GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

SPONSORED PROJECT TERMINATION

Date: 7/24/81

Project Title: Workshop on Interprocess Communication in Highly Distributed Systems

Project No: G-36-632

Project Director: Dr. Phillip H. Enslow, Jr.

Sponsor: U. S. Army Research Office; Research Triangle Park, NC 27709

Effective Termination Date: 8/12/79

Clearance of Accounting Charges: 8/12/79

Grant/Contract Closeout Actions Remaining:

- x Final Invoice and Closing Documents
- _ Final Fiscal Report
- x Final Report of Inventions
- x Govt. Property Inventory & Related Certificate
- _ Classified Material Certificate
- _ Other _____

Assigned to: I & CS (School/Laboratory)

COPIES TO:

Administrative Coordinator
Research Property Management
Accounting Office
Procurement Office/EES Supply Services
Research Security Services
✓ Reports Coordinator (OCA)
Suspense

Legal Services (OCA)
Library, Technical Reports
EES Research Public Relations (2)
Project File (OCA)
Other: _____

AD G36-632

FINAL TECHNICAL REPORT
GIT–ICS–79/11

INTERPROCESS COMMUNICATION IN HIGHLY DISTRIBUTED SYSTEMS

———A Workshop Report———
20–22 November, 1978

By

Philip H. Enslow, Jr.
Robert L. Gordon

Prepared for
U. S. ARMY RESEARCH OFFICE
P. O. BOX 12211
RESEARCH TRIANGLE PARK, N. C. 27709
Under
Contract No. DAAG29–79–C–0010

ARO Project No. P–16334–A–EL
GIT Project No. G36–632

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332

1979

THE RESEARCH PROGRAM IN
FULLY DISTRIBUTED PROCESSING SYSTEMS

INTERPROCESS COMMUNICATION IN HIGHLY DISTRIBUTED SYSTEMS

--- A Workshop Report ---

20 - 22 November, 1978

FINAL TECHNICAL REPORT

GIT-ICS-79/11

Philip H. Enslow Jr.
Robert L. Gordon*

December, 1979

U.S. ARMY RESEARCH OFFICE
P.O. Box 12211
Research Triangle Park, North Carolina 27709

ARO Grant Number DAAG29-79-C-0010
ARO Project Number P-16334-EL
GIT Project Number G36-632

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

* (PRIME Computer, Inc.)

THE VIEWS, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Interprocess Communication in Highly Distributed Systems - A workshop Report - 20 to 22, November 1978.	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 13 Nov 1978 - 12 Aug 1979
	6. PERFORMING ORG. REPORT NUMBER: GIT-ICS-79/11
7. AUTHOR(s): Philip H. Enslow, Jr. Robert L. Gordon (Prime Computer, Inc.)	8. CONTRACT OR GRANT NUMBER(s): DAAG29-79-C-0010
9. PERFORMING ORGANIZATION NAME AND ADDRESS: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709	12. REPORT DATE: December 1979 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): U.S. Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

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

Interprocess Communication
IPC
Distributed Processing
Computer Networks
Distributed Operating Systems

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

Interprocess Communication (IPC) has been recognized as a critical issue in the design and implementation of all modern operating systems. IPC policies and mechanisms are even more central in the design of highly distributed processing systems -- systems exhibiting short-term dynamic changes in the availability of physical and logical resources as well as interconnection topology. A workshop on this subject was held at the Georgia Institute of Technology in November 1979. Four working groups, 1) Addressing, Naming,

20. cont.
and Security, 2) Interprocess Synchronization, 3) Interprocess Mechanisms, and 4) Theory and Formalism, addressed the current state of the art in these areas as well as problems and future research directions. This report incorporates much of the material and working papers from those fields as well as selected references useful in understanding the topic.

ABSTRACT

Interprocess Communication (IPC) has been recognized as a critical issue in the design and implementation of all modern operating systems. IPC policies and mechanisms are even more central in the design of highly distributed processing systems -- systems exhibiting short-term dynamic changes in the availability of physical and logical resources as well as interconnection topology. A workshop on this subject was held at the Georgia Institute of Technology in November 1979. Four working groups, 1) Addressing, Naming, and Security, 2) Interprocess Synchronization, 3) Interprocess Mechanisms, and 4) Theory and Formalism, addressed the current state of the art in these areas as well as problems and future research directions. This report incorporates much of the material and working papers from those fields as well as selected references useful in understanding the topic.

PREFACE

The workshop organizing committee had originally intended to utilize the material developed by the individual working groups to prepare a summary report of the proceedings. This concept was abandoned when it was recognized that a "summary report" would not adequately report on and document all of the work and topics that were covered during the meeting. It was obvious that documentation much more thorough than merely a summary report was warranted, so the members of the organizing committee decided to directly utilize as much as possible of the material and notes prepared by the working groups and assemble and edit that material into an organized workshop report. It was felt that this approach would much better capture the true flavor of the workshop and the breadth of the material covered there.

December, 1979

Philip H. Enslow, Jr.
Robert L. Gordon

ACKNOWLEDGEMENTS

Certainly the most important acknowledgement for assistance in the preparation of this report goes to the working group leaders who prepared the summary reports for their individual groups and to those individuals who acted as recorders during the working groups sessions. To a great extent the material developed by those individuals has been utilized exactly as it was prepared with only minor editing. We would also like to acknowledge the invaluable assistance of two Georgia Tech students who were responsible for the mechanical organization and preparation of the report utilizing our text editing system - Timothy Saponas, who also served as our resident translator for the hieroglyphic notes prepared by session records, and Shelly Smith.

We would also like to acknowledge the support of the U.S. Army Research Office and the U.S. Air Force Office of Scientific Research in funding the Workshop as well as the Office of Naval Research which also partially supported the preparation of this report.

December, 1979

Philip H. Enslow, Jr.
Robert L. Gordon

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

1.1 OBJECTIVES OF THE WORKSHOP

The subject of the workshop was Interprocess Communication Mechanisms with a particular focus on process-to-process communications in highly distributed systems. Highly distributed systems are characterized by very loose coupling between physical resources as well as between logical resources. Such systems also exhibit dynamic, short-term changes in the topology and organization of the total system. These characteristics place new requirements on the design and performance of IPC mechanisms; these requirements are assuming extreme importance in advancing the state-of-the-art in all forms of distributed systems.

1.2 WORKSHOP ORIGINS

The last meeting that focused on interprocess communication was the "ACM SIGCOM/SIGOPS Interprocess Communications Workshop" held 24-25 March, 1975. [IPC 75]

One might conclude from the paucity of material published on this topic since that workshop that the problem is totally under control. (The BBN "Network Operating Systems" study [THOM 78] cites only one reference since 1974.) Such is definitely not the case. Work on IPC's has been covered within projects on operating systems; however, many implementation and performance problems are only partially solved or solved only on an ad hoc basis, and it appeared that the time was ripe to again focus a meeting of specialists onto this topic, especially in view of its key role in the operation and performance of distributed systems.

Since 1975 advances in the field of computer communications have provided mechanisms for connecting computers together in a variety of configurations. For instance, high speed serial communication paths [METC 76, GORD 79] have permitted effective local networks [CLAR 78], in which many computers share specialized resources (storage, printing facilities, etc.), while each node still retains some degree of autonomy. In addition, many mini-computers support large address spaces, and a corresponding high degree of mul-

tiprogramming. One natural way to construct the software for such systems is to base the software architecture on the notion that most tasks will be performed by a collection of communicating asynchronous processes, running on the same or different processors. Such systems are known as "highly distributed systems", and are characterized by a very loose coupling between physical resources as well as between logical resources, and they allow dynamic, short-term changes in the topology and organization of the total system.

The fact that these systems are very loosely coupled, both physically and logically, places quite different demands on IPC from those applicable to more tightly coupled contemporary systems, even those incorporating a local network as the interconnection mechanism. Practical attempts to construct such systems immediately direct ones attention to available Interprocess Communication (IPC) mechanisms and their shortcomings. Lack of well constructed and well understood mechanisms is the root of most of the difficulties in building distributed systems.

1.3 PURPOSE AND SCOPE OF THE WORKSHOP

The "Workshop on Interprocess Communications in Highly Distributed Systems" was intended to bring together a selected group of workers in the subject area to address the five general goals listed below:

1) Assess the present state-of-the-art for IPC mechanisms in distributed data processing systems
2) Identify the data available on the actual performance of various IPC policies and mechanisms.
3) Assess the potential value of various IPC mechanisms satisfying the operational and performance requirements for highly distributed systems.
4) Identify shortcomings in the present state-of-the-art and identify promising areas for future research and experiment on this subject.
5) Identify possible standardization levels of IPC.

The scope of the workshop will be limited to IPC mechanisms for use in distributed systems. (This acknowledges fairly common agreement among the research community that the following are not DDP's --- multiprocessors, computer networks per se, intelligent terminal systems, and satellite processor systems.)

1.4 STRUCTURE OF THE WORKSHOP

Workshop attendees were selected from individuals actively working in the field, and the size of the workshop was purposely limited to approximately 40 attendees. Special attention was given to obtain participants who met one or more of the following criteria:

- Had had practical experience in the design and implementation of IPC policies and mechanisms in highly distributed systems.
- Had analyzed and/or measured the actual performance of various IPC mechanisms.
- Would contribute a written submission to the workshop.

The workshop was held from 12:00 noon, 20-November, thru 12:00 noon, 22-November, 1978, at the Atlanta Townehouse Motor Hotel, immediately adjacent to the Georgia Tech campus.

Before the workshop, invitees were requested to identify their areas of interest. Based on that input, the organizing committee established six working groups:

1) Addressing and Security
2) Fault Tolerance
3) Synchronization, Signalling, and Flow Control
4) Theory and Formalism
5) Hardware and Primitives
6) Programming Issues

However, as often (usually?) happens in such situations, when the groups met and discussed their areas of interest, realignments in the working group organization resulted in four working groups rather than six.

1) Addressing, Naming, and Security
2) Interprocess Synchronization
3) Mechanisms
4) Theory and Formalism

The output of these four groups is the basis for this report.

1.5 ATTENDEES

IPC WORKSHOP

LIST OF ATTENDEES

(* Members of the Organizing Committee)

Hal Abelson
Laboratory for Computer Science
Massachusetts Institute of Technology

Allen Akin
Georgia Institute of Technology
School of Information & Computer Science

Edwin Basart
Hewlett-Packard Co.
General Systems Division

Morton I. Berstein
System Development Corp.

Bill Buckles
General Research Corp.

James E. Burns
Georgia Institute of Technology
School of Information & Computer Science

Gregory Chesson *
Bell Laboratories

Wushow Chou
North Carolina State University
Computer Studies

Phillip Crews
Georgia Institute of Technology
School of Information & Computer Science

Richard A. DeMillo
Georgia Institute of Technology
School of Information & Computer Science

Philip H. Enslow, Jr. *
Georgia Institute of Technology
School of Information & Computer Science

Michael Fischer
University of Washington
Department of Computer Science

Mark Gang
Ford Aerospace & Communications Corp.
Western Development Laboratories

Robert L. Gordon *
PRIME Computers

Jim Hamilton
Digital Equipment Corp.

Mohommad Hassan
MODCOMP

Steven F. Holmgren
Digital Technology, Inc.

Doug Jensen *
Honeywell Research
(Presently Carnegie-Mellon University)

Richard Kain
University of Minnesota
Department of Electrical Engineering

Steve Kimbleton
Institute for Computer Science & Technology
National Bureau of Standards

Peter Koschewa
U.S. Army Institute for Research in Management
Information and Computer Sciences

Leslie Lamport
SRI International

David Lapin
Burroughs Corporation
Computer Systems Group

Thomas Lawrence
Rome Air Development Center
U.S. Air Force

Richard LeBlanc
Georgia Institute of Technology
School of Information & Computer Science

Gerard Le Lann
SIRIUS
IRI (France)

Edward Y.S. Lee
TRW Defense & Space Systems Group

Jon Livesey
University of Waterloo
Computer Communications Network Group

James R. Low
University of Rochester
Department of Computer Science

Nancy A. Lynch
Georgia Institute of Technology
School of Information & Computer Science

Edith Martin
Georgia Institute of Technology
Engineering Experiment Station

Wayne McCoy
Kennedy Space Flight Center
NASA

Nancy Meisner
University of Waterloo
Computer Communications Network Group

Ira Newman
Department of Defense

Richard Peebles
Digital Equipment Corp.

Steve Ratzel
U.S. Army Institute for Research in Management
Information and Computer Sciences

Donald Sharp
Georgia Institute of Technology
School of Information & Computer Science

David Sincoskie
University of Delaware
Department of Electrical Engineering

Stephen W. Smoliar
General Research Corp.

John Staudhammer
U.S. Army Research Office

Carl Sunshine
Rand Corporation
(Present location: ISI, University of Southern California)

Joseph S. Sventek
Lawrence Berkeley Laboratories
Computer Science & Applied Mathematics

P. S. Thiagarajan
Institut fuer Informations-systemforschung
GMD

Virgil E. Wallentine
Kansas State University
Department of Computer Science

Don Weir
Telenet Communication Corp.

Douglas E. Wrege
Georgia Institute of Technology
Engineering Experiment Station

1.6 ORGANIZATION OF THIS REPORT

Following this introductory section, there is a short section on the general background of interprocess communication techniques. The main body of this report is Sections 3, 4, 5, and 6 which cover the results of each of the Working Groups. Within each section, the first material presented is a summary of the Working Group presentation made at the end of the workshop. Following that, there is, in some instances, a collection of amplifying material and selections from the position papers that were prepared prior to the workshop and distributed to the attendees.

Section 7 contains several longer papers that were either prepared specifically for distribution at the workshop or were felt by the authors to be applicable to the workshop and were distributed to the attendees there. Section 8 is a very brief summary and discussion of future directions for IPC and Section 9 contains the references utilized in the report.

SECTION 2

BACKGROUND

2.1 INTRODUCTION

Probably the single most important hindrance to the development of interprocess communication has been the lack of general acceptance and agreement on the notion and abstraction of a "process." Until the "process model" of computation becomes generally accepted and used as the basis of software architectures, there will be little motivation for interprocess communication mechanisms.

In most systems the abstraction of a "process" has not been developed well enough for it to be treated as an "object" in its own right so that "processes" can be used conveniently by system architects and others as building blocks. Primitives for the creation, synchronization, addressing, and communication of processes have in the past only been generally available to operating system developers, and therefore not widely used by application programmers in applications software systems. Unfortunately operating system developers tend to live with and use poorly documented experimental primitives and other ad hoc mechanisms. The notable exceptions to this rule form the core body of classic literature in this field [BRIN 69, DIJK 68b, DIJK 71, DALE 68]. For the most part, application programmers in the past have been restricted to conventional I/O using shared files as a pragmatic method of interprocess communication, with only partial success.

When the notion of a "process" becomes recognized as a fundamental building block for distributed applications, stronger support and documentation will have to be provided by the system suppliers and manufacturers, thus making available to application coders a robust set of "process-based" primitives. After such widespread support materializes, the design experience and performance statistics will provide the basis for a fuller understanding of all aspects of interprocess communication.

A comprehensive survey of the present state-of-the-art in interprocess communication is presented in paragraph 7.6.

2.2 PROCESS MODEL OF COMPUTATION

An excellent survey of the "process model of computation" can be found in [HORN 73]. Prior to this, articles on operating systems developed the notion of a "process" or "task," as an entity that could be scheduled and own other resources in multiprogrammed systems, but they did not treat a process as a structuring methodology in its own right. Examples of these notions can be found in [SALT 66] and [IBM 71].

Access to resources in early operating systems presented the very first examples of interprocess communication, but these early IPC techniques varied widely from one implementation to the next. For example, in most systems, the line printer daemon (or process) owned the line printer, and access to the printer was restricted to ordinary "write" statements at the language level coupled with "logical unit" assignment at the job control of command language level. Other examples may be found where the login process "owns" the communication lines, or a file manager owns the file system as in the MERT operating system [LYCK 78]. An early message-based operating system structured around processes is the RC4000 operating system [BRIN 69, BRIN 70].

Trends in software engineering, applications, and technology certainly point to an increasing awareness of a process as a fundamental method of structuring systems. The proliferation of inexpensive processors and low cost bandwidth suggest a process model of computation, even if there is only one process per processing element, since control and sharing of common resources must be by some form of interprocess communication. New architectures are now being proposed that exploit these trends, e.g. [NELS 78]. The [NELS 78] proposal is based on a high-speed packet-oriented bus interconnecting a large number of processor-memory pairs, termed "cells." Each cell includes a CPU, a primary memory system (typically one or two megabytes), a packet bus node controller, and possibly some peripherals such as disks or communications devices. The architecture supports applications decomposed at the process level; the entire system is viewed as a set of cooperating processes, distributed among the cells to improve performance, cost, or availability.

2.3 HIGHLY DISTRIBUTED SYSTEMS

Highly distributed systems are characterized by very loose coupling between physical as well as logical resources. In addition they exhibit dynamic, short-term changes in the

topology and organization of the total system. The fact that these systems are very loosely coupled, both physically and logically, places quite different demands on IPC from those applicable to more tightly coupled contemporary systems, even those incorporating a "network" as the interconnection mechanism.

Such systems should support multiple name spaces, including the management and translation of file and unit names in these name spaces. In addition, such systems should handle abstractions built from collections of communicating processes and provide mechanisms for addressing and synchronizing groups of processes. High bandwidth message transport mechanisms will potentially allow multiple logical connections between processes to be constructed whenever convenient, but system support must be available for those connections to be useful. To date, very little experience is available to assist a designer attempting to construct complex systems out of communicating processes.

2.4 IPC STRUCTURES

Most existing IPC primitives and structures are based on a "two-party" communication model, in which there is a single "sender" and a single "receiver" for each transaction or message. (This is certainly the basis for IPC facilities built around the X.25 level 3 protocol [CCIT 78].) Other kinds of communication facilities may better support ring, tree and general graph models of process networks. Protocols involving more than two processes are called "N-process" protocols [PARD 79]; they should find use in shared data base and electronic mail systems.

The major functions supporting these protocols are storing, forwarding and routing variable length messages. These functions can be difficult to implement if communication links, processing nodes, or other resources are only partially available.

2.5 INTERPROCESS CONTROL STRUCTURES

Communication links between processes can be allocated strictly to control functions. In fact, the degree of separation of control and data is an important research is-

sue. A path primarily used for the transport of data may have no mechanism for control or "out of band" signalling, which may make error detection and recovery difficult, if not impossible. The system's control path structure is primarily determined by the "control model" used during system development. The "classical" system organizations are a) master/slave, b) hierarchical, c) democratic, or d) autonomous. The first two are well understood and readily implemented, while the latter control organizations are not well understood (in an algorithmic sense) and are the subject of much research [HOAR 78].

SECTION 3

ADDRESSING, NAMING, and SECURITY

3.1 WORKING GROUP SUMMARY REPORT

What are objects

files, processes, devices

Uniform mechanism?

File metaphor -- UNIX

Process metaphor -- MININET, RC4500

Abstractions -- WEB

Worldview: (a la DISY)

Universe >>> Systems >>> Objects

Distinguish between:

NAMES -- what

ADDRESSES -- where

ROUTES -- how to reach

Basic Problem: map

NAMES >>> ADDRESSES

Desirable features:

Generic naming

Context independence

Location independence

Broadcast (group name)

Uniqueness

Path addressing

Other concerns:

Flat vs. hierarthical

Centralized vs. distributed

Steps

Search rules

Connections

Transactions

Merging two systems:

1. one below other
2. both below new prefix
3. corresponding unused addresses

Name >>> Address mapping may be separate from IPC.

IPC between specific addresses

Directory object with well-known address

DISY "MAILBOX"

Generic naming

Location independent

Uniqueness

Object pointer

Resource limits

Access controls

Security

Main attributes of subject:

Logical identity

Physical location

Problems:

1. authentication + access control of location
2. storing authorization on areas outside security environment
3. moving objects if encryption based on location

3.2 AMPLIFYING MATERIAL

What are objects? files, devices, processes

- What things should be in a list of primitive objects?
- Should we choose one object type to represent all objects?

Should there be a uniform mechanism for all objects?

- file "metaphor" - Unix [THOM 74]
- process "metaphor" - Mininet [PEEB 78], RC 4000 (performance?)
- abstractions
    - WEB at DEC (performance?)
    - Capability based systems

Uniform mechanism is a good thing. Being able to do this requires picking one of the above. Not sure we can.

Worldview: ANSI/SPARC/DISY [DESJ 78] or ISO SC 16 model

- Universe consists of multiple systems.
- Systems have many objects.

Distinguish Between Names (what), Addresses (where), Routes (how to reach). (see [SHOC 78])

Basic Problem: mapping NAMES to ADDRESSES.

Desirable features of this mapping:

1) generic naming - many potential servers
    - within one system or across systems
    - selected by server or by requestor ("request for service" facility is just latter [FARB 73])
2) location independence - same name may be used no matter where server is located
3) broadcast - (group name) - communication with multiple servers
4) uniqueness - only one name for given object or set of objects at some level
5) path addressing or source routing - source specifies sequence of addresses to reach ob-

ject. Useful if "system" does not know route, or if destination is outside normal name space.

Additional mapping concepts:

1) Flat vs. hierarchical - latter allows each directory or switch to know only about elements at its own level --> many smaller directories vs. one large one.
2) Centralized vs. distributed - centralized can be reliable, but requires roundtrip delay to get information, high load at center. Distributed may allow local lookup, or may require broadcast. Update more complex.
3) There may be many directories, and many "steps" in the address lookup. Example: "my name" to global name, global name to system address/local name, (send to remote system), local name to local address.
4) Search rules - each user may have rules for tailoring lookup to his needs.

NAME --> ADDRESS mapping may be costly. Hence desire to do it once for many successive messages to same destination. Leads to connection notion. May include route setup. Cacheing of recently used names/addresses also helpful. Connection also needed when desired that successive messages to a given name go to the same object, in order. If transactions are independent, then a different instance of the named object can serve each - no connection needed. [NSW 76]

Problem of merging two previously independent systems:

1) May add "prefix" to all addresses (a higher level in hierarchy) to distinguish systems.
2) Make one system "below" other in hierarchy.
3) Make unused addresses in each system correspond to addresses in other system. Only good for small numbers.

NAME --> ADDRESS translation may be separate from basic IPC which is between specific addresses only. Then directory object (process) with well-known address can be accessed to provide translation, with result returned via basic IPC. Then requestor does basic IPC with specific address of service actually desired. Examples: ARPANET Initial Connection Protocol, Mininet [PEEB 78].

Important Example: Our view of DISY "mailbox" [DESJ 78] has properties or components:

- generic name
- location independent
- uniqueness
- pointer to object (process) mailbox stands for
- resource control (how many in use)
- access controls, owner

Security:

1) Does not include reliability, failure recovery.
2) Does include authentication, access controls, encryption, correctness.
3) Basic goal - allow objects to be accessed only by specified subject.
4) Two main attributes of subject:

   - logical identity
   - physical location

5) Problems:

   a) Allow object to be accessed from one place but not another (e.g., not via dial-in). Must authenticate location as well as identity.
   b) Removable media plus unsecured sources: Can authorization information be stored in areas outside of physical control?
   c) Encryption problem. If authorizations are encrypted based on location of object, how can object move? (Two constraints: need to give authorizations to others, but must not be forgeable (hence encryption)).

3.3 CASE STUDIES

3.3.1 Distributed Data Bases

by

Edward Lee
TRW

Most DDB protocols seem to assume that Data Base Managers can figure out how to communicate between themselves and that naming one another is not a problem. Is it reasonable to assume that file system operations and process IPC are basically the same mechanism? DISY has process as the basic communicating object. You basically open a channel to a process and then communicate directly with it. It is the Session Controller (DISY) which opens the channel for you.

3.3.2 Mininet

by

J. Livesey
University of Waterloo

Mininet is a system in which addressing is basically separate from IPC. In many systems some form of addressing method (name --> address translation) is implicit in IPC.

In Mininet, IPC consists solely of the transmission of a message from a Sender Task to a Receiver Task which has to be identified by an integer Task Identifier (an address rather than a name). In the distributed case the host id is concatenated with the task identifier within the host.

The question then is how to get the task identifier for a task to perform a particular function.

In fact, all system resources (tasks, files, devices, directories, ...) are formalized as tasks. A task has code and data segments. A file, for instance, is a task whose code segments are the Access Method and whose data segments are code segments. A file task gets messages of the form:

```
read (record #)
```

and reacts by returning a message to the user containing the record data.

There is only one well-known task in each host, the Directory Task which has the responsibility to maintain a list relating function name (a character string) to task identifier for each task in this host. As the ultimate parent of each task he can find out their task ids. (Task identifier of a new task is returned to the creating task, the parent.) Now, when user task A, for instance wants to perform

open (filename)

it does so by asking the directory task for the identifier of the "file-open" task. Assuming this exists locally, the directory task returns its task id. The user now communicates directly with "file-open" (a la DISY session) and sends it a message

"open (filename)"

The task "file-open" now creates a file task whose data segments are the data records of "filename" and returns the "file" task identifer to the user task.

The user task now communicates with the "file" task (a second host session a la DISY) with messages

"read (record #)"
"write (record #)"
"close ()"

The "file-open" task handles mutual exclusion on the file (by refusing to create new file tasks for the same file as long as someone has it open to write). The "file" task handles record mutual exclusion.

In the case where no task exists in the local hosts to handle function "X" the local directory task talks to remote directory tasks, who are responsible for knowing which tasks exist in their hosts (and which can be created to do "X"). Directory tasks announce themselves to one another at boot time.

References:

[PEEB 78]

[LIVE 78a]

[LIVE 78b]

3.3.3 Discussion

Meisner:
Is this more complicated than a straight function CALL/RETURN system?

Livesey:
Yes, but more flexible since you can impose a function CALL/RETURN system on top of the basic task/message-passing system using library routines if you want. It is also assumed that we have a homogeneous system.

Sunshine:
Clearly we can have server processes to guard and administer

directories
open function
file tasks
etc.

Lapin:
We need hardware to support process invocation/context switch better than at present.

Livesey:
Yes, but future hardware should not lock us into function call/process invocation capabilities, etc.

Sunshine:
Curiously, in Mininet, every resource (object) is a task (process), but the creation of a process involves reading a file (an object containing its code segments).

Enslow:
Lee says that his distributed data base should be redundant. Does the system itself select the optimal record!

Lapin:
Redundancy increases the reliability of the system.

Livesey:
We have both homogeneous and heterogeneous redundancy here.
Homogeneous

- identical copies of data
- increases reliability

Heterogeneous

- copies of non-identical objects to perform similar functions, eg. FORTRAN compilers

- increases system band width

McCoy:
Can we get a system to give us both!

Sunshine:
To do it across several systems has a cost and we have to ask if the utility of redundancy is worth the cost. The ARPANET Resource Sharing Executive (RSEXEC) was a stripped-down operating system for remotely logged-inusers who actually executed on the first available DEC 10 but never knew which one. This was also an attempt to provide a network-wide file system. Multiple server systems such as the Irvine Net recognize the need to go accross the system to get resources. To use this we may need utility programs to perform

Local COBOL --> ANSI COBOL

and maybe even

ANSI COBOL --> Local COBOL

Livesey:
May also have a network JCL so that a user only uses the JCL of his local machine, and then we need to be able to do the translation

Local JCL #1 --> Network JCL --> Local JCL #2

Lapin:
There are two approaches to a multi UNIX system file system. We can have

/net

as a special file and address files on machines A, B, etc. as

/net/A/pathname ...
/net/B/pathname ...

We can also localize host id in the pathname explicitly

part1/part2
part1: host id part2: pathname

Sunshine:
There is a conflict between REAL and IDEAL worlds. In the Real World, we tend to involve the user in specifying the location of a function (service). In the Ideal World, we would <u>like</u> to give the user <u>abstraction</u>,

generic naming and location independent naming.

Livesey:
Part of the problem is that the concept of the size of the universe (of which the system forms a part) is implicit in the system at a high cost. One is then forced to choose between add-on features such as:

/net/A/resource

which are not location independent on the one hand, and a more or less complete rewrite on the other hand. UNIX is an example of such a system that makes assumptions about the size of the universe.

Meisner:
We now have choices between

i) Centralized Directories
which can now be made very reliable
ii) Distributed Knowledge
iii) Tree Structures

Livesey:
(iii) is just a disguised directory method. There are really two choices: centralized and distributed.

Hassan:
Efficiency may dictate tree structures rather than directory tasks. This was a factor in the MULTICS design.

3.4 POSITION PAPERS

3.4.1 Hamilton

Addressing and Security

by

Jim Hamilton
Digital Equipment Corporation

Because of ever increasing complexity of software development and maintenance, providing any programming environment which complicates software development would be a mistake. This argument leads to a view of distributedness as a property of the implementation of a system, and not of the application development environment.

Addressing and protection are critically important in application development. The above view of distributedness implies that addressing must be location independent. That is, local and remote objects must be addressed identically. Furthermore, I believe that addresses should also be independent of the context of reference (different processes should address the same object in the same way), and uniform across all object types (hardware defined objects, system defined objects, and application defined objects should all be addressed similarly).

I also believe that the use of processes to abstract all other objects is a mistake, for several reasons: 1) it restricts the flexibility of the environment for the execution of functions, 2) it often forces the invention of additional addressing mechanisms within the application, 3) it is inadequate to address system and hardware defined objects (e.g., devices), 4) it inevitably colors the application designer's conceptualization of the system, and finally, 5) it does not appear to be necessary.

To achieve a distributed implementation, it will still be necessary to solve the problems of physical communication and its associated addressing problems at a lower level. But the problems are considerably simplified since the mechanisms can now be highly specialized, because they are not visible to the application designer.

I believe that the notion of capability based addressing, when properly interpreted and implemented, provides all of the properties mentioned above. Moreover, it can be naturally extended to provide capability based protection, which is further discussed below. The challenge is to achieve an implementation which is cost-effective, and which still has all of the necessary properties. A failure in

either domain will be fatal. An even greater challenge is to convince the computer industry that the inevitably higher cost of the basic system will be more than offset by the reduced cost of software.

I believe that the issue of sharing is partially separable from that of addressing. Context independent addressing is a prerequisite for sharing, but its existence does not imply concurrent access by separate processes. Concurrent access to immutable objects should be possible, for performance reasons, but concurrent access to mutable objects now appears to be a dangerous mistake. By precluding this kind of sharing, we also simplify the construction of distributed implementations.

Given an addressing mechanism with the properties mentioned above, a variety of protection mechanisms can be implemented. Capability based protection still seems to be the most promising of these, although it has been criticized as inappropriate for distributed implementations. I tend to reject this criticism, but the notion of self-authenticating capabilities has been developed at Berkeley to address this problem.

The notion of system security has many different aspects. Included among these are physical security, correctness of implementation, and the logical access control model being implemented. In comparison with centralized implementations, distributed ones seem notably weaker in physical security, and possibly weaker in correctness because of greater complexity. The access control model should not, in principal, depend upon the implementation. I believe that these are inherent problems with distributed implementation, but that, with the suitable use of encryption, such systems can still be acceptably secure.

3.4.2 Sunshine

Addressing

by

Carl Sunshine
RAND Corporation

Any discussion of addressing must start by making a clear distinction between NAMES (who), ADDRESSES (where), and ROUTES (how to get there), on which John Shoch of Xerox PARC has written an excellent note. [SHOC 78]

Several key concepts or capabilities must be included in a good distributed IPC system. These include generic naming, location independence, request for service, source routing, and extensibility. Each will be described separately in the following paragraphs, although there are clearly some

relationships between them.

Generic naming is the ability to request communication from a service without specifying the exact process that will provide the service. This is normally useful when multiple instances of a process providing the desired service are available. A specific process is selected (or created) at the time of the initial request, and bound to the source for the duration of the interaction. This binding may require transmitting the specific process ID to the source, or merely keeping it at the destination. The classic example of this facility is a timesharing login service.

Location independence is the ability to request communication with a process by name without knowing its location or address. Since the source user does not supply the address, it must be found by the IPC system in some directory. Such name-to-address directories may be maintained at sources, at a central server, or at destinations (the names are normally handled at the source, with the consequent need to change all tables whenever a host address or name changes or is added; IBM's SNA centralizes lookup in the SSCP; and the Irvine DCS kept name tables in destination machines, requiring broadcast of requests to be recognized by the appropriate destination. The ARPA Internet Name Server proposed by Jon Postel in a recent note is another centralized example. A major feature of location independence is the ability for a named process to move to a different location without its users knowledge. (Of course the directories must be updated.)

Request for service is the ability to broadcast a request for service to an unknown (to the source) number of potential providers of the service, who return bids to perform the requested service, thereby identifying themselves. This is similar to generic naming, but includes facilities for the source to select among multiple bids. Such a facility was implemented in the Irvine DCS.

Source routing is the ability for the source to identify the destination by specifying a route to it. This is necessary in loosely concatenated systems where no global address space exists. The route is given in terms of a sequence of addresses through successive switching points or systems which each have independent address spaces. Hence this concept is also called path addressing. Disadvantages are the need for the source to maintain connectivity information, and the variation of a given destination's "name" (consisting of the route) depending on the location of the source.

Extensibility is the ability to add new users (addresses) to the system. To add new users at an existing level of the address space, sufficient room must be available in address fields, or they must be extensible. Adding additional layers of addressing often proves a bigger problem, for

example replacing a user by a network of many users. If the hierarchy is fixed (e.g., <net/local>), then the bottom "leaves" of the addressing tree cannot be replaced by subtrees. In this case, addressing must be used to deal with networks outside the fixed hierarchy. This is a serious problem with attachment of private networks to public data networks.

Interconnecting two previously independent systems is an important subcase of extensibility. All the users of one system can be given new addresses in the other system if such widespread changes are acceptable. Alternatively, some unused local addresses in each of the systems may be mapped into addresses in the other system if only a limited number of users must be accessable. Finally, if the addressing hierarchy is extensible, one system can be attached as a subtree of the other, or both can be made subtrees of a higher level.

3.4.3 Gordon

Addressing & Security

by

Robert L. Gordon
PR1ME Computers

An extremely important aspect of interprocess communication is the scheme used for addressing and naming the processes and communication paths used. The importance of this subject stems from the fact that in any addressing scheme protection and control mechanisms are explicitly or implicitly present and either aid or hinder the users ability to share objects. Many current systems have inadequate facilities for identifying names and controlling access to the processes within the same host, let alone for processes residing on other hosts. Part of the problem stems from an inconsistent view of the relationship between the names and uses of files, devices, processes, users, mailboxes, generic and specific system services. The utility of abstracting many of the above objects as processes has increased the importance of "process naming" and "process addressing" in overall system design. Therefore until these basic issues are settled the design of specific interprocess communication primitives is difficult since they cannot focus on the fundamental objects that they will be dealing with.

Fault Tolerance & Security

by

Robert L. Gordon
PR1ME Computers

Any communication is inherently an error prone process due to both the natural distortion of the medium and the contextual requirements needed for interpreting the transmitted message. In attempting to design robust interprocess communication primitives one of the more difficult tasks is the defining and handling of the many (natural) errors that can occur. Control of communication mechanisms between processes fundamentally depends on how the designer envisions process relationships. If process relationships are tree structured, then the status and control of a processes' communication with other processes might be monitored and controlled by the parent. On the other hand if each process wants to maintain the concept of sovereignty then the basic challenge is either how to provide the ability for cooperating processes to establish a monitor process that is capable of controlling the communication paths between the processes, or how to build into the communication primitives mechanisms for the detection of and recovery from errors. Since error recovery must make assumptions about lines of authority and responsibility between system components, many of the issues associated with system security are pertinent to this discussion.

3.4.4 Chesson

IPC Opinions

by

G. L. Chesson
Bell Laboratories

Process Naming

Process names, file names, and I/O stream names should reside in the same name space. This avoids the tyranny of the "access method" and attendant problems of making a program that can "talk" to anything in a system. One can allow process names to be passed into processes in the same way that file names and I/O streams are passed around, and this in turn permits progress toward interactive command processors that can set up graph-like structures of processes, file I/O, and IPC streams.

Non-Duplication of Mechanism

A philosophy that has been proven many times over in language design may be stated as follows: it is "bad" to provide more than one mechanism for a particular operation or function. This is a roundabout way of saying that there are benefits to be gained by providing a single IPC mechanism for use by "local" processes, i.e. on the same machine, and "remote" processes on different machines.

Transport Mechanism

It is fine to use shared objects (memory, files) for interprocess communication, but it is important to hide this fact. The reason is that explicit sharing of objects is not portable with respect to different machine and operating system architectures and should be considered a local optimization. Thus, IPC primitives at the compiler or operating system level should appear as I/O-like interfaces that imply copying of data even if they do not actually copy data on some systems.

IPC in Programming Languages

Most IPC proposals for inclusion in programming languages amount to little more than interfaces to subroutine libraries which a) cannot be inherited by processes across process fork operations, b) belong in the operating system anyway, and c) were done better by Burroughs Corp in DCALGOL 10 years ago. The result of adding IPC to a language is analogous and about as useful as the notion of a file system in Pascal. A representation of the fundamentals of IPC that belongs more to the programming language realm than the operating system realm has yet to be demonstrated, and would fill a much-needed gap.

Hardware

There are applications for which IPC bandwidths must approach or exceed disk speeds. It is clear that such performance cannot be obtained with software (or even firmware) alone. Although there may not be much interest in this sort of thing at the IPC workshop, I have been working toward hardware and firmware implementations of my software mechanisms.

Flow Control

Ipc mechanism need flow control. It is better to have a scheme where the sender selfblocks than schemes which depend on "stop" messages from the receiver. For most applications the scheme used in UNIX for pipes and other things would seem to work well: the sender blocks (sleeps) on a queue

length upper limit and is awakened when the queue drains below a lower limit. There exists a timeout call which can wake the writer if the queue drains too slowly or is otherwise delayed. An additional non-blocking mechanism has been built into the mpx software (see section 7.7) which is useful in those few cases where blocking cannot be tolerated -- network servers and the like. This avoids the problems that occur with varying process and communication delays or loss of control messages.

Synchronization

Cognoscienti agree that message-passing IPC schemes are equivalent in "power" to schemes which employ shared objects although the message schemes seem "harder". This has not been proved or disproved mathematically, although there is substantial empirical evidence that pairs of processes can be synchronized by exchanging messages.

Food for Thought

I submit that it is seductively easy to synchronize process pairs, but that strategies are needed for synchronizing groups of processes in various ways. Is it reasonable to set up "overseer" processes that arbitrate and synchronize things, or are there better ways that can be proven correct? For some things, like call-processing in my network I use overseer processes because they reduce complexity and can be made reasonably efficient. For other things, like synchronizing a process group carrying out a parallel computation, I would try to eliminate the Deus ex machina and use direct process to process methods.

Portability

It is important to demonstrate univeral IPC ideas and to distinguish local optimizations and special cases within the universal model. One would hope that a suitable IPC model could be used with protable operating system ideas to bring up compatible IPC mechanisms on dissimilar machines. Section 7.7 on Data Communications Software outlines some ideas that have been partially demonstrated to have portability properties.

SECTION 4

INTERPROCESS SYNCHRONIZATION

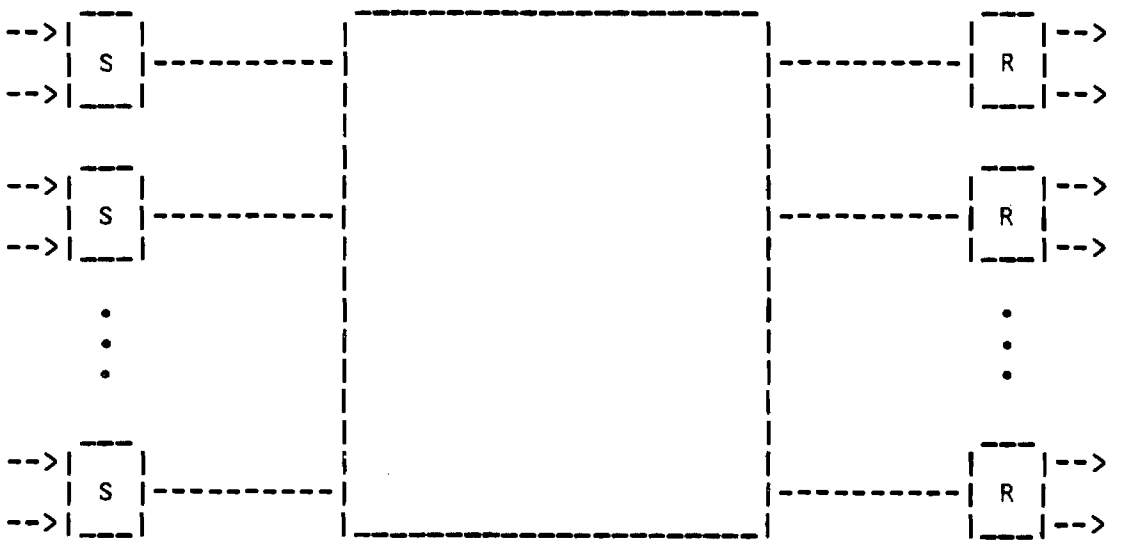
4.1 WORKING GROUP SUMMARY REPORT

4.1.1 Statement of the Problem

1) Synchronization via explicit communication (messages).
2) No global memory.
3) System-wide control with only inaccurate/incomplete information on the system state, without any centralized procedure, data or hardware.
4) Transit delays are: variable, unpredictable, unbounded.
5) Loss, error, desequencing, duplicate.
6) Other failures (processors).

4.1.2 Solution Space

SOLUTION SPACE



GENERAL CONFIGURATION (LOGICAL)
FOR A SINGLE SET OF MESSAGES

MOTIVATIONS:

1) Distributed service.
2) Survive sender/receiver failures.
3) Non-technical reasons.
4) Modularity (growth, ...).
5) Performances.

CONFIGURATIONS:

a) **"Single Sender / Single Receiver"**
Single Path Signalling
End-to-end Synchronization
(Used to achieve flow control for example)

b) **Single Sender / Multiple Receivers**
Multiple Path Signalling

		PROCESSING AT RECEIVERS	
		IDEN.	DIFF.
MESSAGE	IDEN.	1	2
CONTENT	DIFF.	3	4

(1) Pure broadcasting in a fully replicated system.

(2) Pure broadcasting in a heterogeneous replicated data base.

(3) Transaction processing in a homogenous (replicated?) system.

(4) Transaction processing in a heterogeneous replicated data base.

OBJECTIVE: To maintain a unique ordering of incoming messages for all receivers (whether initially fortuitous or enforced).

c) **Multiple Senders / Single Receiver**
Multiple Path Signalling
OBJECTIVE: Reveal/Cause/Express relationships between incoming messages belonging to different flows.

d) **Multiple Senders / Multiple Receivers**
Multiple Path Signalling

1) Fully replicated systems
same objective as (b)
2) Partioned systems
same objective as (c)
3) Mixed systems
same objective as (b) for dynamically changing subsets of receivers plus the same objective as (c)

4.1.3 Some Existing Solutions

a) **Logical Clocks: L. Lamport**
To implement a sequential (T. Ord.) processing in a distributed manner (each process has an image of "The Waiting Queue") - may be used to achieve mutual exclusion.

b) **Physical Clocks: L. Lamport**
How to implement logical clocks on a set of physical clocks (unique physical time frame).

c) **Logical Clocks plus Voting: R. Thomas**
How to resolve conflicts between simultaneous/concurrent processes competing for identical resources (fully replicated systems).

d) **Eventcounts, Sequencers: Reed/Kanodia**
To observe (READ, AWAIT) or to express the occurence of some event (ADVANCE) - to serialize events.

e) **Circulating Token: G. Le Lann**

- Without tickets
  To achieve mutual exclusion.
- With tickets
  To serialize, to express relationships between events

f) **Some "naive" or less general solutions:**

- Shared Variables: E. Dijkstra
- Monitors and Messages: P. Brinch-Hansen

4.1.4 Attributes

a) Response time.
b) Overheads (traffic, processing, storage).
c) Extensibility (is full connectivity required, global knowledge of the system status, ...).
d) Deterministic synchronization / probabilistic synchronization / convergence.
e) Fault tolerance.
   - Detection.
   - Recovery.
f) Simplicity (correctness proving, implementability headaches, ...).

4.1.5 Other Issues

a) Effects of probabilistic synchronization.
b) System considerations:
   - Hard/soft partitioning.
   - Application processing / system processing partioning.
c) Evaluation of solutions with respect to
   - Attribute space.
   - Problem space.
d) Policies (fairness, enforced priorities).
e) Adequacy to resource management.
f) Classification of solutions.

4.2 POSITION PAPERS

4.2.1 Lee

Interprocess Synchronization

by

Edward Y. S. Lee
TRW Defense and Space System Group

My interest in IPC is mainly connected with update synchronization in redundant distributed data bases (DDB). The protocols developed for IPC must be viable and be able to satisfy the following major requirements for DDB operations:

1) Performance (response time)
2) Efficiency
3) Deadlock prevention
4) Error recovery (surviving errors and faults and continue operation)
5) Security

Recent state-of-the-art developments in this area can be divided in two major categories:

1) Protocols associated with a centralized control approach [ALSB 76, BADA 78, ELLI 77, ESWA 76, ROTH 77]
2) Protocols relying on distributed control [GRAP 76, JOHN 75, ROTH 77, STON 78, THOM 77]

However, most of the protocols do not include serious considerations of interprocessor communication, but rather take the approach that some kind of messages can be passed among the distributed processors for communication and let someone else to worry about it.

There are considerable difficulties in taking this kind of approach in a loosely coupled distributed system. Because IPC is the life line of the system, it is needed for the distributed control (operating system), distributed data base operation, recovery of the system as well as the DDB under fail-soft and fail-safe condition, and reconfiguration of the network when one or more processors are disabled. All these essential functions of a distributed system demand efficient and fail-safe IPC mechanisms.

The second obstacle is the lack of evaluation criteria and methodologies to test and measure:

1) Performance
2) Efficiency
3) Validity
4) Verfiability

of any protocol that is being proposed as the best protocol for DDB. There are some efforts present in this area [GARC 78, SUNS 76], but a lot more work will be required.

In a practical system, it is very likely that a mix of several protocols will be used for updating redundant distributed data bases depending on the specific situation and requirement. However, it should be possible to have a unified approach to IPC for all protocols. Additional research in this area is needed.

SECTION 5

MECHANISMS --- IMPLEMENTATION, UTILIZATION, and PERFORMANCE

5.1 WORKING GROUP SUMMARY REPORT

Interesting Issues Not Discussed

Data Interface to program not resolved
Control interface to program
"To poll or not to poll"
Events, interrupts, on-conditions

Mechanisms

Signals
Events
Semaphores
Shared Memory
Monitors
Message Queues
Pipes
Ports
Full Duplex Streams
Virtual Procedure Calls

Characteristics of the Mechanisms

	SHARED OBJECTS	EXPLICIT DATA MOVEMENT	EVENT OPERATING BY	PROCESS CREATION SIDE EFFECTS	EASE OF DISTRIBUTED IMPLEMENTATION
Signals	U	N	na	N	+
Events	U	N	na	N	+
Semaphores	S	N	na	N	-
Shared Memory	S	N	S/R	N	-
Monitors	S	Y	R	N	0
Message Queues	S/U	Y	S/R/T	Y	+
Pipes	U	Y	na	N	+
Ports	S/U	Y	na	N	+
Full Duplex Streams	U	Y	R	N	+
Virtual Procedure Calls	U	Y	T	Y	+

S = Shared
U = Unshared

S = Sender
R = Receiver
T = Transport Mechanism
na = not applicable

Desirable Qualities of Mechanisms

- Performance
  - Bandwidth
  - Delay
- Provability
  - Correctness of use
  - Correctness of implementation
- Security
- Transparency
  - Naming
  - Location (Physical)
  - Environment (Logical)
- Separation of control from data
- Complete and small set of primitives
- Fault tolerance
  - Encapsulation
  - Detection
  - Recovery
  - Size of fault set covered

NOTES: The priorities used to weight these desirable qualities depend on:

- Application
- Level
- Environment

Desirable Qualities of Mechanisms

	Performance	Provability	Security	Transparency (Naming)	Transparency (Location)	Transparency (Environment)	Control/Data Separation	Primitive Completeness/Size	Encapsulation	Error Detection	Error Recovery	Fault Set Covered	Capabilities
Signals	-	-	AD	AD	+	C	+		-	-		C
Events	-	-	AD	AD	+	C	+					C
Semaphores	-	+	AD	AD	+	C	+			+		C
Shared Memory	-	-	AD	AD	-	-	-			-		D
Monitors	+		AD	AD	+	+	+		+			C/D
Message Queues									+	+		C/D
Pipes	+	+	+	+	+	+	+		+	-		D
Ports	+	+	+	+	+	+	+		+	-		D
Full Duplex Streams	+	+	+	+	+	+	+		+	-		C/D
Virtual Procedure Calls	+		AD	AD	+	+	+		-			C/D

AD = Addressing Mechanism Dependent

C = Control only

C = Control
D = Data

Comments on Mechanism Qualities

1) A functionally complete IPC mechanism requires both data and control capabilities
2) All were considered to be "basic" mechanisms -> No embellishments to improve desirable programs
3) Thus ability to recover from faults depends on implementation
4) Another trade - Bandwidth vs. status consistency
5) Perceived hierarchy (in mechanism list)
6) Omissions
   - Broadcasts
   - Addressing
   - IPC mechanisms ??
7) A design exercise to try to overcome "-'s" in table would be interesting --- Also table completion

PROBLEMS

1) Migration of applications from centralized to distributed environment
2) Not enough known about these mechanisms:
   - Complexity of IMPL
   - Size of IMPL
   - Efficiency of IMPL
   - Useful hardware assists
3) Common understanding of all mechanisms
   - Dictionary
4) Lack of a number of implementations
5) Cost / time / complexity
6) Premature standardization
7) Difficulty of modifying / experimenting with hardware support devices
8) Premature vendor mechanism selection
9) Compatibility
   - Obstacle
   - Objective
10) Evaluation criteria
11) Papers don't tell reasons for designs (some designs based on few examples)
12) Definitions of universes

Research Questions:

1) Identify collections of primitives for
   - Easy programmer understanding
   - Efficiency
   - Match to application
   (Answer probably depends on environment)
2) Fault Tolerance of IPC mechanisms not well understood
3) Trade -- User or IPC mechanism?
4) How much must user be aware of process creation/existence?
5) How should responsibility be distributed? Visibility of fault responsibility.
6) How to decouple bindings:
   - Modules to graph
   - Process to nodes
   - Resources to processes
7) What set of IPC mechanisms is
   - Easy to use
   - Complete
   - Efficient
8) Refine virtual procedure call mechanism.
9) Tools for top-down design
10) How to select architectures from option criteria
11) How to decompose applications

5.2 AMPLIFYING MATERIAL

5.2.1 Prepared by the Working Group

An attempt was made to define "a set of primitives that allows an application software engineer to design the best solution for his problem." It was quickly realized that this is not an easy task. Some of the issues involved are:

1) Some applications require highly reliable IPC, while in others, communicated information becomes useless after a certain period of time. A single set of primitives to implement IPC may not solve both types of problems.
2) Should IPC primitives be operating system services or should IPC constructs be parts of various programming languages? A relevant reference to this latter proposal may be found in [HOAR 78].

At this point, it was felt that it was necessary to outline the hierarchy of levels at which IPC mechanisms can be invoked. For each level, we attempted to describe those objects which may be manipulated and those IPC operations which may be performed on each object, if any.

Hierarchy of Levels

Command Level
High Level Languages
Operating System
Instruction Level
Microcode Level
Hardware Level

The description of objects and IPC operations can be enumerated for three different situations:

1) Accepted practice - those commercially available
2) State of the art - current practices of researchers in the field
3) Wish list

Enumeration of Quantities for Accepted Practice

Command Level:

objects - process, file, link, device, program,
task graph, directory

IPC operations -

files: file locks (control function)
pipes

processes: create
delete
link via a pipe
suspend
resume
status

links: creation
temporary files
link management in DEMOS

Reference: [BASK 77].

Note: Though not all types of objects are available on many systems, some of them can be used to emulate those capabilities which are unavailable. For example, temporary files are used in UNIX to emulate pipes.

High Level Languages:

objects - typed objects (integers, reals, characters, etc.)
semaphore
monitors
events
ports
shared common (typed objects)

Except for the use of shared typed objects (via global common areas), current languages commonly available do not use the other objects for IPC (e.g., PL/I). Almost invariably, one must drop into a runtime library routine or to the operating system to perform IPC functions.
PL/I is most progressive
Algol 68 provides some capabilities
APL supports shared variables

Miscellaneous notes:
There was some discussion concerning the two types of commonly used IPC mechanisms: message-oriented vs. procedure-oriented (monitor). A good reference to this area is [LAUE 79].

5.2.2 Prepared by Peebles

5.2.2.1 Introduction and Explanation

The IPC mechanisms described here are known as "primitive" for several reasons; they are primitive in the sense that they are low-level building blocks from which more sophisticated forms of IPC can be built, they are mostly oriented towards two-party communication, the simplest case, and they are mostly derived from existing uniprocessor systems.

5.2.2.2 Desirable Properties

It is fairly easy to list some desirable properties that any interprocess communication mechnisms should have:

Performance -- In terms of bandwidth and also delay. We would like mechanisms with a minimum of overhead, in order to maximize performance. THis should not, of course, reduce functionality.

Provability -- A desirable property for any IPC mechanism should be that it lend itself to the verification of systems which are built up using processes.

Security -- By this we mean protection of two communicating parties from one another, and also with respect to third parties, in terms of leakage and interference.

Transparency -- This refers back to the issues of naming and location. The users of an interprocess communication mechanism sdhould not have to deal with that mechnism at other than the advertised level, nor should they have to be aware of the details of its implementation.

Separation of Data and Control -- It may or may not be a good property of an IPC mechanism to contain elements of both data and control. In some implementations, data and control (signal) transfer from sender to receiver are carried out in the same operation. Separate data and control transfer operations can, of course, be combined in higher-level non-primitive interprocess communication operations.

Completeness and Smallness -- Interprocess com-

munication primitives should certainly be complete, in the sense that one should be able to do any operation which is valid in the given system without introducing new primitive operations. It is not so clear that they should be small, consistent, of course, with performance.

Fault Tolerance --This leads to the concepts of encapsulation and recovery. In order to achieve fault tolerance, an operation should fulfill the following conditions:

- faults should be detected.
- faults should be handled at the appropriate level, and not simply passed back upwards towards the user.
- faults generated at a lower level should not terminate a user session. Instead, they should be recovered at a level close to that at which they occurred.
- in interprocess communication, if data or control transfer fails, it may be sufficient to inform the sender, or, in some critical applications, it may be necessary to inform both the sender and the receiver that some message or control signal did not get through.

The concept of encapsulation suggestes the enforced localization of errors, so that an error in the communication between two proccessors can have no effect on any others. The concept of recovery suggests that whatever errors do not occur are cleaned up in such a way that a consistent system state is restored, and that unresolved error states are not simply passed up the line. Error messages of the form:

SUBNETWORK ERROR - PLEASE LOG IN AGAIN

should never occur.

Cost -- The concept of cost is very difficult to define exhaustively, but one can suggest some kinds of cost which can be incurred:

- implementation
- performance
- application flexibility

Note that in the evaluation of primitive mechanisms given in section 5.1 we assume a fairly standard implementation. The properties above clearly depend in part on implementation

and we cannot give any hard and fast rules.

5.2.2.3 IPC Taxonomy

One of the most obvious dimensions along which to differentiate IPC mechanism is whether they are message-based or not. Mechanisms can, of course, be data-transfer based, without being message-based.

Examples: Pipes, ports, full-duplex streams.

5.2.2.3.1 Non-message-based IPC

These are clearly the IPC mechnisms favored in those distributed systems which are themselves not message-based. Instead of messages, these depend on a variety of communication mechanisms:

1) Signals
   Signals are process interrupts, which can arrive with or without accompanying type information, and perhaps the identifier of the originator. A signal may cause a transfer of control inside the receiver process, and there may be enable/disable mechanisms, analogous to those for hardware interrupts.
2) Events
   An event is a state variable. One should be able to test it and set it. It should be possible to implement a wait on the event by means of a test in a loop.
3) Semaphores
   A semaphore is a storage cell with an associated queue of processes, and with two operations, wait and signal (no relation to signals in section 3.2.1.1) which have side effects.
4) Shared Memory
   Shared memory consists of data cells which are accessible to sending and to receving processes, perhaps with an associated access discipline which is designed to avoid critical section problems in accessing the shared resource.
5) Ports
   Ports are input/output channels belonging to processes. Ports in corresponding processes can be connected together by links to form communiccation channels.
6) Full Duplex Streams
   A full duplex stream is effectively a bi-directional pipe. In place of a sender and receiver, the processes at either end of the full-duplex stream can both send and receive. Naturally, in order to achieve some measure of synchronization, a read should suspend

until a corresponding write is executed at the other end of the full duplex stream, and vice versa.

5.2.2.3.2 Message-based IPC
These are the IPC mechanisms which depend on messages between processes. They can be further subdivided along the following lines:

1) Single send **p1 --> p2**
2) Single receive **p1 <-- p2**
3) Multiple send **p1 --> subset of P**
4) Multiple receive **p1 <-- subset of P**

Blocking and Non-blocking Primitives

A further way of subdividing interprocess communication primitives is on the basis of whether they are blocking or non-blocking in nature. A blocking primitive is one which causes its invoking process to be suspended until the primitive operation is completed. Thus, after invoking a blocking receive, a process will suspend (sleep) until some message does arrive.

Distributed systems have been implemented with blocking send/receive, with blocking send and non-blocking receive, and with non-blocking send/receive.

Virtual Procedure Calls

Virtual procedure calls ccan be viewd as a highly stylized form of message passing but entail a great deal more semantics. They are used in support of an object model - processes access objects and objects are controlled by other processes. IPC consists of one process invoking a function on an object and another process executing that function.

5.2.2.3.3 Higher-level Mechanisms
There are also higher-level mechanisms which can be produced using the primitive operations as building blocks. For instance, one frequently encounters virtual circuits built on message passing combined with signalling.

5.2.2.4 References

The following references may be helpful in explaining the specific IPC concepts identified:

1) Semaphores, Signals, Events, Monitors, Pipes:
[HOLT 78b]
2) Virtual Procedure Calls:
[HAMI nd]
3) Message Passing Operating Systems:
[MANN 77]

4) Message Passing versus Procedure Calls:
[LAUE 79]

5.3 POSITION PAPERS

5.3.1 Peebles

PROGRAMMING ISSUES

by

Richard Peebles
Digital Equipment Corporation

Religious Issues

A Programmer's environment (language, operating system services and model of process structure) tends to be a religious issue. My religion calls for the simplest possible environment by providing a set of "orthogonal basis vectors" for programming. The result is a set of primitives that allows an application software engineer to design the best solution for his problem. Orthogonality of software tools means that one piece, or primitive, does not preempt design choices for the others. This is to be contrasted with another approach to simplicity which preempts almost all choices.

In addition, my religion calls for the removal of representational irrelevancies to the highest degree possible. As a consequence, the underlying process structure is not visible at all to most programmers, nor is the distributed nature of the machine that implements his application.

Practical Issues

The difficult part of religion is applying it to our daily lives. Just what *are* these primitives; what makes an orthogonal set; can we find a set of "basis vectors"? Furthermore, can we reasonably expect to hide the process and machine structure from programmers? In my view, most research in distributed systems is (should be) aimed at answering these questions.

Constraints on IPC Mechanism

The above goals for the programming environment impose several constraints on the IPC mechanism. First it should be location independent. The same mechanism should be used for both inter-host and intra-host communication. This means that a programming decision does not preempt a process-location decision and vice-versa. A more difficult question is whether the IPC mechanism should be visible as such to the programmer. It is possible to provide him with an extended machine in which IPC appears as the application

of an operator to an operand; this is the approach taken in our experimental WEB system. It is a simple matter to construct both datagram and virtual circuit abstractions with this mechanism if "communicating processes" is a relevant abstraction. It is considerably more difficult to provide the operator/operand abstraction mechanism than a simple send/receive mechanism; particularly if abstractions are to be enforced.

State of the Art

In vendor-implemented products neither location transparency nor process structure transparency is usually provided. Research systems have, for the most part, made IPC an explicitly separate concept among other abstract extensions of the operating system. The WEB operator-invocation architecture is seeking to provide a single mechanism that will serve as a general basis for "operating system" and user functions - they are not distinguishable. It is, however, only in the final stages of design - about to be implemented.

Obstacles

The most significant obstacle to providing an IPC mechanism that least perturbs the programming interface is historical artifact. Finding a design that is ideal and that allows reasonably simple migration of customer applications is a hard problem. We may be forced to throw up our hands and call on users to swallow yet another conversion effort. Will we do it again in 1988 when distributed systems go out of vogue? Hence my strong belief in the need for process and machine structure independence of IPC. Early standards will be a hindrance to this but may be inevitable given the state of the art and user impatience to build. If that is accepted, the next biggest obstacles are thin wires and different architectures. Hiding the network structure is hard when physical links are under 100K bps. Then too there is the problem of the complexity of the WEB abstraction approach - it's hard to understand.

5.3.2 Wallentine

PROGRAMMING ISSUES IN DISTRIBUTED SYSTEMS

by

Virg Wallentine
Kansas State University

Problem

The programmer in a distributed processing environment must be provided with a set of facilities which permit easy specification of the distributive properties of his/her program. The word program here is used to refer to either the output of a single compilation or the output of independent compilations of program modules which are to be communicating via an IPC. These distributive properties include the specification of the concurrency, data flow, resource requirements (memory, devices, etc.), and intraprogram (intermodule) protocol properties inherent in the execution of a configuration (system) of cooperating software modules. Given a description of these properties, an operating system must be able to distribute the user's program across multiple machines in a manner which is transparent to the programmer. Traditional approaches to providing these facilities include the concurrency support in high-level languages and the resource allocation and concurrency support in conventional operating systems.

Current Approaches

Several high-level languages such as Concurrent Pascal [BRIN 77] and SP/K [HOLT 78] have incorporated the monitor [BRIN 73] [HOAR 74] concept to provide structured concurrency. This concept is excellent in a centralized system but relies on shared data (and therefore shared memory), and is therefore not an appropriate concept on which to base a distributed system. However, an effort is underway at the National Physical Laboratory [DOWS 78] to distribute a Concurrent Pascal program across loosely coupled microprocessors. The distribution of passive system components (such as monitors) on disjoint machines implies many copy operations for parameters and also additional active system components (processes) which do not appear in the program text.

A much more appropriate high-level language concept for distributed programs is proposed by C.A.R. Hoare in reference [HOAR 78]. Each function is a sequential process which is connected to other communicating sequential processes via input/output. This concurrency support is based on data flow and not shared data; therefore, it is not dependent on shared memory. As a result, each function is distributable. However, it seems that buffering of data between processes is necessary to improve performance in

distributed systems with slow speed connections. Since the compiler for such a language presumably can generate the resource requirements for the program, since processes are identified by name, and since the protocol between processes is fixed, enough knowledge is available to distribute a set of processes which are compiled together.

A second area of programmer concern for distribution occurs because concurrent program functions (modules) may be separately generated (compiled). These may well be existing programs or just separate functions based on programming style. The interconnection of these modules into a program is dynamic and therefore requires operating system support. In early conventional operating systems, the support for combining these functions into a configuration of communicating concurrent software functions is specified at three levels. First, overlap of CPU and I/O are made available for standard I/O file functions. Second, added concurrency is achieved only with unstructured (low-level) facilities for process creation, naming, and communication. Third, complex job control languages are provided to achieve allocation of resources to run these functions. In a distributed system, these JCL steps must be synchronized across machines. Complex resource control in a distributed system should certainly not be the programmer's responsibility. This is alleviated by viewing distributed operating systems and their executable programs as cooperating processes. A highly successful system is the Distributed Computing System of Farber [FARB 73]. In this system, the structure and distribution of the set of processes is transparent to the user; and a high level of concurrency is achieved without use of low-level process control primitives.

Process naming of cooperating processes is still burdensome to the programmer. The same problem also occurs in current "mailbox" schemes as epitomized by the VAX 11/780 system [DEC 77]. The naming or numbering of mailboxes must be known to the programmer or a creating process. This is commonly referred to as the IPC-setup problem, coined by Elliot Organick in reference [ORGA 72]. The designers of UNIX [THOM 74] [RITC 78] sought to alleviate this problem. They invented the "pipe." In UNIX a user program, running in its own process, may take the place of a file in a manner which is transparent to the original program. Each program may have its standard input and output files replaced by programs, thus building via the UNIX shell arbitrarily long linear chains (a pipeline) of programs. UNIX automatically transfers the data between processes and synchronizes the process as it intercepts the standard input and output file operations.

UNIX "pipes" eliminate the need for process naming and treat concurrency, resource allocation, and inter-process protocol as a data flow problem. Interprocess protocols are treated simply as simplex data streams. The job control language provided by the UNIX shell becomes a pseudo data flow language and resource allocation is transparent to the programmer. However, there are a considerable number of programmer protocols which are not served by "pipes." As acknowledged in reference [RITC 78], "pipes" cannot be used to construct multi-server subsystems.

UNIX will support general interprocess communication protocols but these are not generated by the shell. These can be programmed as a set of child processes whose "pipes" have been setup by a parent process.

A Research Direction

If we are to be successful in distributing programs across highly distributed systems, we must provide the programmer of dynamically interconnected cooperating processes a job control language (software configuration control) as easy to use as Hoare's communicating sequential processes. It seems that the most promising direction is to extend the concept of the UNIX shell to automatically generate the more complex protocols available to the parent processes previously described. It must then also be extended to generate (representations of) distributable configurations of communicating processes.

Work in this area is underway at Kansas State University. The project* involves development of a Network Adaptable Executive (NADEX)[YOUN 79]. The attempt is to permit the user to specify data flow at the command level and have the command interpreter generate a distributable software configuration of nodes connected by full duplex data transfer stream connections (DTS connections) to form an undirected graph. In general, a node may be thought of as a process. Each of the connections consists of two independent bi-directional data transfer streams. One of these streams uses small parameters while the other uses a standard-sized data buffer. The data buffers carry along with them size and status indicators whereas the parameter buffers contain only a small amount of user-supplied data.

A user program running in a node performs serial buffered READ and WRITE operations in its various connections. The connections are numbered, and the program attaches particular meanings and implements particular protocols for each of its connections. A connection can connect a node either to a user program or to a system process used to access a file or an I/O device. The program cannot tell the difference between these modes of operation. This clearly provides all of the power of the UNIX pipelines while removing the linearity constraint on the structure of the connec-

tion graph. Also, the connections are bi-directional so that, for example, a write-request/read-response protocol to access a random file can be implemented.

For these serial buffered READ and WRITE operations, a priori protocol knowledge can be specified to the underlying data flow implementation (buffer control) to enable it to maintain a check for validity of user protocol (in terms of data flow) during execution. This protocol checking is critical in "un-debugged" (user-written) nodes. Examples of such protocol violations occur many times in the facilities of SOLO [BRIN 76]. Deadlock detection is also performed based on data flow in a configuration which is distributed across machines connected by a network IPC. Multiserver subsystems, such as a data base management system, are implementable as a configuration with multi-connection READ (multiple condition WAITs) and conditional WRITE operations provided on data transfer streams. Interconfiguration connections are also provided. Finally, the command interpreter and the node interface (PREFIX) provide all the mapping of logical data streams (ports) onto implementation data streams.

* Supported in part by the Army Research Office under Grant Number P-16160-A-EL.

SECTION 6

THEORETICAL WORK

6.1 WORKING GROUP STUDY REPORT

STRUCTURE of Discussion:

Distributed system without central (or any) control
Free ranging, undirected (no standards)
Principles, not mechanisms
Theory, not formalism
Independent of Technology
Outline: Target drawn around arrows

WHAT IS A DISTRIBUTED SYSTEM?

A distributed system is one in which the communication of data between processes takes a significant amount of time compared to the time needed to execute one step of a process.

Example: PDP.10

SPECIFICATION

(Note: Numbers in parentheses are "pointers" to amplifying material in paragraph 6.2.)

Definition: A specification is that which lets one decide if a running system is behaving correctly.

State-free Methods
- Applicative programming (6.2.1.1)
- Teletype paradigm (6.2.1.2)
- Observable I/O behavior (6.2.1.3)

State-based Methods (6.2.1.4)
- State graphs (6.2.1.5)
- Critical sections (6.2.1.3)

Problems
- Avoid explicit state description (6.2.1.6)
- How to specify complex systems (6.2.1.7)

MODELS

Definition: A model exhibits the properties of an implementation

MODELS CONSIDERED (Procedures and Files)
- General test and set model (6.2.2.1)
- Bit transmission model (6.2.2.2)
- Interpretive model (6.2.2.3)

OTHER MODELS (6.2.2.4)
- Actor- induction

- LISP
- etc.

RELEVANCE OF MODELS (6.2.2.5)
PROBLEM AREAS (6.2.2.6)
- Existence of single basic model

ANALYSIS

Inferring a system's behavioral properties
Formal proofs of correctness (6.2.3.1, 6.2.3.2, 6.2.3.3)
Fault tolerance (6.2.3.4)
Performance
- Measurements (6.2.3.5)
- Complexity
  - Space (6.2.3.6)
  - Time (6.2.3.7)
- Data transfer (6.2.3.8)
- Simulation/emulation (6.2.3.9)

Problems (6.2.3.5)
- Trade-off techniques
- Relevance of models

6.2 AMPLIFYING MATERIAL

6.2.1 Specification

6.2.1.1 Applicative Programming

Want to represent a system as composition of side-effect-free functions.

Can extend a "pure" applicative programming language with constructs for multiprocessing:

- Suspended evaluation of subexpressions.
- Multisets - unordered collection of expressions which becomes ordered as evaluations terminate.

Encapsulation of expression evaluations gives alternatives of distribution of compution: factor problem into assigning "capsules" to processing nodes.

Potential disadvantage: in any "real" situation, there is a need for some global reference; such a reference cannot be handled if side-effects are forbidden.

Reference: [BUCK]

6.2.1.2 Teletype Paradigm

All that the user knows about a system is what goes in and what comes out. What happens behind the panels is of no concern to him. This view is captured by the following paradigm. There are N users, each sitting at a teletype. The system behavior consists of the N rolls of paper. The correctness of this behavior must be decidable just from looking at those teletype rolls.

6.2.1.3 Behavior by Interleaved Teletype Rolls

If I/O behavior is to be described in a way suitable for reasoning about composition of systems, it is not sufficient to consider only the separate "teletype rolls." It is possible for two systems with the same individual port behavior to be incorporated as modules in a larger system, causing different external behavior for the larger system. A sufficiently inclusive behavior description to avoid this problem can be given by describing the interleaved teletype rolls. Thus far, such descriptions have been used for simple synchronization and data base behavior, and appear to be

quite natural and usable.

6.2.1.4 State-based methods

A state-based specification method was used for the algorithms in [BURN 78]. There the appropriate mutual exclusion behavior was expressed by grouping process states into "regions" comprising critical states, other program states, and protocol states. Desired exclusion, deadlock-free and fairness behavior was then described in terms of the progress of processes through their regions. Such description led to clean formal reasoning about the processes. The description, however, does not appear to be very easily suited for reasoning about the system as a building block for larger systems.

6.2.1.5 State Graphs

Thiagarajan has used the global state model to give a simple definition of Shapiro's algorithm for the maintenance of redundant data bases in a distributed environment. This permits an elegant and simple proof of correctness.

6.2.1.6 Jellybean Example

There are examples of simple systems in which one cannot talk about the state of the system at any particular point in time. The example involves two processes modifying the number of jellybeans in a factory, and one process counting the total number of jellybeans. The behavior of these three operations cannot be explained by any sequential ordering of their executions. How can we specify correctness of this system in a sufficiently general way to allow this type of implementation?

Reference: [LAMP 76].

6.2.1.7 How to Specify Complex Systems

We are faced with a dilemma. We do not want to have to mention states in our specification. But it is very difficult to write any non-trivial specification without talking about states. For example, try specifying a memory cell without talking about states.

6.2.2 Models

6.2.2.1 The Test-and-Set Model of IPC

The Test-and-Set primitive is a powerful indivisible operation for accessing a shared variable for communication among asynchronous processes. The model treats asynchronous operation by considering timing sequences. Correct algorithms must work for all timing sequences. Fairness properties may require that the timing sequences be restricted to those satisfying "finite delay." A sequence satisfies finite delay if no process has to wait forever for a timing message.

The Test-and-Set primitive is in one sense the most powerful primitive possible. Hence, the lower bounds results for this model apply directly to all weaker primitives.

To model general distributed systems, it is necessary to model processes and significant-distance communication. To model a message channel in the simplest and most natural way, we think of it as a special type of process with access to two variables, one at each of its ends. The process simply reads the contents of one of the variables and writes the result in the other variable, ad infinitum. We imagine this process to be asynchronous with respect to the other processes in the system. Thus communication delays are assumed to be arbitrary. This model seems simple and general enough to provide a basis for simulating and comparing distributed systems of practically any type.

6.2.2.2 Bit Transmission Model

Lamport favors a more low-level IPC model: transmission of 1 bit of information from one process to another. Requires a 1 bit storage device which can be written by process A and concurrently read by process B. Non-trivial to implement on atomic register which acts as if reads and writes are totally ordered. Some results are in [LAMP 77], others are unpublished.

6.2.2.3 SS Model

The applicative technique uses an interpretive language to describe a distributed system. An interpreter for applicative language may then serve to model system behavior. The unordered evaluation of expressions in a multiset becomes implemented as a scheduler. Communication may be modeled in terms of the elapsed simulated time associated with each parameter passing operation.

6.2.2.4 Other Models

Certain models, although significant, failed to receive attention due to the lack of advocates in the group. Most notable were the Actor-Induction Model of Carl Hewitt and Petri Nets.

6.2.2.5 Relevance of Models

Models of distributed systems are abstractions of real or hypothetical systems. The relevance of any abstraction depends strongly on its intended application -- the abstraction should preserve the important features of the situation being modelled and discard the unimportant. Models reflecting details of current technology are appropriate for understanding present-day distributed systems but they become quickly obsolete as the technology shifts. Models attempting to capture the universal constraints on any system imposed by basic laws of physics are more fundamental, but evaluating their relevance to digital systems requires a considerable understaning of electronics and physics, and they will likely be too primitive and detailed to shed much light on higher-level issues such as those discussed elsewhere in this report.

For example, most models of parallel systems include some sort of synchronization primitive whether it be P and V, monitors, message-passing, or whatever, and most practical systems have hardware which implements these primitives satisfactorially. However, the glitch problem aparently prevents the construction of a perfect arbiter (as oppsed to one which is satisfactory because its probability of failure is infinitesimally small), so any physical realization of an arbiter has a possibility of failure through infinite delay. The test-and-set model and the 1-bit transmission model can both describe perfect arbiters and so both must be considered only approximations to reality. While test-and-sets seem at first sight to be far from primitive, they encompass operations such as read, write, increment memory, etc. which might or might not be atomic in a given system, so lower bounds on complexity apply to all such weaker models. The fact that a fair arbiter is needed for a hardware realization of the model does not detract from its usefulness in describing solutions to the critical section problem, for building critical section solutions with strong fairness properties (bounded-waiting, FIFO) from arbiters only known to be free from lockout is a non-trivial task.

6.2.2.6 Problem Areas

Although a number of models were proposed for interprocess communication, we observed that there was no "basic unit" by means of which all of them could be implemented. Identifying such a basic unit would give a uniform scale for comparing different communication mechanisms.

6.2.3 Analysis

6.2.3.1 State Graph Analysis

See 6.2.1.5

6.2.3.2 Critical Region Algorithm Proof

A formal proof has been developed for one of the mutual exclusion algorithms given in [BURN 78]. Although the proof follows the general format of invariant-assertion proofs, the major ideas in the parts of the proof that deal with fairness are contained in precisely-stated lemmas which mirror natural intuitive understanding of the algorithms. The parts of the proof that deal with reachability of states have a less intuitive and more case-analytic flavor. A current effort is to decompose the invariants in a way that will allow reachability properties also to be verified in a way that accords intuition.

6.2.3.3 Global Assertions

There are well-developed techniques for proving correctness properties of non-distributed multiprocess programs. Lamport used to feel that they were not good for distributed systems because (1) they used global assertions which imply a global system state, which is undesirable (see 6.2.1.6), and (2) they require that communication arcs be represented by processes, which means lots of processes. However, he has recently discovered that these techniques do work well, since (1) there seem to be a class of "good" global assertions, and (2) you have to specify the communication arcs very carefully anyway.

6.2.3.4 Fault Tolerance

We consider two types of failure: unannounced halting (sleeping) and announced shutdown (dying). Peterson and Fischer [PETE 77] and Rivest and Pratt [RIVE 76] give critical section algorithms in a shared-variable read/write model that are immune to process dying, i.e., the remaining processes continue correct operation.

Performance and tolerance to failure by sleeping are closely related. If one process can be hung up forever because it is waiting for a failed process, then its performance will be degraded by a non-failed process that is simply running very slowly.

We have algorithms for the test-and-set model solving the k-critical section problem which in a sense have k independent paths to the critical section. That is, even if k - 1 processes fail, the other processes will not be waiting on them and will continue operating and gaining access to the remaining resources.

6.2.3.5 Measurements

The traditional measures of "time" and "space" do not form an adequate framework for assessing the complexity of distributed computations. In order to understand the "cost" of a distributed computation, we need to enlarge and refine our collection of cost measures. For example, "time" may refer to total time or time measured at an individual site. Similarly "space" could refer to either the size of the total system, or the size of individual sites. In addition to the "time" and "space" required to perform a computation, we should also consider the "amount of interprocess communication," both the total traffic flow over the whole system, and the bandwidth requirements of individual channels.

In analyzing sequential processes, we are used to thinking in terms of time-space tradeoffs. Are there analogous tradeoffs for distributed systems? For example, one can usually get by with smaller individual processors if one is willing to have more processors, and consequently, more interprocessor communication. Can this tradeoff of interprocess communication vs. complexity of individual process be made precise? Again, one usually has the choice of either implementing shared global resources or duplicating these resources at different sites. Are there guidelines for deciding which of these strategies to pursue? In general, we need to deal with the following sorts of questions: (i) What are the characteristics of those problems which allow one to make effective use of distributed computation? (ii) Conversely, can we learn to recognize problems whose solution would require such large amounts of interprocessor communication as to render these

problems inherently unsuited for solution in a distributed manner? (iii) Can we identify techniques for tailoring distributed architectures to the solution of particular computational problems? (iv) Can we formulate a theory which combines concerns for time-space complexity with concerns for minimizing interprocess communication, thus providing an adequate framework for assessing the complexity of distributed computations.

6.2.3.6 Space Complexity for IPC

In measuring space complexity for IPC, the shared variable models provide a natural measure - simply the number of states necessary in the shared variables. Tight upper and lower bounds on the communication space required have been demonstrated for certain synchronization problems using the Test-and-Set model. Additional bounds are anticipated for other problems and primitives.

Reference: [BURN 78]

6.2.3.7 Time Complexity Measures for IPC

A great deal of work has been done in the time complexity of sequential algorithms. Synchronous parallel computations commonly use a "tree depth" measuere for the time complexity. These techniques do not extend easily to asynchronous parallel processing because there is no direct measure of global time directly derivable from the steps of the individual processes. For example, if any process reaches a state where it must wait for communication from another process, it may take an unbounded number of steps before the remainder of the system changes state. Since a simple sum of all processor steps would often give unbounded lower bounds for many problems, (and hence are uninteresting), new measures are needed. Current work is proceeding examining time bounds of test-and-set algorithms using the following types of bounds.

1) Count the total number of "transitions" between two events of interest.
2) Count the number of transitions of a particular process between two events.
3) Count the total number of transitions between two events divided by the number of processes involved.

(A "transition" is a step of a process which causes a change in the shared variable) Each of these bounds appears to be of interest.

6.2.3.8 Data Transfer Performance

Abelson [ABEL 78] has recently developed techniques for proving inherent lower bounds on the amount of interprocess communication required for performing computations in a distributed system. Using these techniques, he has analyzed distributed systems which perform matrix operations and solve systems of linear equations. His work shows that, from the point of view of minimizing communication, the obvious techniques are optimal.

6.2.3.9 Performance Results

An alternative (perhaps a copout) to formal analysis is to use a simulation or emulation. This, however, is not an entirely straightforward proposition. First, a suitably accurate description of the distributed system must be derived and second, the artificialities of the simulation/emulation must be factored out.

6.3 POSITION PAPERS

6.3.1 Abelson

Theoretical Issues in Distributed Computation

by

Harold Abelson
MIT

Current research in the area of distributed computation focuses almost exclusively on algorithms and systems, while the problem of determining the inherent complexity of distributed computations remains virtually unexplored. Moreover, most theoretical work in the area of parallel processing relies on a model of computation in which all processors have ready access to all memory registers --- an assumption which is unrealistic when dealing with distributed computations. For example, although the solution of n linear equations in n unknowns can be accomplished in order (log n)**2 steps if one ignores information transfer, it can be shown that, for typical interconnection configurations among n processors the interprocessor data transfers alone require on the order of n steps.

We need to address directly the problem of interprocessor data transfer and to establish bounds on the amount of communication required for a wide variety of problems in a wide variety of distributed architectures. In general, we need to deal with the following sorts of questions: (i) What are the characteristics of those problems which allow one to make effective use of distributed computation? (ii) Conversely, can we learn to recognize problems whose solution would require such large amounts of interprocessor communication as to render these problems inherently unsuited for solution in a distributed manner? (iii) Can we identify techniques for tailoring distributed architectures to the solution of particular computational problems? (iv) Can we formulate a theory which combines concerns for time-space complexity with concerns for minimizing interprocess communication, thus providing an adequate framework for assessing the complexity of distributed computations.

6.3.2 Fischer

Time Complexity of Distributed Computations

by

Michael J. Fischer
University of Washington

A fundamental question in the theory of distributed computing is how well a particular system does its job. To determine this, one needs a specification of the job and a means of comparing the efficiency of the given system with other candidate systems.

Three aspects of distributed systems complicate considerably the specification of the desired behavior. First of all, non-terminating computations tend to be the rule rather than the exception, so infinite execution sequences must be described. Secondly, because of variablity in the relative speeds of the different processes, the system is inherently non-deterministic. While determinate behavior is nonetheless possible, it may not be required, so the specification must allow for variablity in the observed behavior. Finally, the inputs and outputs of a distributed system may be dispersed over a number of sites, and the communication aspects of the problem need to be captured in a natural way.

Finding a satisfactory time measure for distributed systems is much more difficult than for sequential programs. In the latter case, elapsed time is just the sum of the times of the basic instructions. With parallel computations, certain steps may execute concurrently, so the simple linear dependence of elapsed time on the instruction speed is lost. For this reason, it becomes attractive to look instead at the dependencies between steps of various processes rather than at elapsed time. When these dependencies are represented as a partial order, the longest path through the order gives a natural measure that reflects the time necessary, assuming maximum concurrency.

Once we have a satisfactory notion of the execution time for a particular interleaved sequence of steps, it is still not clear how to base a comparative analysis of systems on this information, for different systems solving the same problem will not necessarily exhibit the same interleavings. What is needed is a set of parameters common to all solution systems in terms of which the time can be expressed.

Finally, the relative efficiency of a system may depend strongly on whether one is interested in some notion of total system throughput or in response time at a given site (or in some other quantity).

6.3.3 Lamport

Theory and Formalism

by

L. Lamport
SRI International

Formal methods are needed to specify and prove the correctness of distributed systems. The primary requirement for a specification is that it be understandable by humans, since only a human can determine the correctness of a specification. Moreover, a specification involving program variables does not meet this criterion, since program variables are part of the solution, and are of no concern to the user. There has been very little progress in this area. It is rare to find even a precise informal statement of what a simple distributed algorithm is supposed to do -- let alone a specification of an entire system.

A formal specification is useful only if there is some formal method for deciding if a system meets its specification. Currently, there exist formal methods for proving properties of non-distributed multiprocess systems. We need to discover how these methods can be extended to distributed systems, or else develop new methods. There has been some progress in this area, but we are very far from being able to handle real, complex systems.

I feel that in order to make progress in these areas, it is necessary to be able to deal formally with non-atomic operations -- to describe the system as a collection of operations which do not act as if they were executed in any sequential order. I have some vague, preliminary ideas on how this can be done.

6.3.4 Lynch

Complexity Theory of Distributed Systems

by

Nancy Lynch
Georgia Institute of Technology

Most of the current work in theory of distributed systems seems to me to focus on a rather high level of programming. Virtual machines and networks, Hoare-style communication mechanisms which combine powerful synchronization and value-passing behavior, related mechanisms which assume preservation of unbounded numbers of messages, serializers, abstract data types with "nonatomic" elements, etc. are all user-oriented abstractions which allow logical organization of complex algorithmic behavior without concern for troublesome

implementation detail. Unfortunately, there are good reasons why such detail cannot entirely be suppressed. Efficiency of operation of a distributed system is of paramount concern to the user. There are so many more possible variations in implementation in a distributed environment than in more traditional computing environments that knowledge of the implementation method cannot help but influence the user's program design; indeed, some such knowledge is probably necessary for even acceptably efficient use of the system.

It is important to complement high-level theoretical and language-design work with a firmly-based theory of lower-level distributed programming, geared particularly to measurement of the efficiency of performance. Very simple and general primitives such as shared variables and one-way arbitrary-delay communication channels should be used as a general basis for such a theory. Various appropriate measures of resource use and performance (e.g., communication "bandwidth", total number of changes to variables that occur, total "depth" of the computation) can then be defined precisely. Then the costs of implementing the various high-level mechanisms mentioned above can be assessed objectively and compared. While the user might not need to know precise implementation details, he would at least benefit from knowledge of these <u>costs</u> in resource use, for the various available mechanisms.

As for sequential computing, the theory of distributed systems will not ultimately be concerned with implementation of different system primitives, but with efficient fulfillment of application requirements. Thus, the theory can be expected to focus on design and analysis of systems exhibiting certain desired behavior, in application areas suitable for distributed computing (e.g., load-sharing, multiple use of databases, mail communication, synchronization). A low-level model and elementary complexity measures such as those described will form a useful basis for such analysis, with higher-level constructs used along the way. Also important for such a theory will be the development of reasonably consistent means of specifying desirable behaviors for systems. Such behaviors might involve the input-output interface of a system or the internal state behavior of processes.

A prototypical development has been carried out (jointly with Michael J. Fischer and graduate students J. Burns, P. Jackson, and G. Peterson) for simple mutual exclusion behavior. Further work is currently in progress.

6.3.5 Smoliar

Theory and Formalism

by

Stephen W. Smoliar

Conventional modes of programming and algorithmic specification have many potential shortcomings in the design and implementation of distributed systems. In his 1977 ACM Turing Award Lecture, John Backus cited seven "inherent defects at the most basic level" in traditional programming languages: "their primitive word-at-a-time style of programming inherited from their common ancestor--the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs." Unfortunately, a good deal of thinking about distributed systems has become bogged down precisely because of a preconceived commitment to these same inherent defects.

A fruitful alternative is the functional style of applicative programming. The central idea is that all programs are expressed as functions. The coupling of a function with its arguments constitutes an expression, and a process is that computational activity involved in the evaluation of an expression. The most important aspect of this approach is that it has eliminated the need for the assignment statement, since the only allowable assignments are parameter bindings. Recursive composition of functions eliminates the need for loops (and with it many of the concerns of structured programming). Finally, input/output functions may be transcended by a view of files as arguments and values of expressions.

Multiprogramming concepts may be best expressed in applicative terms by introducing a data structure known as a multiset. A multiset may be viewed as an unordered collection of expressions whose evaluations may proceed in parallel. Retrieval of data from a multiset is contingent upon termination (also known as convergence) of at least one evaluation process; and retrieval effectively transforms a multiset from an unordered collection of expressions into an ordered sequence of values. Furthermore, multisets may be constructed through multiple applications of the same function to each of the elements of an already-constructed multiset. Finally, the conventional conditional expression may be extended to control whether or not an evaluation process ever converges: if the predicate of a guarded conditional is not true, then the evaluation process automatically diverges.

It is thus possible to formulate algorithms for distributed systems in terms of a rather simple applicative language. In fact, the applicative language provides a very powerful tool for the study of distributed systems; this tool is the language's interpreter. Such an interpreter must know how to implement the evaluation of expressions; but, more importantly, its definition must include a protocol for how multisets are constructed and how their elements are evaluated. This protocol may be instrumented to reflect the behavior of a real-time environment. The interpreter thus provided a basis for simulation experiments within which one may investigate how multiple processors may be profitably applied to multiset interpretation.

SECTION 7

CURRENT TECHNIQUES AND EXPERIENCE

7.1 A PROCESS BASED COMPUTER SYSTEM

An Informal Paper

by

Ed Basart
Hewlett-Packard Company

Processes are the basic entity in our computer system. When a program runs, it exists as a process, and gives a program the illusion that it has its own private processor. The system is then constructed to support processes effectively by making process communication and switching efficient and inexpensive. As a consequence, multiple processors can be used to increase the parallelism of the processes running in the system.

The advantages of such a computer system are program modularity, increased performance through parallelism, growth by adding processors, and physical distributability of functions. Processes are used as the single "object" that unifies operating system services and resources. The operating system exists as a collection of processes, and process primitives are used as the kernel of the operating system.

Processes communicate using queues and the send and receive primitives. Multiple queue writers are permitted, while only a single queue reader is allowed. Send and receive handle the details of the path between processes for any arbitrary hardware configuration of processors. This includes providing mutual exclusion for processors sharing memory and invoking data communication drivers in systems not sharing memory. The data communications processes resolve the connection between processors, whether the connection is a high speed bus, through telephone lines, or an indirect path through more than one processor.

In order to send a message to another process, the sending process must first establish a link to a receiving process queue. Links are made by the file system. Opening a link is very much like opening a disc file. Capabilities and access rights to queues are checked at open time by the file system, which eliminates message verification for the send and receive primitives, and also for the communicating processes.

After a link is open, the sending process sends a message to a receiving process by specifying a link number, along with the data. The receiving process reads its queue by specifying its queue number and issuing a receive. The receiving process creates a queue initially by asking the file system to allocate space for the queue and grant the receiver "queue" access. Linking a sending and a receiving process establishes half duplex communication. Full duplex communication may be established by creating another queue and opening another link in the opposite direction between the two processes.

As the file system opens a link, it determines whether the two processes are residing on different computers. If so, the address placed in the link is that of a surrogate process, a data communications driver that handles the details of the communication line. At the other end of the line is another surrogate data communications process. This process has a link pointing to the receiving process queue. This mechanism allows uniform process communication for both local and remote processes.

Creating a single queue for multiple writers seems to be a mixed blessing. One advantage is that the system makes a single space allocation for the queue, and no new allocations need to be made for each writer. Another advantage is that the reader goes to only one location to read messages. This is particularly important when the writers and reader exists on different computers.

The disadvantage of a single queue is that a "mad" writer can clog the queue. There are two solutions to this problem. The system can be made cognizant of a writer's "message rate," and a process can be given lower execution priority if its rate becomes too high. The other solution is to maintain a message count for each writer. The reader then decrements the count as the queue is read.

Neither of these solutions is very attractive. They both suggest high cost to protect against the mad writer. For the present the approach is to make queues large enough to absorb an initial outburst from the writer. The reader is given a "break link" function that disallows any further messages from a particular writer. This forces detection of the problem on the communicating processes while relieving the send and receive primitives of an added complication.

Three similar computer systems have been influential in the design of our system. They are: 1) the Tandem 16 computer system manufactured in Cupertino, California, 2) the Demos operating system for the Cray-1 computer at Los Alamos, New Mexico, and 3) the Thoth operating system developed at the University of Waterloo, Ontario.

Our system has two primary differences from the mentioned ones. The first is in handling all types of physical processor interconnections at the primitive level, rather than doing it in the operating system. The second is in making much greater use of processes and messages. All of the above systems break away from their message systems for certain types of functions that are considered to be too expensive to be done in a message system.

7.2 IPC IN HETEROGENEOUS DISTRIBUTED COMPUTER NETWORKS

HETEROGENEOUS DISTRIBUTED COMPUTER NETWORKS AND INTERPROCESS COMMUNICATION THEREIN

by

J. S. Sventek
Lawrence Berkeley Laboratory

7.2.1 Introduction

The primary focus of the Advanced Systems Group in CSAM is the question of distributed processing in a network consisting of hosts with vastly differing architectures. Our main goal, at this point in time, is to provide a distributed environment which is easily used by people with very diverse needs; for example:

1) a research group developing a distributed relational database system
2) administrative personnel maintaining current accounting databases
3) graphics researchers exploring new and novel representations
4) high energy physicists designing systems to collect and sample on-line vast quantities of experimental data

In order to achieve the goal of easy use, we are somewhat less concerned with "efficiency" issues than with merely making the system functional. From empirical studies of a working system, we hope to discern the "inefficient" aspects of the system, and may devise algorithms to alleviate the problems. Efficiency, in this context, is only concerned with throughput.

Two entities must exist before an easily used distributed system can be realized:

1) a common shell (command line interpreter). It is of somewhat limited utility to provide virtual terminal capabilities on the hosts in the network if the user must learn a different language to communicate with each one. Much of our recent research has been in the development of just such a portable shell. A prototype of this shell is currently running on the following systems: PDP-11/780 (VMS), PDP-11/70 (IAS), CDC 6600 (homegrown operating system).
2) a common file naming convention. Current research (based on a pathname structure) is

progressing in this area, and a prototypical system is operational on the PDP-11/70 (IAS) system.

The rest of the discussion will asume that these two entities exist on all hosts in the network.

7.2.2 Fundamental Quantities in a Computer System

There are three basic quantities in a civilized computer environment which a programmer must be able to manipulate. They are:

1. file - this category includes non-file structured devices (e.g., tt0, mt0, etc.), data files, and executable image files.
2. process - this entity describes an image file plus its context (standard input, output, and error files, default directory, priveleges, etc.) which is currently in a schedulable state or waiting upon some resource in order to become schedulable in a particular host.
3. vertex - this "virtual" entity allows two processes to extablish an interprocess communication link.

Several operating system primitives are necessary to allow a programmer to manipulate these quantities.

File oriented

open	open a file
close	close a file
create	if file exists, open it; else create it
delete	delete file
rename	rename file
getc	get a character from a file
putc	put a character into a file
mark	note current position in a file
seek	position a file
prompt	output string with no terminating carriage control

Process oriented

spawn	spawn process, sending specified arguments to it
pstat	query status of a process
kill	terminate process
suspnd	suspend process
resume	resume suspended process

Vertex oriented

pipe create a vertex and open a link to it

A few more words concerning vertices are in order. A vertex is a valid input parameter to the open and close primitives. In this way, subprocesses may be linked together by redirecting the respective standard outputs and standard inputs to a vertex. The subprocess itself is oblivious to the source or destination of its information. A vertex is also a transitory quantity, in the sense that when all links to it have been terminated (via a close operation), it vanishes. All I/O through a vertex should be synchronous to avoid all of the problems inherent in buffering asynchronous I/O in dynamic system memory.

7.2.3 Naming Conventions

Files are known globally by their pathnames:

/hostname/default directory/filename

Once a process has established a link to a file (via an open or create), the file is then known internally to the process by the id returned as the value of the primitive function invoked.

Processes are known globally by the id returned as a parameter of the spawn primitive:

/hostname/processid

Vertices are known globally by the following pathname:

/hostname/processid/vertexname

One sees that as long as the first field of a file pathname can never assume the value of a process id field, this naming convention uniquely identifies all quantities.

7.2.4 Implementation in a Distributed Environment

A skeleton of a typical primitive would look as follows

```
if (local (ARGUMENTS) == YES)
   {
   perform function
   }
else
   {
   reformulate request (if necessary)
   forward request to KERNEL
   wait for result
```

}

The purpose of the local function is to determine if the request can be performed within the requesting process. (File and process oriented primitives can usually be performed locally if they involve local files and processes.) If it cannot be performed internally, the request may have to be reformulated to include process context information, and is then forwarded to the KERNEL, which is an extension of the native operating system. Due to differences in the services provided by most native operating systems, one sees that the local function will be system dependent. The KERNEL is a separate process, one per host, which has access to the physical links of all hosts in the network which are directly connected to the current host. The KERNEL fields three types of requests:

1. local requests for local services not provided by the native operating system
2. local requests for services on remote hosts in the network
3. remote requests for local services on behalf of a requestor on a remote host

For the first type of request, the KERNEL will perform the service, and return status and any other information to the requestor. The last two types of requests are linked in their function. For type 2, the KERNEL forwards the request to its counterpart, which receives a request of type 3. This request is performed, and return information is forwarded to the original requestor through the network.

All types of distributed activity are then supported in such a network environment. The following examples will serve to emphasize this point.

7.2.5 Examples

Virtual terminal

User is currently interacting with the shell on host A with standard input, output, and error files being ttn, and default directory DEFAULT. User wishes to establish virtual terminal connection with host B. To do so, he/she issues the following command at his/her terminal

```
% B/shell
```

A/shell detects that this is a request to spawn a process at another host, so it reformulates the command as

```
B/shell <A/ttn >A/ttn >*A/ttn {DEFAULT}
```

and forwards request to A/KERNEL, which, in turn, forwards the request to B/KERNEL, which performs the service and returns status to the requesting process via A/KERNEL. The next prompt that the user sees will be that of the shell operating on host B, with the shell on A being suspended until B/shell has received an end of file on the standard input.

Host transparency to native utilities

User on host A wishes to copy a file from host A to host B; he issues the following command:

```
% copy file B/path/file
```

The shell will spawn copy, copy will open file, and attempt to open B/path/file. The open request will be forwarded to A/KERNEL, which in turn forwards request to B/KERNEL. B/path/file will be opened, and all writes to it will be directed through the KERNELs and the network link.

Interprocess communication between processes on different hosts

User on host A wishes to analyze a data file with a utility available on host B, directing the output of that utility to a graphic display program on host A which displays the results on the user's graphics terminal.

```
% B/analyze <mydata | A/graphit
```

A/shell will issue a spawn request to A/KERNEL with the following command line

```
B/analyze <A/DEFAULT/mydata >A/shellid/pipe1 &
```

where A/shellid/pipe1 is a vertex created by A/shell. The ampersand (&) indicates that A/shell does not wish to wait for the completion of the spawned process. A/shell will also spawn A/graphit, redirecting its input to A/shellid/pipe1. A/shell can then sit back and monitor the progress of the two cooperating processes, regaining control when they complete or terminating them if errors occur during their execution.

7.3 PROTECTED MAILBOXES AS AN IPC MECHANISM

by

R.L. Gordon
PRIME Computer, Inc.

Keywords: mailbox, IPC primitives, switch-board tasks, access lists

7.3.1 Introduction

It is the thesis of this short note that IPC facilities built around the notion of a protected mailbox could provide the basis for a robust set of primitives. Robustness, in this case, implies their utility in conventional multiprogrammed uniprocessor systems as well as shared memory multiprocessors, loosely coupled multiprocessors and local and long haul networks. The proposed mechanism can support different communication forms (N-process protocols), addresses security issues, and assists users in the synchronization of what is basically an asynchronous phenomenon (process communication).

7.3.2 Proposed IPC Primitives

Mailboxes are created by a process "P" executing a primitive of the form:

u = create(Access_List, T)

which is sufficient to bind the process name "P" to the unique descriptor "u" of the created mailbox, and associate the list of processes appearing in the "Access_List" with the mailbox "u". In addition the create primitive specifies a maximum time "T" between mailbox use (I assume mailboxes that are not used are not useful). Thereafter, if the identifier "u" is valid, (e.g. not equal to ERROR) then any process "P'" appearing on the "Access_List" and wishing to send mail to the process "P" would use a system call of the form:

send message(buf, u)

and continue execution. This primitive would have the effect of eventually placing the contents of "buf" in the mailbox "u" of process "P" along with the name of the sender "P'". Process "P", wishing to receive messages in mailbox "u", would make a system call of the form:

receive message(buf, u)

which would prohibit any further progress of "P" until either a message is received from a process on the "Access_List" or no message has been received during the time interval "T", specified in the "Create" primitive. Notification of this fact would would appear as a message in "buf" if the user had included a system process responsible for communication monitoring in his "Access_List". [See Section 7.3.6 on Fault Tolerant Aspects.] To complete the set of primitives a system call of the form:

delete(u)

would cause the mailbox "u" to be retired forever.

7.3.3 Initialization

Initial dialogues are established by "receiving" an identifier "s" of the current system mailbox in a mailbox "r" that was originally created with only the name of a well known system process on the access list. The system mailbox identifier "s," would then be used to send messages to the system kernel, with replies being received in mailbox "r".

One of the more difficult issues is with the design of the mechanism needed to establish communication with generic processes, (e.g. processes that represent a single service but may have multiple instantiations) and with discovery of newly created processes. The trouble stems from the fact users are incapable of establishing a dialogue with any process not known to them, and therefore cannot include them on the access list. For these reasons, it seems desirable to provide a "switch-board process" whose sole function is to provide a generic to specific name mapping. For example, such a service would be used to return the specific process name (or names) of the latest version of a fancy text formatter, when supplied with the generic name "format".

7.3.4 Security

A unique descriptor represents a sort of capability (at least for communication purposes) since possession of a mailbox identifier provides the possesser with the potential for sending messages and requests to the process bound to the identifier. However, if the target mailbox does not have the sender on the access list the message may be discarded by the system, thus essentially controlling communication through the maintainence and enforcement of the "Access_List." It is clear, therefore that security issues revolve around the ability to control changes to the "Access_List," an issue already explored by file system designers.

If one takes the view that a message is an attempt to access an object by a principal [GRAH 72], then this facility contains all the elements of the access matrix model [LAMP 71] of protection. By having different processes act as monitors of objects one has a formalization of the access model since the identification of the accessor and the object being sought are both available to the monitor process.

7.3.5 Synchronization

The availability of the senders identification coupled with the access control list provides the means to achieve solutions to synchronization of processes and to detection of boolean combinations of events. Creation of mailboxes with only one process name on the "Access_List" provide the facilities for a simple "pipe" (one way communication channel) that can be used to construct a self clocking "pipeline" with the "send" and "receive" primitives. Logical "or"-ing of the input from two processes, say A and B, can be accomplished by simply including A and B on the "Access_List." More complicated forms of synchronization can be accomplished by creation of an intermediate process that performs the appropriate level of demultiplexing. Broadcast transmissions are simply achieved by iteration over a set of available mailbox identifiers.

7.3.6 Fault Tolerant Aspects

There appear to be many forms of communication errors that are recoverable by the technology underlying the IPC level. Failure of underlying mechanisms can easily be reported to a process if it opens a channel for that purpose by including the name of a system process on the "Access_List" on an already opened mailbox, or opening one for just that purpose. It seems to me that users who do not want to be concerned with error handling, should not be forced to carry along a lot of extra apparatus for those who do. One nagging concern of mine is whether the system should force error messages (especially for timeouts) into mailboxes that have not included the communication monitor on the "Access_List."

Positive acknowledgement is purposefully not included in this scheme, but is left to the user to construct his own by setting up a duplex path between processes. As an aid, the design of the "create" primitive must have a value "T" for the maximum time between messages. Since the primitives are designed to be used over a wide range of situations most applications will have some knowledge of how long it is reasonable to wait for a reply or input from a cooperating process.

7.3.7 Summary

A set of primitives for interprocess communication have been proposed that seem suitable for implementation in a wide variety of circumstances. Only briefly mentioned however, is the issue of process addressability when communication is desired between several processes. The solution of this problem requires the development of a name space architecture that tackles the relationship between files, devices, processes, users and many other system objects, certainly beyond the scope of this short note.

7.4 BRIEF DESCRIPTION OF DSYS-PLITS

by

James R. Low
University of Rochester

The Model of Computation

The model of interprocess communication that we use in DSYS-PLITS has evolved from that used in the RIG (Rochester Intelligent Gateway) Operating System. Basically, we think of a program being composed of several independent processes (we call them "modules") communicating with each other only through messages. There is no directly shared memory. Processes are relatively stable and to "fork" a process means to create a totally new environment independent from that of the creator. Our basic model does not force any hierarchy on the processes though it is relatively easy for a programmer to think in terms of hierarchies if he wishes.

DSYS (Distributed System)

DSYS is basically a set of facilities added to existing programming languages and operating systems to support inter-process communication across a network of heterogenous machines (DEC PDP-10 running DECSYSTEM-10, Data General ECLIPSEs running RIG, and XEROX ALTOs running the ALTO operating system). DSYS consists of operating system interfaces and user interface procedures.

Processes communicate via messages. The SEND primitive supported by DSYS takes three parameters: the message to be sent; the process identifier of the destination (originally obtained through interactions with a name service process, or provided in a message from some other process); and a transaction key (analogous to a "port"). All connections between processes are implicit. If a process has obtained another process's name it can send that process a message without any explicit "open" command. Of course, the processes themselves may ignore messages which do not conform to higher level (user-specified)) protocols. Transaction keys are used to separate various conversation streams. DSYS will guarantee that all messages with a specific transaction key sent from one particular process to another will arrive in the proper order. No guarantee is made about messages with different transaction keys. Details of the reliable transmission and flow-control mechanisms in the DSYS subnet may cause messages from one process to another with different keys to arrive in a different order than they were SENT.

Selective reception of messages is provided. A process may state that it wishes to receive only messages from a specific set of other processes or about specific transaction keys. Thus the general form of RECEIVE is

RECEIVE msg FROM (sndr1, sndr2,... sndr3)
ABOUT (trns1, trns2...)

If there is more than one message that has suitable SENDER and TRANSACTION, an arbitrary one is selected (subject to the constraint of ordering within a SENDER-TRANSACTION pair mentioned above). If the user wishes to enforce more general priority mechanisms he may use the PENDING construct to see if there are suitable high priority messages before he receives lower priority ones. PENDING takes the same arguments as RECEIVE and returns TRUE if there are suitable messages and FALSE otherwise. It does not actually perform the RECEIVE so the message queues are left intact. If all else fails and the user wants more general reception criteria then he can ask to receive all messages and then do his own local queing. We believe this to be very rare and have not seen this done in the programs coded so far.

DSYS performs all queue management, reliable transmission, and flow control. Application programs are notified of exceptional conditions (communication lines going down, other processes in the "distributed job" breaking) via emergency messages.

PLITS Messages

DSYS itself considered messages as just strings of bits. We have found it desirable to provide higher level message support to applications programs. This higher level message support is called PLITS.

Traditionally, fixed message formats have been used for application programs. To design a new message type, a programmer would lay out an explicit template for his data. He would have to state the number of pieces of data, their data-types; the external representation of the data type; and the translation routines to use to translate between the external (used in messages) representation and the internal (used in his program variables) representation of the data.

In PLITS, we try to remove the burden of message template design. By automating the process we also remove one class of possible errors. In PLITS, the applications programmer sees a message as a set of keyword value pairs. We call these pairs, "slots". To construct a message he specifies the particular set of slots he desires. The receiver can determine (for individual messages) which slots are present

and their values. Thus, a message to a file server might look like:

```
SEND (action ~openfile, mode ~update, name ~"MYFILE",
      directory ~"<mydir>", initialposition ~0, bytesize ~8)
      TO FileServer ABOUT OPNTransaction;
```

"action", "mode", "name" and so forth are the keywords (or slotnames). The message would be identical as far as the receiver were concerned if the sender had specified a different order of the slots. We do not require that every message contain a specific set of slots, but of course it is an error if a process attempts to fetch the value on a non-existent slot. Defaults may be easily implemented using the PRESENT IN primitive. For example, the file server above might wish to assume that the directory is "<SYSTEM>" if none is specified.

```
RECEIVE msg FROM ANYSENDER ABOUT ANYTRANSACTION;

IF NOT (directory PRESENT IN msg) THEN
      PUT (directory~"<SYSTEM>") IN msg;

thedirect := msg.directory;
```

When a user wants to use a slot in his program he must declare the keyword and the type of its value both in the sending and receiving process.

```
STRING SLOT filename;

MODULE SLOT continuation;
```

In the existing implementation of PLITS (see below) the data-type of each slot is sent in the message and consistency is checked during the translation from the external format of messages to the internal format of messages during reception of the message. Implementation is underway to have a "loading" time (when a process joins a "distributed job") when the consistency of slot definitions would be checked. Small identifiers for each slot would also be given at this time. This would decrease the overhead of the slot mechanism (currently in addition to the data, a type code and a character string are sent for each slot).

In the current implementation the "data-type" of a slot implies the external representation of the value of the slot within messages. Thus we have several INTEGER types.

```
INTEGER16 SLOT small;

INTEGER32 SLOT large;
```

with implied external representations of sixteen and thirty-two bits. Note: this does not imply that the internal representation for the value of the two slots above must necessarily be different. For example, in the PDP-10, both values would be represented using 36-bit integers. When a message is sent, however, a check is made during the encoding into the external format that the values are in the appropriate ranges. Future implementations may have a "negotiation" phase during "loading" in which the various processes "agree" on the external precision necessary for each data value (one "negotiation" strategy would be to use enough bits for the maximal declared range).

Current State of Implementation

The DSYS has been running since last Spring on the PDP-10 and ECLIPSE computers. A distributed vision application was encoded this past Summer. Recently an ALTO DSYS support package has been used to link ALTO's to the ECLIPSE. The PLITS message format has been running on the PDP-10 for over a year (using a preliminary version of DSYS that ran only on the PDP-10). A design for the support facilities necessary for PLITS on the ECLIPSEs and ALTOs has been completed.

Almost all the support software has been written either in SAIL (on the PDP-10) or BCPL (on the ECLIPSEs or ALTOs).

7.5 MODELS OF CONCURRENT COMMUNICATION ACTIVITIES

PARAMETRIC MODELS OF CONCURRENT COMMUNICATION ACTIVITY

by

Bill Buckles
General Research Corporation

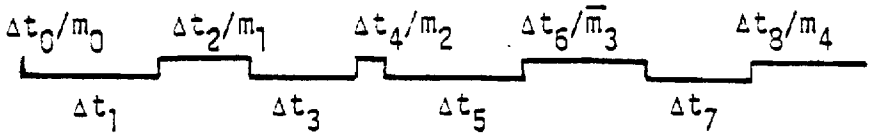
INTRODUCTION

Using a distributed system to feign, simulate, or emulate a second distributed system is of interest primarily to those engaged in design. The principal problem in this approach is the inherent timing discrepancies between the existing and target systems. Lamport [1] has made invaluable contributions applicable to this area and this study is directed at specializing his results to emulation.

MODELS AND STATES

The goals are to determine (1) what aspects of communication behavior can be observed from an emulation? (2) what ancillary relationships must be embedded in an emulation to assure that the primary behavioral attributes can be extracted? and (3) if the ancillary relationships are not exact, how much confidence may we place in the extracted primary behavioral attributes? In order to achieve this, a definition of process state has been derived that deals only with aspects of inter-process communication. The target process state is distinct from the emulation process state, but the former is embedded within the latter. Additionally a progression of six communication models have been defined, each an elaboration of the previous one.

Model 1 is a single process emulating itself. It may be schematically represented as



* Work sponsored by the Ballistic Missile Defense Advanced Technology Center, P. O. Box 1500, Huntsville, Alabama 35807 under contract number DASG60-78-C-0058.

where Δt_i denotes a time interval, m_i a message, and the even intervals denote active communication periods. Model 2 is a single process emulating a second process with uniform time distortion (either rate increase or decrease). Model 3 is a single process emulating a second process with both uniform time distortion and non-uniform perturbations (strictly slow-down). In this model, the emulation process may contain more periods than the target process. However, there must exist an order-preserving mapping from the target process periods to the emulation process periods. Model 4 advances to multiple processes with equal time distortions and perturbations. Model 5 relaxes the equality constraints on distortions and perturbations, but requires the two be balanced. That is, inequality among the time distortions of various processes must be offset by perturbation. Model 6 is completely unconstrained with respect to both distortion and perturbation.

The state of a single target process, i, at time period j is denoted by the pair $s_{ij} = [\Delta t, \eta]$ where Δt is the duration of the most recently completed period and η is the information sent or received. The state of the target system is denoted $S = [s_{1j_1}, s_{2j_2}, \ldots, s_{nj_n}]$. The state of a single emulation process i after time period k is denoted by the 5-tuple $\sigma_{ik} = [s_{ij}, \Delta t', \mu, r, \rho(k)]$ where s_{ij} is the state of the target process, $\Delta t'$ is the duration of the most recently completed period, μ is the information sent or received during the last period, r, a constant, is the uniform time distortion, and $\rho(k)$ is the instantaneous perturbation at the beginning of the current period. A system state is denoted by $\Sigma = [\sigma_{1k_1}, \sigma_{2k_2}, \ldots, \sigma_{nk_n}]$. A system state change occurs when exactly one σ_{ij} assumes a new value.

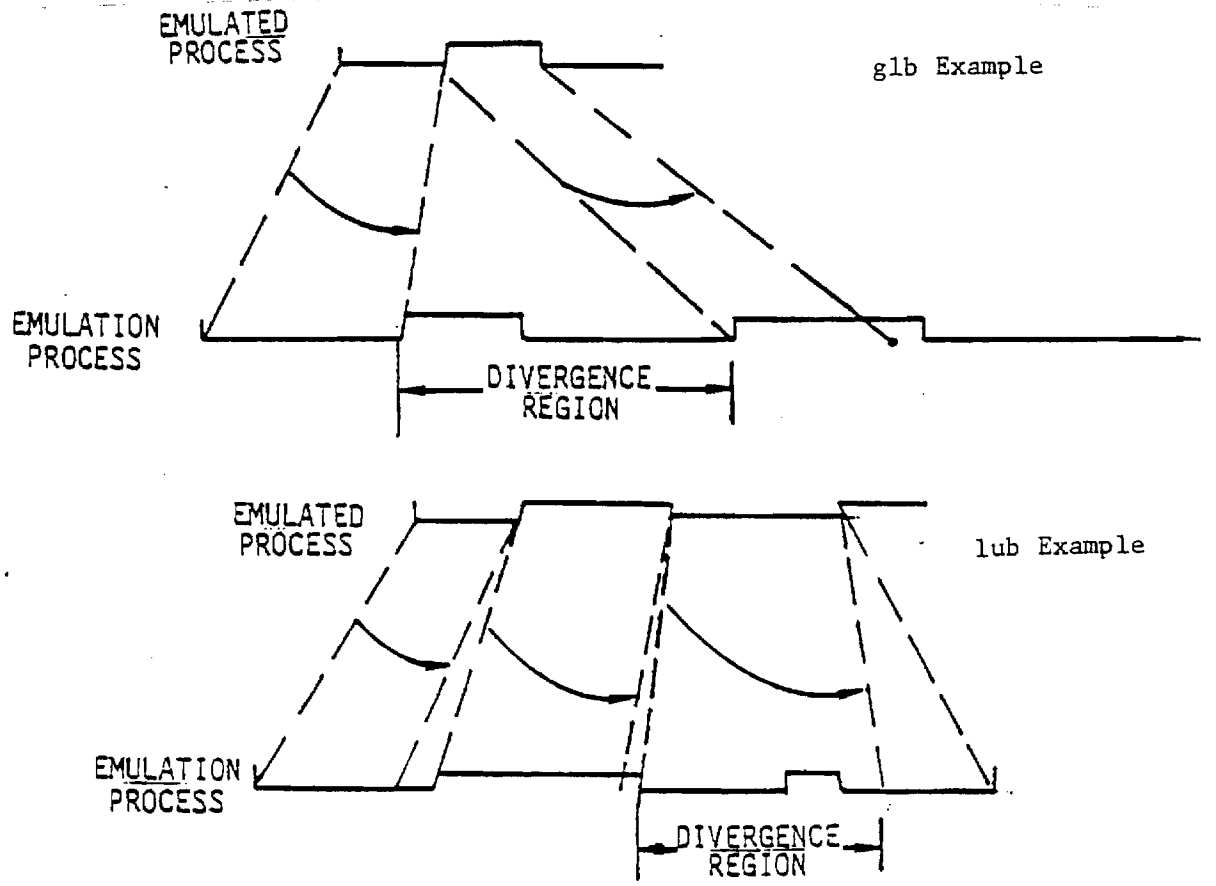
PRELIMINARY RESULTS

Time models are inherently continuous while the state model described above is discrete. Lower and upper bounds on the time relationships are desirable to fix the amount of error between state changes. Because r (the distortion) is constant, only ρ (the perturbation) may introduce error:

$$\text{glb}(\rho) = \rho(n)\left[1 - \left(\Delta t'_n \Big/ \sum_{i=1}^{n} \Delta t'_i\right)\right]$$

$$\text{lub}(\rho) = \rho(n) + \left[\Delta t_{v+1} \Big/ r\sum_{i=1}^{n-1} \Delta t'_i\right]$$

Unfortunately, lub(ρ) required the prediction of the period duration, Δt_{v+1}, of a current target process. An assumed order-preserving mapping illustrating the lower and upper bound errors follow.



Model 6, being the most general, is of interest. For example, determining what measures must be taken to preserve the state transition ordering in the emulation to reflect accurately the state transition ordering in the target process is necessary. If $S_a < S_b$ in time and the transition to S_a is embedded in Σ_x and the transition to S_b is embedded in Σ_y then we would desire that $\Sigma_x < \Sigma_y$. Let ${}_x\sigma_{ij}$ be the specific substate that changes value at Σ_x and ${}_y\sigma_{km}$ be the specific substate that changes value at Σ_y. Both $S_a < S_b$ and $\Sigma_x < \Sigma_y$ if

$$ {}_y\psi_{ij} \cdot \sum_{w=x}^{y-1} \tau_w > [{}_y\psi_{ij} - {}_x\psi_{km}] \cdot \sum_{w=0}^{x-1} \tau_w $$

where ${}_p\psi_{qv} = {}_p\sigma_{qv}(\rho(v)) \cdot {}_p\sigma_{qv}(r)$ and τ_w is the normalized elapsed emulation time in period w-1. In symbols:

$$ \tau_w = r \cdot \sigma_{ij}(\rho(w) \cdot \sigma_{ij}(s(\Delta t)). $$

CONCLUSIONS

These and other relationships dealing with the communication behavior of emulation processes have been formally proved. Some knowledge on the problem of what information to collect and how to analyze it has been gained. It is believed that future investigation will further strengthen the utility of the models.

REFERENCES

1. Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM 21, 7 (July 1978), 558-565.

7.6 PRIME IPC CONFERENCE REPORT

by

Robert L. Gordon
and
Jack A. Test

The enclosed Prime research note is partly based upon a couple of early 1978 internal Prime R&D meetings concerned with "Task Control and Communication for Multiple Processor Systems". It discusses the synchronization and interprocess communication mechanisms used in a number of important operating systems and explores the importance of these mechanisms for the development of future computer systems, and is offered as additional material for the current techniques and experience section of the conference report, since it summarizes a review of mechanisms used in several well known systems.

7.6.1 Introduction

Two in-house meetings concerned with "Task Control and Communication for Multiple Processor Systems" were held on January 11, 1978, and March 22, 1978. The purpose of the meetings was to provide a forum for the discussion of existing operating system mechanisms for process management and interprocess communication as related to Prime's efforts in process-based computer network architectures.

The two meetings consisted of a series of informal presentations by members of Primes R&D staff on other systems and discussions on related PRIMENET communication meetings. The particular topics were: (1) "Process Communication In DEMOS", (2) "Process Control And Communication In UNIX", (3) "TANDEM And VAX Process Structure", (4) "The Multics IPC Facility", (5) "Event Counting And Sequencing In Distributed Systems", and (6) "Communication Primitives For PRIMOS".

The purpose of this note is to discuss the synchronization and interprocess communication mechanisms developed for the systems mentioned above and to explore future directions in the development of process-based computer networks. Observations concerning the IPC facilities of the operating systems discussed are based upon the authors' knowledge of the systems, available literature, and the Prime Conference talks. Accordingly, Section II of this note presents brief summaries of the IPC facilities, and Section III states some conclusions and future directions. The References & Selected Readings, at the end of this note, lists several articles pertinent to the study of Interprocess Communications.

7.6.2 Synchronization/IPC Facilities

Included in this section are discussions of the synchronization/ipc mechanisms developed for the systems mentioned in the Introduction. For additional information regarding each system, refer to any of the pertinent references.

7.6.2.1 Process Communication in DEMOS

DEMOS is an operating system under development at the Los Alamos Scientific Laboratory for the CRAY-1 computer [BASK 77]. A task or process in DEMOS consists of a program and its associated state information which includes a link table. The primary mechanism for communicating between user and operating system tasks is by passing messages over links. Links are associated with, but maintained outside the address space of sender tasks and are essentially one-way (simplex) communication paths. All operations on links are performed by the kernal of the operating system which insures their integrity.

Appropriate standard links are provided by the system for user tasks requesting operating system services. These are provided in an automatic and transparent way, one such standard link being to a switchboard task. Switchboard tasks can arrange to get two or more mutually cooperating processes together, and since tasks may under certain conditions pass link identification information as a message, dynamic process networks may be easily constructed.

Links resemble capabilities, so their management must take into account many of the well known difficulties of managing capabilities. Some of these, such as lack of control over link passing and link duplication have been partially alleviated by classifying links into specific types and restricting specific operations to these types. Other facilities include data segment links and channels that are associated with links in order to provide facilities for multiple event handling and windows into task address spaces.

The communication mechanism of DEMOS is not pure in several ways. First, data segments are an escape from communication only by messages; and second, conditional receives and channel interrrupts provide an escape from the sychronization provided only by message primitives. However, with proper hardware support these escapes might not be necessary.

7.6.2.2 UNIX Process Control/Communication

The UNIX system was developed at Bell Telephone Laboratories for the DEC 11/40, 45, and 70 minicomputers. The basic literature reference to the system [RITC 74] provides a good explanation of the principle ideas incorporated in the UNIX design.

In UNIX, a "process" is defined to be the execution of an "image" where an image is a computer execution environment, namely: allocated core, register values, open files, etc. Images are small in UNIX, roughly 32K words + status information, and the system is oriented around their execution manipulation.

Processes are organized in a parent-child tree-structure within the UNIX system environment. Parent processes can spawn (create) child processes dynamically through a fork system call. Initially, the child process is a copy of the parent process but with a different return value from the fork call. The child inherits the parent's environment (i.e. open files, register values, etc.) but does possess its own memory image. Typically, a child process will initiate an exec system call which will overlay the child image with the startup image of a program named in the call. In this manner, a parent process can create any child process it desires.

The main form of communication between parent and child processes is accomplished through pipes created by the parent process. Since the parent's environment is lost when a child process overlays itself, the pipe descriptor must be passed as an argument to the overlaying "exec" system call. Pipes serve as serial data paths with one "write end" and one "read end". Multiple processes can write or read a single pipe but data can be intermixed if the pipe is not locked on writes. In addition to the pipe mechanism in the original release of UNIX, new versions of the operating system allow processes to communicate through messages that are routed and queued for unique process identifications. Messages in UNIX serve as a more general form of interprocess communication than pipes since "unrelated" processes can communicate using them. For mutual exclusion and synchronization purposes, the UNIX system provides both wait/signal and counting semaphores for use by user processes.

There are a number of limitations to the current IPC mechanisms available in UNIX. Specifically, pipes, because of their serial nature, must be used carefully in order to avoid mixed streams on the write end or lost streams on the read end. In addition, the message mechanism in UNIX requires the process-id of sending and receiving processes. Unfortunately, this information is not available through any system administered switchboard and must be handled by the processes themselves in some arbitrary manner. The naming

of processes, therefore, is not adequately addressed in UNIX.

In summary, the UNIX timesharing system provides a dynamic and flexible process environment with a high degree of modularity. Some notable shortcomings in the UNIX IPC facility (in addition to the problems discussed above) are: (1) the inability of a process to wait for multiple piped or message inputs, (2) the small address space available per process, admittedly a PDP-11 imposed limitation, and (3) the lack of any network process management capability.

7.6.2.3 **Interprocess Communication in TANDEM**

The Guardian Operating System [BART 77] for the Tandem Computers model 16 computer has as its foremost goal the maintainance of a failure-tolerant computing environment. Even though the underlying Tandem hardware consists of multiple computers and multiple dual-ported I/O devices, the operating system is designed to give the appearance to the user of a unified system through the novel application of several software abstractions.

The first abstraction provided is that of a process. Each processor module may have one or more processes residing on it, however a process may not execute on any other processor than the one it was initially created on. Each process in the system has a unique identifier or process-id of the form: <cpu #, process #>, which allows it to be referenced on a system wide basis.

Process synchronization primitives include counting semaphores and process local event flags. Semaphores may be only used for synchronization between processes within the same processor and are typically used to control access to resources such as resident memory buffers and message control blocks. Event flags are predefined for up to eight different events and are signalled within a processor by either hardware events, such as device interrupts, or by the function AWAKE. All event signals are queued so that they are not lost if the event is signaled when a process is not waiting on it, and a process may wait for the first of one or more events via the function WAIT. Processes may also specify a maximum time to block which, if exceeded, results in the return of an error condition to the process that requested it.

The message system used for communication between processes residing on different processors uses five primitive operations: LINK, LISTEN, READLINK, WRITELINK, and BREAKLINK, to implement what can be best thought of as dialogues between requestor/server pairs. Messages are queued for processes and result in the setting of an event flag for processes wanting to "LISTEN".

With the implementation of processes and messages, processor boundaries effectively disappear. System wide access to I/O devices is provided by the mechanism of process pairs. An I/O process-pair consists of two cooperating processes located in two different processors that control a particular I/O device. One of the processes is considered the "primary" one and the other the "backup" process. The primary process handles requests sent to it but sends information to the backup process via the message system in order to assure that the backup process will have all the information needed to take over control of the device in the event of an I/O channel or device error. Because of the distributed nature of the system, it is not possible to provide a "block" of driver code that could be called directly to access the device. While potentially more efficient, such an approach would preclude access to every device in the system by every process in the system.

Processes are not grouped in classical ancestry trees. No process is considered subservient to any other process on the basis of parentage, and two processes, one created by the other will be treated as equals by the system. When a process "A" creates another process "B", via a call to the procedure NEWPROCESS, no record of B is attached to A. The only record kept is in process B where the creation "id" of A is saved and is known as B's "mom". When process B stops, a STOP message is sent to process A. If B wants to know whether A has stopped it must "adopt" its mom.

The innovative aspects of the Guardian Operating System lie not in any new concepts, but in the synthesis of pre-existing ideas. Of particular note are the low level process and message abstractions. By using these, all processor boundries can be hidden from both the application programs and most of the operating system. These initial abstractions are the key to the system's ability to tolerate failures and provide the configuration independence necessary to run over a wide range of system sizes.

7.6.2.4 Process Communication in Vax

The VMS operating system architecture [DEC 77] supported by the VAX hardware is a process structured system. Because of this, the designers of VMS were motivated to look for and evaluate the utilization of alternate process communication schemes in order to ease the design and implementation of VMS. It is significant that this study resulted in three different mechanisms for process comunication in order not to force-fit applications into using any one particular type.

The three interprocess communication facilities provided by VMS are all software implemented. The first facility is apparently used for trusted processes (e.g. Kernal processes) and consists of the notion of event flags, event flag clusters, and masks that allow boolean combinations of event flags. Since it is well known that this form of (semaphore) type communication can be easily abused by naive users it apparently is restricted only to trusted processes.

The second type of interprocess communication used in VMS (internal communication) consists of send receive queues that have implicitly associated event flags. This mechanism serves as a way of passing variable quantities of data between trusted processes with a fairly high degree of efficiency. Each user process builds its own buffer (data packet) and sends it to a "receive" queue, which then sets the associated event flag for the receiving process.

The third type of interprocess communication mechanism (generalized communication) consists of primitives for handling mailboxes. Mailboxes can also be thought of and implemented as queue or FIFO files, thus they can use the same protection mechanisms as files. Of course mailboxes, like files, can be classed as both temporary and permanent so that interprocess communication can take place while processes are "absent" or dormant, a useful feature for writing to logged out terminals. In addition, processes communicate with mailboxes in a fashion similar to record-oriented I/O thus providing a framework for advanced concepts such as I/O redirection.

VAX/VMS supports not only processes, but also jobs that constitute a collection of subprocesses and groups that are sets of processes that share resources. Subprocesses can be spawned and can have the rights of the creator as well as the rights of the spawned image thus allowing a form of enhanced rights.

It seems that the VMS operating system provides a rich set of interprocess communication primitives; whether it is a consistent set and can be managed over the life of the system remains to be seen.

7.6.2.5 The Multics IPC Facility

The interprocess communication facility supported by the Multics system is based upon the concept of event channels. The primary purpose of an event channel is to provide synchronization between processes.

Event channels (which can be thought of as a numbered slots in the ipc-facility tables) are either event-wait or event-call channels. The event-wait channel receives events that have occured and awakens the process that established the channel if it is blocked waiting for an event on that channel. The event-call channel responds to the occurence of an event by calling a specified procedure if the process which established the channel is blocked waiting for any event.

For events to be noticed by explicitly cooperating processes, event channel identifier values are typically placed in known locations of a shared segment. Processes can block waiting for an event to occur or can explicitly check to see if the event has occured. If an event occurs before the target process blocks, the process is immediately awakened when it does block.

In summary, the event-channel facility in Multics provides a flexible synchronization mechanism. Typically, processes establish channels and wait for events on one or more of the channels they have created. The utility of this approach is clearly demonstrated by the use of the ipc-facility throughout Multics for all user process coordination and terminal I/O handling.

7.6.2.6 Event Counting and Sequencing

Synchronization of concurrent processes is usually required for the relative ordering of events internal to each process. Most currently favored synchronization techniques such as monitors [HOAR 74] and semaphores involve mutual exclusion, a technique that only indirectly notes the occurrence of an event. A alternate set of synchronization primitives have been proposed by Reed and Kanodia [REED 77] where a process controls its synchrony with respect to other processes by observing and signalling the occurrence of events through operations on objects called eventcounts. An eventcount is an abstraction representing the number of events in some class of interest that have occurred. Operations on eventcounts are: ADVANCE(E) - Signal one event; READ(E) - Return the number of previous ADVANCES on E; and AWAIT(E,V) - Suspend a process until READ(E) >= V. ADVANCE purely transmits information, READ and AWAIT purely observe. In contrast the P operation on a semaphore is not a pure observation primitive since it can modify the semaphore. Pure observation or signalling primitives are more attractive for use in secure systems [LAMP 73]. If only one process executes ADVANCE operations on an eventcount, ADVANCE and READ can be concurrent. If more than one process does ADVANCES, a different eventcount can be given to each process, and the sum of those eventcounts gives the total number of events in the class.

When mutual exclusion is needed (when events must be ordered dynamically, such that the ordering is not known in advance), a sequencer can be used. A sequencer operates like the ticket machine in a bakery, and has one operation called TICKET, that returns the number of previous ticket operations on that sequencer. An eventcount and a sequencer can be used to implement a semaphore. Several eventounts and sequencers can be used to implement semaphores that allow a process to wait for several different events.

There seem to be at least two attractive advantages over other alternate synchronization schemes that eventcounts have for distributed systems. The first advantage is that the ADVANCE operation affords a natural broadcast mechanism to all processes that might be waiting on an event, because unlike simple semaphores the signaller need not know the names of the intended observers. The second advantage is the avoidance of mutual exclusion where only the relative ordering of events is required, thus tending to limit the amount of serialized code in systems, code that often results in performance bottlenecks. Eventcounts and sequencers could be used by an operating system, instead of user-visible semaphores, for implementing more general interprocess communication mechanisms with shared files and this mechanism could be made available to the user to coordinate the use of shared resources.

7.6.2.7 Intertask Communication Primitives For PRIMOS

Several intertask communication capabilities currently exist within the Prime operating system (PRIMOS). Both lock/unlock and counting semaphores, are implemented at the microcode level, and are available for system and user tasks. In addition to these basic synchronization primitives for communication between processes on the same processor PRIMOS supports a set of PRIMENET inter-process communication capabilities based on x.25 flavored "virtual circuits". These capabilities allow a user process to establish a full-duplex virtual connection to another user process whether local or remote.

Virtual circuits can be managed at the user program level by the proper use of a collection of subroutine calls to PRIMOS and provide a "Level 3", X.25 Interprocess Communication Facility (IPCF).

The major services provided are for forming a connection, breaking a connection and transmitting or receiving data. Generally, two different forms of a service are provided. The first form is an abbreviated calling sequence, with only a minimum amount of information needed to be supplied by a user in order to establish and use a virtual circuit. The second form is a more detailed one that allows a user full access to all fields of the X.25 "Level 3" defined packet

formats. The latter form is intended primarily for users wishing to form X.25 connections to non-Prime hosts on Public Data packet networks.

Eleven network primitives currently compose PRIMENET and provide capabilities to: establish status as a network user (X$ASGN), establish a network connection (X$CONN), get local connect information (X$GCON), accept a connection (X$ACPT), clear a connection (X$CLR), hand off a connection (X$GVVC), receive via a connection (X$RCV), transmit via a connection (X$TRAN), wait on transmit or receive (X$WAIT), get network status (X$STAT), and terminate network user status (X$UASN). This set of PRIMENET primitives is based upon the X.25 protocol and is due for release under REV 17 of PRIMOS. The chief shortcoming to the current PRIMENET set of primitives is the inability to support multiple readers and/or multiple writers per connection.

The addressability defined in the basic X.25 specifications refers only to a single 14-digit address per host, although it is not uncommon for a host (like PRIMOS) to handle multiple processes and users. Therefore, in order to decide which user or operating system service should control a connection, each incoming "call request packet" in PRIMENET must specify a network "port." This port, coupled with the 14-digit address of the target system, designates a target process.

Each host in Ringnet has a pool of 255 available ports that may be assigned to any process on a first come, first served basis by a call on the operating system. However, only ports 1 through 99 are available for users; the rest are reserved for system use. Permanent port assignments to a process are possible by controlling the order in which processes are initiated just after system startup; otherwise, there is no absolute guarantee that a particular process is associated with a given port number.

The short form of the initial connection protocol uses an ASCII host name (e.g. "ENG.15") instead of the long 14-digit address and a port number previously acquired by the target process. The "connect" function is typical of the IPCF primitives and the request for it is shown as a partial example of how a circuit is formed at the program level.

```
CALL X$CONN (VCID, PORT, ADR, ADRL, VC_STAT)
```

The variable ADR points to a string containing the name of the intended host (i.e ENG.15), ADRL contains the length of the name (6), and VC_STAT represents the status of the requested service. Upon completion of a successful connection, a "virtual circuit identifier" (VCID) is returned that can be used for the subsequent transmission of data. Incoming calls for a particular port in a host are queued on a first come first served basis. Information concerning a call request at the head of a port queue can be obtained via

a system call, so that connections can be accepted, refused, cleared, etc. Calls are kept pending for 90 seconds, during which the requestors' status is that of "connection in progress." Other X.25 services are provided to users that allow for waiting on the completion of a network event, accepting or clearing a call, passing off a virtual circuit to another process in the same host, and obtaining status information about a particular circuit.

At a level above the PRIMENET primitives, PRIMOS supports a remote-login capability (RLOGIN) and a network file-access-method (FAM). The File Access Manager (FAM) is a PRIMOS subsystem that extends the functions of the PRIMOS file system to a network of hosts. Virtualization of the file system is accomplished by permanently assigning a port (255) to the local FAM process of each host, over which virtual circuits to neighboring FAMS are used to accomplish remote file operations on behalf of a user.

A FAM process in a host fields requests from local users for file operations on remote hosts, handles incoming file requests from remote hosts, and maintains status and update information concerning the current state of network connections and file system devices. When the PRIMOS supervisor decides that a particular user request is destined for a remote device, it queues the request for the local FAM process and suspends the user. FAM packages this request in a message and passes it off to the appropriate remote FAM, which performs the requested file operations on behalf of the user. The remote FAM process sends the original request and the requested data back to the local FAM, which copies the returned values into the user's address space and causes the user to be rescheduled. Because certain file primitives are guaranteed to be "atomic" operations, all file functions are performed to completion just as if they occurred locally, even if they require multiple messages or updating of local supervisor tables.

Since both local and remote operations on a particular file are handled through the file system of the host that owns the particular file, all of the normal file protection and other mechanisms, such as locking a particular record while writing, are automatically accomplished. Applications using remote data as well as local data run without any change.

In a similar fashion, the ability of a user to "remotely log-in," as if their terminal were physically attached to the host of their choice, is achieved by the operating system multiplexing all remote terminal traffic through port "0." When a user "logs in," they may designate a system to be attached to as:

```
LOGIN SMITH -ON ENG.15
```

At this point the local login server establishes a virtual circuit to the target host and requests the initiation of,

and connection to, a process in the remote host. From then on the local terminal buffers are effectively diverted to the input and output buffers of the remote process running on the selected node.

A proposal for an implementation of pipes [SCHE 78] was discussed as an alternative to virtual circuits. The pipe mechanism does allow multiple readers and multiple writers and thus, together with the X.25 PRIMENET, would facilitate most applications that demand IPC facilities incorporating multiple readers and writers.

In summary, the current PRIMOS interprocess communication capabilities allow local and remote process cooperation through X.25 flavored "virtual circuits", in addition to the semaphore primitives for local communication. These "point-to-point" mechanisms may not suffice for distributed process applications demanding N-process protocols; however the set of applications demanding such protocols at this time seem small.

7.6.3 Conclusions and Future Directions

As this report has illustrated, the process concept has become increasingly central, in recent years, to the design of computer systems both at the hardware and software levels. There are many reasons for this development, two important ones being: (1) the continuing decomposition of systems and applications problems into sets of cooperating parallel programs for greater modularity, functionality, flexibility, and maintainability; and (2) the increasing cheapness of processors and memory allowing the assignment of processes to processors in an economical way. As processes have become "cheaper" to create, maintain, and destroy, the flexibility, scope, power, and economy of interprocess communication mechanisms has become increasingly central to the effectiveness of multi-process systems.

A wide variety of mechanisms for interprocess communication have been surveyed in this report. Perhaps the major reason for such a variety comes from a desire to provide in one set of primitives: (1) flexible process synchronization tools, (2) data transfer mechanisms, and (3) communication control and error recovery. Some of the major issues involved in the design of interprocess communication mechanisms are briefly discussed below.

1. Process Naming: Many systems have inadequate facilities for identifying names of processes within the same host, let alone for processes residing on different hosts. Part of the problem stems from an inconsistent view of the relationship between the set of allowable names for files, devices, processes, users,

mailboxes, generic system services, and specific system services. Until this problem is settled the design of specific interprocess communication primitives cannot focus on the set of fundamental objects that they will be dealing with. This is a difficult issue, since it is here that many of the system security issues are also addressed.

2. Control Of Links Between Processes: Control of communication paths between processes fundamentally depends upon the nature of process relationships. If process relationships are tree structured, then the status of a child's communication with other processes might be monitored and controlled by the parent. On the other hand, if each process wants to maintain the concept of sovereignty then the basic challenge is how to provide the ability for cooperating processes to establish a monitor process that is capable of controlling the communication paths between them.

3. Control Of Data Flow Between Processes: The need for a flexible set of operations to control data-flow between processes is of major importance in the design of IPC mechanisms. This issue involves providing processes with the ability to: control multiple links, respond to out-of-band signals, receive/transmit/flush stream and message data types, and receive/transmit link capabilities. A number of additional capabilities might also be considered, such as allowing processes to define data-type-links that facilitate the passing and manipulation of complex data structures.

4. Synchronization Of Processes: Clearly, a major function of interprocess communication is to provide either explicit or implicit synchronization between processes. Early forms of interprocess communication depended only on the correct use of explicit synchronization primitives for sharing sections of main memory. In some systems, temporary files serve as synchronizing points between job steps (implicit), while in other systems processes synchronize and exchange data by signalling (explicit). Whether explicit or implicit synchronization primitives should be provided is still very much an open question.

With the advent of cheap communications and distributed systems these issues are becoming more important each day to both the manufacturers and users of computer systems. A workshop addressing IPC design is, therefore, scheduled to be held in Atlanta, Georgia, on the 20-22 of November, that will bring together a selected group of researchers in this subject area to address the five general topics listed below:

(1) Assess the present state-of-the-art for IPC mechanisms in distributed data processing systems.

(2) Identify the data available on the actual performance of various IPC policies and mechanisms.

(3) Assess the potential value of various IPC mechanisms satisfying the operational and performance requirements for highly distributed systems.

(4) Identify shortcomings in the present state-of-the-art and identify promising areas for future research and experiments on this subject.

(5) Identify possible standardization levels in IPC design.

Some of the issues the workshop is intending to examine in detail are: addressing issues, hardware support, transport mechanisms, flow control, out-of-band signalling, fault tolerance, security, synchronization, and performance and application programming impact. Prime Research is actively participating in this workshop which also has the support of both IEEE Computer Society and the three ACM Special Interest Groups, SIGOPS, SIGARCH and SIGCOMM.

In conclusion, there are far reaching ramifications to the demand for, and the development of, interprocess communication facilities and cheap processes. At the user level, a greatly enhanced system functionality and flexibility can be achieved, and at the operating system and hardware levels, the need to efficiently support this functionality is leading to new architectures and OS designs. As the section on PRIMOS in this report suggests, Prime is developing new IPC mechanisms for the enhancement of current systems and is attempting to incorporate some of the ideas developed in other systems. In addition, as new computer architectures are explored at Prime, the need to include hardware support for critical IPC functions is an area that requires study and understanding.

7.7 DATA COMMUNICATION SOFTWARE

DATA COMMUNICATION SOFTWARE

by

G. L. Chesson
Bell Laboratories

Introduction

Distributed computing environments are based upon, and wholly depend upon, data communications. Although there exists a sizable and growing hardware technology for data communication, software has not generally kept apace in recent years. Better software tools and techniques are needed in order to experiment with the new hardware devices that are available in the laboratory as well as to improve the capabilities for cooperation between our normally monolithic operating systems. These notes outline the direction and status of communication-oriented software research with the context of the 7th edition of the UNIX operating system.

Several software components are being experimented with in computer systems at Murray Hill, including a PDP-11/45, 11/70's, an Interdata 8/32, and LSI-11's. Some of the software is part of the UNIX kernel, or resident operating system, and the remainder consists of programs that utilize the new kernel facilities. The software components in the kernel include:

1) primitives for managing intermediate-sized contiguous areas of kernel data space,
2) a "packet driver" which can be used to impose framing, sequencing, checksumming, and retransmission procedures on a character device,
3) multiplexed and non-multiplexed interprocess communication channels.

The salient characteristics of these components are described in the next three sections. The organization of the higher-level codes which use these components will not be discussed here.

Space Management Primitives

The previously existing space-management procedures in the UNIX kernel were used to implement the terminal character lists and the disk buffer cache. Since the size of an allocation permitted by these routines is either one byte or 512 bytes, it is not surprising that an additional mechanism was needed for data communications. There are but two

primitives needed: one to allocate and one to release. The new primitives manage contiguous memory segments that are some multiple of 32 bytes in size up to a maximum of 512 bytes.

It was intended that the buffer management primitives be fast enough to be invoked from within interrupt routines. This means that recombination or garbage collection must also be capable of being done at interrupt time. These considerations lead to a strategy which employs a few judiciously chosen bit-map tricks in conjunction with the constant allocation sizes mentioned above.

The allocator may be called with a flag which directs whether it should sleep when space is not available or whether it should return a failure indication. This was built in because the allocator must not be allowed to sleep when called from an interrupt routine. However, it may be equally distressing to have it fail. Current practice involves building strict space bounds into interrupt processes that cannot live with allocation failures. This way space requirements are known in advance, and the allocator is used to dedicate a private buffer pool where it is needed.

Although the new space management primitives are useful for allocating "ordinary" I/O buffers, their real usefulness is in supporting the fifo queues needed for data rate balancing between readers and writers. Because of the address-space limitations of the PDP-11, memory is a critical resource, and it is not possible to devote as much space to data queues as many high-bandwidth applications require. As the software described below matures, it will become necessary to extend fifo mechansims to secondary storage or to non-kernel memory space. The methods used in the current primitives can, and probably will, be applied in these other circumstances.

Packet Driver

The packet driver consists of a group of routines similar in name and function to the parts that make up the typewriter control software; namely, there are open, close, read, write, ioctl, read interrupt, and write interrupt entries. A software switch, called the line-discipline switch, placed at the proper locations in a character device driver selects whether a call should be made to the standard system control routines, or to the corresponding entries in the packet driver or other line-discipline. This switch mechanism may be thought of as a bidirectional filtering process which may be selectively inserted between a device driver and a user program.

The packet driver is designed to operate character devices in a packet mode with the error checking and flow controls that are necessary for reliable data communication. The implementation is organized so that flow control functions are at a high level and are independent of framing and other details of link control. This means that device characteristics are transparent at the flow control level, allowing the code to be used in different contexts - e.g. with both bit-oriented and byte-oriented lines, or DMA and non-DMA devices. Also, implementations exist for the UNIX kernel, as a user-level subroutine package, and currently for one non-UNIX system. Emphasis has been placed on learning how to produce communication software that is operating system-independent as well as machine-independent. In practice this means that the packet driver implementations listed above consist of protocol routines which are common in all cases plus io and clock routines which are system dependent. Since protocol changes invariably affect only the common code, the logistics of making network-wide improvements or repairs simplify to updating a common file and reloading the appropriate system programs.

There exist numerous link control and flow control procedures, however they were judged not suitable for our uses for a variety of reasons. Some typical complaints are that flow control procedures are not really end-to-end, packet formats are complicated and verbose requiring a fair amount of real-time scanning, multiplexing is usually defined in immutable ways, and error control, framing, multiplexing, and flow control are usually mixed together instead of being separated where possible. These considerations led to the following:

1) flow control is based on a sliding "window" of sequence-numbered packets. The numbers are modulo-8, the maximum window size is 7, and the window sizes are controlled by the receivers. The retransmission strategy uses either "go-back-N" or selective single packet retransmission at the receiver's discretion.
2) packet sizes and window sizes are negotiated between two communicating packet drivers. The packet and window sizes in each direction need not be the same.
3) packets may range in size from 32 bytes to a maximum of 4096 as determined by the formula $32 * (2 ** k)$ where k is an integer, $0 \leq k \leq 7$.
4) all message headers are the same size, unlike X.25 and other similar protocols.
5) it is possible to multiplex the link at the packet level, or within packets, or both.

The software overhead of running the packet driver on 9600 baud lines is quite low. The implementation is efficient enough that data rates exceeding 50K baud have been demonstrated with this software using a a PDP-11/45 and non-DMA devices. As one would expect the overhead at higher data rates consumes the available cpu resources. For this reason the packet driver is looked upon as an algorithmic testbed and intermediate step toward improved computer peripheral hardware for communications.

Interprocess and Process-device Communication

Multiple independent asynchronous data streams and events comprise the greater part of the environment for data communication software. It has been observed many times that "blocking" I/O as implemented in the UNIX timesharing system does not provide direct methods for dealing with these entities, and there are sound architectural reasons why it does not. Nevertheless, a process that must read from more than one source sould not have to wait on idle data sources since input data will be missed or delayedd on lines that are actively producing data while the process is blocked. (It is assumed that polling techniques are unacceptable.) Also, the flow-control scheme used throughout the system causes writer to block if the total amount of written data exceeds a threshold. Such processes sleep until the corresponding reader (process or device) consumes some or all of the waiting data. A communications process typically must write to several processes and/or lines at once. It is somewhat inefficient to force such a process to block on a "slow" device or process when there are other readers that can be written to. Thus it would apppear that an operating system must provide techniques for dealing with asynchronism and blocking or flow-control problems as well as supply a useful means for establishing data bpaths between the various data sources and sinks. The mechanism outline below accomplishes these immediate goals in a simple and direct manner.

Two entities are defined: **channels** and **multiplexed channels**, also called **channel groups** or **groups** due to the similarity with existing notions in telephony. A channel consists of a pair of full-duplex communication paths. One pair is designated as the "data" path and the other as the "control" or "signaling" path. This architecture explicitly recognizes the need for what is usually called "out-of-band" signalling by dedicating a communication path for the purpose. In the implementation, each path has some amount of fifo or data queuing built into the transport mechanism. However, the actual data transport is dealt with indirectly: in order to avoid unnecessary copying of data from place to place within the system, the data is placed somewhere using a buffering mechanism, tokens indicating where the data can be found are passed from place to place. This decoupling of

the fifo and buffering functions from the data transport mechanism increases the efficiency of data movement and permits insertion of or tuning of buffering mechanisms in a transparent manner.

A channel can be thought of as a software null-modem: a null-modem consists of two plugs connected by some wires (fifo/buffering) so that data and signals transmitted at one plug are received at the other and vice versa. In the hardware world one may connect computers, computer terminals, and various other digital devices to one another via null-modems. In the software world one may attach processes, devices, other channels, and groups (see below) to the ends, or plugs, or a channel.

The multiplexed channel construct is a bundling mechanism ("Bundling" is a convenient term to describe a construct which fans-in, fans-out, or otherwise merges data. Examples include the PORT mechanism developed at RAND and elsewhere, certain aspects of the C.mmp system, and the UNIX timesharing system tee command.) which supplies both a multiplexing discipline for merging data from many channels and the inverse mechanism for sending data to the individual channels in a bundle, or group. A process can arrange to have various devices and processes "plugged-in" to the ends of channels and bundle all the opposite endings together in a multiplexed channel, or group. In this way a read command issued on the multiplexed channel will return any and all data (up to the requested limit) available from all the attached channels. This eliminates the blocking reader problem mentioned above.

It is possible to bundle the multiplexed stream associated with a group into another bundle, or super-bundle. This allows tree-structured data path networks to be built up. The maximum tree height and fan-in at each group is fixed at 4 and 16 respectively. By numbering the channels bundled into a group, a unique name for every possible tree node is defined as the pathname, or sequence of channel numbers encountered along a path from the "top," or root, of the tree to any particular node. The pathname or sequence numbering of a particular node is referred to as an **index**. (An index is represented as a 16-bit quantity interpreted as a sequence of 4-bit numbers.) All exchanges between the operating system and a process owning channels and groups are carried out using indices.

Multiplexed channels are created using the following C code:

```
fd = mpx ("name",mode);
```

which has the same effect as **creat** ("name",mode) in that "name" is placed in the file system. In addition reads and writes on "fd" are translated by the operating system into I/O operations on channels attached to the group.

I/O operations on a group are carried out via the standard UNIX timesharing system calls:

```
cc = read (fd,buf,count);

cc = write (fd,buf,count);
```

The contents of "buf" are a concatenation of some number of variable-length structures each having the form of an index followed by a byte count followed by the indicated number of data bytes. (Control channel data is distinguished from data channel data by an escape convention based on the message byte count. If the count indicates a zero-length message, then the actual byte count follows the zero and is in turn followed by control channel data.) The "buf" formats for reading and writing are identical, and in both cases "cc" indicates the number of bytes actually transferred out of a total request of "count" bytes. (Another form of **write** is provided in which "buf" consists of indices, byte counts, and pointers to the actual data. This format reduces the buffer filling overhead on output and improves the performance of certain programs.) On write operations if "cc" < "count" and the contents of "buf" were destined for more than one channel, then it is known that at least one channel fifo threshold was exceeded or some error condition was encountered. Precise information can be obtained by reading the group because the system immediately passes back status information. The index numbers of blocked channels and the number of data, one ,essage for each blocked data channel. When the previously written data is finally consumed, another control message is sent to the group owner indicating the readiness of a channel to accept data. These "blocking" and "unblocking" messages allow a process to continue to serve channels even though it temporarily cannot transmit to all its channels. A complementary function is provided whereby a process can enable or disable incoming data transfers on selected channels.

If "d" is a character device file descriptor obtained via a call resembling

```
d = open ("/dev/name",2);
```

then a channel can be created and the character device attached to the channel by executing

```
ch = join (d,xfd);
```

where "xfd" is the file descriptor for the multiplexed channel and "ch" is the new channel number.

Multiplexed channels may be joined or "bundled" to other channels by using the join primitive as outlined above and letting "d" be the file descriptor of a multiplexed channel. There are additional primitives for "unbundling" and manufacturing file descriptors that map into channels. Moreover the non-multiplexed file descriptors for channels may be used as the standard input or output for any UNIX program. (The multiplexed file dexcriptors provide direct access to the control paths of channels, but this not meaningful for the non-multiplexed case. Currently, **ioctl** commands on the non-multiplexed end of a channel are treated as messages on the control path of the channel.) The preceding discussion indicates how channels and devices can be attached to groups. It remains to indicate how channels are attached to processes. There are two techniques. One involves using the **extract** primitive, which is a converse of the **join** operation, to manufacture a file descriptor from a channel. Using standard techniques found, for example, in the UNIX shell one arranges fro an extracted file descriptor to be the standard input and output for a new process by executing UNIX **close** and **dup** calls usually followed by **fork/exec**. The second method has more interesting properties - if "name" is the name of a group, then

```
fd = open ("name",2);
```

triggers the following sequence of events:

1) the kernel notices that an open is being done on a group rather than an ordinary file.
2) if a new channel cannot be joined to the group or if the process which created the group is no longerrunning, the **open** fails immediately.
3) otherwise, a message is sent on the control channel of the group to the owner process stating that an **open** was requested. The effective UID of the opening process as well as the index, x, of a new channel are included in the message.
4) the owner process may respond with either **attach(x)** or **detach(x)** which respectively complete the job of hooking channel x between the group and returning file descriptor fd, or cause the open to fail.

An open sequence as described above results in the creation of a channel. The file descriptor returned to thr process executin ght open will be "plugged-in" to one end of the channel, and the other end of the channel will be attached to the group. A read on the file descriptor will be satisfied by writing on the channel through the group, and conversely for writing on the file descriptor and reading the group. An immediate application of this facility is in im-

plementing virtual terminals, or a "telnet server" as it is called by the Arpanet community. A process first establishes a group and arranges for one channel to be a data path to a similar process runing on another computer. If the remote process sends a message asking that an interactive environment be established, then the local process forks, opens its own group, and starts up the shell with the file descriptor returned from the open as the standard input and output. Meanwhile the original local process arranges to copy data from the newly created channel to the remote computer and vice versa. Of course there are certain niceties involving access permission, process groups, and other details which are not explained here, but they can all be handled neatly within the channel/group organization.

The method outlined above provides a form of "port" facility. Its main disadvantage is that one must know a port name. System or network-wide services would presumably have well-known names, but it is important to have a class of unbound names that the system can recognize. Interpretation of such names might require searching for a remote machine having a certain service facility or might require a simple translation of some sort. In order to accomplish this a mechanism has been established whereby a multiplexed channel may be designated as the unique interpreter for all such unbound port names. In the operating system any **open** requests on names containing "!" are treated as open requests on the special channel. One use of this mechanism is to treat "name1!name2" as a request for a file with name name2 on a machine designated by name1. Since strings of this form may be passed in to any program on the system, one may write

```
diff machine1!file1 machine2!file2
```

and expect the UNIX **diff** command to be run with input from machine1 and machine2.

For some applications the bandwidth that can be achieved by implementing data stream switching between channels in a user process, implying a copy operation from the kernel to the switch process and back to the kernel and then a final copy to the destination process or device, may be quite adequate. The primary example is the virtual terminal scheme outlined above. However this is not true for many other applications especially those involving file transfer or file access. For these cases a **connect** primitive is supplied which establishes a "short-circuit" connection in the kernel between a channel and file descriptor. That is, at the place in the operating system where data buffered in a channel would be copied to a user process as part of a read operation, the data is handled as though a write on the file descriptor had been done. The connect primitive specifies whether the symmetric short-circuit path is also meant to be established - that is, whether writes on the file descriptor should induce a direct copy to the agent reading the "other"

end of a channel. A **disconnect** operation is also provided to break open short circuits.

The semantics of carrying out a normal open call on a multiplexed channel name provide a useful range of interprocess communication capabilities. This is what one expects from a process communication system. However, by making slight adjustments to the name recognition algorithms in the system a wider class of file names can be "trapped" by the open routines in the kernel and passed as messages to a program for further interpretation. This comprises a very powerful mechanism for distributing system functions in interesting and useful ways: once a channel has been established via this name translation procedure, subsequent I/O on the channel by the process can be redirected to other computers or other process at will and without modification to the initiating program.

7.8 DISTRIBUTED IPC AND SIGNALLING

DISTRIBUTED INTERPROCESS COMMUNICATION AND SIGNALLING

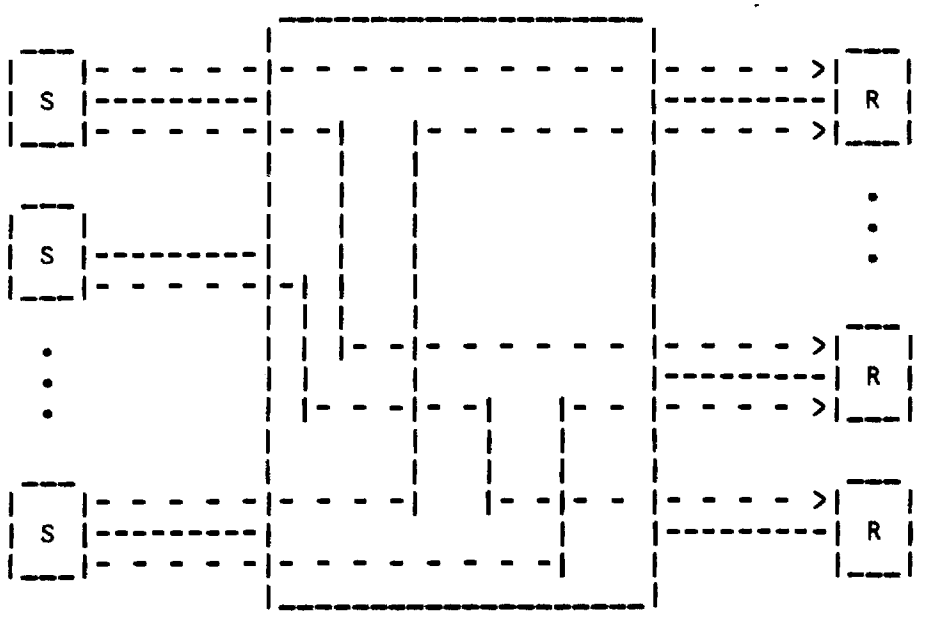
by

G. Le Lann
IRIA/SIRIUS

7.8.1 The General Context

Let us consider a system including several processors being linked together through an interconnection structure. We will distinguish between processors being accessed by external users who wish to initiate activities and processors which run these activities and may return results to some external users. Initiation of activities, execution control and transmission of data are accomplished through transmission of messages. In the following, we will refer to these processors respectively as senders and receivers of messages (see figure 1). We will not make any assumption regarding the size of these messages.

Figure 1 = A Schematic Representation of the System



senders **interconnection structure** **receivers**

- - - - - - - - > Flow of messages

Our assumptions will be:

- senders and receivers may be micro, mini or maxiprocessors,
- these processors may fail,
- the interconnection structure is any resilient hardware structure (using alternate routes in telecommunication networks, multiple busses/cables in multiprocessors/multicomputers, radio frequencies, etc.),
- errors, duplicates and losses are possible during the transmission of messages,
- message transit delays are variable,
- there is no privileged processor in charge of handling either communication or interprocessor cooperation.

We would like first to describe some of the problems we see to exist in such systems and, second, to present a solution.

7.8.2 The Problems

7.8.2.1 Multiple Sender/Single Receiver Systems

Let us consider a system as depicted in figure 1 but including only one receiver. We can identify two different problems:

i) for any sender, it may be necessary to maintain a strict sequencing of messages being sent to the receiver
ii) the various message flows converging at the receiver may have to be serviced by the receiver according to a particular discipline, which may be dynamically changed and not be known statically or guessed by the receiver.

Problem (i) is a problem of end-to-end signalling or single-path signalling (sps). Solutions to the sps problem are well known. The "window" technique is an example of such a solution.

Problem (ii) raises the issue of multiple-path signalling (mps) that is the problem of serializing incoming messages issued in parallel by different asynchronous sources. A mechanism is needed whereby senders may enforce distantly a particular serialization of messages at any time. For example, this is needed when two senders A and B wish to establish a particular ordering for initiating activities (e.g., A before B).

7.8.2.2 Multiple Sender/Multiple Receiver Systems

Let us now consider a system including several receivers. We will distinguish between two cases:

i) Fully redundant systems

Major motivations for running several identical receivers are to make the system able to survive receiver failures, to provide for a geographically dispersed but unique activity visible from various locations (receiver areas), or to relax constraints regarding system maintenance.

The serialization of incoming messages (either fortuitous or enforced) must be unique for all receivers. This is an mps problem.

ii) Partially redundant systems, partitioned systems

These systems include several receivers running activities which may be strictly identical for some of the receivers, as well as activities which are different for all receivers.

In addition to the motivations already mentioned, other reasons for considering such systems are to provide for various activities being run in parallel and to allow for a modular and dynamic growth of the system. In these systems, an activity being initiated by a sender may span several receivers. This raises the need for coordinating the various individual serialization processes over these receivers. Finally, according to user requests, the mapping between senders and receivers, i.e. the need to set and reset cooperation paths between senders and receivers will be constantly changing with time.

To summarize, we want to maintain a unique serialization of incoming messages for those receivers which act as "twins." In addition to this, we want to be able to achieve:

- For every receiver, a specific and local serialization of messages in step with the dynamically changing subset of senders it is cooperating with
- decentralized coordination between those receivers which have to serialize messages related to multi-receiver activities in order to avoid conflicts between such activities.

This is again an mps problem.

7.8.3 Looking for a Solution: Requirements

Potential advantages of distributed computing systems are numerous. However, it is not so simple to find a solution to a particular design problem which does not annihilate some of these advantages. A number of requirements which are considered to be of primary importance for a "distributed solution" to the mps problem are listed below.

7.8.3.1 Parallelism and Response Time

A solution should take full advantage of the parallel nature of the system; parallelism in processing as well as in communication may result in a good resource utilization ration. This has a non-negligible impact on system costs and response time.

7.8.3.2 Resiliency

A solution should survive failures. Actually, we need a more precise measurement of such a property which would express the number of simultaneous failures a solution may survive. This is the notion of resiliency.

7.8.3.3 Overhead

Costs of a solution may be low, monstrous, or acceptable. It is necessary to evaluate overheads as regards traffic (number and size of additional messages), processing (handling of additional messages) and storage (for "control" information).

7.8.3.4 Permanent Rejection

When conflicts occur (between "simultaneous" activities, for example), how does a solution lend itself naturally to avoid infinite waiting, without resorting to any exotic or ad-hoc mechanism?

7.8.3.5 Fairness

Again, when conflicts occur, a solution should not favor systematically the same processor(s).

7.8.3.6 Extensibility

If a solution may keep on working under dynamic system reduction (failures), then it is necessary to show how this solution matches the requirement of dynamic system extension. What this means is that it should be possible to reinsert or to add processors to the system without disrupting the functioning of the system.

7.8.3.7 Simplicity

When time has come to implement a system, problems of understanding, specifying, debugging and maintaining the software corresponding to a particular solution become preponderant. This last requirement may well be one to look at very carefully when considering to build a real system.

7.8.4 A Solution

We have seen that an mps mechanism is needed if one wishes communications between several senders and receivers to exhibit some specific properties. Obviously, signalling in a distributed system will be accomplished through the exchange of messages, i.e. signalling will rely on communication.

This apparently recursive problem requires some structuring. We will then assume that any convenient technique is used in the system for solving the sps problem.

On top of this "layer," we will build our mps mechanism.

7.8.4.1 A Virtual Ring Structure

Sending processors are given permanent identities. If n is the predicted maximum number of these processors, identities will be integers belonging to the interval [0, n - 1]. As a result, it is possible to view these processors as being sequencially located along a virtual ring. Each processor i has a well known predecessor and a well known successor, i - 1 and i + 1 in the absence of failure (the marks - and + stand for operations modulo n). There is no assumption made regarding the mapping of processor identities on physical addresses. In other words a virtual ring strructure does not assume any particular physical topology.

As processors are located on a virtual ring, it is only needed for each of them to know the identity of their respective predecessor (pred) and successor (suc).

A permanent and virtual communication path is established between adjacent processors. A message sent on such a path may travel over different physical links as provided by the interconnection structure. Specific techniques may keep the failure of a particular link transparent to processors. However, occurrence of one or several failures may preclude communication between adjacent processors. Detection of a communication path breakdown as well as detection of a processor failure can be achieved by using one of the following techniques.

7.8.4.1.1 Mutual Suspicion
Every processor sends regularly "life messages" to its successor on the ring. These messages should be acknowledged. If the successor fails to return acknowledgements for a given period of time, it is declared dead and its predecessor undertakes a ring reconfiguration. Actually, there is no difference between an abnormal behaviour of a successor and a breakdown of a communication path. In both cases, the successor should not be maintained on the ring.

Acknowledgement of life messages is bound to some internal checking procedure which, if successful, indicates that the processor is safe. In order to achieve correctness checking transitivity along the ring, it is necessary to bind the transmission of life messages to this checking procedure as well.

Consequently, a processor cannot be returning acknowledgements to its predecessor and fail in checking its successor.

7.8.4.1.2 Explicit Message Acknowledgement
It may be required for messages sent over a communication path to be acknowledged. A number of retransmissions are allowed before deciding that the communication path is broken. Numerous examples of protocols aimed at monitoring transmission on various transmission media can be found in the literature. They will not be detailed here. Also, it may happen that messages are not acknowledged because the successor has failed. As explained before, whatever the case, that successor should not be kept on the ring any longer.

Thus, every processor on the ring must be provided with a reconfiguration protocol to be used every time a failure leads to a ring breakdown. A simple example of such a protocol is given below.

7.8.4.2 Ring Reconfiguration

Let us consider a situation where processor i and processor i+2 are respectively predecessor and successor of processor i+1 when this processor fails or when the communication path between i and i+1 is broken. It is only necessary for processor i to send to i+2 a specific message, to be referred to as a reconfiguration message, meaning that from now on predecessor or processor i+2 is processor i. This message must be acknowledged by i+2. If an acknowledgement is not received by i after several attempts, i will send a reconfiguration message to i+3, thus excluding i+2 from the ring. The extreme situation is that of a ring including only one processor.

The decision of initiating a reconfiguration being taken exclusively by one processor for any particular failure, it is easy to infer that no incoherence can arise because of the exclusion of a processor from the ring. Because it is required for a reconfiguration message to be acknowledged, it is possible to devise some more elaborate scheme (for instance, utilizing passwords) to avoid the possibility of having a single faulty processor excluding all the others from the ring. An example of a protocol using passwords is given below.

7.8.4.3 The Extensibility Property

If processors are allowed either to fail or to leave, it should be possible to reinsert on the ring a processor which has been repaired or which decides that it is "on" again. Also, we want it possible to expand the system while the system is running. To this end, a three-party protocol is needed such that the ring is always correctly configurated. This protocol must survive failures itself and should entail as small a disturbance as possible. Let us assume that processor j has to be inserted on the ring.

To this end, j must send a specific message, called an "insert" message, containing its identity j to its potential successor (j+1, J+2, ...). Let us assume that k is on the ring. Processor k knows the identity of its current predecessor. Let us assume that pred [k] is processor i.

Upon receiving such a message, k checks that the following condition holds:

pred [k] < identity within insert message < k

(< is modulo n).

If it is so, k checks for an exchange of m life messages with j and then sends to i a message meaning that i should accept j as its new successor. This message contains a password X. Upon reception of this request, i checks for an exchange of m life messages with j. When this is completed, i sends to k a "switch" message containing the password X. This message is intended to avoid processors i and k being fooled by a malicious processor j and it is also used as a means to perform safely message transmission switching on the new path (i, j, k) as explained below.

Upon receiving the "switch" message, k acknowledges it and listens to j to detect the reception of a message containing code X.

Upon receiving this acknowledgement, i performs the update suc (i) := j; the first message to be sent to j is a message including code X. This message and other subsequent messages are passed on to k by j.

When receiving a message with code X, k updates pred [k] with value j and then stops listening to i.

There is no interruption of message transmission on the ring. If something goes wrong with j no disturbance is introduced on the existing ring. The message containing code X is a good vehicle to maintain a FIFO message transmission on the ring should this be required. There is no special provision made to guarantee that loss of messages does not occur between i and k just before or after reconfiguration of the ring performed by k. Loss of control messages is accepted on the ring and is harmless as will be shown later.

If transmission between i and j or between j and k turns out to be impossible, then a normal ring reconfiguration is undertaken.

7.8.4.4 The Control Token Mechanism

Cooperation between processors located on a virtual ring can be achieved by providing them with some control privilege. The solution suggested here is to have a particular message, called the control token, circulating on the ring. Only when holding the token should a processor be allowed to initiate some specific activity. Upon completion, the token is sent to the successor. Obviously, in the case the token is lost, it should be possible to regenerate it.

We begin by describing how the control token mechanism is made resilient. Then, we show how this mechanism can be used to solve the mps problem.

7.8.4.4.1 Resiliency

We assume that every processor owns a timer and that timer values being used by the various processors on the ring are not necessarily identical. Processors are allowed to read headers of messages circulating on the ring.

Transmission of a token between adjacent processors is monitored through a positive acknowledgement + retransmission protocol. The token carries with it an integer value, called the cycle number, which is incremented for every complete revolution on the ring. This incrementation is performed by processor x such that x > suc (x). At any time, this processor is unique. Also, the numbering cycle to be used should be chosen so that duplicate detection can be performed safely. This is possible if maximum "hardware" transit delays are known.

Timer values being used by processors correspond to the expected round-trip time with the successor on the ring. A timer is reset when the token has been acknowledged by the successor.

Each processor keeps a recording of the value (N) carried within the token during its last visit. Next real token to be received (not duplicates) must carry value N + 1. When the sender's timer awakes, transmission is tried again, up to a maximum number of attempts. Should this limit be reached, a ring reconfiguration is undertaken. The token is not lost.

If failure of a processor is noticed through the mutual suspicion protocol, then it may be the case that the token was held by this processor which failed. Detection of such a situation and regeneration of the token can be performed as follows.

Let h be the identity of the predecessor of that processor which has failed and i the identity of the successor. Processor h undertakes a ring reconfiguration. The reconfiguration message carries with it value N(h), last token value known in h. Upon reception of this message, processor i runs the following algorithm:

```
if (i > h and N(h) ≠ N(i)) or
   (i < h and N(h) = N(i)) then
create token N(I) := N(i) + 1;
```

With such an algorithm, it is possible to assert that a token is never lost and that, at any time, there is only one such token circulating on the ring (or zero for a finite and hopefully short period of time).

7.8.4.4.2 Distributed Signalling

A simple way to achieve a specific signalling sequence in a distributed system is to have the processors serializing themselves so that at any time, only one processor is "acting." This can be done very simply by using the control token as a vehicle to achieve mutual exclusion between these processors. However, the speed of this signalling technique is very much dependant on the time spent within the critical section. The problem is that very often, both the number and the nature of mutually exclusive actions are given beforehand and it may be very difficult to adjust the size of the critical section so that response time requirements are matched. Such a technique could slow down a system artifically.

Instead of this, it is suggested to uncouple completely the signalling mechanism and the execution of the critical section. As a result, mutually exclusive actions will be initiated in parallel. A proper sequencing can be built by assigning identifiers to them. The control token will be

used for the purpose of distributing sequencial identifiers within the system. These sequential identifiers will be referred to as tickets. Every message issued by a sender must be ticketed.

If we want receivers to service incoming messages according to a purely sequential ordering, then we need one ticket space per receiver category. In a fully redundant system, we have only one category of identical receivers. One ticket space is needed. In a partitioned or partially redundant system, we need one ticket space for each partition. Then, according to the system under consideration, the token will carry either a ticket value or an array of ticket values.

It has been shown how the birtual ring + token structure can survive failures. But ticket allocation must also be resilient. To this end, one may require that a processor should be either selecting tickets or using them but not both. What this means is that those tickets which are selected by a processor should not be used until the token has been acknowledged by the successor. As a consequence, should a failure occur in the midst of ticket selection, the correct ticket value or array of ticket values can be regenerated with the token exactly like this is done for the cycle number (see 7.8.4.4.1). Another issue is that of failures interrupting processing at random. In particular, what should be done with those messages which have been issued by a processor which failed later on? Another problem is what to do with tickets not being used because they were held by a processor which died.

Actually, the whole issue would require a complete discussion which is out of the scope of this paper.

7.8.4.4.2.1 Fortuitous Serialization
i) Signalling within fully redundant systems

The broadcasting of a ticketed message to all receivers may be done by the sender (parallel broadcasting). The usual problem with this technique is that the sender may fail while issuing messages. However, because tickets must be sequential, it is simple for a receiver to detect such an unsafe situation. A copy of the missing message may be obtained from another receiver.

Another approach to broadcasting consists in organizing receivers along a virtual ring. This ring is intended to be a resilient vehicle for message broadcasting. Only one copy of a message must be created by the sender which hands it over to one of the receivers. This receiver is then in charge of initiating the revolution of the message on the ring.

ii) Signalling within partioned or partially redundant systems

The transmission of ticketed messages is done by the sender which selects tickets from the ticket spaces correspodning to the relevant partitions.

7.8.4.4.2.2 Enforced Serialization
Let us assume that two senders A and B want the receivers to process messages issued by A first and then messages issued by B. This is done very simply by having A sending to B a "go-ahead" message after A has ticketed its last message. There is no need for serializing the related activities outside the system (for example, A waits until its activity is over and then sends a message to B).

Senders A and B may also wish to initiate co-related activities which, in a partitioned system, share at least one partition. These activities are such that the message from A should be serviced before the message from B and also the message from B should not be processed if the activity initiated by A could not be completed.

The following protocol may be suggested. In the "go-ahead" message, A stores the value of the ticket used for its message. It is then only needed to provide for a flag and a field in message headers to be used as follows. When a message M is received with the flag set, the receiver should read the ticket value stored in the field. If the corresponding activity could not be completed, message M is discarded and the sender is told that its activity was not initiated.

7.8.4.4.2.3 Performance Considerations
We want the signalling mechanism not to put any artificial limitation upon the system performances. Consequently, this mechanism should not be dependent upon the rotating time period of the token on the virtual ring. Senders should be able to ticket and to issue messages at any time. This means that senders should be allowed to select tickets not only for pending messages but also for "future" messages, i.e. messages to be created and issued between two successive visits of the token.

Let p be a sender. At token visit #i, let C.i(p) be the exact number of messages which are pending when the control token is received, f.i(p) be the predicted number of future messages, T.i(p) be the current value of the relevant ticket space upon reception of the token and T$i(p) be the new ticket value when the token is sent on the ring.

Sender p is allowed to acquire C.i(p) + f.i(p) consecutive tickets, starting from T.i(p). Ideally, during token revolution #i+1, P needs exactly f.i(p) tickets. Clearly, predictions are only predictions. Furthermore, the token circulating speed is variable. Hence, it is necessary to

consider two possible situations:

- p runs short of tickets: it has to wait for reception of the token.
- some tickets are not used when the token is back: let u.i(p) be the number of unused tickets. Because of the mutual independence principle, these tickets should be used up immediately. For that purpose, we provide for the utilization of a no-operation code. Exactly u.i(p) "fake" messages carrying a NOP code will be isued by p.

When needed, and as long as tickets are available, new messages are issued.

Probably, this will achieve a good parallelism between senders but it is not clear whether or not this will result in a good average response time. Response time for a given sender is dependent on how fast predecessors use up their tickets.

Should such an interference be judged unacceptable, another solution is needed.

What we would like to build is a mechanism whereby current pending messages and future messages are distinguishable, so that current pending messages for any sender receive tickets "smaller" than those given to future messages.

Let us make it clear that we do not attempt to build a perfect chronological ordering of messages. We only try to achieve some system-wide statistical FIFO service so that the average response time for every sender can be kept below a reasonable value.

The way this can be done is rather simple. It is only needed to maintain two ticket values T and θ, in the token instead of one (or two arrays instead of one). T as above, is to be used for ticketing current pending messages and θ for ticketing future messages. By the time the token is back in p, only one of the three following conditions can hold:

- u.i(p) = C.i(p) = 0 (ideal case)
- C.i(p) messages are waiting because p is lacking tickets, u.i(p) = 0, C.i(p) > 0 (under-estimation)
- u.i(p) tickets are still available, u.i(p) > 0, C.i(p) = 0 (over-estimation).

A requirement regarding the ticketing function is that the two sets of numbers being used to assign a value to T and θ should not be overlapping.

Two numbering cycles N(T) and N(θ) should be chosen so that tickets lifetime is conveniend (see computations below).

As T-ticketed messages and θ-ticketed messages will be received interleaved by receivers, it is necessary to provide for some means whereby receivers are able to decide when to stop processing T-ticketed messages and start processing θ-ticketed messages as well as the reverse.

Such a "switching" should correspond to a complete revolution of the token on the virtual ring. We need a sender to flag the corresponding T and θ ticket values.

That sender could be x such that successor (x) < x. Due to the properties of the virtual ring, this processor is unique and always exists.

The algorithm to be followed by sender p upon reception of the token is described below (+ and - operations are modulo N(T) or N(θ)).

```
BEGIN
   IF suc (p) < p and C.i(p) = 0 THEN
      BEGIN
         C.i(p) := 1;
         creat Fake message
      END;
   IF C.i(p) > 0 THEN T*.i(p) := T.i(p) + C.i(p)
(acquisition of tickets #T.i(p), ..., #T.i(p) + C.i(p) - 1)
   ELSE IF u.i(p) > 0 THEN
      send u.i(p) Fake messages (ticketed with the u.i(p)
         highest θ-tickets obtained during the last
         token visit);
   assign a value to f.i(p);
   IF suc (p) < p AND f.i(p) = 0 THEN
      BEGIN
         f.i(p) := 1;
         create Fake message
      END;
   θ*.i(p) := θ.i(p) + f.i(p)
(acquisition of tickets #θ.i(p), ..., #θ.i(p) + f.i(p) - 1);
   IF suc (p) < p THEN Flag messages carrying tickets
      #T.i(p) + C.i(p) - 1 and #θ.i(p) + f.i(p) - 1;
END
```

The algorithm to be followed by a receiver is given below.

Notations:

X stands for either state T ("current") or
state θ ("future");

X~ = T if (X=θ),
= θ if (X=T);

t(X) is a local variable containing the ticket value of the last processed message, i.e. t(T) or t(θ).

```
WHEN IN STATE X DO
LOOP: Scan for, or wait for reception of message
   X-ticketed t(X)+1;
CASE1 (X-ticket t' > t(x)+1 is received):
                    mod
   Record request;
CASE2 (X~-ticket is received):
   Record request;
CASE3 (X-ticket t(x)+1 is present or received):
   BEGIN initiate processing;
      IF message t(X)+1 is flagged
      THEN
         switch to state X~
      ELSE
         t(X) := t(X)+1
   END
CASE4 (timeout):
   Marks itself out of synchronization and initiate a
      recovery procedure.
```

A simple way to provide for two separate numbering schemes of equal length is to use one bit to distinguish between T-tickets and θtickets. However, one should mention that, if predictions are not too inaccurate, θ-tickets are to be used up more rapidly than T-tickets. Then an equal share of the ticket number space may not be the best solution.

We will discuss only briefly the issue of fairness in estimating f.i(p). We consider two cases:

- senders are processors (maxis, minis, micros) cooperating within a distributed computing system to be viewed as a unique system by users. Algorithms to be followed by senders are designed by system builders who are responsible for choosing convenient values for f.i(p).
- senders are computers connected on a computer network. Over-estimation is costly to senders because (i) processing wasted in handling NOP

messages cannot be used to process useful messages (throughput is lower), (ii) a sender is billed for messages carrying NOP code and for the corresponding processing in the distant computer.

Because of the "pipe-line" nature of this mechanism, there will be no interruption of message transmission. What this means is that receivers may be kept as busy as desired. If used cleverly, the signalling mechanism using anticipation can achieve any desired throughput.

Tickets Lifetime

For 16 bit tickets, values are re-used after 65 seconds if ticketed messages are issued every millisecond for the whole system, after 18 hours and 12 minutes if ticketed messages are issued ever second.

For 32 bit tickets, lifetime is much longer. Values are re-used respectively after 1 hour and 12 minutes, 119 hours or 136 years when ticketed messages are issued every microsecond, 100 microseconds or second in the whole system.

7.8.5 Conclusion

In this paper, a solution to the problem of multiple-path signalling in distributed computing systems has been described. This solution is based on the utilization of a particular control structure which can achieve a distributed and resilient generation of sequential identifiers. In addition to solving the mps problem, this solution can be used in distributed systems which should be resilient and where unique names need to be generated dynamically. Also, a side-effect of this solution is to allow for a safe detection of duplicate messages at a high level in the system.

SECTION 8

SUMMARY AND FUTURE DIRECTIONS

8.1 GENERAL OBSERVATIONS AND CONCLUSIONS

The idea of a process has not been fully absorbed by programming languages or by modern hardware. Consequently, the concept of an abstraction of a process and its support is left to the realm of operating systems (which sit between the language and the hardware), resulting in little or no standardiztion of a "process" (especially when compared to the level of standardization enjoyed by other features or aspects of higher level languages and hardware). Nevertheless, as this report has illustrated, the process concept is becoming central to the design of computer systems both at the hardware and software levels. There are many reasons for this development, probably the two most important ones being: (1) the decomposition of systems and applications problems into sets of cooperating parallel processes for greater modularity, functionality, flexibility, and maintainability; and (2) the increasing cheapness of processors and memory allowing the assignment of processes to processors in an economical way.

As processes have become "cheaper" to create, maintain, and destroy, the flexibility, scope, power, and economy of interprocess communication (IPC) mechanisms has become an important key to the effectiveness of multi-process systems in general, and highly distributed systems in particular. However, there currently exists a wide variety of mechanisms for interprocess communication, resulting in what one researcher [SALT 79] has termed the "IPC Jungle". Perhaps the major reason for such a variety comes from a desire to provide in one set of primitives all of the following capabilities:

1) Flexible process and/or data synchronization tools,
2) Data transfer mechanisms, and
3) Communication control and error recovery mechanisms.

Surprising to some researchers at the workshop was the lack of attention paid to security, fault tolerance, and error recovery; however, this may be taken as an indication of the general state of affairs of a young technology. In such cases, attention is usually first focused on achieving a certain level of functionality before much effort is devoted to engineering those features that make the technology robust enough to be put into wide-spread use.

Finally, dissemination of information about IPC techniques and options with respect to both implementation and performance has been extremely poor in the past, and there do not appear to be any immediate advances being made on this aspect of the problem.

8.2 WORKSHOP SUMMARY

Below is a summary of the major focus areas of the workshop and their conclusions.

8.2.1 Addressing, Naming, and Security

Many systems have inadequate facilities for identifying names of processes within the same host, let alone for processes residing on different hosts. Many existing systems almost totally sidestep the naming issue. Part of the problem stems from an inconsistent view of the relationship between the set of allowable names for files, devices, processes, users, mailboxes, generic system services, and specific system services. As Livesy pointed out during the workshop, the concept of the size of the naming universe (of which the system forms a part) is implicit in the system at a very deep level. One is forced to choose between "add-on"naming techniques such as:

/net/A/resource

which are not location independent on the one hand, and a more or less complete redesign of the naming architecture on the other hand. UNIX is an example of a system that makes assumptions about the size of the universe. Until this problem is settled, the design of specific interprocess communication primitives cannot focus on the set of fundamental objects that must be dealt with. This is a difficult issue, since it is here that many of the system security issues must also be addressed.

8.2.2 Interprocess Synchronization

Clearly, a major function of interprocess communication is to provide either explicit or implicit synchronization between processes and/or access to shared data. Early forms of interprocess communication depended only on the correct use of explicit synchronization primitives for sharing objects (usually sections of main memory). In some systems, temporary files served as synchronizing pointes between job steps (implicit), while in other systems, processes explicitly exchange data by signaling. Whether synchronization primitives should be explicit or implicit is still very much an open question.

It is also becoming clear to some of the researchers in the field that error recovery may be integral to the question of synchronization. Visibility of the state of a computational process is at the heart of the synchronization and error recovery issues. Concern over the "atomicity" of an operation is becoming more of a focal point for distributed systems as the dimensions of time and space for com-

putational operations begin to change by orders of magnitude. This concern is reflected in the recent literature concerning synchronization in distributed systems (see the 1978-79 references), and in some of the recent theoretical work. However, their effectiveness using current technology is largely unknown until prototype implementations appear.

8.2.3 Interprocess Mechanisms

At least ten currently used IPC mechanisms were identified along with some estimate of their support of certain qualities deemed desirable by the workshop attendees. There was more agreement on the set of desirable qualities than there was on which mechanisms fulfilled those qualities. It was also obvious that none of the present mechanisms did everything that everybody hoped for, which should tell us that we have yet to obtain maturity of abstraction (in the sense that the abstraction of a subroutine is well understood) for a general IPC mechanism. For these reasons, it seems reasonable to keep exploring new mechanisms while we also continue to build real-world systems with the best techniques we have heard about.

In addition it appears important to devote some additional work to selecting the factors to be utilized in assessing trade-offs between provability versus convenience of implementation and use. Many of the mechanisms discussed at the workshop present enormous obstacles to rigorous proof.

8.2.4 Theoretical Work

Distributed systems present new theoretical challenges to researchers, largely because the specification of a distributed computation involves time and space boundaries that are difficult to define, and may be constantly changing. Variability in speeds and state definition may even make a "system" inherently non-deterministic. Such difficulties throw much of the previous work in proogram specification and correctness into disarray when applied to distributed systems. There is little agreement whether to approach the problem using "state-free" or "state-based" descriptions, or whether to grapple with atomic or non-atomic actions, or even what are relevant measures of "time" and "space". Once again, this seems to reflect the immaturity of the whole field of distributed systems.

8.3 CONCLUSIONS AND RETROSPECT

Lastly, we should be honest as to how well we achieved our original goals. Each goal is repeated here with a short comment as to our view of the level of success we enjoyed and the reasons for it.

1) Assess the present state-of-the-art for IPC mechanisms in distributed data processing systems.

   *** Successful. A reading of many of the enclosed working papers and the references should adequately reflect the present state-of-the-art.

2) Identify the data available on the actual performance of various IPC policies and mechanisms.

   * Unsuccessful. An attempt was made, however lack of agreement on appropriate measures (see mechanisms) has probably prevented any great data base being built up.

3) Assess the potential value of various IPC mechanisms in satisfying the operational and performance requirements for highly distributed systems.

   ** Moderately successful. Many of the advantages and disadvantages of the functional aspects of current mechanisms in use were examined, although, obviously, more thorough operational and performance assessments must await more "distributed" implementations.

4) Identify shortcomings in the present state-of-the-art and identify promising areas for further research and experiments on this subject.

   *** Successful. A reading of the report reflects many of the shortcomings of current techniques. Promising areas for further research were not specifically addressed in all areas; however, they are indirectly identified by many of the authors.

5) Identify possible standardization levels in IPC design.

   * Unsuccessful. The plethora of available abstractions and the notable lack of any single outstanding set useful for distributed

applications reflect the immaturity of the field and possible premature standardization.

SECTION 9

SELECTED READINGS AND REFERENCES

9.1 SELECTED READINGS

On Process Models and Structures

[HORN 73]
[DIJK 68a]
[HOAR 78]

On Addressing and Naming

[SALT 78]
[SHOC 78]

On Theoretical Considerations

[MILN 77]
[ZAV 76]

On Process Synchronization

[DIJK 68b]
[HOAR 74]
[HABE 72]

On Message Based Operating Systems

[BRIN 69]
[BRIN 70]
[BALL 76]
[LYCK 78]
[NELS 78]
[FARB 73]

On Local Networks

[CLAR 78]
[METC 76]
[GORD 79]

On Ports, Pipes and Virtual Circuits

[WALD 72]
[THOM 74]
[CCIT 78]

On the Early Treatment of Processes and IPC in Operating Systems

[DALE 68]
[SALT 66]
[DIJK 71]
[IBM 71]

On IPC Protocols

[PARD 79]
[DESJ 78]

9.2 LIST OF REFERENCES

[ABEL 78] Harold Abelson, "Lower Bounds on Information Transfer in Distributed Computations," Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science, October 16-18, 1978, pp 151-158.

[ALSB 76] P. A. Alsberg, G. C. Belford, and S. R. Brunch, "Synchronization and Deadlock," Center for Advanced Computation, Doc. No. 185, University of Illinois, March 1976.

[BACH 78] Charles W. Bachman, "Provisional Model of Open System Architecture," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 29-31, 1978.

[BADA 78] D. Z. Badal and G. J. Popek, "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 29-31, 1978.

[BALL 76] J. E. Ball, J. Feldman, J. R. Low, R. Rashid, and P. Rovner, "RIG, Rochester's Intelligent Gateway: System Overview," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, December 1976, pp. 321-328.

[BART 77] J. F. Bartlett, "A 'NonStop' Operating System," Proceedings of the Hawaii International Conference of System Sciences, January 1978.

[BASK 77] F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in DEMOS," Proceedings of the Sixth Symposium on Operating System Principles, 6-18 Nov 1977. Reprinted in Operating Systems Review, vol. 11, no. 5, November 1977.

[BOBR 72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX - A Paged Time Sharing System for the PDP-10," Communications of the ACM, Volume 15, Number 3, March 1972.

[BRIN 69] Per Brinch Hansen, "RC 4000 Software: Multiprogramming System," Regnecentralen, Copenhagen, Denmark, April 1969.

[BRIN 70] Per Brinch Hansen, "The Nucleus of a Multiprogramming System," Communications of the ACM, vol. 13, no. 4, April 1970, pp. 238-50.

[BRIN 73] Per Brinch Hansen, Operating Systems Principles, Prentice-Hall, 1973.

[BRIN 76] Per Brinch Hansen, "The SOLO Operating System," Software Practice and Experience, vol. 6, no. 2, April-June 1976, pp. 141-206.

[BRIN 77] Per Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall, 1977.

[BURN 78] J. E. Burns, M. J. Fischer, P. Jackson, N. A. Lynch, and G. L. Peterson, "Shared Data Requirements for Implementation of Mutual Exclusion Using a Test-and-Set Primitive," Proceedings of the 1978 International Conference on Parallel Processing, August 22-25, 1978, pp 79-87.

[CCIT 78] CCITT, Provisional Recommendations X.3, X.25, X.28 and X.29 on Packet Switched Data Transmission Services, Geneva, 1978.

[CLAR 78] D. C. Clark, K. T. Pogran, and D. P. Reed, "An Introduction to Local Area Networks", Proceedings of the IEEE, vol. 66, no. 11, November 1978, pp. 1497-1517.

[DALE 68] R. C. Daley and J.B. Dennis., "Virtual Memory, Processes, and Shaping in Multics"., Communications of the ACM, vol. 11, no. 5, pp. 306-12, May 1968.

[DEC 77] VAX11 Software Handbook, Digital Equipment Corporation 1977.

[DESJ 78] Richard desJardins and George White, "ANSI Reference Model for Distributed Systems," Proceedings of COMPCON 1978, Washington, D.C., September 1978, pp. 144-149.

[DIJK 68a] E. W. Dijkstra, "The Structure of the 'THE' - Multiprogramming System," Communications of the ACM, vol. 11, no. 5, May 1968, pp. 341-346.

[DIJK 68b] E. W. Dijkstra, "Cooperating Sequential Processes," in Programming Languages, (Editor: F. Genuys), Academic Press, New York, 1968.

[DIJK 71] E. W. Dijkstra, "Hierarchal Ordering of Sequential Processes," Acta Informatica vol. 1, no. 2, 1971, pp. 115-38.

[DOWS 78] M. Dowson, "The DEMOS Multiple Processor Technical Summary," National Physical Laboratory Technical Report, NPL Report 101, April, 1978, Teddington, Middlesex TWII OLW, UK.

[ELLI 77] Clarence A. Ellis, "A Robust Algorithm for Updating Duplicate Databases," Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 25-27, 1977.

[ESWA 76] 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, pp. 624-633.

[FARB 73] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, C. Larson, C. Loomis, and L. A. Rowe, "The Distributed Computing System," Digest of Papers from COMPCON 73, San Francisco, California, 27 February - 1 March 1973, pp. 31-34.

[GARC 78] Hector Garcia-Molina, "Performance Comparison of Two Update Algorithms for Distributed Databases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 29-31, 1978.

[GORD 79] R. L. Gordon, "Ringnet: A Packet Switched Local Network with Decentralized Control," 4th Conference on Local Computer Networks, Minneapolis, Minn., October 1979, pp. 13-19.

[GRAH 72] G. S. Graham and P. J. Denning, "Protection -- Principles and Practice," AFIPS Conference Proceedings, 1972 SJCC, pp. 417-429.

[GRAP 76] Enrique Grapa, and Geneva G. Belford, "Techniques for Update Synchronization in Distributed Data Bases," unpublished paper, 1976.

[HABE 72] A. N. Habermann, "Synchronization of Communicating Processes," Communications of the ACM, vol. 15, no. 3, March 1972, pp. 171-76.

[HAMI nd] J. Hamilton, "The Functional Specification of the WEB Kernel," Digital Equipment Corporation, Coproate Research Group, ML3-2/E41, no date.

[HOAR 74] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," Communications of the ACM, vol. 17, no. 5, October 1974, pp. 549-557.

[HOAR 78] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, vol. 21, no. 8, August 1978, pp. 666-677.

[HOLT 78] R. C. Holt, G. S. Graham, E. D. Lazowshka, and M. A. Scott, "Announcing Concurrent SP/K," Operating System Review, vol. 12, no. 2, April 1978.

[HOLT 78b] R. C. Holt, et al, Structured Concurrent

Programming with Operating Systems Applications, Addison-Wesley Series in Computer Science, 1978.

[HORN 73] J. J. Horning, and B. Randall., "Process Structuring," ACM Computing Surveys, vol. 5, no. 1, May 1973, pp. 5-30.

[IBM 71] IBM System/360 Operating System Supervisor Services, IBM Systems Reference Library, Order Number GC28-6646-4, 1971.

[IPC 75] ACM SIGCOMM/SIGOPS WORKSHOP, ACM SIGOPS Review, MARCH 1975.

[JOHN 75] P. R. Johnson and R. H. Thomas, "The Maintenance of Duplicate Databases," RFC No. 677, NIC No. 31507, January 1975, ARPA Network Information Center, SRI-Augmentation Research Center, Menlo Park, CA 94025.

[JONE 77] A. K. Jones, R. J. Chansler, I. Durham, P. Feiler, and K. Schwans, "Software Management of Cm* - A Distributed Multiprocessor," AFIPS Conference Proceedings, Volume 46, 1977 NCC.

[LAMP 76] L. Lamport, "Towards a Theory of Correctness for Multi-user Data Bases," Mass. Computer Associates, Inc., CA-7610-0711, October 7, 1976.

[LAMP 77] L. Lamport, "On Concurrent Reading and Writing," Communications of the ACM, vol. 20, no. 11, November 1977, pp. 806-811.

[LAMP 71] B. W. Lampson, "Protection," Proceedings of the Fifth Annual Conference on Information Sciences and Systems, Department of Electrical Engineering, Princeton University, March 1971 pp. 437-443.

[LAMP 73] B. W. Lampson, "A Note on the Confinement Problem," Communications of the ACM, vol. 16, no. 5, October 1973, pp. 613-615.

[LAUE 79] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," Operating Systems Review, vol. 13, no. 2, April 1979.

[LIVE 78a] N. J. Livesey and E. G. Manning, "Protection in a Transaction Processing System," Proceedings of the 7th Texas Conference on Computing Systems, October 1978.

[LIVE 78b] N. J. Livesey and E. G. Manning, "What Mininet Taught Us About Programming Style," Proceedings of COMPSAC 78, Chicago, Illinois, November 1978, pp. 692-697.

[LYCK 78] H. Lycklama and D. L. Bayer, "The MERT Operating System," The BELL System Technical Journal, vol. 57, no. 6, Part 2, July-August 1978, pp. 2049-86.

[MANN 77] E. G. Manning and R. W. Peebles, "A Homogenous Network for Data Sharing: Communications," Computer Networks, Vol. 1, No. 4, 1977, pp 211-224.

[METC 76] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Communications of the ACM, vol. 19, no. 7, July 1976, pp. 395-404.

[MILN 77] G. Milne and R. Milner, "Concurrent Processes and their Syntax," University of Edinburgh, Department of Computer Science Report CSR-2-77, May 1977.

[NELS 78] D. L. Nelson and R. L. Gordon, "Computer Cells - A Network Architecture for Data Flow Computing," Proceedings of COMPCON 78, Washington, D.C., September 1978, pp. 296-301.

[NSW 76] NSW Protocol Committee, "MSG: The Interprocess Communication Facility for the National Software Works," BBN Report No. 3483, Massachusetts Computer Associates Document No. CADD-7612-2411, December 1976.

[ORGA 72] E. I. Organick, The MULTICS System: An Examination of Its Structure, MIT Press, 1972.

[PARD 79] R. Pardo and M. T. Liu, "Multi-Destination Protocols for Distributed Systems," Proceedings of the 1979 Computer Network Symposium, Gaithersburg, Md., December 1979.

[PEEB 78] Richard Peebles and Eric Manning, "System Architecture for Distributed Data Management," Computer, vol. 11, no. 1, January 1978, pp. 40-47.

[PETE 77] Gary L. Peterson and Michael J. Fischer, "Economical Solutions to the Critical Section Problem in a Distributed System," Proceedings of the 1977 Ninth Annual Symposium on Theory of Computing, May 2-4, 1977, pp 91-97.

[POWE 77] M. L. Powell, "The Demos File System," "Task Communication in DEMOS," Proceedings of the Sixth Symposium on Operating System Principles, 16-18 Nov 1977. Reprinted in Operating Systems Review, vol. 11, no. 5, Nov. 1977.

[REED 77] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers," Operating Systems Review, vol. 11, no. 5, pp. 91-92.

[REED 78] D. P. Reed, "Naming and Synchronization in a Decentralized Computer System," MIT LCS Report MIT/LCS/TR-205, September 1978.

[RITC 74] D. M. Ritchie and K. L. Thompson, "The UNIX Timesharing System," Communications of the ACM, July 1974.

[RITC 78] D. M. Ritchie,"A Retrospective on the UNIX Timesharing System," The Bell System Technical Journal, vol. 57, no. 6, part 2, July-August 1978.

[RIVE 76] R. L. Rivest and V. R. Pratt, "The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report," Proceedings of the Seventeenth Annual Symposium on Foundations of Computer Science, 1976, pp 1-8.

[ROTH 77] J. B. Rothnie and N. Goodman, "A Survey of Research and Development in Distributed Database Management," Proceedings of 3rd International Conference on Very Large Data Bases, Tokyo, Japan, October 1977.

[SALT 66] J. H. Saltzer, "Traffic Control in a Multiplexed Computer System," Project MAC Technical Report MAC-TR-30 (Thesis), Massachusetts Institute of Technology, July 1966.

[SALT 78] J. H. Saltzer, "Naming and Binding of Objects," in Operating Systems - An Advanced Course, R. Bayer, R. M. Graham, and G. Seegmuller (eds.), Berlin, Springer-Verlag, 1978, pp. 99-208.

[SALT 79] J. H. Saltzer, Comments at the "7th Symposium on Operating Systems Principles," November, 1979, concerning distributed systems.

[SCHE 78] L. Scheffler, "Pipes - Interprocess Communication for PRIMOS and PRIMENET," (PE-T in final preparation).

[SHOC 78] John F. Shoch, "Inter-Network Naming, Addressing, and Routing," Proceedings of COMPCON 78, Washington, D.C., September 1978, pp. 72-79.

[STON 78] Michael Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 29-31, 1978.

[SUNS 76] Carl A. Sunshine, "Survey of Communication Protocol Verification Techniques," Proceedings of the Symposium on Computer Networks: Trends and Application, Gaithersburg, MD, November 17, 1976.

[THOM 77] Robert H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Data Bases," Bolt Beranek and Newman, Inc., BBN Report No. 3733, December 1977.

[THOM 78] Robert H. Thomas, Richard E. Schantz, and Harry C. Forsdick, "Network Operating Systems," Bolt Beranek and Newman, Inc., BBN Report No. 3796, March 1978.

[THOM 74] K. T. Thompson and D. M. Ritchie, "The UNIX Time-sharing System," Communications of the ACM, vol. 17, no. 7, July 1974, pp. 365-375.

[WALD 72] D.C. Walden, "A System for Interprocess Communication in a Resource Sharing Computer Network," Communications of the ACM, vol. 15, no. 4, April 1972.

[WILK 79] Maurice V. Wilkes and D. J. Wheeler, "The Cambridge Digital Communication Ring," Proceedings of the Local Area Communication Networks Symposium, Mitre Corporation and National Bureau of Standards, Boston, May 1979.

[WULF 74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernal of a Multiprocessor Operating System," Communications of the ACM, Volume 17, Number 6, June 1974.

[YOUN 79] R. Young and V. Wallentine, "The NADEX Core Operating System Services," Kansas State University Department of Computer Science Technical Report, no. CS 79-11, November, 1979.

[ZAVE 76] P. Zave, "On the Formal Definition of Processes," Proceedings of the Conference on Parallel Processing, Wayne State University, IEEE Computer Society, 1976.