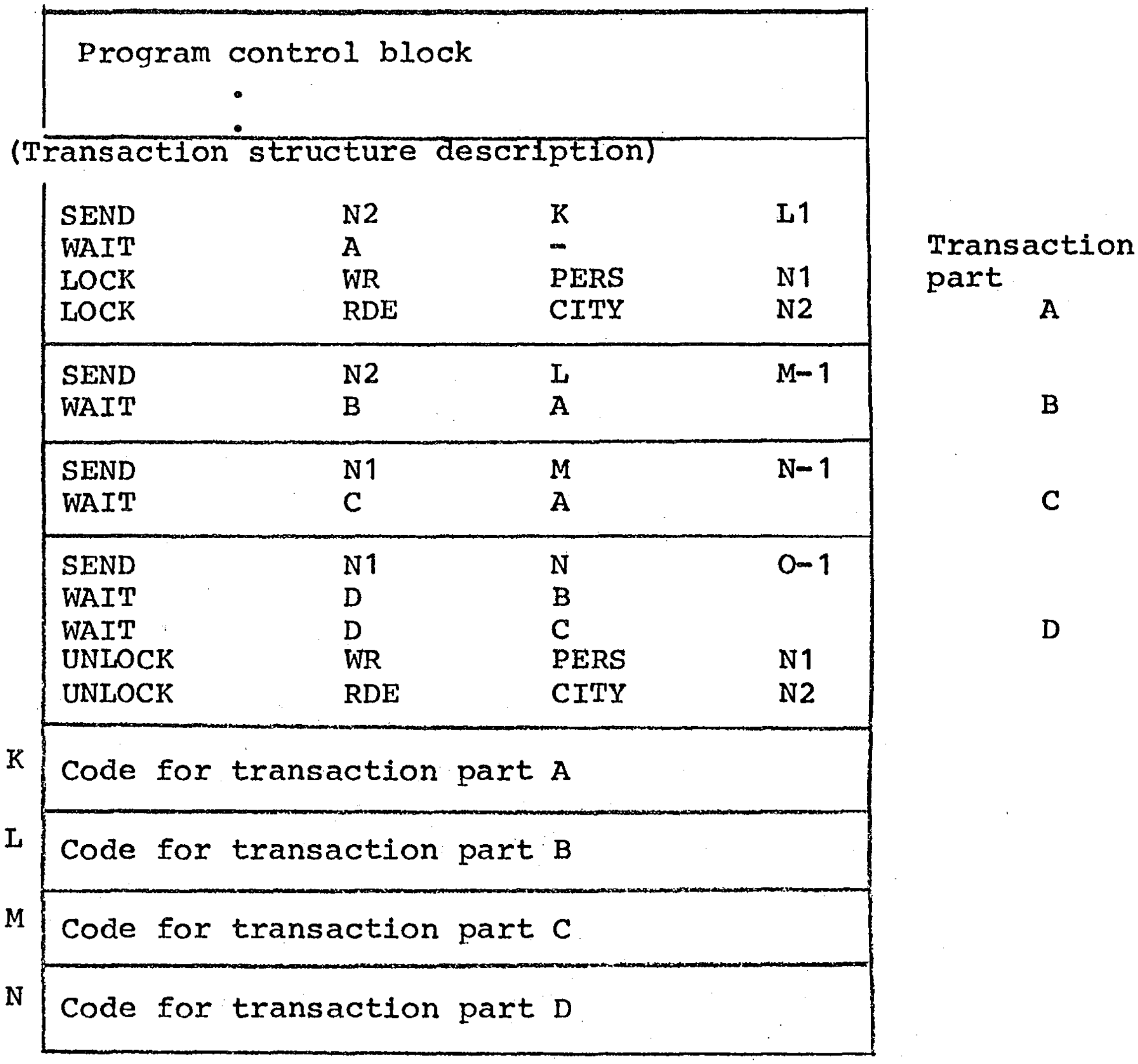
A user working with dialog process DIA1 on node N1 is starting an RDBL compilation (1). After NUA has done the syntactical and semantical processing for a non-distributed query the user request is passed to the NOA to be analyzed in the network environment (2). After this work is finished, the created object code goes back to the user as immediate execution was not identified (3).

Let us suppose the users' request results in four transaction parts A, B, C, D which are dependent as is shown here.



The associated internal transaction description is shown in Figure (3).

Later the user starts the execution phase for his query. As changes in the actual data base environment may affect a successful execution the runtime checker analyses the RML-code again (4) Let us assume no data base reorganization has taken place. Therefore the code is given to the monitor (5) which sends transaction part A to destination node 2 (6) whose monitor, as an intermediary, starts execution on base machine 2 (6) which after termination retransmits the ready message to MON1 via MON2 (8 + 9).



```
 Program control block
        .
        .
(Transaction structure description)

 SEND      N2    K      L1      Transaction
 WAIT      A     -              part
 LOCK      WR    PERS   N1          A
 LOCK      RDE   CITY   N2

 SEND      N2    L      M-1
 WAIT      B     A                  B

 SEND      N1    M      N-1
 WAIT      C     A                  C

 SEND      N1    N      O-1
 WAIT      D     B
 WAIT      D     C                  D
 UNLOCK    WR    PERS   N1
 UNLOCK    RDE   CITY   N2

K  Code for transaction part A

L  Code for transaction part B

M  Code for transaction part C

N  Code for transaction part D
```

Figure 3 : The transaction description

Corresponding to the timing constraints the monitor after termination of transaction part A can start B and C in parallel, B is sent again to node 2 (10), C to the local base machine (not further illustrated here).

The remote monitor routes B to the base machine (11). Let us suppose transaction part B sends results directly to the user (12), who acknowledges correct reception of data. Transaction part B now finishes (like 8 and 9). And so on ...

	ORIG	ACSEN	RMON	MKIND	DEST
1	DIA1	DIA1	-	TRANS	-
2	DIA1	NUA1	-	TRANS	-
3	DIA1	NOA1	-	READY	-
4	DIA1	DIA1	-	EXEC	-
5	DIA1	RTC	MON1	EXEC	-
6	DIA1	MON1	MON1	EXEC	-
7	DIA1	MON2	MON1	EXEC	-
8	DIA1	BM2	MON1	READY	MON1
9	DIA1	MON2	MON1	READY	-
10	DIA1	MON1	MON1	EXEC	-
11	DIA1	MON2	MON1	EXEC	-
12	DIA1	BM2	MON1	RESLT	-
13	DIA1	DIA1	MON1	ACK	

Figure 4. Message flow history for the example

6.2 Embedding the monitor into the system's architecture

When installing the monitor in the system the question arises how many of them do we need? There are two alternatives: one per given request (either transaction or program) created dynamically when a new request arrives and only responsible for this input alone, or one for each site.

As a consequence of the first alternative many processes will run concurrently and for systems like ours, which have restrictions on the number of processes, this may not be the right way. On the other hand this solution requires then an additonal site unique module (like the scheduler, lockhandler etc.) which coordinates the log tape writing activities of the monitors for recovery purposes.

For POREL we have chosen the second way, one monitor per node. That is, we have a unique coordination process on each node for all inputs of all local users. This approach was found not to cause too much traffic load for a single process as the monitor was freed from all result handling work as mentioned above. We now were able to integrate the log tape handler into the monitor as an internal routine which needs no further synchronization.

6.3. Current and future research

There are two features we intend to incorporate into our system. One is the concept of delayed updates. This means that, when working with a data subset, we require at least the original to be locked, together with at most one copy at the processing site. Updates on the locked data are immediately executed. Updates for all other copies can be delayed and executed whenever there is time for or when a request is made for data for which there still exist delayed updates. Using version numbers an effective algorithm can be developed for the necessary synchronization. When locking another copy the lockhandler has to compare its version number with that of the original to be sure that the copy is up to date, if not the remaining pending updates are executed before granting the lock request. The second feature is an alternative mode for improving data transfer. Up to now all transaction parts are sent to their destination if all preconditions are fulfilled. We can also select a different approach:at transaction start all parts for one site are collected and sent together to their destination. Later on this is followed only by a short activation message to start a specific part whenever its wait conditions are satisfied.

Using this strategy we even can go further by decentralizing the transaction based execution monitor scheme somewhat further. Partial trees of the execution sequence which are to be processed on a single node could be given in total to the remote node which then also gets control over that tree and sends a status message to the original monitor only at the end of its processing. These subtrees usually are the result of the optimization algorithms in the NOA level which break up object code into maximally parallel executable streams on concurrently working base machines. But the side effects of such a partial decentralization have still to be studied in detail, especially with respect to more dynamic locking strategies and recovery.

7. CONCLUSION

In this paper we have now presented an overview of the execution monitor used in the POREL system. Of course much more could be said about this module and many questions have to remain unanswered, in the space available. We hope however to have given sufficient detail to grasp the decentralized execution control mechanisms used in POREL and the way this control unit interfaces with the other parts of the system. In case of deeper interest please consult the technical reports (10), which contain the design specifications of the system and other publications about POREL.

8. ACKNOWLEDGEMENT

The authors would like to thank all other members of the POREL research group (K. Boehme, Th. Olnhoff, G. Peter, S. Poschik, R. Proell, R. Studer, B. Walter, I. Walter) and for the partial support by the 'Bundesministerium fuer Forschung und Technologie BMFT' under grant 0815010, by the 'Deutsche Forschungsgemeinschaft DFG' and by the European Research Office of the US Army.

9. REFERENCES

(IFI means Institut fuer Informatik, University of Stuttgart, Azenbergstr. 12, D-7000 Stuttgart 1. Many of the papers marked "in German" are to appear in English too but have not yet been issued).

[1] Astrahan et al.: System R: A Relational Approach to Data Base Management, IBM Res. Lab. San José, RJ 1738, 1976

[2] Biller/Peter: The Distribution of Data and the Control of Copies in a Relational DDBS, IFI (in German), 1977

[3] Fauser, U./Neuhold, E.: System Architecture of the Distributed Relational DBMS-POREL, Working Paper for IGDD, Workshop on Distributed Systems, Aix-en-Provence, 1979

[4] Fauser, U.: Data Base System Features, IFI Course Notes (in German), 1978

[5] Fauser, U.: System Catalogues in the DDBMS-POREL, IFI Working Paper, 1979

[6] Hiller/Neuhold/Peter: Calculation of Minimal Transportation Cost in a Distributed Data Base System, IFI Working Paper, 1979

[7] Neuhold/Biller: POREL: A Distributed Data Base on an Inhomogeneous Computer Network, Proc. of VLDB, Tokyo, 1977

[8] Neuhold/Peter: The Distribution of Data in the DDBMS-POREL, IFI Working Paper, 1978

[9] Olnhoff/Päusch: Efficient Retrieval Operations for a Relational Data Base System, IFI Working Paper, 1979

[10] POREL: Preliminary Design, Technical Report IFI (in German), Jan. 1978

POREL: Design Specification, Technical Reports (in German), IFI, Dec. 1978
78/ 4 POREL System Architecture
78/ 5 The Relational Data Base Language
78/ 6 The Relational Machine Language
78/ 7 Network Independent Analysis
78/ 8 Host Language Preprocessor
78/ 9 Compatible Data Base Functions
78/10 Network Oriented Analysis
78/11 Execution Monitor
78/12 Relational Base Machine
78/13 Communication System

[11] Poschick, S.: User Oriented Data Clustering in a Relational DB, IFI (in German), 1977

[12] Poschik, S.: A Portable Relational Interface for the DDBS POREL, IFI, 1977

[13] Rothnie/Goodman: Overview of the Preliminary Design of SDD-1, A System for Distributed Data Bases, Proc. of 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, 1977

[14] Stonebraker/Neuhold: A Distributed Data Base Version of INGRES, Proc. of 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, 1977

[15] Studer, R.: Functional Specification of a Decision Support System, IFI Working Paper, 1979

[16] Weber, W.: A Subsystem for Maintaining Semantical Integrity in Distributed Relational DBS, Technical Report 17, Part 1 and 2, University of Karlsruhe (in German), June 1978

ESA

An Evolutionary System Architecture for a Distributed Data Base Management System

Herbert Weber
Dieter Baum
Radu Popescu-Zeletin

Hahn-Meitner-Institut
Berlin

Abstract

A distributed data management system on a heterogeneous computer network is presented. It consists of two components: a front end system composed of a number of application specific data base management functions and of a general purpose kernel. This architecture has been developed (1) to provide for system evolution upon changes of user requirements, (2) for simplicity of the system structure, and (3) for a high system performance. The paper elaborates on the rationale for the approach. It encompasses a description of the gross architecture of the system, reflects on the most critical design issues for distributed data management systems on heterogeneous computer networks and explains the developed solutions.

*   *   *   *

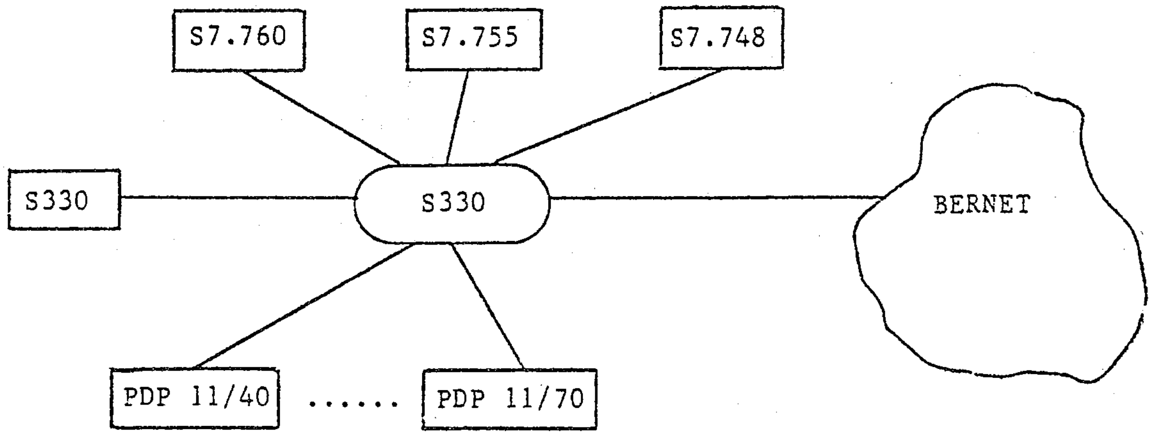
1. INTRODUCTION

Decentralized computing systems seem to be attractive for a great number of applications. It is obvious that the existence of some kind of Distributed Data Management System (DDMS) is inevitable for most applications /WE 78/.

At present a number of distributed data base management systems are under development but none exist as a product on the market. Since an agreement on a coherent set of requirements for DDBMS has not been reached yet, each of the different design experiments starts from a particular designer's perception of the possible environment in which the DDBMS would be used. Consequently, a great number of obviously different, or sometimes not so different architectures have been proposed (a comprehensive list of references may be found in /AD 78/).

The nonexistence of a unique and coherent set of requirements and objectives for DDBMS on the one hand, and our customers´ demands for a maybe simple but soon usable Distributed Data Management System on the other hand, are the determining factors for the development of an alternate design and implementation strategy. This paper elaborates on the rationale of this strategy and presents the resulting Evolutionary System Architecture (ESA).

The system being presented was developed jointly by the Hahn-Meitner-Institut for Nuclear Research, and Siemens AG. It was sponsored by the German Ministry for Research and Technology (BMFT). This effort resulted in a star-shaped computer network -the HMINET- which has been operating since 1976. The network connects a large number of process control computers of different manufacturers, and three time-sharing mainframes (SIEMENS 7.760, 7.744 and 7.748) via a central switching node (SIEMENS 330) using high speed data lines (up to 200 kb/s). The process control computers serve as data acquisition devices for nuclear physics and radiation chemistry experiments. The installation of an interconnecting link from the HMINET to an external network in Berlin (BERNET), which will connect research institutes and university computing centers is planned. The entire network structure may be depicted as follows.



The description of the ESA system in this paper is organized in the following way. Chapter 2 contains a more elaborate description of our motives and goals and relates them to those of other researches. The applied design strategy is then explained in chapter 3. Chapter 4 encompasses a description of the gross architecture of the ESA system, focussing first on a so-called ESA frontend system and describing ESA's kernel system in the sequel. This chapter reflects on the most critical design issues for distributed data management systems on heterogeneous computer networks and explains the solutions developed in our project.

2. RATIONALE

Our concept of a distributed data base management system is based primarily on an analysis of our in-house applications in scientific data processing and some application in some other environments:

(i) The acquisition, storage and pre-processing of large bulks of data local to data producing scientific experiments require local front end data storage and retrieval capabilities. The archival and large scale evaluation of experimental data requires back end data management capabilities on main frame computer systems. Thus a DDBMS should provide means for an interactive data analysis incorporating both access to local and remote data.

Similar requirements seem to determine some other important applications:

(ii) Applications in government administration.
There is a need for data-exchange capabilities among independently operating data base systems for different governmental agencies. The connection of these systems into a DDBMS is the only economic way to exchange data. Very often, those networks will encompass different types of computer systems and differently structured data bases.

(iii) Applications in business

* Office automation

The term stands for a number of efforts with the goal of supporting the office work of executive secretaries. This application is considered to be the most expanding area of computer applications in the 1980's. Local office computer systems support document processing (dictation, text editing, copying, printing, photo composition, etc.) message processing (sending, receiving and distributing of memoranda, letters, proposals, drawings, etc.) information archival, and providing access capabilities to the company's main data base on mainframe computer systems.

* Executive decision making
Decision making depends on the availability of appropriately prepared data. It is assumed that computer-supported decision-making will be one rapidly growing computer application area. It will depend on locally and remotely accessible data.

(iii) Applications in Medicine: Practice Automation
Besides tasks similar to office activites, like billing and accounting, doctors will use systems to support diagnoses. For both activities, access to local data on small office systems and remote data on mainframe systems will be necessary.

Based on this analysis we have drawn the following conclusions:

C1: Computer networks of small dedicated computers and mainframe computers are most appropriate to support a great number of different applications in different environments.

C2: For most of the considered applications a DDBMS should primarily support the rather frequent access to "own" data on local computer systems and should allow for the rather infrequent access of "common" data on remote computer systems.

Our concept is also very strongly based on a thorough analysis of possible difficulties because of a number of so far unsolved problems for the development of DDBMS:

The differences between centralized and distributed data base systems with respect to a number of fundamental technical problems have been reported repeatedly in the literature /AD 78, R 77, PM 78/. It is obvious that solutions to some of them, like synchronization of concurrent updates, query processing, handling of component failures, and in particular the conversion of data and operations in heterogenous networks will also result in systems of high complexity and cause high internal administration overhead. For its expected complexity it does not seem to be clear to us as to whether it is yet feasible to design and implement an efficient general purpose distributed data base system for all kinds of applications. We, in our effort, tend more to the assumption that a suitable architecture framework encompasses both a general purpose kernel system providing a machine-oriented data management facility (similar to the one usually attributed to an internal schema of a data base system) and function units for special applications. Our Evolutionary System Architecture was therefore developed to enable the development of the kernel first and of arbitrary application units on top.

It is still a fact that a great number of applications use the operating systems' data management functions only and do not depend on a data base system at all. It is therefore our intention to devolop with our Evolutionary System Architecture a concept for the integration of different data management capabilities into a netwide data manager on top of the operating system level. Nevertheless --to avoid any misunderstanding-- it is not our intention to develop a distributed operating system. We are much more interested in a system for the administration of large amounts of data by rather complex retrieval and update operations. These requirements are - as stated in /RG 77/ - fundamentally different from those for operating systems. The resulting architecture must reflect these differences. We expect the kernel system to provide data structuring capabilities and data manipulation capabilities comparable to those in contemporary operating systems. Higher order structuring and manipulation means are expected to materialize in the aforementioned application- oriented function units.

According to these considerations our system concept is based on the following additional conclusions:

> C3: A suitable architecture of a DDBMS consists of a general purpose kernel system and of application specific function units.
>
> C4: A suitable DDBMS should support both a low-level user interface providing efficient data base access capabilities, and higher-level user interfaces for higher user convenience.

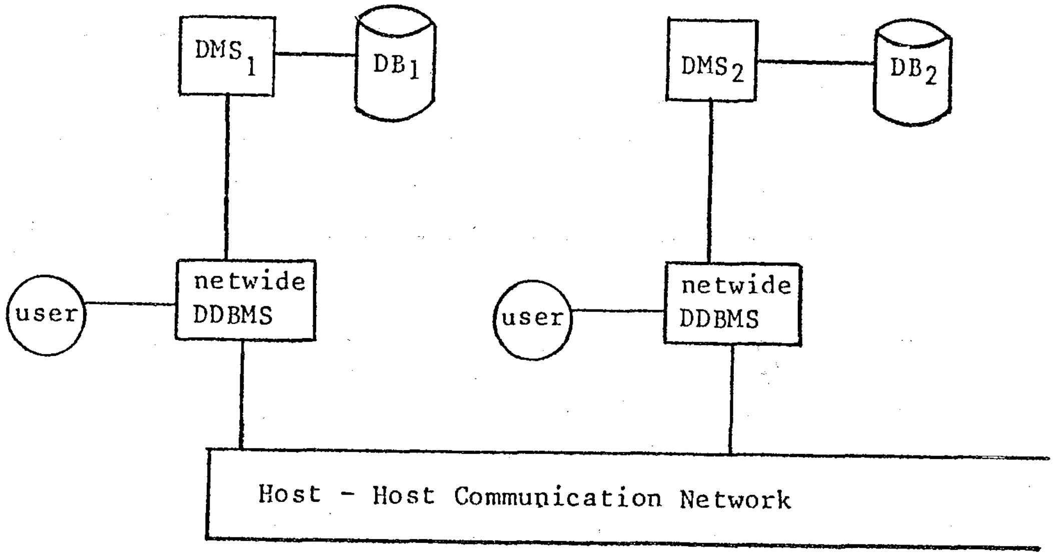
3. ESA DESIGN STRATEGY

Two basic design strategies have been proposed for the design of a DDBMS /AD 78/: A uniform integrated system may be built in an overall top/down design process (T-systems in /AD 78/) in the one case, or a number of different DBMS´s implemented on different host computers will be integrated afterwards in a bottom-up fashion (B-systems in /AD 78/) in the other case.

This of course does not imply that a different software design methodology - top down or bottom up - has to be applied in the design of these systems. To avoid any confusion we would prefer to call the resulting systems pre-integrated and post-integrated systems respectively.

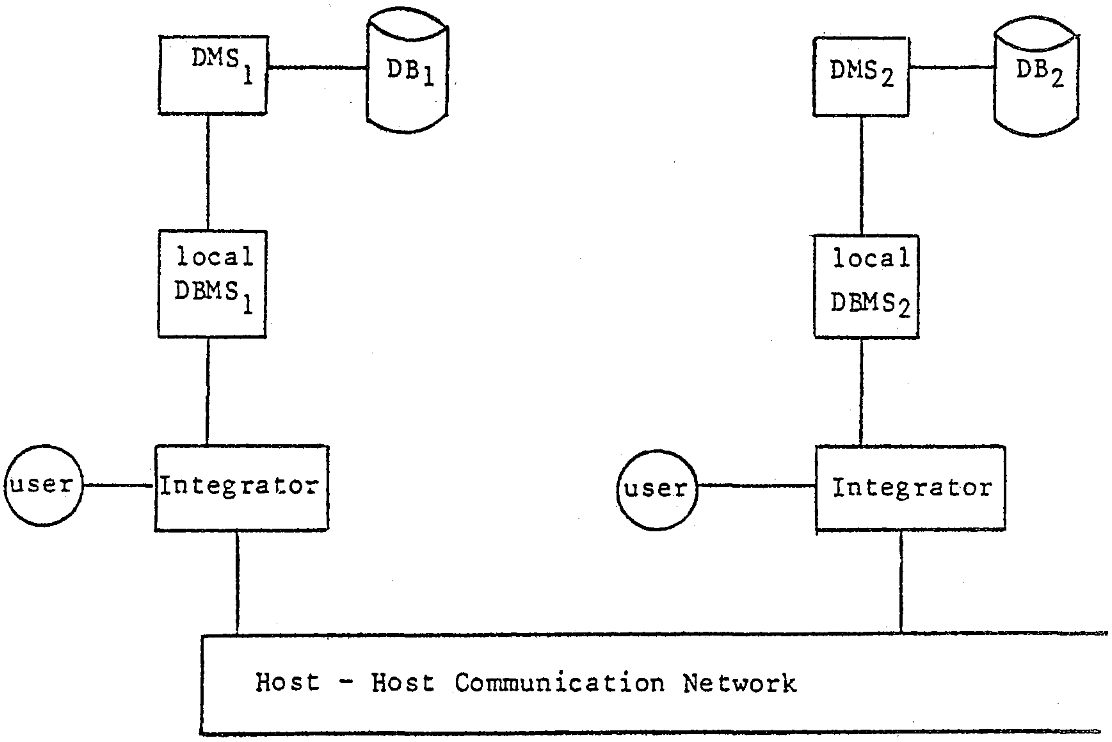
The two resulting architectures may be depicted by the following graph as proposed in /PM 78/.

Pre-integrated System Architecture



In this concept a user has access to a uniformly designed DDBMS which uses local data management services (DMS_i) or provides access to remote data management services.

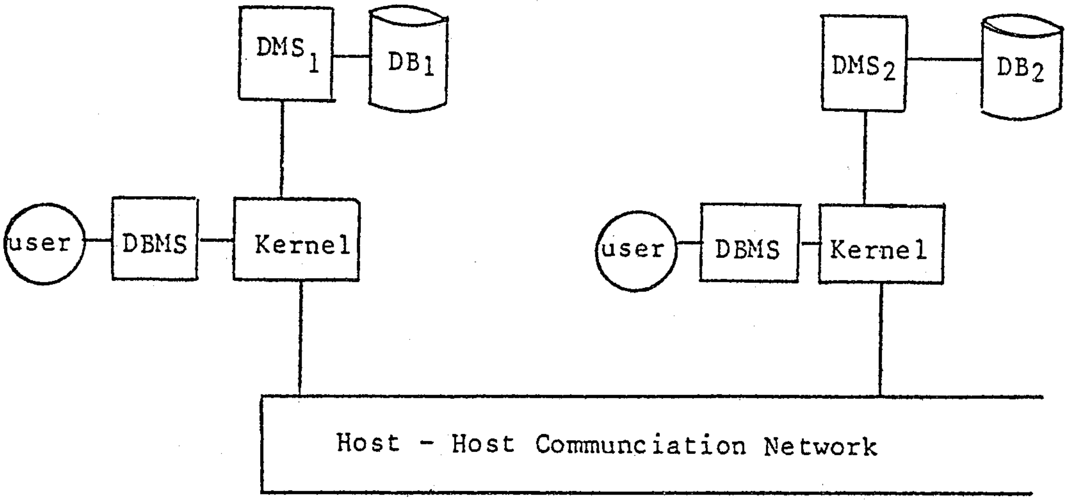
Post-integrated System Architecture



In the second concept an integrator will be provided on top of existing local DBMS´s. A user gets access to the uniformly designed integrator which uses local DBMS services and provides access capabilities to remote DBMS´s.

Our design strategy represents a position in between the two extremes. A number of different data management functions in different operating systems will be integrated resulting in a post-integrated kernel system. The kernel system uses data management services provided by local or rnmote operating system data management functions. The kernel provides services to human users or to one of the following front end systems: application specific data management functions, a netwide general purpose DBMS or different local DBMS´s.

Evolutionary System Architecture

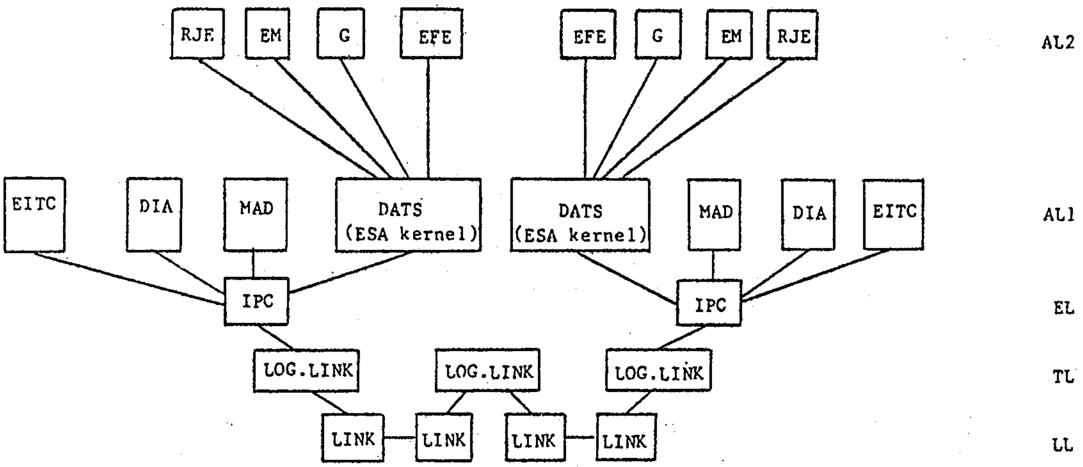


After the design of its kernel ESA may be extended into one or all of the aforementioned directions. A more elaborate discussion of the resulting systems will be given in the next section of the paper.

4. ESA GROSS ARCHITECTURE

ESA consists of a front end system (or of a number of co-existing frontend systems) and of a kernel system.

ESA's embedding into the entire network system may be depicted in the following graphical representation of the network software structure.



As usual, the network is designed in a layered fashion providing a number of different functions on each layer. The functions on each layer will be performed by using services provided by the functions of the next lower layer. Four layers may be identified in the current state of development, e.g., the application level, the end-to-end level, the transport level and the link level. Communication between corresponding functions will be enabled by communication protocols associated with each level in the network.

The two bottom layers - the link level (LL) and the transport level (TL) - provide a datagram service controlled by a SIEMENS NEA2 protocol. Their function and internal structure is not of particular interest for our further discussion.

The interprocess communication facility (IPC) is provided on the end-to-end level (EL).

Several basic functions are provided on the application level 1 (AL1), e.g.:

the system component called EITC (Extended Intertask Communication System) facilitates the communication between programs which run on different hosts;

DIA (Dialog)
permitting accesses from remote terminals (i.e., accesses to any timesharing host from all terminals connected to the network);

MAD (Maintenance, Administration and Demonstration)
performing control functions like load measurement, network administration, and information display;

DATS (Data Access and Transfer System)
providing the kernel functions for ESA and some other application systems.

A number of functions on application level 2 (AL2), like a Remote-Job-Entry-System (RJE), an electronic mail system (EM), a distributed graphics system (G) and last but not least, ESA´s front end system (EFE) are all based on services provided by DATS.

The following discussion will now be restricted to a more elaborate description of ESA´s front end system and the kernel DATS.

4.1 ESA Frontend System

In its current state ESA is meant to support rather simple application-specific data base management functions (ASF´s) first, which may be extended later on into a general purpose DDBMS. The simplifications introduced in the design of ASF´s become apparent in a comparison of its characteristics with some interface characteristics of general purpose DDBMS.

General purpose DDBMS´s provide access to all data stored in the distributed data base from any host in the network. This may be accomplished by maintaining directories accessible from each host in the network. A number of different strategies have been proposed for the allocation of data in a computer network, e.g.,

(i) all data may be duplicated and located at each node of the computer network (i.e., the fully redundant case);

(ii) data are stored partially redundant;

(iii) data may be stored non-redundant;

and for the location of directories in a computer network, e.g.,

(i) a directory containing entries for all data in the distributed data base is maintained centrally at one host computer; access to data is provided through this centrally located directory only;

(ii) a directory containing entries for all data will be duplicated and maintained on each of the host computers.

Any combinations of these alternatives for the placement of directories and data in the network has a great impact on the solution of the concurrent update problem on query processing problems and on the handling of component failures.

ESA materializes a somewhat simpler and restricted concept which is based on a somewhat different perception of the requirements for many applications. In our concept the front end system consists of a number of rather autonomous application-specific data base management functions (ASF´s) which share just some data and maintain the major part as "own" data. They form altogether an application system. A more sound definition of an application system will be given in the sequel.

An Application System, AS, consists of a time variant set P_t of programs p_{ti} and of a time variant set D_t of data repositories d_{ti} (which altogether form the data base) which are used by the programs p_{ti}. One may then write more abstractly:

$$AS_t = (P_t, D_t, R_t)$$

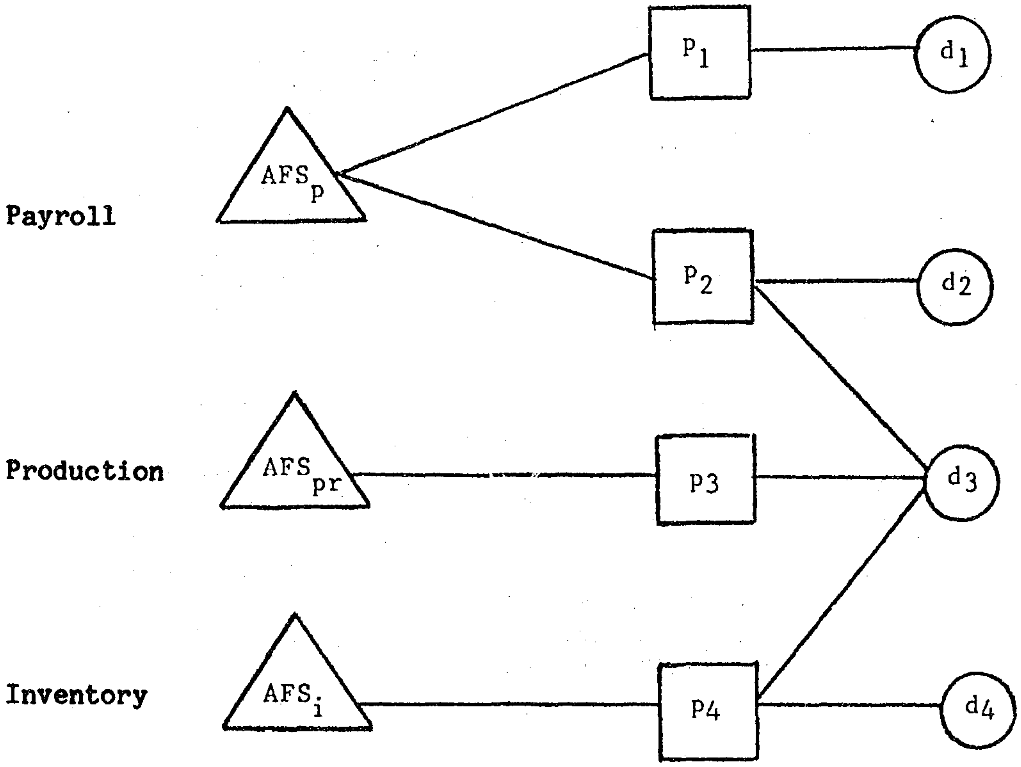
$$\text{where } R_t \subseteq P_t \times D_t$$

$$\text{with } p_{ti} \in P_t$$

$$d_{ti} \in D_t$$

Each program p_{ti} may have access to a number of data repositories $d_{ti},\ldots,d_{tj}$ during its execution and each data repository d_{ti} may be used by a number of programs $p_{ti},\ldots,p_{tj}$. Because of this characteristic, the "access" relationship R_t is said to be n:n.

This definition of an Application System is in accordance with information processing practice. For example, an enterprise´s payroll AFS (index p) uses its own data repositories and some of the production department's repositories. And, conversely, the production department's data repository is used by the payroll ASF and by the inventory control ASF's programs. This sample AS may be visualized by the following graph.



The foregoing definition captures the very basic fact that application programs are not independent of each other, even though they have been independently developed. All the programs which use a common data repository interfere with one another via the common data.

However, the partial sharing of data among different application programs of different AFS's is the very basic characteristic which distinguishes application systems and general purpose data base management systems. Each application-specific data base management function ASF$_i$ provides just a window to the data base offering

a limited access capability to a part of the data base relevant to this application only. The amount of sharing of data among application programs determines the degree of autonomy between ASF´s: a small number of shared items cause high autonomy and vice versa.

The placement of indexes and data of an AS in a network may now be based on the following assumptions:

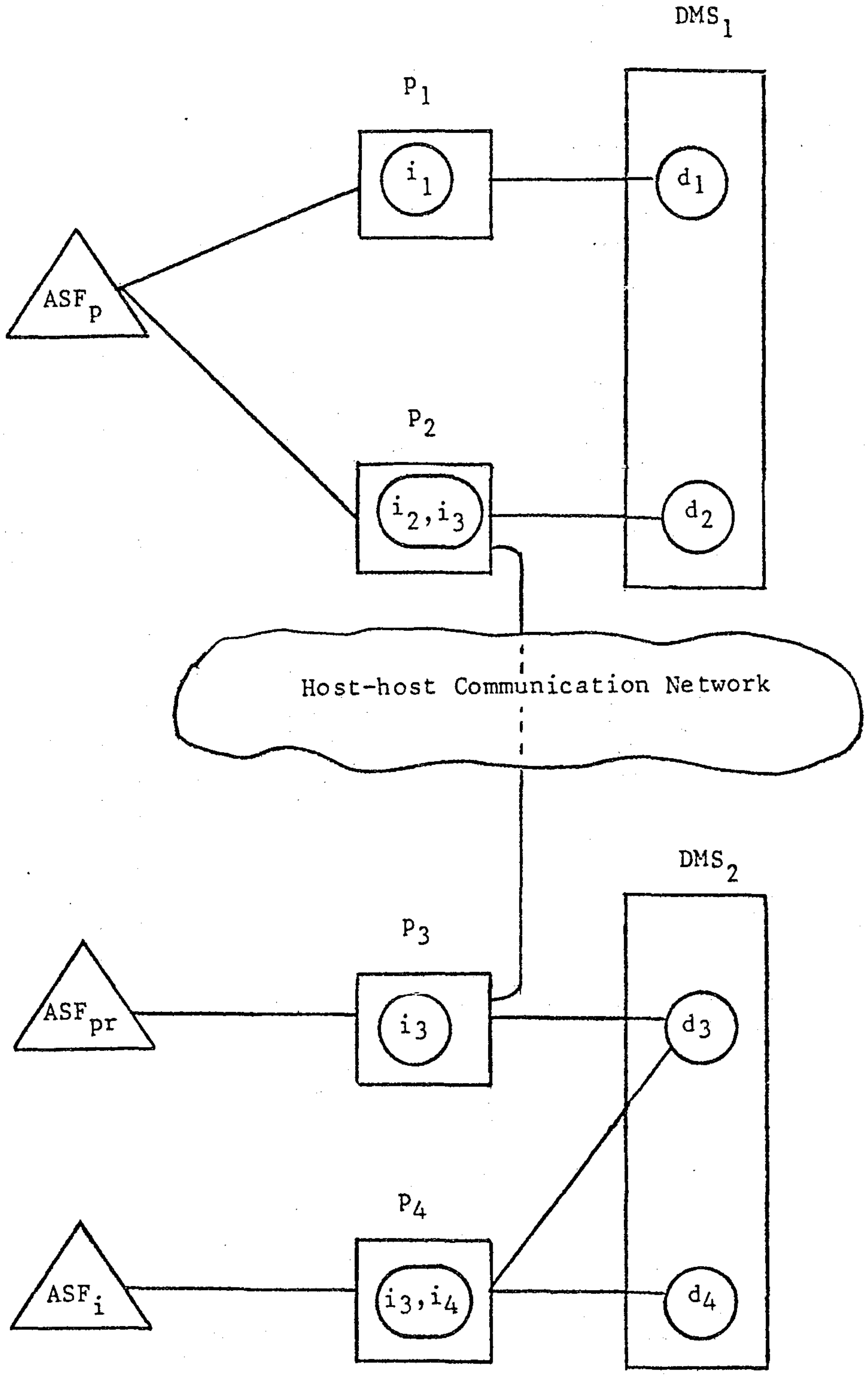
(i) Most of the Application Systems for the kind of applications considered in Section 2 of the paper consist of highly autonomous ASF´s.

(ii) In most cases a host computer will be provided for each ASF.

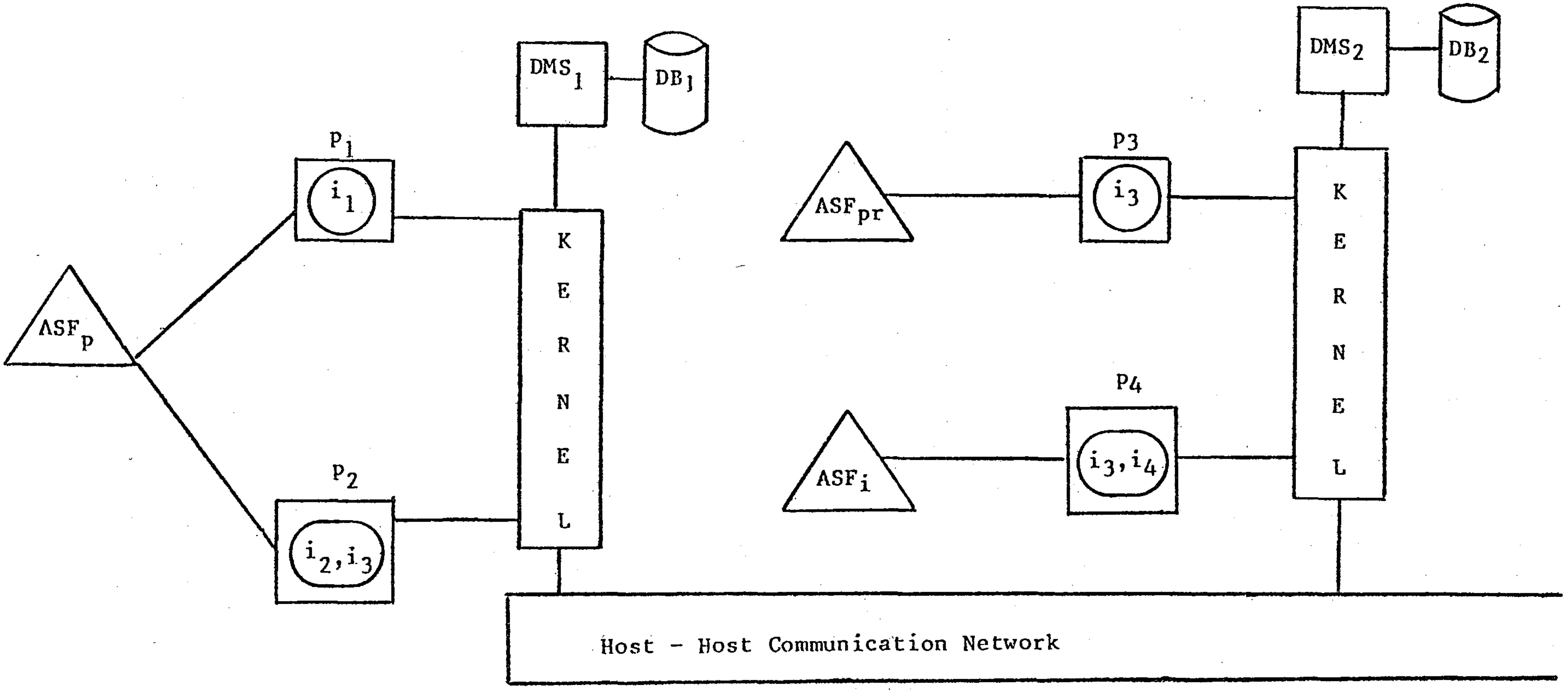
(iii) Most of the data will be stored locally on the host computer allocated to the ASF, only a small portion of the data relevant to an ASF will be stored remotely if the data is shared and accessed more frequently at the other host. The data will be stored in a non-redundant fashion.

(iv) Directories will be maintained on each host containing entries for all data repositories relevant to the ASF allocated to this host. Because of the sharing of data between ASF's only partially redundant subdirectories of the data base directory will be maintained at each host.

Based on these assumptions ESA´s entire frontend system may be constructed of a number of application programs p_i with a directory i_i associated with each of them. The application programs and the associated directories may be partitioned according to the number of ASF´s allocated to different hosts in the network. For their execution, application programs perform then service requests to a local data management system provided in the kernel system or to one or more remote application programs. This may be depicted for the sample system introduced above in the following graph.



To enable the communication between application programs and local data management systems on the one hand, and between application programs on different hosts on the other, ESA´s gross architecture has been designed as depicted in the following graph.


DMS1
DB1
DMS2
DB2
P1
i1
P3
i3
ASFpr
ASFp
K E R N E L
K E R N E L
P4
i3,i4
ASFi
P2
i2,i3
Host - Host Communication Network

4.2 Provisions for Evolution

For its evolution ESA provides means for changes of its front end system without causing any modifications of the kernel. Changes of the front end application system can be facilitated by composing it of quasi-standardized modules which are designed according to a uniform generic module design concept. The concept provides means for AS modifications by adding and removing modules /WE 78/ /WE 79/. According to this concept modules are defined to exhibit clear and simple interfaces and internal structures which make them function the same way in all possible environments. Thus, modules may be connected to other modules and connections may be removed without any effect whatsoever on the functioning of any other module.

To guarantee this invariance of a module's functionality, a module is defined to exhibit the following characteristics:

Modules identify a particular type of data and all the operations applicable to data of that type. Data and operations defined in a module must fit together properly.

The rather imprecise statement that data and operations have to fit together needs some further explanation: Data are representations of real phenomena. To meet these phenomena, the representing data must assume certain characteristics. Data are, on the other hand, subject to changes. It is therefore important to distinguish two different types of characteristics: the extension, and the intension of data. The term extension refers to the time dependent aspects of the information contents (e.g., the actual set of tuples in a relation at a certain point in time). The term intension refers to the time invariant aspects of the information content (e.g., two domains of a relation are in a functional dependence at any time). Data may then be manipulated (i.e., their extension may be changed) according to their time invariant characteristics (i.e., according to their intension). It is consequently necessary for the design of Application Systems to ensure that all operations will be designed to be in accordance with the data´s intension.

A module design methodology which imposes the necessary discipline is therefore developed along the folllowing guidelines:

(i) Each individual type of data object and all the permissible operations on data objects of this type will be defined together in a module.

The data object will be manipulated by those operations only. Different users may manipulate the data object by invoking one of the predefined operations. This makes the module a self-contained entity which will display the same time invariant characteristics in all environments. The data are called encapsuled by the permissible operations within the module.

(ii) A module definition encompasses the definition of rules for the preservation of the semantic integrity of the type of data defined in the module.

Changes of data are constrained by restrictions which are to be obeyed in order to preserve the semantic meaning of the data. For example: An inventory department´s data repository may contain data about parts on hand. The order department is consequently not allowed to change the data in this repository after an order of new parts has been issued, but only after the new parts arrive. Thus, data changes may be tolerated if the preservation of the semantic integrity of the data is guaranteed.

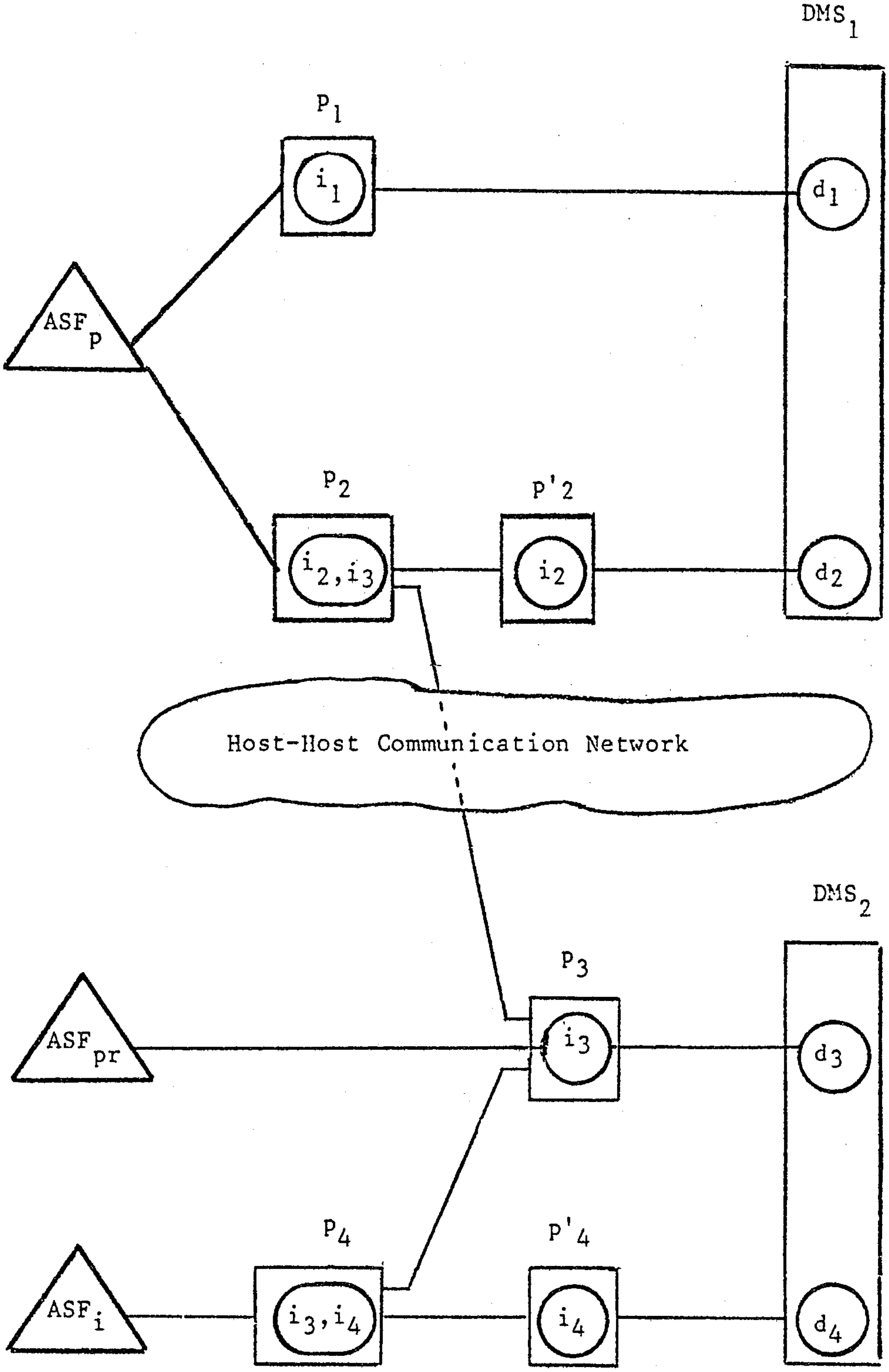
(iii)A module definition encompasses the definition of rules for the execution of concurrent execution requests for its operations.

Changes of data are also constrained by restrictions for the concurrent execution of operations on common data repositories. It is necessary in this case to guarantee that the outcome of the execution of an operation is the same as it would be if the operation were not interleafed with any other operation. Thus, concurrently performed data changes may be tolerated if the preservation of the consistency of data is guaranteed.

Application systems when designed with the aforementioned methodology allow changes of the system by additions and removals of modules as desired.

The concept is applied in the design of an AS in ESA in the following way:

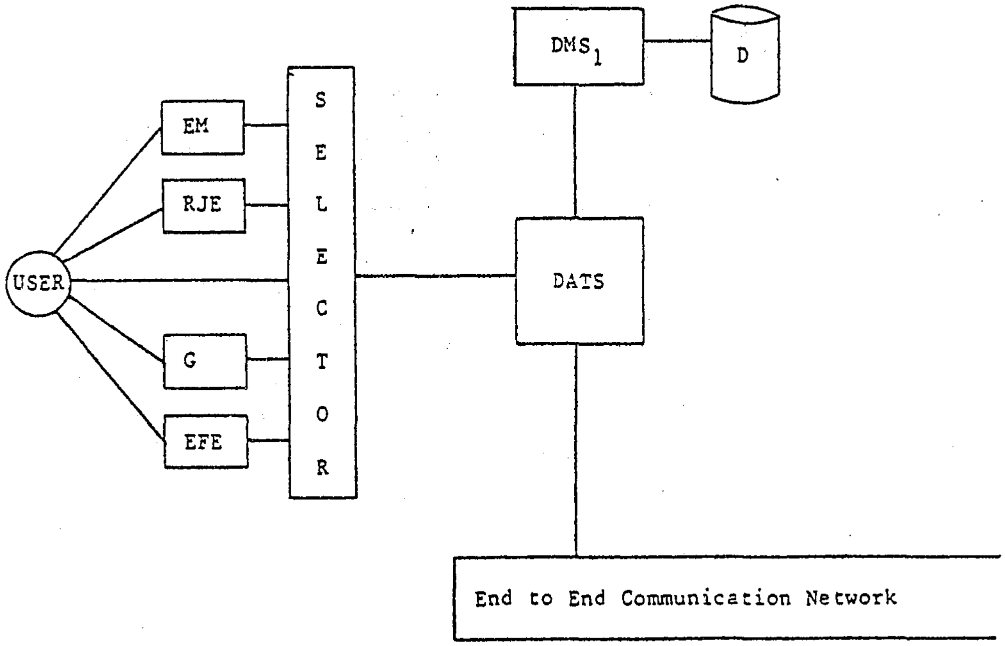
The AS is constructed in a hierarchic fashion by the composition of modules out of other lower level modules. As a consequence, an AS is built as a hierarchic composition of indexes (representing a hierarchic structuring of data types) and of associated operations (representing the hierarchical structuring of application programs). Modules in the hierarchy are related to one another by a so-called "use-relationship" indicating that higher level modules use the services of lower level modules to complete their task. This hierarchic composition may be illustrated with the previously introduced sample AS in the following way.



For system changes, arbitrary modules may be added to the hierarchy or removed from the hierarchy if the modules are not referenced anymore in any other modules.

4.3 ESA Kernel System

The Kernel System consists of a component called DATS (Data Access and Transfer System) and of a SELECTOR component. The DATS component has been designed to provide general purpose functions for a rather great number of services like maintaining distributed data representing graphic information, facilitating electronic mailing of information between different hosts in the network, facilitating remote-job-entry, maintaining distributed data repositories, etc. It was one of the main goals of the design to build DATS on top of unchanged existing operating systems of the various host computer systems participating in the HMINET. Thus the embedding of DATS into the network system may be depicted as follows.



Based on this simplified depiction DATS may be considered as consisting of two (distributed) communicating processes residing on two -maybe different- host computer systems. For its expected function the process pair is designed to provide as proposed in /K 78/:

- a mechanism for the selection of requested data;

- a mechanism for the transfer of data;

- a mechanism for the transformation of data structures according to the different data structuring capabilities on the different host computers.

In order to provide these functions the interacting processes have to accomplish tasks as different as data link control, system and data resource allocation, addressing, interprocess synchronization, error recovery, etc. A more detailed description of these tasks will be given in the protocol definition for DATS after some further explanation of DATS' internal structure in the following section.

4.3.1 DATS' Architecture

The kernel component DATS in turn consists of four components with a somewhat overlapping function: Remote File Transfer (RFT), Remote Data Access (RDA), Virtual File (VF) and Remote Execution Request (RER). All four of them are composed of two processes residing on different hosts and they communicate in accordance to a specified protocol. The function of these components may be explained as follows.

Remote File Transfer:

RFT may be initiated by a user process UP and

- it establishes a connection between a local RFT process called FTM (File Transfer Master) and a remote RFT process called FTS (File Transfer Slave);

- it transfers a remote file transfer request;

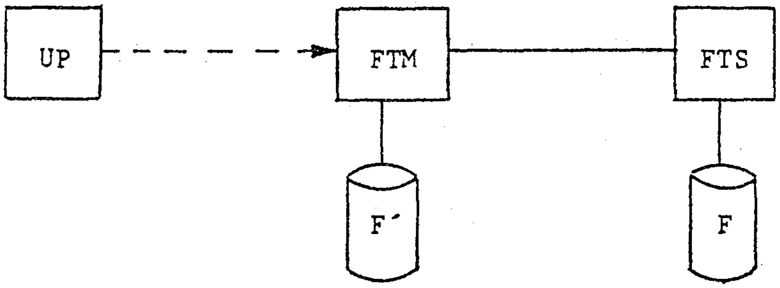
- it performs an access to the requested file;

- it transfers a message containing the requested file to the requesting host;

- it converts data formats of the transferred data into the formats of the requesting host;

- it stores the transferred copy at the requesting host.

This communication pattern may be depicted in the following graph.



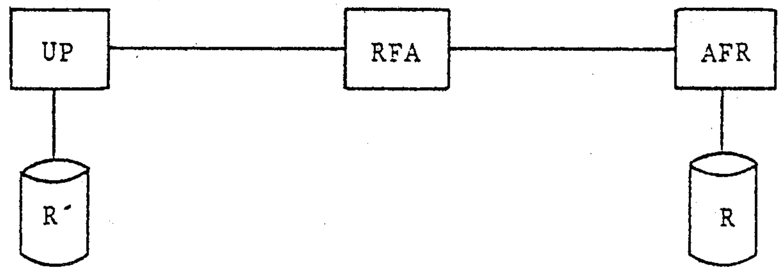
After the initialization of RFT by UP the two processes may continue to execute in an asynchronous fashion.

Remote Data Access:

RDA may be initiated in a user process UP and

- it establishes a connection between a local RDA process RFA, and a remote RDA process AFR;
- it transfers a remote data access request;
- it performs an access to the requested record of a remote file;
- it transfers a message containing the requested record to the requesting host;
- it converts data formats of the transferred data into the format of the requesting host;
- it provides the transferred copy of the requested record to the requesting program.

The communication pattern may now be depicted as follows.



After the initialization of RDA by UP the user process halts until the termination of the remote data access.

Virtual File (not fully designed yet)

VF may be initiated in a user process UP and

- it establishes a connection between a local VF process VFL, and a remote VF process VFR;
- it transfers a virtual file request;
- it maps an access request on a virtual file into an access request on a local file;

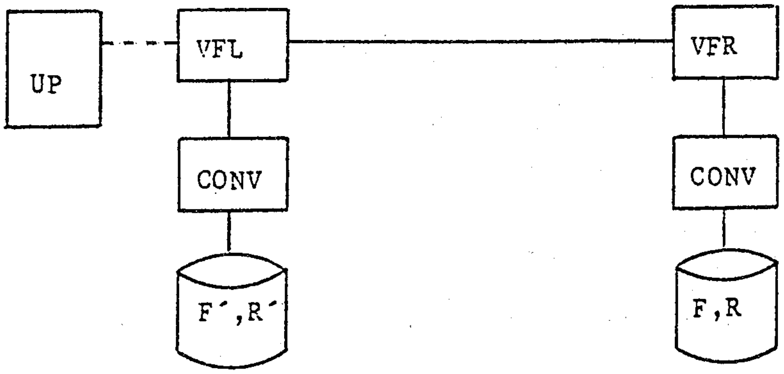
- it performs an access to the requested file or records on the remote host;

- it maps the format of the accessed data into the virtual format;

- it transfers the requested data to the requesting host;

- it stores the transferred data in a data repository on the requesting host.

This communication pattern may then be depicted as follows.



After the initialization of VF by UP the two processes may continue to execute in an asynchronous fashion.

Remote Execution Request (not fully designed yet)

RER may be initiated in a user process UP and

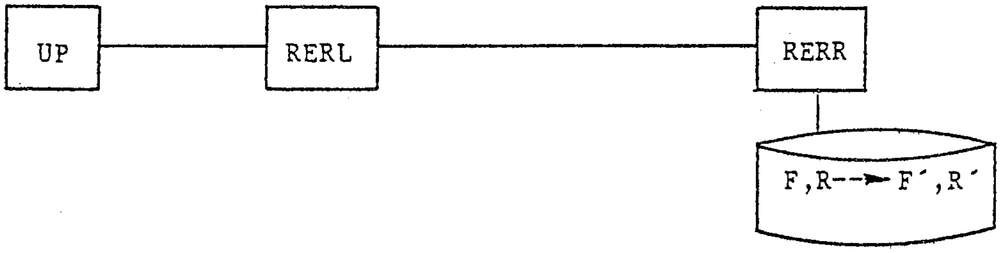
- it establishes a connection between a local RER process RERL and a remote RER process RERR;

- it transfers a remote execution request;

- it performs manipulation on data identified in the remote execution request;

- the results of the remote execution are transferred to the user process (UP)

This RER communication pattern may be depicted as follows.



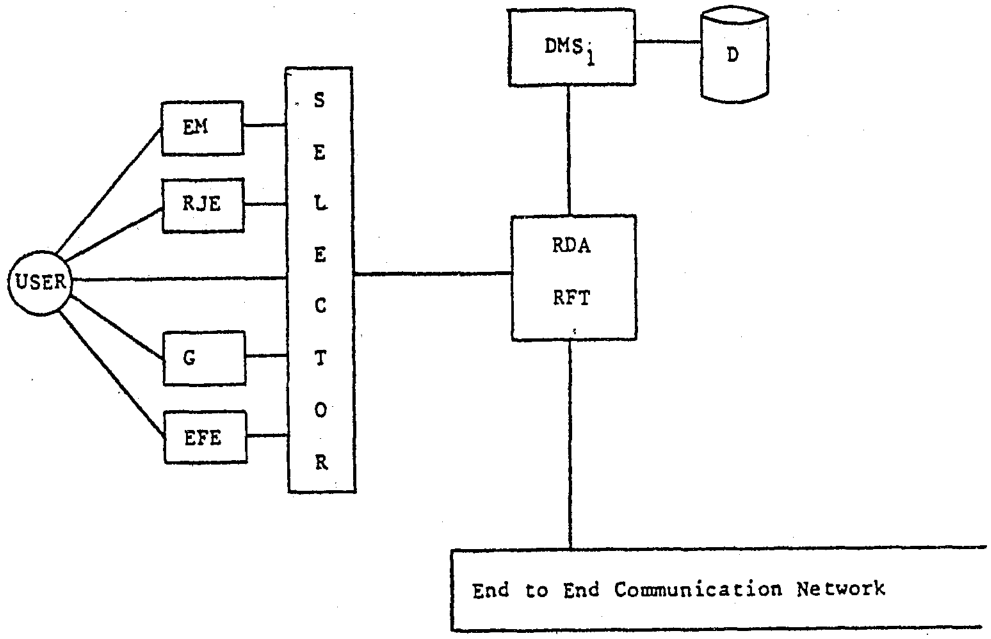
After the initialization of RER by UP the two processes may continue to execute in an asynchronous fashion.

The first three components RFT, RDA and VF may all be used to perform similar tasks. The decision of which component will be used in the execution of a certain user transaction will be based on performance and availability criteria. These criteria have been determined in system modeling, protocol modeling and system simulation studies /PZB 78, CB 77, BB 79/ discussed below. A selector component of the Kernel System decides during the execution of user transactions on the selection of one of the aforementioned components.

The VF component is being developed in the system because the HMINET will be connected to the highly heterogeneous BERNET system. In that system all data management services will take place in terms of a standard virtual data format as the one adopted for VF.

The RER component is meant to provide a capability for the efficient change of remotely stored data.

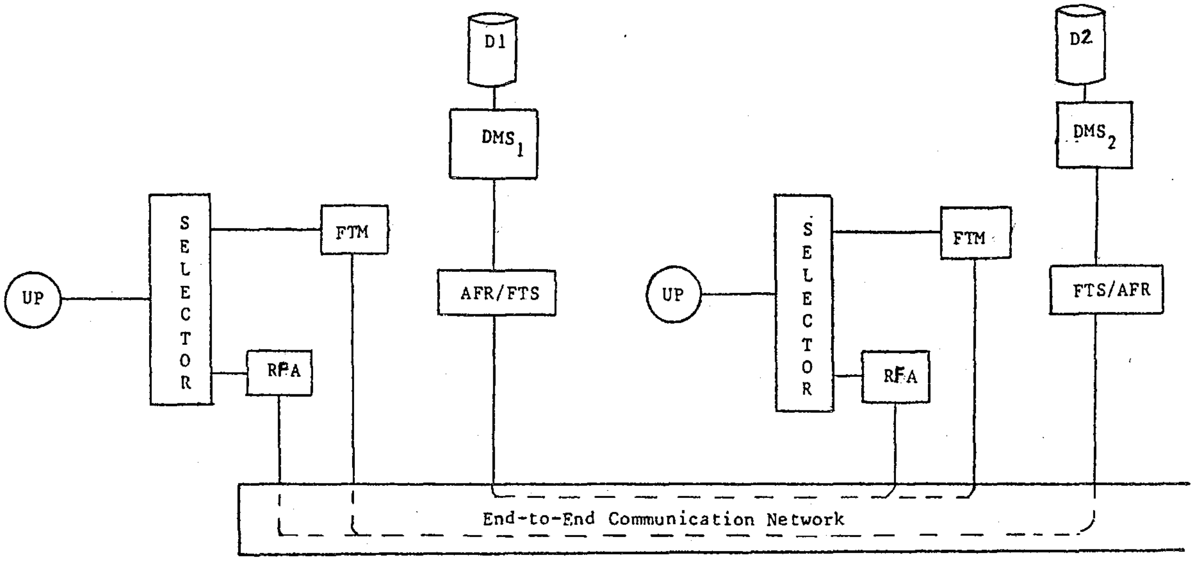
The kernel system in its current implementation may then be depicted as follows.



As mentioned before, the kernel provides services to a number of application systems. A user may also gain direct access to the DATS system via a simple DATS command language introduced below.

4.3.2 DATS' Execution

Several processes communicate during the execution of DATS. The basic communication pattern will be depicted in the following graph.



A user process UP requesting the service of DATS issues a request to the SELECTOR component of the kernel system. The SELECTOR, after deciding which of the DATS components to use, passes the user request to that component (RDA or RFT in the currently implemented version). For the execution of either RDA or RFT a communication channel --a so-called "coded connection"-- will be established between the RDA/RFT component residing on the calling host (FTM/RFA) and the RDA/RFT component residing at the remote host (FTS/AFR). This will be accomplished in a two-phase process:

To initiate the communication the user process issues in the first phase --the so-called remote connection phase-- a call of an INIT statement (via the selector component) to the local component of the RDA or RFT. Together with this call the user process supplies the local component of the RDA or RFT with its own identifier, with the identifier of the called host, and with the size of a resource set (i.e., the maximum number of resources) required (i.e., remote files or remote record-oriented devices) as parameters.

Based on this information the local component FTM/RFA in executing the INIT statement establishes together with the remote component FTS/AFR, a connection between the user process and the requested remote resource set.

In the second phase --the so-called association phase-- the user process supplies the local component FTM/RFA via an ROPEN statement with the identifier of some particular resources out of the resource set as parameters. Based on this information the local component FTM/RFA establishes together with the remote component FTS/AFR a connection between a user process and a particular resource.

After this communication connection has been established the user process may issue calls on action primitives whose execution results in transfers of data from the local to the remote host and vice versa. A more complete list of primitives to initialize a connection and to perform data transfers may be found in Appendix A.

4.3.3 Kernel Selector Functions

As mentioned above, a special kernel component, the selector, decides on the selection of one of the DATS components with similar functionality for the execution of user transactions. This decision is primarily based on performance criteria as, for instance, response time behaviour, local station throughput and buffer space requirements, but is, of course, also dependent on the availability of the respective DATS components in the local or remote hosts. For the latter case, the selector component holds and updates a table of DATS components status and configuration information. Whenever one component, RDA for instance, is known to be unavailable or is known to be not functioning locally or at the remote host, another DATS component, RFT for instance, may be selected to provide the needed service.

We call a component selection of that kind an "availability based selection" in contrast to the "performance based selection" (which shall be described in the following) and the "user intention based selection", as, for instance, the selection of the Virtual File component in highly heterogeneous networks.

The Multiclass Concept

Any host in the network is denoted as "local system". Let there be four distinct user classes defined for local systems: CPU bound jobs, two types of I/O intensive jobs and "normal" jobs. The latter are representatives for the average job profile of some scientific computing center or other. The two kinds of I/O intensive jobs are defined in the following manner:

Access to a whole file - requiring the record by record transfer from device to core - is modelled by the service of a "particular abstract I/O device", that is, any file access request of that kind (called "bulk access request") will be represented by a request to this particular abstract device whose service times are appropriately determined as the estimated value of time needed to perform the whole access request. Here the files are assumed to exhibit constant mean lengths, which has been observed and estimated with relative accuracy.

The customer class of I/O intensive jobs of the first kind is now associated with the class of jobs which will set bulk access requests with some significant probability. I/O intensive jobs of the second kind are defined as processes exhibiting high access rates to different storage devices.

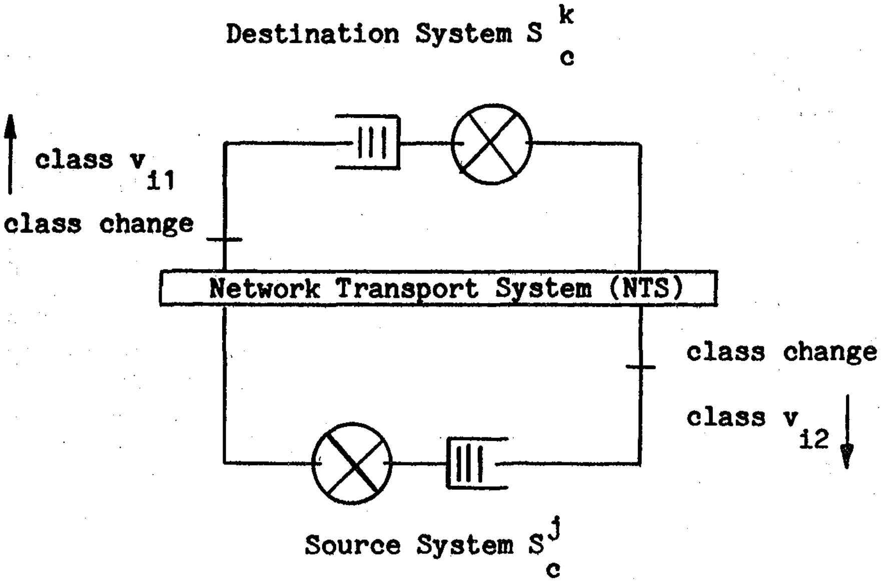
This constitutes the following associations:

Customer class 1	◄---►	CPU bound jobs
Customer class 2	◄---►	normal jobs
Customer class 3	◄---►	I/O intensive jobs of 1st kind
Customer class 4	◄---►	I/O intensive jobs of 2nd kind.

The set of all modules materializing the transport of some message through the network is called "network transport system" (NTS). The NTS normally is a complex system of its own and can be modelled by a special network of queues. In our case, nevertheless, we decided to model it by one single server station (a detailed description may be found in Appendix B). This station NTS is of paticular importance for our investigation of different user classes, since different kinds of net traffic have to be considered. The distinctions between the traffic characteristics of the NTS for the RFT, RDA, VF and RER are primarily due to the different lengths of messages in the respective DATS component. Furthermore, the traffic characteristics for a DATS component are different for different transfer directions (i.e. for transfers from the initiating to the executing process or for transfers in the reverse direction).

These differences are captured in properly estimated class specific NTS-server rates for each of the DATS components. To be more precise, we associate two different customer classes with each traffic type indicating the different characteristics of initiating local processes at some source system S^k and of executing processes at some destination system S^j (j,k= 1,...,h; h the total number of hosts in the network). Thus, eight customer classes v_{11}, v_{12}, v_{21}, v_{22}, v_{31}, v_{32}, v_{41}, v_{42} may be defined in the net, each of them being associated with one of the aforementioned (local) customer classes at the source and the destination system for the execution of the DATS components.

During the execution phase of one of the DATS components a certain customer class v_{i2} will be associated with that component at the source and a different one v_{i1} at the destination system. Thus, a customer class change from v_{ij} into v_{ik} ($i=1,\ldots,4$; $j,k \in \{1,2\}$) happens in between the execution of the DATS component at the source and the destination. This may be illustrated in the following figure.



An association list for the four possible traffic types and the possible associated customer classes may be given as follows.

traffic type	associated local customer class in S (k=1,...,h)	class change when leaving NTS to class
tr_1 = RER	1 (destination) 2 (source)	$v_{12} = 2$ $v_{11} = 1$
tr_2 = VT	2 (destination) 2 (source)	$v_{22} = 4$ $v_{21} = 3$
tr_3 = RFT	3 (destination) 3 (source)	$v_{32} = 6$ $v_{31} = 5$
tr_4 = RDA	2 (destination) 4 (source)	$v_{42} = 8$ $v_{41} = 7$

The depicted associations may be interpreted as in the following example. The list indicates for the RDA system: the traffic type v_{41} of the NTS for the traffic between a source system and the destination system is associated with network user class 7, and traffic type v_{42} of the NTS for the traffic between the destination and the source is associated with network user class 8; the associated local customer class in the destination system is 2 (i.e. normal jobs) and in the source system is 4 (i.e. I/O intensive jobs of the 2^{nd} kind). That means, that in its execution the RDA is of user class 4 at its source, of user class 7 in NTS on its transfer from source to destination, of user class 2 at its destination and of user class 8 in NTS on its transfer from destination to source.

This multiclass concept is used to define customer classes of a queueing model which is built up to analyze the dynamic behaviour of the communication system. It yields formulas for the computation of relevant performance measures. In our system the kernel selector computes on the basis of these formulas approximate values for the following performance measures: response time, local station throughput, and needed buffer space (average queue lengths). This is done according to some known network status and configuration parameters and local load parameters which have to be delivered by other modules or are predefined.

4.3.4 Data Representation and Conversion

Different representations of data on different hosts in the network result in incompatibilies in the data access and transfer system. Files and records may be internally structured in terms of bit strings, character strings, integers, real numbers.

Their representation is not identical on each machine and usually another set of physical data formats is associated with each operating system connected to the network.

These incompatibilities may be resolved in two different ways:

- by a declaration of a netwide standard format;
  A translation from standard format to the local format has to be done on each host in the system. This may lead to a loss of precision for the data representation on certain systems or may cause on overhead for others;

- by permitting all kinds of data representations in the network;
  Conversion routines must then exist in every host to convert from all existing data representations into the local representation.

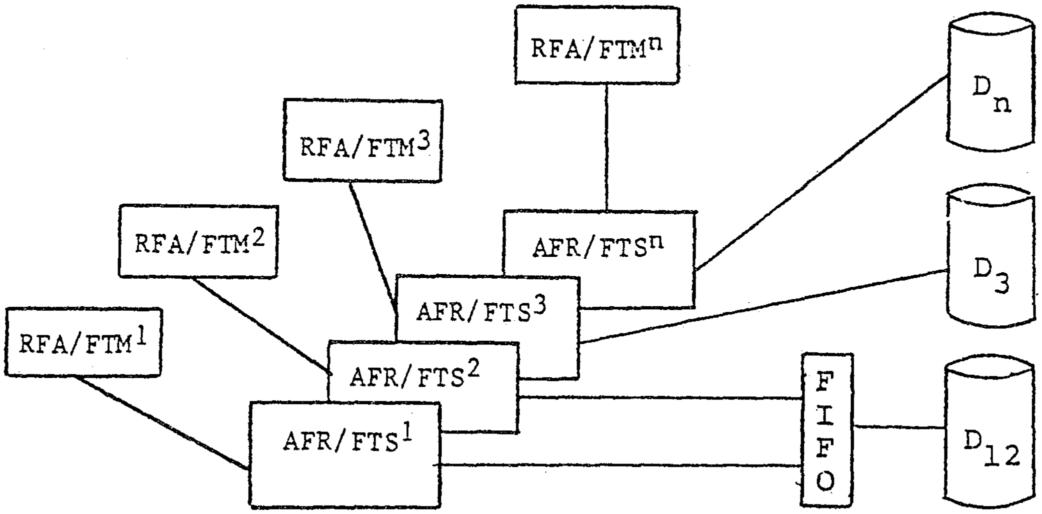
Because of the low heterogeneity of the network and the needed precision the second solution has been chosen in our system.

Another category of inconsistencies results from different operational capabilities on different hosts. Access methods like sequential, indexsequential or different types of random access techniques may exist as standard features of the local DMS or not. Furthermore the same access techniques are implemented differently on different hosts. For the execution of a data transfer request the mapping from one access techniques into another one on a different host with the same effect will be performed in one case. A mapping from one implementaion of a given access technique into another implementaion with an equivalent effect will be performed in the second case.

For conversion of formats, access operations and access right indicators the system will be supplied with parameters from higher level software layers or from a user programm indicating the source and destination formats for the conversion. Depending on these parameters the conversion will than take place either on the source or on the destination host.

4.3.5 Synchronization of Concurrent Accesses

Data access requests may be issued concurrently from a number of source host computer systems to one destination host. To allow shared access to data in the destination host a locking mechanism provides means to assure all users get access to consistent data. For that purpose user request (i.e., each association phase in the DATS protocol) is accompanied by a lock/unlock command (i.e., open/close command in the kernel command language) at its start and termination. This guarantees an exclusive write access or a shared read access to data allocated to the requesting process. The coordination of the concurrent accesses may be achieved as follows: For each user request a different instantiation of a remote data access process AFR/FTS is created on the destination host. Each of these instantiations gets the requested data allocated exclusively for writes and in a shared mode for reads. If several instantiations of remote access processes request accesses to common data concurrently they will be served in some priority order (e.g. FIFO) as defined in the remote data management system. This may be depicted in the following graph:



Instantiations of remote processes AFR/FTS[1] other than those requesting common data may be executed in an asynchroneous fashion. All of them are dedicated to perform a particular remote (i.e., remote to a calling RFA/FTM process) task. For completion of these tasks they are never impelled to call upon the service of any other remote processes on other hosts. Thus cyclic access may not be created and consequently deadlocks may not occur.

4.3.6 Resiliency Provisions

A rather great variety of resiliency provisions have been implemented for the back up of the system after failure in one of its function units. In principle resiliency will be gained by applying an error detection and signalling schema and by a time out mechanism. The implemented resiliency mechanisms are defined to provide means for detection of the following kinds of failures:

-detection of local failures by the RFA/FTM process
-detection of transmission errors by the end-to-end transport system
-detection of remote failures by the AFR/FTS process.

A netwide error code has been defined for the detection of the failure locality and the signalling of failures between different hosts in the network. A failure in the transmission of failure signals will be resolved by the time out mechanism.

5. Conclusion

The concept of a data management facility on a computer communication network has been introduced. The concept is different from others since it is primarily aimed for system evolution over time with changing user requirements. The resulting Evolutionary System Architecture (ESA) has also been developed for the sake of simplicity of the system structure and for high system performance.

The system is composed of two main components, the ESA front and system and the ESA kernel system. The front-end system consists of a number of highly autonomous application specific function units ASF's. The kernel system encompasses a number of components with different but somewhat overlapping functionality. Only one of the components will be selected for the execution of a data access request in a user program. The selection is based on performance criteria. This decision will be made in the so-called selector component of the kernel.

An evolutionary change of the system will be enabled by the composition of the front-end of highly autonomous modules. Modules may be added and removed from the system for its changes. The module concept also incorporates means for the preservation of the consistency of data during its concurrent execution. For the lack of space in this paper the required front-end level synchronization mechanism has not been described. The concept is to some extent similar to the "event count" notion in /KR 79/ and the interested reader is referred to a paper in preparation.

The main emphasis has been placed on the description of the kernel system. This is of particular importance since this system component has been in operation for some time now and its well-functioning has been validated. Performance considerations play an important role in the efficient use of data management capabilities on computer networks. For that reason the provision for an efficient computation of data access services in the kernel system is discussed in some detail. This problem is particularly acute for heterogeneous networks and their additional intrinsic overhead for the execution of format conversions. The kernel has consequently been designed to provide means for dealing with this problem by offering different options for the execution of data access requests with different performance characteristics.

In its continuation the project should lead to some improvements of its current version and to its extension by new components. The design of a well performing system usable in a realistic application environment will be our continuous concern.

6. BIBLIOGRAPHY

/AD 78/ Adiba, M. et. al. "Issues in Distributed Data Base Management Systems" Proceeding VLDB 78, Berlin 1978

/BB 78/ Butscher, B., Bauerfeld, W. L-., Popescu-Zeletin, R., "Data Access in Heterogeneous Computer Networks" Proceedings of the GI Symposium on "Distributed Databases" Karlsruhe, Germany, April 1978

/BB 79/ Baum, D., Bauerfeld, W.L-., Popescu-Zeletin, R., K. Ullmann "End-to-end level data flow analysis for communication networks" Proceedings of the International Symposium on Flow Control in Computer Networks, Versailles, France, Feb. 1979

/BS 78/ L-. Bauerfeld, W., Strack-Zimmermann, H.W. "The Hahn-Meitner-Institut Computer Network" Workshop on Data Communication, IIASA, Laxenburg, Austria, Sept. 1978

/CB 77/ Chandy, K.M., Baum, D.,-Bauerfeld, W.L-. Popescu-Zeletin, R., Ullmann, K. "Data Flow Analysis for File Transfer and Remote Data Access Protocols" to be published

/CH 75/ Chandy, K. M., Herzog, U., Woo L. "Parametric Analysis of Queueing Networks" IBM J. Res. Develop., Vol 19 (1975)

/HB 77/ Heinze, W., Butscher, B. " File Transfer in the HMI Computer Network" Third European Network User's Workshop, IIASA, Laxenburg, Austria, Apr. 1977

/K 78/ Kimbleton "Network Operating System - An implementation Approach" NCC - 1978

/KR 79/ Kanodia, R.K., Reed, D.P. "Synchronization with Eventcounts and Sequencers" Communications of the ACM, Vol. 22 (1979) No. 2

/KL 76/ Kleinrock, L., W. E. Naylor, H. Opderbeck "A Study of Line Overhead in the ARPANET" Communications of the ACM, Vol. 19 (1976) No. 1

/PB 78/ Popescu-Zeletin, R., B. Butscher "Specification of the RDA System in the HMINET
HMI-Report B 279, Okt. 1978

/PM 78/ Peebles, R., E. Manning "System Architecture for Distributed Data Management"
Computer Vol. 11 (1978) No. 1

/PZ 78/ Popescu-Zeletin, R., B. Butscher "A Study of Efficiency and Overhead of Data Access and Transfer Systems in the Heterogeneous Computer Networks"
German Chapter of ACM-Workshop on Communication Networks, Wiesbaden, Germany, June 1978

/PZB 78/ Popescu-Zeletin, R., B. Butscher "Access Systems for a Distributed Data Base"
Proceedings of the International Congress on Data Processing, Berlin, Germany, Sept. 1978

/RG 77/ Rothnie, J. B., N. Goodman "A Survey of Research and Development in Distributed Data Base Management"
Proceedings VLDB 11, Tokyo, Japan (1977)

/WE 78/ Weber, H. "Design of Coherent Application Systems by Autonomous User's in Hansen, H. R.
"Entwicklungstendenzen der Systemanalyse"
Oldenbourg Verlag München 1978

/WE 79/ Weber, H. "Modularity in Data Base System Design: A Software Engineering View of Data Base Management"
in Weber, H., Wassermann, A. I., "Issues in Data Base Management".
North Holland 1979

APPENDIX A

DATS Operations

A communication connection will be established upon a user request in a two-phase process.

The first phase - the remote connection phase - establishes the connection user/remote host. Information about user rights on the remote host, about the resources which should be allocated and granted to DATS for this user, and the maximal common transmission buffer between the two involved processes are the main negotiation subjects of this phase.

In the second phase, i.e., the association phase, a connection user/remote file is established. User access rights on the remote file will be checked in this phase. Several association phases are allowed during a remote connection phase.

After that the requested data are read or written by the remote DMS in the data access and transfer phase.

A user program communicates with the DATS by means of a set of language primitives.

a) Initialization primitives: they are associated to the remote connection phase and perform the following function:

INIT - defines the user access rights on the remote host;
- allocates a resource set in the RDA system;
- establishes buffer conventions between RFA and AFR.

DISCON - terminates remaining association phases;
- releases the resource set;
- ends the remote connection phase.

b) Control primitives: they may be executed in the association phase and operate in the following manner:

RFCB - generates a Remote File Control Bloc, (RFCB) which is the information carrier between a user and a remote DMS; the parameters of this primitive describe the attributes of the remote file.

ROPEN - establishes a connection user-file;
- transfers the Remote File Control Block to AFR;
- prepares the remote file (local open).

RCLOSE - terminates the remote file handling;
- returns information about the state of the file;
- cancels the connection.

The status primitive generates a temporary association phase:

RSTAT - establishes a temporary connection user file;
- returns information about the remote file;
- cancels the connection.

c) Action primitives: they may be executed in the data access and transfer phase and perform the following tasks:

RPUT - writes or reads a data structure in a user;
RGET - specified format to or from the remote system;
RCNTR - controls a remote file (pointer position, etc.).

These action primitives are translated locally into DMS specific access programs.

Appendix B

The queueing model

It is well known that an exact analysis of general queueing networks is either impossible up to now or leads to an unacceptable amount of computation. For queueing networks of GORDON-NEWELL type, for instance, the necessary amount of computation rises significantly with the number of service stations and circulating customers. The authors, therefore, decided to model the real system by a closed local balanced queueing network of the above mentioned type and to use a decomposition method suggested by Chandy et al. /CH 75/. It has already been shown for the case of end-to-end level data flow that this simplification leads to acceptable results (see /BB 79/). The main idea is to represent each local system in the network by a central server queueing system with exponentially distributed service times (CPU as central server), and to compose then each such local system into one single queue. An equivalent exponential central server queueing network is constructed this way in which the central server represents the network transport system (NTS). The average class dependent service rates of the composite queues are calculated as the values of class dependent throughput rates of the shortened "rest of the network" queues (see /CH 75/ and /BB 79/). They in fact are dependent on the vector $M = (m(1),\dots,m(V))$ of actual numbers $m(v)$ of customers of class v in the very local system under consideration ($v=1,\dots,V$, with V the total number of customer classes in the network).

In order to reduce the enormous difficulties in calculating the throughput rates for such complex server activities the customer number specific means $\mu_c^k(s)$ of the service times $\mu_c^k(K;s)$ are used instead of the $\mu_c^k(K;s)$ itself in further calculations (the error seems to be negligible); here K denotes the vector of numbers $k(v)$ of net customers of class v visiting the local system S^k which in turn is characterized by an upper index $k \geq 1$. The upper index 0 denotes the NTS-server. Each $k(v)$ corresponds to some "fixed" number $b(v)$ of "background customers" (local programs or tasks or requests, etc.) in the local system, such that $b(v)+k(v)=m(v)$.

Since k(v) is the only variable number (v=1,...,V), it is allowed to write $\mu_c^k(K;s)$ instead of $\mu_c^k(M;s)$. The lower index c always points towards the fact that composite queues S_c^k are considered to represent the actual local system S^k. Let K(s) be the total number of circulating net customers of class s and let

$$P^k[k(1),\dots,k(s-1),k(s)=n,k(s+1),\dots,k(V)] =: p_s^k(n)$$

denote the probability that there are k(v) customers of class $v \neq s$ and n customers of class s in the system S^k $(v \in \{1,\dots,V\})$. Then, using $\mu_{cs}^k(n)$ for $\mu_c^k(k(1),\dots,k(s)=n,\dots,k(V);s)$, we have

$$\mu_c^k(s) = \sum_{Q(s)} \sum_{n=1}^{K(s)} \mu_{cs}^k(n)\; p_s^k(n)$$

(Q(s) is the set of all vectors (k(1),...,k(s)=n,...,k(V)) with $0 \leq k(v) \leq K(v)$ for $v \neq s$).

Moreover, let $\Pi^k(K;s;n)$ denote the probability that there are n=k(s) customers of class s in the system S^k: $\Pi^k(K;s;n) = \sum_{Q(s)} p^k(n)$, then

$$\bar{N}^k(K;s) = \sum_{n=1}^{K(s)} n\,\Pi^k(K;s;n)$$

is the expected number of customers of class s waiting or being served in queue S^k. Based on an elaborate analysis (the interested reader is referred to a paper in preparation) this yields

$$T^k(K;s) = \frac{\{1-\Pi^k(K;s;0)\}\; \bar{N}^k(K;s)}{\sum_{v=1}^{V} \mu_c^k(v)\; \bar{N}^k(K;v)}$$

for the class dependent throughput through queue S_c^k.

Based on Little's result for the relation between average response time, average queue length and average arrival rate in steady state we may get

$$ {}^{k}(K;s) = \frac{\sum_{v=1}^{V} \bar{N}^{k}(K;v)\mu_{c}^{k}(v)}{1-\hat{\pi}^{k}(K;s;0)} + \mu_{c}^{k}(s) $$

for the value of the average response time of class s concerning queue S_c^k .

These results and other formulas (for needed buffer space, average server utilization, waiting times etc.) may be used upon selection of DATS components:

$$ U^{k}(K;s) = 1-\hat{\pi}^{k}(K;s;0) \qquad \text{(utilization)}, $$

$$ T^{K}(K;s) = \frac{U^{k}(K;s)\ \bar{N}^{k}(K;s)}{\sum_{v=1}^{V} \mu_{c}^{k}(v)\ N^{k}(K;v)} \qquad \text{(throughput)}, $$

$$ w^{K}(K;s) = \frac{N^{k}\ (K;s)}{T^{k}(K;s)} \qquad \text{(waiting time)}, $$

$$ \tau^{k}(K;s) = w^{k}(K;s) + \mu_{c}^{k}(s) \quad \text{(response time)}. $$

Obviously the response time for each traffic type also includes the service times of the server NTS for the transfers in both directions, i.e. for both user classes v_{i1} and v_{i2}. Thus, for the execution of the RFT component for example, the RFT response time may be computed from

$$R^k(tr_3) = {}^k(K;v_{31}) + \mu_c^0(v_{31}) + \mu_c^0(v_{32}),$$

provided that there are K customers at all travelling through the network at that time.

Example

In the following example we now restrict our attention to the selection of one of the two RFT or RDA in the execution of a user transaction. Files to be manipulated from remote are supposed to be of fixed average length f, i.e. some constant r shall represent the estimated average number of records in a file. Each record in turn is assumed to be of fixed length p equal to the data length of one "transport element" in the network; thus, f=r p. For the passing of parameters together with each RDA request let us suppose furthermore that each such request will also correspond in length to one transport element, independent of the type of the requested operation in the data access and transfer phase; thus,

$$\mu_c^0(v_{41}) = \mu_c^0(v_{42}).$$

Then, RDA and RFT performance can be compared with respect to the response times as follows:

If m records of some file residing at system S_c^k shall be manipulated, the expense of time D^k (including network delay) for the RDA system is

$$D^k(K;tr_4) = m_0 \left\{ \tau^k(K;v_{41}) + 2\mu_c^o(v_{41}) \right\},$$

A Concurrency Control Mechanism for Distributed Databases Which Uses Centralized Locking Controllers
H. Garcia-Molina
Stanford University, Stanford, California

On Efficient Monitoring of Database Assertions in Distributed Database Systems
D.Z. Badal
University of California at Los Angeles, Los Angeles, California

3:30 p.m - 4:00 p.m. Break

4:00 p.m. - 5:30 p.m.

PROTOCOL MODELING
Chairperson, to be announced

A Study of the CSMA Protocol in Local Networks
S.S. Lam
University of Texas, Austin, Texas

Global and Local Models for the Specification and Verification of the Distributed Systems
M. Gouda, D. Boyd, and W. Wood
Honeywell, Bloomington, Minnesota

Protocols for Dating Coordination
D. Cohen and Y. Yemini
USC-ISI, Marina del Rey, California

Wednesday, August 29, 1979

9:00 a.m. - 10:30 a.m.

MULTIPLE COPY CONTROL TECHNIQUES
Session Chairman: Mr. Ed Birss
Hewlett Packard, Cupertino, California

Distributed Control of Updates in Multiple-Copy Databases: A Time Optimal Algorithm
R.J. Ramirez and N. Santoro
University of Waterloo, Waterloo, Ontario, Canada

Concurrency Control in a Multiple Copy Distributed Database System
W.K. Lin
Sperry Research Center, Sudbury, Massachusetts

Concurrency Control Algorithm for Distributed Database System
T. Minoura
Stanford University, Stanford, California

10:30 a.m.- 11:00 a.m. Break

11:00 a.m.- 12:00 m.

LOCAL NETWORKS PANEL

Session Chairman: Dr. John Shoch
Xerox-PARC, Palo Alto, California

12:00 m. - 1:30 p.m. Lunch

1:30 p.m. - 3:00 p.m.

DATABASE MACHINES PANEL

Session Chairwoman: Dr. Paula Hawthorn
Britton-Lee, Berkeley, California

The designers of state-of-the-art database machines will discuss the roles of their machines in future distributed systems. Also discussed will be the major design differences and target applications for the machines.

Panel:

Stewart Schuster, Tandem Computers
Harvey Freeman, Sperry Research
Mike Stonebraker, U.C. Berkeley
David DeWitt, University of Wisconsin

2:30 p.m. - 3:00 p.m. Break

3.00 p.m. - 5:00 p.m.

NETWORK RESOURCE ALLOCATION

Session Chairman: Dr. Yogen Dalal
Xerox-SDD, Palo Alto, California

Synchronization of Distributed Simulation Using Broadcast Algorithms
J.K. Peacock, E. Manning and J.W. Wong
University of Waterloo, Waterloo, Ontario, Canada

The Updating Protocol of the ARPANET's New Routing Algorith A Case Study in Maintaining Identical Copies of a Changing Distri ed Data Base
E.C. Rosen
Bolt Beranek and Newman, Cambridge, Massachusetts

The NIC Name Server -- A Datagram Based Information Utility
J.R. Pickens, E.J. Feinler and J.E. Mathis
SRI International, Menlo Park, California

A Protocol for Buffer Space Negotiation
D. Nessett
Lawrence Livermore Laboratory, Livermore, California

6:00 p.m. -10:00 p.m. Bay Cruise and Dinner

Thursday, August 30, 1979

9:00 a.m. - 10:30 a.m.

IMPLEMENTATION OF DISTRIBUTED SYSTEMS - II
Session Chairman: Dr. Daniel Sagalowicz
SRI International, Menlo Park, California

Labeled Slot Multiplexing: A Technique for a High Speed, Fiber Optic Based, Loop Network
S. Blauman
TRW, Redondo Beach, California

A Distributed File Manager for the TRW Experimental Development System
S. Danforth
TRW, Torrance, California

Network Support for a Distributed Data Base System
L.A. Rowe and K.P. Birman
University of California at Berkeley, Berkeley, California

10:30 a.m. - 11:00 a.m. Break

Transaction Processing in the Distributed DBMS-POREL
U. Fauser and E. Neuhold
University of Stuttgart, Stuttgart, W. Germany

An Evolutionary System Architecture for a Distributed Data Base Management System
H. Weber, D. Baum and R. Popescu-Zeletin
Hahn-Meitner Institute, Berlin, W. Germany

U.S.GPO:1979-692-475/323

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.