

CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/0 9/2  
DECENTRALIZED RESOURCE MANAGEMENT IN DISTRIBUTED COMPUTER SYSTEMS--ETC(U)  
FEB 82 H L APPLEWHITE, R GARG, E D JENSEN F30602-78-C-0099

**RADC-TR-81-203**

ML

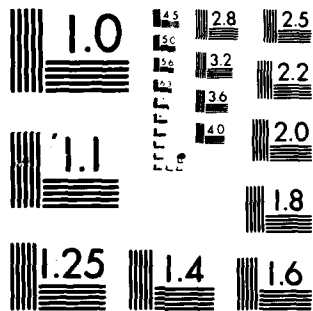
176  
 2000

[illegible]

END  
DATE  
FILMED  
5 -82  
DTIC

5 -82

DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

AD A113255

**RADC-TR-81-203**  
Final Technical Report  
February 1982



12

# **DECENTRALIZED RESOURCE MANAGEMENT IN DISTRIBUTED COMPUTER SYSTEMS**

**Carnegie-Mellon University**

**Hugh L. Applewhite  
Ravi Garg  
E. Douglas Jensen**

**J. Duane Northcutt  
Lui Sha  
James W. Wendorf**

**APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED**

**DTIC**  
**ELECTE**  
**S** **D**  
**MAR 9 1982**  
**A**

DTIC FILE COPY

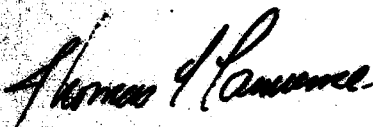
**ROME AIR DEVELOPMENT CENTER  
Air Force Systems Command  
Griffiss Air Force Base, New York 13441**

**82 04 12 006**

This report has been reviewed by the RADC Public Affairs Office (PAO) and is releasable to the National Technical Information Service (NTIS). As NTIS is open to release to the general public, including foreign nations.

RADC-PA-61-263 has been reviewed and is approved for publication.

APPROVED:



THOMAS F. LAWRENCE  
Project Engineer

APPROVED:



JOHN J. MARCINIAK, Colonel, USAF  
Chief, Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS  
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COFD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

This report is loaned to you unless contractual obligations or notices of copyright require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-203	2. GOVT ACCESSION NO. AD-A113 255	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DECENTRALIZED RESOURCE MANAGEMENT IN DISTRIBUTED COMPUTER SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report May 80 - October 80
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Hugh L. Applewhite J. Duane Northcutt Roli Garg Lui Sha E. Douglas Jensen James W. Wendorf		8. CONTRACT OR GRANT NUMBER(s) F30602-78-C-0099
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie Mellon University Computer Science Department Pittsburgh PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 558121S3
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COTD) Griffiss AFB NY 13441		12. REPORT DATE February 1982
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		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)  Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Thomas F. Lawrence (COTD)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Decentralized Control Decentralized Resource Management Distributed Computer Systems Control Computer Network		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This is the first technical report from the Archons project, which is performing research in the science and engineering of what we term "distributed computers". By this we mean a computer having highly decentralized (e.g., consensus) resource management at every level of abstraction from the executive down. This fiscal year-end report provides a snapshot of several incomplete, ongoing investigations supported by the Rome Air Development Center; (over		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

decentralized synchronization; the requirements for simulation of decentralized resource management algorithms; and the facilities to be provided by a decentralized executive. Other areas we are investigating, but which are not discussed herein, include: decentralized control in general; distributed computer architecture; and the implications of highly decentralized control on hardware/software implementation tradeoffs.

We begin with a summary of our views on decentralized resource management and control, and the implications of physical communications on control (especially at the executive level). Then we briefly survey several other distributed system projects. This brings the Archons project into closer focus, as their orientations and objectives are considerably different from ours.

Synchronization (the induction of a common, consistent ordering on events) is the essence of decentralized control. New concepts and techniques are required to achieve synchronization in distributed computers without reliance on any decentralized entity such as a semaphore, monitor, sequencer, or bus arbiter.

Recommendation For	
CLASSIFICATION	
Distribution/	
Availability Codes	
Avail and/or	
Special	
Dist	



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

# Table of Contents

<b>Abstract</b>	<b>1</b>
<b>1. Introduction</b>	<b>2</b>
<b>2. Project Background</b>	<b>5</b>
2.1 Distributed Computer Systems	5
2.2 Project Overview	7
2.2.1 Archons System Structure	8
2.3 Comparisons With Other Systems	11
2.3.1 Distributed Computer System -- DCS	12
2.3.1.1 System Goals and Objectives	12
2.3.1.2 Physical Attributes	12
2.3.1.3 Operating System Structure	12
2.3.1.4 Interprocess Communication	12
2.3.1.5 Decentralized Resource Management	12
2.3.2 Arachne (nee' Roscoe)	14
2.3.2.1 System Goals and Objectives	14
2.3.2.2 Physical Attributes	14
2.3.2.3 Operating System Structure	14
2.3.2.4 Interprocess Communication	14
2.3.2.5 Decentralized Resource Management	15
2.3.3 MicroNet	16
2.3.3.1 System Goals and Objectives	16
2.3.3.2 Physical Attributes	16
2.3.3.3 Operating System Structure	16
2.3.3.4 Interprocess Communication	16
2.3.3.5 Decentralized Resource Management	16
2.3.4 MuNet	18
2.3.4.1 System Goals and Objectives	18
2.3.4.2 Physical Attributes	18
2.3.4.3 Operating System Structure	18
2.3.4.4 Interprocess Communication	18
2.3.4.5 Decentralized Resource Management	18
2.3.5 Cm*	20
2.3.5.1 System Goals and Objectives	20
2.3.5.2 Physical Attributes	20
2.3.5.3 Logical Attributes	20
2.3.5.4 Medusa	21
2.3.5.5 StarOS	22
<b>3. Synchronization Techniques in Distributed Computer Systems</b>	<b>23</b>
3.1 Introduction	23
3.2 The Functional Classification of Synchronization Techniques	23
3.2.1 Access Synchronization	23
3.2.2 coordinating Synchronization	25
3.2.3 Meta-synchronization	26
3.3 Desirable Attributes of Synchronization Mechanisms	27
3.3.1 Reliability	27

## Archons

3.3.2 Modularity	28
3.3.3 Practicability	28
3.3.4 Performance	29
3.4 Access Synchronization Techniques	29
3.4.1 Access Synchronization in Shared Memory Computer System	30
3.4.2 Concepts and Issues in Distributed Access Synchronization	31
3.4.2.1 The Simultaneity Assumption in Atomic Transaction	31
3.4.2.2 Transaction Ordering, Update Set and Distributed Access Synchronization	32
3.4.3 Overview of Distributed Access Synchronization Techniques	34
3.4.3.1 Logical Clocks	34
3.4.3.2 Physical Clocks	35
3.4.3.3 Circulating Tokens	35
3.4.3.4 Sequencers	36
3.4.3.5 Eventcounts and Sequencers	37
3.5 Conclusion	38
4. ICON - Archons Simulation Facility	40
4.1 Introduction	40
4.2 Scope of Icon	40
4.2.1 Primary Application Area	41
4.2.2 Other Contributions	41
4.3 General Requirements for Icon	43
4.4 Analysis of Requirements	44
4.4.1 Environment	44
4.4.1.1 Archons	44
4.4.1.2 Icon	45
4.4.2 Ease of Implementation	46
4.4.2.1 A Powerful User Interface	46
4.4.2.2 Data Collection	46
4.4.2.3 Statistical Functions	46
4.4.3 Expandability and Adaptability	46
4.5 Specific Requirements for Icon	47
4.6 System Structure	47
4.6.1 System Processes	48
4.6.2 User Processes	48
4.6.3 Experimenter Process	49
4.7 User Functional View	49
4.8 Implementation Environment : Cm* versus Simula	50
5. ArchOS: An Operating System for Archons	52
5.1 Introduction	52
5.2 Goals and Objectives	53
5.3 ArchOS PAM Concepts: Issues and Design Decisions	56
5.3.1 Nature of the Program's Abstract Machine	57
5.3.1.1 Spectrum of Choices	57
5.3.1.2 Resolution	63
5.3.2 Nature of Program Structuring	64
5.3.2.1 Spectrum of Choices	64
5.3.2.2 Resolution	67
5.3.3 Further Issues in the Design of the ArchOS PAM Concepts	68



Archons

5.4 Future Work and Research Directions	68
Acknowledgements	70
I. Decentralized Resource Management and Control	71

## Abstract

This is the first Technical Report from the Archons project, which is performing research in the science and engineering of what we term "distributed computers". By this we mean a computer having highly decentralized (e.g., consensus) resource management at every level of abstraction from the executive down.

This fiscal year-end report provides a snapshot of several incomplete, ongoing investigations supported by the *Rome Air Development Center*: decentralized synchronization; the requirements for simulation of decentralized resource management algorithms; and the facilities to be provided by a decentralized executive. Other areas we are investigating, but which are not discussed herein, include: decentralized control in general; distributed computer architecture; and the implications of highly decentralized control on hardware/software implementation tradeoffs.

We begin with a summary of our views on decentralized resource management and control, and the implications of physical communications on control (especially at the executive level). Then we briefly survey several other distributed system projects. This brings the Archons project into closer focus, as their orientations and objectives are considerably different from ours.

Synchronization (the induction of a common, consistent ordering on events) is the essence of decentralized control. New concepts and techniques are required to achieve synchronization in distributed computers without reliance on any centralized entity such as a semaphore, monitor, sequencer, or bus arbiter.

Studying decentralized synchronization and other resource management algorithms requires a facility on which to execute and measure them. Until our own distributed computer testbed hardware is constructed, we will conduct simulations on the PDP-10 and Cm\*. We are now beginning to define the requirements for this facility.

We are interested in the possible roles which decentralized control might play, and how it can best influence the services and attributes of executives. Our initial application environment is real-time control. The empirical aspects of this effort are provided through the design of ArchOS, the initial "default" operating system for the Archons testbed machine.

# 1. Introduction

This fiscal year-end report is a snapshot of incomplete, on-going research sponsored by RADC in the field of distributed computer systems: decentralized synchronization, distributed executive user interface, and decentralized algorithm simulation facilities.

The Archons Project is aimed at producing a large-scale distributed computer system testbed which can be used for research and evaluation of distributed computer systems. In the context of this testbed, we intend to address the major issues of distributed computers, such as decentralized control, reliability, and hardware/software relationships. We have chosen the area of real time control as our initial applications area of distributed computer systems for several reasons: first, it is a particularly stressing environment in terms of response time, throughput, and reliability; second, the field is underdeveloped as compared to other areas of computer usage (e.g. general-purpose systems); third, there is a clear appreciation on the part of users for greater understanding and use of these systems.

There are many areas in the field of distributed computer systems which we do not touch on in this report--for example, distributed computer systems architecture, programming languages that support distributed computation, bus allocation, and so forth. This is not due to lack of interest or oversight, but rather due to the fact that our research effort is just beginning, and distributed computing is such a large and unexplored field that we cannot address all problems at once.

Chapter 2 is an overview of the total Archons Project. It describes the fundamental research issues which face distributed computer systems, and possible approaches to these issues. The background of the Archons project is briefly reviewed. An alternative model of distributed computer systems, with layering different from that of the International Standards Organization (ISO) Open Systems Interconnect (OSI) is sketched. The implications of this structure on the Archons system architecture are considerable. First, the Archons model is less strictly hierarchical--the model's orientation is process-to-process, with processes communicating at several layers of the system. Second, the Archons model is more appropriate for a tightly-integrated hardware/software implementation. Finally, several other projects in the area of distributed computer systems/local networks are described.

Chapter 3 reports the status of our review of decentralized synchronization. Synchronization can be defined as the induction of a consistent ordering on events, and is the fundamental mechanism which underlies all resource management. At the process and communication level, events occur at

random times, communication is probabilistic, and processes execute concurrently. It is the synchronization mechanism that provides structure and orders process interaction. It thus gives rise to a stable and consistent system behavior. In this section the concepts and principles of various synchronization techniques are developed. They are then used to analyze well-known synchronization techniques.

Synchronization techniques may be broadly classed into two groups, according to major purpose. The first type of synchronization is *access synchronization*. Here, the major synchronization requirement is access to some shared data object. The second type of synchronization is *coordinating synchronization*. Coordinating synchronization guarantees the correct ordering (*i.e.* precedence) of processes.

*Metasynchronization* is the specification of the logical relations which the synchronization must guarantee. Metasynchronization is important for two main reasons. First, as a means of specifying synchronization, it is the touchstone to which correct system operation must be related. Second, if the specification is simple and succinct, then system reconfiguration might be guided by it. Of course, reconfiguration must not violate (at least in the long run) the metasynchronization relations if graceful degradation is to be accomplished. Access to the metasynchronization specification must be properly controlled (*i.e.* synchronized) in order that reconfiguration results in a consistent system.

Chapter 4 documents the current status of the simulation environment requirements study which we performed. In order to gain a better understanding of decentralized algorithms such as synchronization, it is necessary to actually observe their behavior. Two different ways of observing the synchronization behavior of systems are instrumentation of an existing system and simulation. The instrumentation approach is hampered by the difficulty of implementing different algorithms on an actual system, suitably instrumenting the system, and finally interpreting the results. Simulation, on the other hand, lends itself well to the broad-brush experiments we wish to perform initially. As greater fidelity is desired, we will move to an emulation on the Cm\* multiprocessor system. Finally, the experience gained in the simulation and emulation will be used to design the stimulation and instrumentation facilities of the Archons testbed.

Chapter 5 deals with the implications of decentralized resource management upon the user's (programmer's) view of a distributed computer system. Two essentially different views are possible; they greatly impact both the implementation and performance of the system. The first view is that a programmer basically sees a standard sequential machine with no possibility of concurrent operation. The second view is that the programmer is aware that the machine which executes his programs is a distributed multi-computer system, and thus supports concurrent operation at all levels.

The first view has the advantage that it is no different than the view a contemporary FORTRAN programmer possesses; thus new users may easily move over to such a system. It also preserves the existing software investment. The disadvantage, though, is severe. Analyzing the programmer's intent and converting it into a concurrent form suitable for execution on the actual computer system is difficult and will not yield maximum concurrency. It is also difficult to take advantage of redundancy in the system in order to yield a more robust, fault tolerant application. The full capabilities of the system would not be utilized. The advantages and disadvantages of the first view are reversed in the second view. Initially, a computer system such as Archons might be more difficult to use, but the potential payoff is substantial.

ArchOS is an operating system for Archons which implements the second view. In a sense, it is the *default* set of executive software mechanisms for Archons. It is not the only operating system which might be implemented on Archons, but will provide an example of one way that a distributed computer can be utilized by an operating system<sup>1</sup>. ArchOS is valuable in that experience in actually implementing an operating system will provide feedback necessary to judge the usefulness and practicality of distributed computer concepts. In passing, we note that the operating system is a computer system's *only* user; applications programs use the operating system.

## 2. Project Background

In this project we are performing research on the architectures and operating systems of what we term "distributed computer" systems. We define a distributed computer system as a collection of processor/memory pairs, having extensively decentralized system-wide control at the executive levels and below. We are interested in determining the implications of decentralized resource management (particularly at the executive levels) on the architecture of both the processor and the interconnect structure. There are rational arguments, and even some evidence, for the hypothesis that decentralizing various types of system control offers significant potential benefits (e.g., improved robustness and modularity).

### 2.1 Distributed Computer Systems

Due to the historical centralized uniprocessor evolution of software in general (and operating systems in particular), the more decentralized alternatives for control are only recently beginning to be perceived in any sort of conceptual fashion. The term "decentralized control" is being used with increasing frequency, but unfortunately there tends to be little common understanding of what the term means, and thus is used to represent a number of very different (and occasionally centralized) concepts. Thus, our initial step was to identify the factors which contribute to the degree of control decentralized thereby illuminating the whole spectrum and delineating the region in which we are interested. Our definition of decentralized control is based on a conceptual model [21] which (while not intended to provide quantitative evaluations) provides a framework for determining the degree of decentralization of control of given entities. In our view, decentralization is based largely on the rejection of two fundamental (but usually transparent) premises which underlie almost all traditional operating system concepts and mechanisms.

The first of these premises is the existence of a unique, special entity (e.g., monitor, sequencer, bus arbiter) which enforces the state consistency of an activity. We replace this with the idea of *multilateral* control, where a number of entities participate in jointly managing the resources associated with an activity (a similar view is expressed in [27]). The *degree* of an activity's decentralization is determined by the number and inter-relationships of the lower level entities which enforce its consistency. Our model proposes that the degree of decentralization is directly related to the fashion in which controllers participate (both individually and in conjunction with other controllers) in the management of resources. For example, autocracy, functional partitioning, and democracy are increasingly decentralized forms of control [21], and in each case the number and manner in which controllers manage resources differ.

## 6 Background

The second premise which we discard concerns the (usually implicit) dependencies of most control concepts and mechanisms (particularly at the executive levels) on physical communication characteristics. Communication in computer systems involves two conceptually distinct aspects: the production, and the manifestation of signals--the relationship between these we term *observability*. Cases of poor signal observability include not just erroneous bits, but more importantly, semantic inaccuracies caused by signals having variable, unknown delays and inconsistent sequencing at different destinations. When the physical communication mechanism is shared primary memory, signal observability is normally very good--the performance and even correctness of nearly all conventional centralized executive control techniques rely on that. But in multicomputers where primary memory is physically partitioned and communication is by I/O (e.g., buses), signal observability is typically much worse. Protocols at several levels in a system can help alleviate parts of the problem (e.g., packet switching errors which result in lost, duplicate, or missequenced packets), but exacerbate other parts (e.g., delays). Thus, executive control techniques originally intended for shared primary memory may be inappropriate or counterproductive if naively reimplemented in systems with reduced signal observability. The quantitative observability differences between memory and I/O communication (as conceptually uninteresting as they may seem) will generally necessitate qualitative differences in control, particularly at the executive level--just as different memory technology speeds lead to different management approaches for primary and secondary memories. Concepts and mechanisms which are specifically intended to function correctly and efficiently despite inaccurate and incomplete information must employ algorithms which make "best effort" resource management decisions. We believe that the use of "best effort" type algorithms is an essential feature of the operating system of a true distributed computer system.

In order to create a practical distributed computer system, thoughtful (and, not coincidentally, unconventional) hardware/software tradeoffs are needed. Furthermore, in order to optimize overall cost-effectiveness it is necessary, when making hardware/software tradeoffs, to recognize that decentralized executive control is a *system* rather than a software matter [22]. A major portion of our research is concerned with the implications of decentralized control at the executive levels and below on the design and implementation of processors and interconnection mechanisms specifically for distributed computers. It is clear that, because all other facilities are built on them, physical and logical communication play a key role in decentralized control. Dynamic port/link creation and deletion, message addressing/routing, type checking and protection, priorities, and buffer management are examples of traditionally expensive software functions which can be effectively supported by hardware mechanisms without necessarily restricting policy decisions. Decentralization also emphasizes the importance of creatively implementing higher level functionality, including

synchronization, atomic operations, fault detection and recovery, and process management. System control appears to be an exceptionally promising opportunity for VLSI technology.

## 2.2 Project Overview

While this project is intellectually novel, another important aspect which sets it apart from others is the design and implementation of innovative new hardware on a large scale. A second-generation (HXDP [19] was the first) laboratory testbed is being created to permit experimentation with not only the design but also the hardware/software implementation of decentralized control mechanisms and facilities from the lowest levels (e.g., physical communication protocols) to the operating system user interface. While our primary interest is research, this testbed will also be valuable to industry and government laboratories as a distributed system development vehicle. It will consist of a logical and physical framework with which a wide variety of distributed computer systems can be designed, implemented, configured, programmed, exercised, and measured. The foundation of the system is the Archons "control and communication subsystem" -- a logical and physical framework containing both standard and experimenter-provided software and hardware primitives which can be flexibly interconnected for carrying out the broadest possible range of decentralized control policies at levels from the executive down to the processor interconnection.

Where the requirements of an application environment should be considered, our choice is real-time control (with special emphasis on high performance aerospace and military systems of the future [35], [13]). One of the primary motivations for this choice is that this environment has (and recognizes) a great need for the potential benefits of distributed computer systems; furthermore, the range of real-time control needs is so wide that it encompasses the limited capabilities afforded by the current state of the decentralized control art, as well as the most ambitious aspirations for the future.

The distributed computer testbed is oriented towards multicomputer systems in which the constituent "application subsystems" have disjoint primary memories, and communicate via explicit I/O. Archons interconnects as many as 256 application subsystems, which may be heterogeneous, general- or special-purpose (e.g., signal processing, sensor/actuator data multiplexing, etc.), and contain any number of processors each. To accommodate such a broad span of application subsystems with a minimum of cost and effort, descriptions their I/O interfaces are written in the ISPS language [4] and compiled into code which programs the Archons interface to match. Our laboratory application subsystems have not yet been selected, but will likely be some readily available machines (e.g., PERQ's, VAX's) together with one or more new ones designed by the project expressly to study the relationships between decentralized executive control and application processor architecture.



Each application subsystem is connected to Archons through a "control and communication unit" in which resides much or all of the system-wide executive. This makes the system more independent of application subsystem hardware and software, allows the incorporation of special hardware to facilitate decentralized control, and provides concurrent execution of application and executive. However, an application subsystem must have certain features if its resources are to be managed entirely externally. The application subsystems of our own design will have these features, but no currently available products do, so those require that a portion of the system executive reside inside them. Furthermore, there are other circumstances under which an application subsystem may include a private executive--for example: when it has resources which must not (for reasons such as security) be managed globally; or when preserving some existing application subsystem run-time utility software outweighs the reduced benefits of local control. Every control and communication unit is itself a highly capable machine with a multiplicity of functionally dedicated processors in a data-flow architecture.

### 2.2.1 Archons System Structure

The Archons structure is layered, but according to a philosophy (beyond the scope of this paper) we feel is more suited to decentralized control (especially at the executive levels) than are others, notably the ISO Open Systems Reference Model [50]. While our model is outside the scope of this document, one difference is that levels need not be strictly hierarchical, or even totally ordered: the abstractions at one level may not involve all (or even any) of those at some particular level below it; at times, one level may employ another, while at other times that relationship may be reversed (e.g., the interprocess communication and synchronization levels). Another departure is that messages may be initiated at any level and may terminate at any other level -- this allows a more natural representation of important behavior such as that of our acknowledgement mechanisms.

The design of Archons is proceeding "outside in", but even so the lowest levels bear the closest resemblance to conventional technology and thus more is currently known about them. Their facilities include but are not limited to the following.

Layer 1 is for the physical path, which is a set of autonomous buses, each connecting any (but usually all) control and communication units. Up to 16 buses may be implemented, for both reliability and bandwidth reasons. The global bus topology provides good modularity, low latency, and system-wide visibility of communication to assist in achieving decentralized control [1]. The buses are bit-serial and as long as 1 KM in length to accomodate moderate physical distribution. Every bus has a data rate of at least 10 Mbs or more, depending on the state of bus technology.

Layer 2 encompasses bus control (i.e., allocation and communication). Each bus is half duplex, and any pairs may be used in full duplex mode. Bus allocation is done in the time domain by a special machine whose hardware and software are programmable to efficiently implement a wide variety of different algorithms [31] [44], including our own which is designed for optimal dynamic priority responsiveness. The communication protocols at this level are also totally programmable, and the facilities are oriented towards a state-exchange model of balanced, n-party protocols. The valid sequences of messages allowed for various types of bus transactions are defined within one processor of each Control and Communication Unit. The intended recipient(s) of a message may be determined by destination, source, or message names, which are software defined and independent of physical bindings--thus, the names may correspond to logical entities (e.g., processes, ports, links), services, or even global or local conditions which must be met (type checking, protection, guards, etc. are special cases). Unintended entities (and thus Control and Communication Units) may eavesdrop on messages to improve Archons' view of the system state--at this level, that ability helps detect and recover from communication (e.g., protocol) errors. Message fields may be arbitrarily located, and of any length--this information is hidden from the rest of the control and communication unit by a format handling processor which uses it to create a standardized description of the message organization.

Level 3, the Transport level, is for implementing communication among the executive kernel instantiations. It transforms the pool of autonomous buses into one virtual, higher speed, more reliable bus on which messages are sent according to priorities and precedence constraints. An atomic broadcast (i.e., all intended recipients correctly receive the message or none do) is provided, subject to certain restrictions. As a mode, or on a per-message basis, buses may be selected or excluded for transmission or reception of messages or message types. Archons provides a system-wide interprocess communication facility (among others), as opposed to an application subsystem physical communication subnetwork--therefore, every message is normally transmitted on a bus, regardless of whether the source and destination(s) happen to be bound to the same application subsystem at that time. This supports dynamic reconfiguration for modularity and robustness, albeit at the cost of higher bus bandwidth. If this cost is considered too high for particular circumstances, messages among physically co-located entities may be looped back inside a control and communication unit. It is also at this level that arbitrary size messages are subdivided into packets of dynamically determined optimal length, and reassembled when received by the destination(s).

The fourth (the IPC) level is devoted to communication among logical entities such as processes, ports, links, etc., as determined by the experimenter. Its mechanisms are not yet completely defined--

10  
Background

they attempt to efficiently support all proposed models of interprocess communication, and facilitate the invention of new ones specifically for decentralized control situations.

The layers from the fifth up deal with synchronization, atomic operations, and other more general forms of decentralized resource management. Their nature is the subject of much of our research, and as such are not clearly defined at this point.

## 2.3 Comparisons With Other Systems

As stated earlier, a number of very different systems are being called distributed computing systems. In many cases, it is not possible to distinguish between systems previously known as networks, and those called distributed computers on the basis of a system's physical attributes. In an attempt to remedy this problem we have defined distributed computer systems to be those systems that have decentralized control at the executive level and below. Executives which are significantly decentralized according to our perspective are not presently within the state of the art. For this reason it is not realistically feasible for most multicomputers to have a distributed, conceptually singular operating system in the same sense that a uniprocessor or multiprocessor can--i.e., one which integrates the processors into a *computer* by managing all the executive level resources in the best interests of the whole system. Instead, most are constrained to being what we define as a *network*, i.e. a system in which there is a separate operating system for each processor which manages a local partition of the system's resources in its own best interests. In many applications (e.g., real time control), having only local and no global executive control greatly reduces the extent to which certain important attributes (such as robustness and modularity) can be achieved on a system-wide basis.

Our definitions are such that it is not possible to classify a given architecture as being either a network or a distributed computer without considering the nature of its operating system. It is the purpose of this section to help distinguish our system from other current efforts in the general area. The primary factor which sets our research apart from that of others is our interest in pursuing decentralized control from the executive level down. For this reason, the emphasis in the following comparisons is on the degree of executive level decentralization.

### **2.3.1 Distributed Computer System -- DCS**

#### **2.3.1.1 System Goals and Objectives**

DCS [14] was designed as a system to provide highly reliable user services from a network of up to thirty minicomputers. It was intended that DCS perform a dedicated task (which was originally text processing) without the use of any central controlling processor. Furthermore, it was desired that (for extensibility reasons) it be simple to add processors to the system, and that (for reliability and performance reasons) it be easy for processes to migrate from one processor to another.

#### **2.3.1.2 Physical Attributes**

DCS is a ring-structured loosely-coupled collection of processor memory pairs, with a physical distribution of a few square miles. In DCS, there are seven homogeneous minicomputers, five of which interface to a pair of unidirectional rings through simple buffer and shift-register interfaces. Communications is accomplished by processors putting messages on a ring when an empty slot is detected and removing the message when it has made a complete cycle around the ring. When a processor removes its own message from the bus it is compared with the original message to determine if an error has occurred in the transmission of the message.

#### **2.3.1.3 Operating System Structure**

The operating system for DCS consists of replicated kernels which reside on each node in the system. The DCS operating system is structured in a layered fashion similar to Dijkstra's T.H.E. operating system.

#### **2.3.1.4 Interprocess Communication**

Messages in DCS are addressed to processes, as opposed to physical processors (to allow process mobility). Broadcasts of messages are permitted to classes of processes, as well as individual processes. While communication between processes is carried out independent of the physical location of the destination process(es), messages destined for processes co-residing on a processor are not actually placed on the ring.

#### **2.3.1.5 Decentralized Resource Management**

Resource management at the executive level is accomplished through a bidding scheme based on a simple economic model. Processes which require resources broadcast a "request for quote" (RFQ). Upon receipt of an RFQ, those nodes that have the desired resource and are willing to supply it, return a bid (which consists of an estimate of the node's cost to supply the resource). Finally, the

node desiring the resource waits for a period of time and then selects (based on some criteria) one of the bids it receives. In this fashion a "contract" is made which must then be okayed by a centralized process known as the notary before the transaction may be carried out.

While the only resource that is managed through bidding in DCS is memory, bidding schemes can be used to manage other resources multilaterally [39]. In a bidding scheme such as the one implemented on DCS, it should be noted that the degree of multilateral resource management in DCS is low. This is due to the fact that the decision to send an RFQ and the decision to bid on an RFQ are made on a local basis -- a node permits others to engage in multilateral resource management only when it sends an RFQ, and a node need not necessarily bid on an RFQ if it chooses not to. Furthermore, the node requesting a service decides locally which of the bids it will choose, it is not the case that the requesting node need choose the "lowest" bid each time; it may choose one bid over a group of others based on some local information about the bidding nodes.

In DCS the only time multilateral management occurs is when the nodes bid to manage the system's memory. While the bidding scheme is more decentralized than a centralized entity managing the resources, bidding is clearly not a very decentralized form of resource management.

14  
**Background**

**2.3.2 Arachne (nee' Roscoe)**

**2.3.2.1 System Goals and Objectives**

The **Roscoe** system [40] was designed as a general purpose computing facility in which objects are shared in a distributed, non-hierarchical fashion.

**2.3.2.2 Physical Attributes**

The hardware for **Roscoe** consists of a connected network of five LSI-11's which interface to one or more nearest neighbors by way of parallel line units. All communication between nodes is in the form of messages which are transferred around the network in a store and forward fashion. Routing in the **Roscoe** system is performed by fixed routing tables at each node.

**2.3.2.3 Operating System Structure**

A kernel resides on each node which is responsible for providing low level functions. The entities defined in **Roscoe** are files, programs, core images, processes, links, and messages. The kernels' primary responsibility is to provide support for the message passing and the link mechanism. However, the kernel also has mechanisms to create and destroy processes, maintain clocks and interval timers, and establish interrupt handlers for devices.

In **Roscoe** only the most rudimentary functions are implemented in the kernel, all other (higher level) functions are implemented by ordinary processes. **Roscoe** provides the user with a set of processes known the utility processes. Among the utility processes are terminal drivers, file managers, and resource managers. Each utility process is localized, e.g., file management processes exist only at nodes that have mass storage. Furthermore, **Roscoe** combines the use of kernel service calls and utility processes to provide the user with a limited number of higher level functions. These functions are called library routines and provide useful, higher level functions for directory manipulation, file or terminal I/O, and other functions of this nature.

**2.3.2.4 Interprocess Communication**

To the user, the **Roscoe** system appears as a single machine in which all processes communicate by way of a uniform interprocess communication mechanism (regardless of whether the destinations are physically local or remote). In **Roscoe** the kernels are designed to support both message passing and communications link management. The link mechanism used in **Roscoe** is very similar to that of the CRAY-1 Demos operating system [5].

### 2.3.2.5 Decentralized Resource Management

The original resource management scheme proposed for Roscoe consisted of a simple mechanism for managing the system's processes, processor cycles, and memory. When a new process is spawned, the resource management process at that node makes a decision as to whether or not the new process will be run locally. If it is decided that the process is not to be run locally, a request to run the new process is passed to the next node in a virtual ring. Each node that receives a request makes a local decision as to whether it will accept the process or pass on the request. The request to run the new process returns to the initiator around the virtual ring if no node accepts the request. In such a case, the node which initiated the request must decide whether it should run the new process locally, resubmit the request to run the new process remotely, or kill the new process.

Recently there has been a proposal for a new resource management scheme to be implemented on Roscoe [9]. This scheme is considerably more complex than the original one, and consists of two separate activities. The first activity consists of calculating the load of a processor at a given point in time. The second activity is the cooperation of processors in balancing their loads. The method by which processors cooperate is through pairing with nearest neighbors and transferring tasks from the more heavily loaded processor to the lighter loaded processor. This pairing process occurs repeatedly throughout the system, and processors remain paired only long enough to transfer task descriptors.

Decentralized resource management (in varying degrees) occurs in either of the two resource management schemes proposed. However, in the simple case there is only a small degree of decentralization due to the fact that the management decisions (e.g., to accept or reject a request) are made by each node individually, based on local considerations and there is only one manager for the resources at any given point in time. It is possible for all managers to be involved in the management of a given resource (as in the case where a request to run a process successively passes to each node in the virtual ring). However, the degree of equality of authority each node has and the degree of interaction between nodes is clearly quite limited.

The proposed (more complex) resource management scheme is somewhat more decentralized than the simple case, as managers cooperate in successive pairs to manage their collective resources. However, only a small portion of the system's controllers take part in any resource management decision, and again there is a large difference in the equality of the management authority possessed by each of the nodes. Therefore, the degree of resource management decentralization (at the executive level) is moderately low.



### 2.3.3 MicroNet

#### 2.3.3.1 System Goals and Objectives

MicroNet [47] was designed to support multiple, general-purpose users, each running parallel programs, on a network of from two up to ten thousand processing nodes. It was intended that MicroNet tolerate the loss of nodes, allow for a wide variety of interconnect topologies, and adapt to dynamic variations in loading. The designers of MicroNet explicitly reject the concept of master/slave relationships or star configurations as being too centralized to achieve their goals of reliability and extensibility.

#### 2.3.3.2 Physical Attributes

In MicroNet, a node consists of an LSI-11 with memory and a Z-80 based communications front end which connects the node to the front ends of two other nodes. It is intended that a total of sixteen nodes will be connected in a regular grid configuration. The front ends can be used to implement a store and forward network of arbitrary topology, using word-parallel buses.

#### 2.3.3.3 Operating System Structure

In the MicroNet operating system (i.e., MicroOS), a kernel resides on each node and provides support for communication and a command language interpreter (i.e., the CLI). A CLI is provided at each node to permit the user to set up, spawn, and interconnect processes. There exists a global name space which contains unique names for every addressable entity in the system. All entities in the system appear uniformly as processes, which have the same capabilities as does the user. This means that processes use the same interface as is provided to the user by way of the CLI.

#### 2.3.3.4 Interprocess Communication

Interprocess communication is carried out in a uniform (to the user) fashion, regardless of the location of the destination processes. The kernel provides support for the interprocess communication mechanism.

#### 2.3.3.5 Decentralized Resource Management

In MicroNet, global resource management is carried out by means of a hierarchy of unilateral decisions. The designers of MicroNet recognized that it is not practical (or desirable) to have a centralized table that maintains a record of the state of the system resources. Thus, they decided to distribute the state information about the system in a hierarchical fashion such that as one progresses up the control hierarchy, the scope of the control information increases while the detail of the

information decreases. The hierarchy is structured in a fashion such that each manager is responsible for either a group of lower level managers, or a group of application processing nodes. This means that the actual, user visible resources exist only at the leaves of the control hierarchy. To avoid the problems presented by having a single manager at the top of the control hierarchy, it was proposed (and subsequently abandoned in favor of a single manager with a hot standby) that the topmost control level consist of an oligarchy of controllers. The control hierarchy is constructed without regard to the physical topology of the network, but an effort is made to ensure a high degree of connectivity between controllers at each level and their resources.

For the most part, **MicroNet** contains few examples of decentralized resource management. At the lower levels in the control hierarchy, each resource is managed by only one controller. This is true at all levels in the system up to the highest level of the control hierarchy where the oligarchy of controllers was to manage the resources below them. Thus, executive level resource management in **MicroNet** is not fully centralized in that there is no centralized entity which controls all of the system's resources, but it is also not very decentralized as there exists no instances of multilateral resource management.

### 2.3.4 MuNet

#### 2.3.4.1 System Goals and Objectives

**MuNet** [16] was designed to be a program-transparently scaleable architecture, with the intention of combining a potentially large number of processors into a single, general purpose machine. It was desired that processors be added to increase performance incrementally without adding to the communication load (as is the case in bus-structured systems).

#### 2.3.4.2 Physical Attributes

In order to accomplish the desired goal of scaleability, processors in **MuNet** connect only to a small number of other processors in the system. **MuNet** has admittedly very simplistic hardware, as the goal of **MuNet** is the development of software methodologies for and the organization of such networks. **MuNet** currently has "several" (sic) LSI-11's connected, in an arbitrary configuration, to a maximum of four other processors by bit serial lines.

#### 2.3.4.3 Operating System Structure

In **MuNet** each node has a kernel which is used to support the user interface known as the virtual interface to **MuNet** (i.e., the VIM). The VIM provides a virtual machine to the user, through which user programs can invoke internal **MuNet** functions. The VIM defines two entities-- objects and events. Objects are defined as files (containing data or algorithms) and events are defined the same as conventional processes. Additionally, process/processor binding is carried out by **MuNet** in a user transparent fashion.

#### 2.3.4.4 Interprocess Communication

All interprocess communication (and object sharing) is accomplished by way of reference trees [15].

#### 2.3.4.5 Decentralized Resource Management

In **MuNet** load balancing is the only instance of multilateral resource management. Events are migrated to balance processing load, while objects are migrated to balance memory load. The kernel at each node in **MuNet** maintains an "event list", which is a list of the events (i.e., processes) a particular processor is to execute. Each node also sends information concerning the size of its event list to its neighbors, in addition to gathering such information from its neighbors. Thus, when a node's event list exceeds a given value, it ships some of its events to a neighboring, less loaded node. The choice of a node to ship events to is based on the length of each of the neighbors' event list.

There is a low degree of decentralized resource management in **MuNet** due to the fact that management decisions are made on a very localized basis with no more than a small number of controllers participating in the decision making process. However, insomuch as there is some degree of cooperation between neighboring nodes for the purpose of load balancing, it can be said that some form of decentralized resource management occurs at the executive level in the **MuNet** system.

## 20 Background

### 2.3.5 Cm\*

#### 2.3.5.1 System Goals and Objectives

The Cm\* multiprocessor [42] was designed and built to perform research on multiprocessor systems that have close cooperation between a large number of small processors. It was also intended that Cm\* serve as a testbed for multiprocessor hardware and software experimentation.

#### 2.3.5.2 Physical Attributes

The Cm\* hardware consists of a two level hierarchy of buses connecting five clusters of ten computer memory pairs (which are known as computer modules or Cm's). Each Cm is made up of an LSI-11, a local memory, and a local switch (the Slocal) which is interposed between the two. Up to fourteen Cm's can be connected by an intracuster bus to a switching node (known as a Kmap), thereby forming a cluster. The Kmaps are used to effect connections between clusters by interfacing to (at most) two intercluster buses. Both the intracuster and intercluster buses are word parallel. The Kmaps not only perform the basic communications functions, but they also provide support for some low level operating system functions.

#### 2.3.5.3 Logical Attributes

The logical structure of Cm\* can be altered by changing portions of the system software and firmware. It is possible for Cm\* to be either a shared main memory multiprocessor, a network, or a distributed computer system depending on the Kmap microcode and the operating system executive. The current Kmap microcode is such that Cm\* is a shared main memory multiprocessor (all processors share a virtual memory space). Nonlocal memory references are intercepted by the Slocals and sent to the cluster's Kmap, which then routes the access request to the appropriate cluster, and the Kmap for the destination cluster directs the request to the proper Cm. Thus, memory access is uniform across the entire virtual memory space, however access times are non-uniform (and vary by an order of magnitude).

#### 2.3.5.4 Medusa

One of the operating systems experiments being performed on Cm\* is the design and implementation of the Medusa operating system [33].

##### Goals and Objectives

The main objective of Medusa was to perform experiments in operating systems structure for multiprocessors. It was explicitly desired that the Medusa operating system fully exploit the potentials of the Cm\* hardware. To accomplish this it was decided that operating system level resource management decisions should be made independantly, as opposed to by centralized entities.

##### Operating System Structure

Each Cm has a replicated kernel that provides simple functions such as devices handlers and simple process multiplexing. The Medusa operating system consists of the replicated kernels and a collection of relatively autonomous utilities, each of which provides a single, dedicated function. Among the utilites provided to the user by Medusa are memory management, file system management, and task force management utilites. All utilities in Medusa are software structures known as task forces. Task forces consist of a collection of concurrent activites that execute single tasks. Activities are roughly analogous to processes in other systems, however activities in Medusa are statically assigned to processors.

##### Interprocess Communication

Interprocess communication in Medusa is accomplished by sending and receiving messages through simple pipe mechanisms [37]. The Kmaps provide low level support for the message passing mechanism.

##### Decentralized Resource Management

In Cm\* under the Medusa operating system there is little decentralized resource management at the executive level. This is due to the fact that each resource management activity is carried out by an activity of a task force, and each activity serves a different request. Furthermore, activities within a task force only interact with one another in the activity of locking to provide mutually exclusive access to shared data structures.

#### 2.3.5.5 StarOS

The second of the two operating system projects for Cm\* is the StarOS operating system project [24].

##### Goals and Objectives

The main objective of StarOS was to develop an operating system that would support large collections of concurrently executing, cooperating processes (i.e., task forces). It was furthermore desired that StarOS provide facilities to develop and experiment with task forces.

##### Operating System Structure

All but the most primitive functions in the StarOS operating system are implemented as task forces. Only a few, lower level functions are not task forces, but are implemented by the Kmaps. Each task force in StarOS contains a complete complement of the operating system utilities. That is to say, each operating system utility is an individual task force activity. Furthermore, each cluster has a complete task force (and thus a complete operating system). Additionally, task forces and utilities are statically assigned to Cm's.

##### Interprocess Communication

In StarOS, interprocess communication is performed through mailbox objects which are defined by the system and supported by the Kmaps.

##### Decentralized Resource Management

In StarOS there is little decentralized resource management at the executive level. This is seen in the fact that each cluster has a task force which consists of managers for the cluster's resources. Each resource in a cluster is managed by a utility within the task force that resides at that cluster. The only interaction between task forces (and therefore clusters) occurs when a request for a resource in another cluster is made. In this case, the local resource manager forwards the request to the manager in the cluster with the desired resource, and it is that manager which then makes a local resource management decision.

### 3. Synchronization Techniques in Distributed Computer Systems

#### 3.1 Introduction

The system behavior of a distributed computer system is characterized by the interactions of concurrent processes. At the process and physical communication levels, events occur at random times, communication is probabilistic, and processes execute concurrently. It is the synchronization mechanism that provides structure and order to interactions, and thus gives rise to a stable and consistent system behavior. In this report we present the concepts and principles of synchronization, and then use them to review some well known synchronization techniques. In the next stage of our research, we will investigate various decentralized algorithms in order to design better synchronization mechanisms.

#### 3.2 The Functional Classification of Synchronization Techniques

Synchronization techniques well suited<sup>2</sup> for regulating access to shared resources will be classified as "access synchronization" techniques. Synchronization techniques well suited for maintaining specified temporal orders between *cooperating* processes will be specified as "coordinating" synchronization techniques. Finally, techniques used to change the logical relations, which are maintained by the synchronization techniques, are classified as "meta-synchronization" techniques.

##### 3.2.1 Access Synchronization

One important aspect of a modern computer system is resource sharing among various processes. For example, in a general purpose multi-programming environment, although programs are usually independent at the user level, they still interact owing to the sharing of resources at the system level. Due to the independence at the user process level, requests for the common resources are independently generated and can conflict with each other. An access synchronization mechanism is therefore needed to order those independently generated requests so as to maintain the consistency of the shared resources. A generic case of access synchronization in a distributed computer system

---

<sup>2</sup>Since one synchronization mechanism may accomplish more than one function, the classification is according to its main function



environment is the access of a distributed data base, and we will use this case as an example to illustrate various distributed access synchronization techniques.

Access synchronization is, in fact, a mapping from a randomly generated temporal order of requests into an organized temporal order of accessing so as to maintain the consistency of the shared resources. When there are several temporal orders which can satisfy the consistency requirement, the synchronization mechanism is free to randomly choose one. For example, consider the case of producers and consumers sharing a bounded buffer. When the buffer is empty, the synchronizer is free to randomly choose one of the producers. When the buffer is full, the synchronizer is free to choose one of the consumers. Otherwise, the synchronizer is free to choose one of the producers or one of the consumers. As we can see the temporal accessing order is the *means* to achieve the *ends* of the consistency requirement.

The parameters of such a mapping are:

1. the type of request, such as read or write;
2. the sharing policy, such as priority;
3. the state of a shared resource, such as the shared buffer being empty or full;
4. in a distributed computer system, there is also the logical dependence between distributed resources. When members of a set of distributed resources are update dependent, we call such a set as an "update set". For example, distributed and replicated files form a update set, because when one member of this set of files is modified, the others have to be modified accordingly. The logical dependence between distributed and shared resources is at the heart of distributed access synchronization.

A serialization mechanism is required to order the requests one by one. In a shared primary memory computer system, special hardware supported indivisible operations such as test-and-set are usually employed. In a distributed system environment, various serialization mechanisms such as circulating tokens [28] have been proposed. One important point is that only the *mapping process* needs to be serialized, not the resulting accessing order. Failure to separate these two will result in loss of concurrency. We will illustrate this point via the examples of monitors [17] vs serializers [3] in shared memory systems and circulating tokens [28] vs eventcounts and sequencers [36] in a distributed computer system environment.

### 3.2.2 coordinating Synchronization

One of the central tasks of a distributed computer system is to maintain close cooperation between distributed processes. Certain processes have to execute together, while the execution of some processes have to precede others. That is, we want to maintain the specified precedence relations (temporal orders) among distributed processes. For example, we may want a set of distributed radar and signal processing systems to work cooperatively and concurrently. When they reach a consensus that some unidentified flying objects are approaching, they will simultaneously activate a set of geographically distributed processes which represent the appropriate response. Techniques used to maintain such a specified system temporal order are defined as coordinating synchronization techniques.

In order to formalize the concept of cooperation, we introduce the notion of *logical simultaneity*. A set of processes is said to be initiated in logical simultaneity if and only if they are activated by the same initiating event. A set of procedures is said to be terminated in logical simultaneity if and only if they are terminated by the same terminating event. A set of procedures is said to be executed in logical simultaneity if and only if they are initiated in logical simultaneity and terminated in logical simultaneity. Note that logical simultaneity and the physical time simultaneity do not imply each other. For example, two events occurring simultaneously in physical time may be just due to co-incidence and does not reflect any cooperation at all. On the other hand two distributed processes activated by the same event will not start their executions at the same exact physical time due to randomized communication delays in receiving the remote activating event and due to drift in the distributed clocks. Once the logical simultaneity is guaranteed, the physical time simultaneity can, however, always be approached. For example, if we want some distributed processes to be initiated by the same remote event A and we want them to start the execution in physical time as close as possible, we can simply defer the event A's initiation effect by, say, X time units. That is, if event A occurs at, say, 11:00, messages will be sent to inform all the relevant processes to start execution as soon as their local clocks indicate 11:X. Suppose the probability that all relevant processes successfully receive the message by 11:X is, say, 0.99. Under these conditions, these processes will start execution, with probability 0.99, as close as the accuracy of synchronization among distributed clocks.

One consequence of the definition of logical simultaneity is that it can be represented by precedence relations between distributed processes. For example, instead of qualitatively stating that the monitoring and the tracking processes work concurrently, we can specify that the initiation of these processes is *preceded* by the event X and their termination is *preceded* by the event

Y. Therefore, we can formally specify the cooperation among distributed processes by only the precedence relationship.

In order to maintain the precedence relations among distributed processes, we have to send relevant events to relevant parties and define the response to received events. The important parameters of coordinating synchronization are therefore:

1. on the sender's side, a list of receivers of a given event;
2. on the sender's side, an exception handling routine when the expected response from the receivers does not occur;
3. on the receiver's side, a list of authorized senders for each type of event in order to guard against unauthorized sending, and a routine to help the operating system to find out why unauthorized sending happens;
4. on the receiver's side, a set of procedures to operate on the received events.
5. An inter-process communication protocol to handle the communications between the event senders and event receivers.

### 3.2.3 Meta-synchronization

Access synchronization and coordinating synchronization techniques are used to ensure the logical relation among shared resources and processes. However, we may want to dynamically change the existing logical relations without stopping the system. This could be the result of some system element failure, so we have no other way but to alter the logical relations in order to adapt to the remaining hardware; or this could be the result of system expansion, and we want to add new processing elements but it is too expensive to stop the system as a whole. In any event, techniques to ensure a smooth transition from old relations to new relations among concurrent processes and among distributed data bases are vital. Since these techniques address the question of how to modify the logical constraints which are maintained by synchronization mechanisms, we call them "meta-synchronization" techniques. For example, the time-varying logical relations which are maintained by synchronization mechanisms can be illustrated by the problem of time-varying precedence relations which are maintained by a coordinating synchronization mechanism.

From a theoretical point of view, the coordinating synchronization mechanism provides a state exchange mechanism between distributed processes. For example, when we say that sending process A produces a event, it is equivalent to saying that process A enters a special communicating state. When the receiving process receives the event and sets variables which represent the effect of

the event, the effect is that the state vector of A at the communication state is mapped onto those state variables of the receiving process. Since the precedence relation represents the cooperation among distributed processes, and since the coordinating synchronization mechanism maintains the *precedence relations by means of supporting the state exchanges among distributed processes*, we conclude that the mappings of states among distributed processes completely specify the cooperating behavior of distributed processes.

In the previous section, we have implicitly assumed that the precedence relations between distributed processes are somehow completely and correctly specified under all circumstance, so we *can use coordinating synchronization mechanisms to carry them out*. In practice, defining static precedence relations can be relatively simple, such as in many signal processing applications. It can also be extremely complex in some advanced real time application, where system elements are subject to destruction and system reconfiguration is needed, or when autonomy of system elements is required. In either case, the precedence relations are time-varying and depend on dynamic system events. The value of a state exchange view is that it allows us to develop a formal meta-synchronization mechanism based on the exchanged states (mappings of states) among distributed processes.

### 3.3 Desirable Attributes of Synchronization Mechanisms

In order to compare various synchronization mechanisms, we have to identify a set of desirable attributes. However, the objective parameters and weights associated with them depend on one's application environment. Our basic assumptions about the application environment are as follows: 1) our model is a distributed computer system without shared memory; 2) the computer system must be able to maintain its main functionality despite the failure of individual system element; 3) the major difficulty in the system development will be software development; 4) the system is used in some demanding real time applications. The important system attributes are, therefore, reliability, modularity, performance and practicability.

#### 3.3.1 Reliability

One of the primary motivations for building a distributed computer system is reliability. Synchronization mechanisms, as a key element in maintaining the system behavior, are therefore required to be able to deal with the disruptive effect of a system element failure. We would like the access synchronization mechanism to be capable of maintaining the consistency of distributed data bases and the co-ordinating synchronization mechanism to be capable of maintaining the

precedence relations among co-operating processes. Furthermore, we would like the meta-synchronization mechanism to provide the necessary state information for such operations.

In any complex system, there will always be a chance that a fault occurs and it is neither tolerable nor recoverable. Under this circumstance, we would like the damage to be confined.

### 3.3.2 Modularity

In many modern computer systems, the cost of software has far exceeded the cost of hardware. One major tool used to combat the software complexity is modularity. A piece of software is nodular if its properties are context independent. That is, the functional behavior of an object can be understood, specified, and implemented without reference to the context.

The requirement of context independence logically leads to the requirement that the synchronization mechanism is an integral part of the definition of a resource module. Only the specification of the abstract resource and the allowable operations should be visible at the user level. Resources presented to the user must be properly synchronized at the implementation level, and "no synchronization code need be included in programs accessing the resource" [6]. This is because if every user writes his own piece of synchronization code, one can never verify anything about the property of the shared resource independent of its context.

### 3.3.3 Practicability

A given synchronization method will be practical if it is easy to use at the user level. At the user level, the user should be required only to specify the logical dependence between distributed resources and the precedence relations between distributed processes. For example, users should be allowed to generate accesses to replicated files without worrying about synchronization details.

Therefore we measure the practicability by the complexity faced by a user. Consider the classical example of critical regions. The pure software solutions are rated low in practicability, because users have to program the procedures to interact in a complicated way to obtain the mutual exclusion. Semaphores simplify the job by providing simple and clearly defined P and V primitives. The monitor further improves practicability by reducing the user's task of using a shared resource into calling appropriate monitor procedures.

### 3.3.4 Performance

The performance of a given synchronization mechanism is measured by its expressive power and the amount of system resources it consumes.

The two main parameters of expressive power are concurrency and priority. For example, we can maintain the consistency requirement of shared resources by passing a single token around all the potential users. The user currently holding the token can access the shared resources. Such a method, however, neither supports concurrent access, nor supports any priority policy other than *round-robin*.

Synchronization procedures, as any computational procedures in a distributed computer system, require computational resources, storage resources, and communication resources. Under the assumption that VLSI technology can provide inexpensive local computation and storage, we are not too concerned about the amount of local computations and local memory a given synchronization method consumes. The major concern will, rather, be the system level response and throughput.

However, the system level response and throughput are affected by many factors other than the synchronization techniques employed. A reasonable approach to deal with the performance problem, we believe, is to measure the non-local and system wide impact generated by a given synchronization method. Since interprocess communication is a well known overhead which seriously affects the performance of multi-processor/computer system [Chu80], we will use the communication load generated by synchronization methods as the indicator of system resource consumption.

## 3.4 Access Synchronization Techniques

As computer systems evolved from uniprogramming to multi-programming systems with shared primary memory, and further evolved to distributed computer systems without shared primary memory, many underlying system characteristics changed. This requires corresponding changes in the construction of synchronization techniques. We will, therefore, briefly review some familiar techniques of shared primary memory computer system from an evolutionary perspective so as to reveal the underlying concepts of synchronization mechanisms. We will then discuss some pertinent issues of access synchronization in the environment of distributed computer systems without shared memory. Finally, we will give an overview for some representative distributed access synchronization techniques.

### 3.4.1 Access Synchronization in Shared Memory Computer System

In the early uniprogramming systems, there was only one active process. The access synchronization problem was non-existent. The development of multi-programming systems in the 1960's made the synchronization problem much more complicated. Since more than one process may compete for some shared common resources, the so-called "critical section" problem arises.

In the early days pure software solutions [Shaw] were proposed to solve the critical region problem. The inherent difficulties associated with this approach stem from the implicit assumption that the access synchronization should be solved by directly exchanging *user state* information. However, there is no direct mapping from user state to the state of shared resources, unless users are co-operating in a very special order to achieve mutual exclusion. Designing such a run time interaction order has proved to be a very difficult task.

One of the well known solutions to the access synchronization problem was Dijkstra's semaphore [12]. The importance of the semaphore is that it correctly addresses the basic issues of access synchronization ---- *the serialization of conflicting access requests and the representation of the state information of the shared resource*. In the construction of a synchronization primitive, the key element is usually a hardware supported non-divisible operation ( e.g. test-and-set ) on a shared variable. This not only serializes the concurrent accesses but also communicates the state of the shared resources in one step. Compared with pure software solutions, the semaphore takes the serialization task away from the users, encapsulates the implementation details, and presents a higher level instruction pair, P and V to the user. This is a significant step toward the modularization of synchronization mechanisms.

A further development of the encapsulating concept in synchronization mechanisms was the monitor concept developed in the 1970's. Extending Dahl's class module [11], Brinch Hansen and C. A. R. Hoare developed the monitor module which encapsulates the serialization mechanism, the shared state variable, the critical region, the scheduling queue and all the operations on the shared data structure [17]. For example, a monitor module for a disk file defines procedures for open, close, read, write and defines entry procedures to ensure that users use the monitor one at a time. The monitor encapsulates the implementation detail of the resources, frees the user from synchronization details, and disciplines him to use well defined operation procedures. This usually leads to better modularity and structure in the construction of operating systems.

However, there are still some problems with the monitor. For examples, there is potential deadlock in nested monitor calls [30], and the inconvenience of explicit signalling instead of automatic

signalling in certain applications. However, a properly constructed monitor can avoid deadlock in almost all cases, and an explicit signal, although requiring users to do more work, is much more efficient than automatic signalling [6]. The major problem is that a monitor does not separate the mapping process from the resulting order. Since users have to use monitors one at a time, true concurrent operations on the shared resources are not possible if the resource is encapsulated by the monitor. This is not a real problem in uniprocessor systems, but in multiprocessor systems, where true concurrency is possible, it is an important problem.

The serializer [3] was a proposed solution to the above problems. In the serializer construction, the mapping and the resulting access are separated. Processes have to enter the serializer one by one as in the monitor; the difference is that if conditions permit the entering process to access the shared resources, the process will access the resources and will leave the mapping mechanism. A new process can then enter the mapping mechanism. Therefore, more than one process can concurrently access shared resources within the serializer, whereas by definition, processes in a monitor are mutually exclusive. In addition, the serializer uses automatic signalling rather than explicit signalling.

### 3.4.2 Concepts and Issues in Distributed Access Synchronization

#### 3.4.2.1 The Simultaneity Assumption in Atomic Transaction

One generally accepted assumption is that all the actions of an atomic transaction have to be done *simultaneously* with respect to other transactions. That is, atomic transactions are considered as disjoint sets of inseparable actions. This assumption underlies many distributed access synchronization techniques such as eventcounts and sequencers. This is certainly a reasonable assumption in many situations, and our discussion in the following sections will be consistent with it. Nevertheless, we also want to point out that there are important application areas in which this assumption is unfounded and the design of the access synchronization mechanism should be different.

Situations in which this assumption is unfounded often arise from some special property of the data base, such as the property of *unordered operations*. It is defined as the following: If there are operations  $op_1, op_2, \dots, op_n$  on the data base  $B$ , the result is identical for any permutation of  $N$  operations. Noted examples are the checking accounts and inventory accounts. Financial transactions are strict atomic transactions, because money moved out from one account has to move into a receiving account or nothing is done and the accounts are still balanced. But the actions of this type of atomic transaction do not need to be done simultaneously with respect to other transactions.



Accounts are updated as the checks come in from the mail. Financial transactions have been done in a distributed and decentralized way via an unreliable communication channel (post offices) successfully for a long time without locking up the corresponding accounts in other banks during the updating of its local accounts.

#### 3.4.2.2 Transaction Ordering, Update Set and Distributed Access Synchronization

In order to guarantee the consistency of an update set and to guarantee the consistency of the results of accessing, all the operations of a transaction must access the same version over the entire update set. For example, an update set consists of three files and is in version one. A writer should be guaranteed to update all three files of version one and the entire update set becomes version two. A reader should also be guaranteed to read all three files of the same version one, although he will not affect the consistency of the update set.

This can be achieved by means of mutual exclusion among users. A typical example is the method of circulating tokens [28]. The consistency requirements can also be achieved by parallel access methods. The idea is to order all the transaction requests to an update set into a deterministic sequence in which the next request number can be determined by the previous request numbers. Since the synchronization mechanism in each member of the update set knows the next access request number, all it has to do is regulate the access to proceed as defined in the deterministic sequence. In this way, all the operations of a transaction will always bind to a single version of the update set, although the accessing of the update set is done in parallel. For example, if the transaction request sequence is the natural numbers and the update set is initialized as version one, the synchronization mechanism in each member of update set will only allow transaction one to access version one. After the accessing, version one becomes version two and it is ready for transaction two. Although each member of the update set is accessed in parallel and the change over of version numbers is done asynchronously among members of the update set, transaction X will always operate on version X. A typical example is eventcounts and sequencers [36].

It is worthwhile to point out that the transaction request sequence must be a deterministic sequence defined as above. An arbitrary total ordering is neither necessary nor sufficient for distributed access synchronization. To show it is not necessary, we first demonstrate that no transaction ordering is necessary. Assume that there is an unique naming mechanism such as the process name plus a local transaction request count, so that all the transaction requests will be uniquely named. To achieve the effect of mutual exclusion, all we have to do is randomly pick one of those uniquely named requests by means of communication protocols. For example, users can

directly send their uniquely named requests to all the members of the update set. These requests will first be put into local queues of the members of the update set. The synchronization mechanism of the members of the update set will exchange the information about their local queues. Once they find out that there are some requests across all the queues, they can randomly pick one (such as by the use of pseudo-random number generators with identical seeds). It is true that any arbitrary total ordering plus a communication protocol to find out who is at the head of the request queue according to the arbitrary total ordering can also provide the mutual exclusion effect.<sup>3</sup> The point is that an arbitrary total ordering is not a necessary building block in constructing a distributed synchronization mechanism. So far as guaranteeing the consistency requirement is concerned, randomly picking one in the request queue is just as good as picking the head of the request queue defined by the arbitrary total ordering.

An arbitrary total ordering, unlike a deterministic sequence, is also insufficient to guide the parallel accessing of an update set. The primary reason is that a member synchronization mechanism can not determine the next element in the arbitrary total ordering from previous elements. To illustrate the point, we use a "ticket machine" for all the update sets in the system. The "ticket machine" will label all the transaction requests as a sequence of natural numbers. Although the transaction request number sequence for any update set is a monotonically increasing integer series (total ordering), the interval between any two consecutive requests for any update set will depend on the users' run time choice among different update sets. Such an interval is, in general, unpredictable for the update set synchronization mechanism. For example, transaction one and three may go to, say, update set A, while transaction two and four may go to update set B. Now assume that update set B has just been created and it is in version one. When a member of update set B receives transaction four, it will have a difficult time deciding whether to accept transaction four. If it decides to wait to see whether there are lower number transactions, it may wait forever. If it decides not to wait and to accept transaction four, then it is possible that transaction three will arrive later. Some members of the update set will be modified by transaction three while this place will be modified by transaction four and the update set results in an inconsistent state. When the interval between transaction requests is unknown, one can never be sure whether accepting a transaction will result in a consistent state or not.

---

<sup>3</sup>A typical example is Lamport's logical clocks which will be discussed in the following section.

### 3.4.3 Overview of Distributed Access Synchronization Techniques

#### 3.4.3.1 Logical Clocks

The logical clock system [26] is a set of counters synchronized with respect to observable events. Each process has a local counter which is incremented by one with respect to each internal event. The value of a counter is called logical time, and each message sent is stamped by the logical time. Upon receiving a message, the time stamp of the message is compared with the local logical time and the counter is set to an integer value which is greater than or equal to the old original internal value and greater than the time value stamped on the message. The logical time plus priority, which artificially assign a precedence to time stamps with the same value, defines an arbitrary total ordering. Such an arbitrary total ordering, unlike a deterministic sequence, is neither necessary nor sufficient for distributed access synchronization.

As any arbitrary total ordering, it can, however, provide the effect of mutual exclusion if additional communication protocol is used to find out the earliest request defined by the total ordering and let it proceed first. Under the assumption that some type of interlock communication protocol is used to guarantee messages are received in the same order as they are sent, the communication protocol for a single resource or update set is as follows: 1) when a process requests a resource, it will send its request to all other processes and put his own request in its local request queue; 2) all the process receiving the request will acknowledge the sender and put the request in their local queues; 3) if a process has received messages from all other processes and its own request is the earliest compared with time stamps of all received messages( including requests ), then its request is the earliest and it can use the resource; 3) when it finishes, it will remove its request from its local queue and send a message to inform all other processes that it is done; 5) upon receipt of the message, all other processes will remove the request from their queues.

The same algorithm can be easily extended to the multiple update set case. The only change is to add a parameter to the request to identify which update set that it is requesting. When a process receives at least one message from all other processes and still finds its own request for an update set is the earliest in logical time, it can be sure that its request for that update set is the earliest. Because if there is any other process making a request earlier, then under the assumption that messages are received in the same order as they sent, this request should already be received and its own request cannot be the earliest. Since a logical counter is merely a integer variable, a more modular way without much additional cost is to use one set of clocks for each update set and perform the basic algorithm independently for each update set.

Since counters are maintained by users, and since a user may need as many counters as the update sets he accesses (or his counter is shared by all the update sets), single user failure may affect multiple transaction orders. Since every correct access requires the correct co-operation of all the processes, the failure of any process may cause multiple failures in the system.

Since the state of the shared resource depends on activities of all the processes in the system, it does not meet the context independent requirement of modularity. Since all the synchronization tasks are carried out by the user processes, it does not meet the practicability requirement. So far as the performance issues are concerned, the logical clock system is a mutual exclusion method with respect to each update set. Therefore it is low in concurrency. The logical clocks system has no special provision for priority expression, but it favors slow clocks. This is because transactions with smaller value time stamps will be ahead of the access queues. Finally, it has an very high interprocess communication load compared with most known schemes.

#### 3.4.3.2 Physical Clocks

Lamport has proposed the use of a set of distributed and synchronized real time clocks as a distributed synchronization mechanism [26]. In the same paper, an algorithm to synchronize distributed physical clocks is also given.

Time stamping plus priority, which assigns an artificial precedence to time stamps with the same value, define a total ordering of transaction requests for any update set. Since events occur at random times, the interval between any two consecutive requests is unpredictable. Therefore, it is neither necessary nor sufficient for distributed access synchronization. As any arbitrary total ordering, it can provide the effect of mutual exclusion if additional communication protocol is introduced to find out which one is the earliest request defined by the total ordering, and allow the earliest request to proceed first. For example, the communication protocol for logical clocks can be used. The most unique feature of physical clocks system is that the system implements a first request first serve policy. The reliability, modularity, practicability and performance is similar to to the logical clocks.

#### 3.4.3.3 Circulating Tokens

A simple way to obtain mutual exclusion among users is to pass a token, which represent the right to access, around all the potential users of the shared resource. The one who currently has the token can access the shared resource. When he finishes, he passes the token to the next user. The method of establishing a token and a communication(virtual) ring for each update set was first proposed by Le Lann [28].

Since a user can access multiple resources, virtual rings intersect. The failure of a single user may affect more than one ring. Since there is no protection at the data bases, when a user fails during the updating phase, the data bases will become inconsistent and may not be recoverable. That is, this method does not have special provision to support fault tolerance or fault recovery. However, the connectivity of the communication ring and the uniqueness of the token on each ring can be made secure by embedding a high level communication protocol into each user [28].

Since the knowledge of logical dependence between distributed resources is embedded in the users access codes rather than encapsulated by the synchronization mechanism, this method does not meet the context independent requirement of modularity. For example, an update set consists of ten distributed and replicated files. A new user is added to the system, if he somehow, by mistake, only updates nine of them, then the update set will be in an inconsistent state. Since we can not verify the property about the update set without making assumptions about the users, the context independent requirement not meet. To make it context independent, a update routine, which either updates all ten of them or updates nothing, should be included as part of the synchronization mechanism, and all the updating must be done via this routine.

Compared with the logical or physical clocks system, this method is easier to use at the user level. This method is fair to all the users, because it is a round-robin algorithm. It does not support other priority policies. The circulating tokens provide the synchronization by mutual exclusion among users, therefore, it is low in concurrency. The communication load generated by this method has characteristics which is the same as a bus polling scheme for it is a round-robin polling method, that is, it depends on the ratio of active users to the population of the ring. If the accessing pattern is bursty, users may uselessly pass tokens most of the time. However, it generates less communication load than that of logical clocks system.

#### 3.4.3.4 Sequencers

LeLann proposed the use of a circulating sequencer in [28, 29] as distributed synchronization mechanism. Since he uses only one sequencer which does not bind to any update set, the resulting order is an arbitrary total ordering which, by itself, is neither sufficient nor necessary for distributed access synchronization. In order to obtain the effect of mutual exclusion, he uses a two phase commit communication protocol at the update set. [29]

Another way to use a sequencer is to assign each sequencer to an update set. Since each update set now has its own sequencer, the resulting transaction order is the natural number series. The synchronization processes in each member resource will regulate the access order accordingly. For

example, assume that when the update set is created, it is labeled as version one. All the members of version one will accept access of transaction one only; after it is accessed by transaction one, it becomes version two and will only accept transaction two only. Although resources are managed individually, transactions will be always done as soon as possible and the update set will be always consistent. This is because each transaction will only access a single version which equals to its transaction number.

The reliability, modularity and practicability of this method are similar to that of circulating tokens. But it has higher concurrency. A special problem associated with this method is that each local member of the update set may need a large buffer to hold those outstanding requests. For example, assume that the update set is version three, but somehow access requests of transaction four, five and six arrive first. They have to be held in the buffer and wait for transaction three to arrive and proceed first.

#### 3.4.3.5 Eventcounts and Sequencers

Reed and Kanodia developed eventcounts and sequencers as basic mechanisms for communicating the state of shared resources and for serializing the conflicting concurrent requests [36]. Four primitive operations are as follows: 1) Ticket(<sequencer>) provides a monotonically increasing integer to serially order concurrent requests; 2) Advance(<Event>) increments the event variable by one; 3) Read(<Event>) gives the current value of the Event variable; 4) Await(<Event>, <Value>) blocks the calling process until the Event variable is greater than or equal to the <value> specified. Eventcounts and sequencers, unlike the semaphores, do the serialization and access control separately. This separation provides the user with more detailed control over the ordering of concurrent activities.

In distributed computer systems, the importance of this method is that these four primitives represent distributed mechanisms by which we can encode the information of the logical relations among distributed data bases and processes. For example, we have a set of update-dependent files, file1, file2, and file3, distributed across three computers. When a user needs to access these files he will first get a ticket from a sequencer associated with the set of files. He then "watches" the messages from the eventcounts associated with each file. If any eventcount has a number equal to his ticket number, he accesses the corresponding file. When he finishes with that file, the eventcount associated with that file will be incremented by one. Although a user will be served as soon as one of the files is ready, the update set will always be consistent, because he is guaranteed access to a single version (equal to his ticket number) of these three distributed files.

This method has better fault confinement properties than circulating tokens/sequencers. This is because a user interacts with the resource only when he needs it. A user can fail without affecting others who just share the resources with him, if he is not currently accessing any common resources. They are, however, similar in other reliability aspects. For example, if a user fails during the updating phase, the data bases may not be recoverable. This is because the data bases are not encapsulated and protected. Therefore, this is still a primitive method in the sense that it does not have special provision to support fault detection, tolerance and recovery.

So far as the context independence requirement of modularity is concerned, this method is similar to the semaphores in the shared primary memory system. It provides powerful primitives to the users who now need only have a ticket and then wait for the message. However, it is not an elaborate method like monitors in a shared primary memory computer system. It does not encapsulate the internal inter-dependence among distributed resources. The operations on shared resources are also not disciplined.

Eventcounts and sequencers provide better concurrency than that of circulating tokens. Eventcounts and sequencers also tends to produce less communication traffic than does circulating tokens/sequencers, if the access pattern is bursty. This is because there is communication only when someone is using the shared resources. The use of eventcounts also eliminates the need for buffers at the resource management process. This is because each user will hold his request until his turn comes. Eventcounts and sequencers do not have special provision to support access policies such as priority, but process in a fair first-come-first-serve fashion.

This method is the easiest to use at the user level, and is not difficult to support at the operating system level. In summary, we find this method is the best among those in our review, and yet it is, like semaphores in a shared primary memory computer system, relatively primitive.

### 3.5 Conclusion

When we examine distributed access synchronization techniques, it seems as if the development pattern of the multi-programming era is being repeated in the context of distributed computer system. The logical clocks method corresponds to the pure software solution in the sense that synchronization is obtained by means of directly exchanging user state information. Eventcounts and sequencers correspond to semaphores in the sense that synchronization primitives are used to provide the basic serialization and to represent shared resource state information.

None of the mechanisms currently proposed completely encapsulate the resource, free the user from synchronization details, discipline the users to use only the allowed operations, and support various reliability measures. The next phase of this research will be directed at development of such a mechanism.



## 4. ICON - Archons Simulation Facility

### 4.1 Introduction

Icon is a facility for designing and testing decentralized algorithms. It is a tool to be used in the implementation of Archons. Since Archons embodies completely decentralized control at all levels, we need a convenient facility for experimenting with different solutions to problems in a decentralized framework. This includes the areas of synchronization, resource management, scheduling of processors and processes, as well as the creation and deletion of processes. Icon addresses itself to the implementation of algorithms in the above areas.

The philosophy behind Icon is to provide a rational back-drop for decisions regarding the mechanisms which should be implemented in Archons. Archons is the kernel of an operating system (in the same sense as Hydra), which will provide various *mechanisms* as primitives. These mechanisms can be used to implement any policies desired by the user. One reasonable way of designing mechanisms is to try various algorithms to get a precise idea of the kernel facilities which will be needed to support algorithms of this nature. This philosophy requires the existence of a facility for conveniently experimenting with different decentralized algorithms.

Icon can be viewed as a simulation of Archons. If we look at simulation as a hierarchy, in which the lowest levels are closely bound to the system characteristics (such as registers, buffers, protocols, delays) and the higher levels look only at the abstracted view of the system, then Icon fits into the highest level simulation of Archons. Icon is a functional simulation of Archons, which means that it provides the same functionality as Archons, but abstracts away the implementation details.

### 4.2 Scope of Icon

The primary contribution of Icon is to provide a convenient facility for testing decentralized algorithms. In a broader sense, however, it has many other contributions to make to Archons. It is the starting point for the implementation of Archons, as it will be using several of the concepts which will be used in Archons. This will be discussed in detail in a later section.

#### 4.2.1 Primary Application Area

The primary application area of Icon is decentralized algorithm design. Consider some of following problems for which Icon could be used.

1. **decentralized synchronization** : Several different processes need atomic actions, i.e., either all of them do a particular action or none of them do. For example, either all missiles are fired or none are. These processes do not have a global view of the system state. Any synchronization they can perform is by explicit message passing. What algorithms would be appropriate?
2. **decentralized creation, deletion and scheduling of processes**: When a new process has to be created, deleted or scheduled, how do the many executive instantiations co-operate to decide which one of them will actually perform the task? How would load-sharing be done in this system?

Experimentation in the area of decentralized or probabilistic algorithm design would consist of determining whether the algorithm works or not, and then measuring its performance.

To summarise the scope of Icon:

- Initially, it will provide an environment for testing out algorithms for decentralized resource management. The experience derived from this will help in building a first order implementation of Archons.
- At all stages of the development of Archons, the Icon facility will be available to conveniently and efficiently try out any new algorithms or ideas in decentralized decision-making.

#### 4.2.2 Other Contributions

Icon can be viewed as the first step in the implementation of Archons. It embodies many of the concepts of Archons on a much smaller scale. Even though its primary aim is to experiment with decentralized algorithms, it will contribute not only in terms of the ideas for the kernel primitives, but also in the implementation approach and ideas for Archons.

Archons is a research test-bed, which has completely decentralized control. The aim of the system is two-fold. Firstly, it will serve as an example of an actual system which has been built using decentralized control at all levels. Secondly, it will be a research facility for experimenting with algorithms and techniques for the decentralized management of resources.

Icon is directly related to Archons in two ways. Firstly, it will be emulating the environment of Archons. Secondly, it is a mini test-bed in its own right. It will allow some experience in the design of

an effective experimenter interface and allow us to explore the facilities needed for experimentation in the area of decentralized decision-making. It should also be possible to develop the system incrementally. By this we mean that we should be able to develop very rudimentary facilities, which allow for the testing of more complex algorithms and development tools. These tools can then be implemented in the system. This creates a better, more hospitable environment in which more complex tasks can be carried out.

In order to better appreciate the issues involved, it may be a good idea to quickly run through the key features of Archons. The details are given in other portions of this report. The aim of this project is to conduct research in the area of "Distributed Computer Systems."

These systems are characterised by multilateral resource management; where several controllers manage each resource on a democratic, *equality* basis, and most resources are multilaterally managed. Distributed systems have a large number of controllers. The memory is completely partitioned and all communication is by means of messages only. There is no unique, globally available system state.

The result of this is that such systems are characterised by incomplete information and poor signal observability. They have high latency and low synchronization; there is only a certain probability that information will be available when needed, and in the required order. Management for this environment requires completely different algorithms and techniques.

The Archons system has an interconnection mechanism consisting of very powerful Communication and Control Units. These units run most of the executive and are connected together by several buses. Each CCU will also be connected to its own user processor. The message subsystem will consist of buses and their allocation and protocols at the lower levels, and the executive primitives to handle communications at a higher level.<sup>4</sup>

The operating system ArchOS being built on Archons will be process oriented, where all system and user programs will be structured as a large number of communicating sequential processes executing in parallel. We will need to experiment with the structuring of processes in a distributed environment.

Several features of Icon resemble Archons and this adds to the utility of Icon as a first step in the

---

<sup>4</sup>These levels are logical, not implementation dependent - e.g. each level could be implemented in hardware or software

implementation of Archons. Firstly, Icon emulates the environment of Archons, so that it is effective as a simulation facility for decentralized algorithms. Secondly, like Archons it is also a test-bed, and will allow experimentation in the design of an effective user interface, as well as tools for useful experimentation. Thirdly, the system will have to be modular and expandable so that new features can be added easily, both in terms of measurement facilities as well as experimentation at different levels of the Archons system. Fourthly, Icon is also process oriented and can provide some insight into the advantages and disadvantages of this approach.

The advantages which accrue from this approach are:

1. **Ease of development** : It is easier to handle smaller problems at a time, and slowly build up to getting better algorithms and techniques for solving problems.
2. **Modularity** : Since we will be extending the system incrementally, we will be forced to adopt a modular approach to system design. It will be easier to ensure modularity, which is one of the high priority goals of the project .
3. **Hospitable User Interface** : We will be working on the user (experimenter) interface we develop for further experimentation. We will be aware of the problems of the environment we provide. This will enable us to provide a more hospitable environment. If this is not the case, the user environment frequently suffers.

### 4.3 General Requirements for Icon

Since Icon will be used primarily for trying out algorithms in decentralized management for Archons, it must incorporate the following features:

1. **Environment of Archons**: Icon should be able to abstract the environment that Archons provides, at the level at which experimentation is to be carried out. For example, if the user wishes to try out synchronization algorithms, he should feel that he has a large number of processes, which execute only local code, and communicate by means of messages, available to him. This would be characteristic of the Archons environment at that level.
2. **Ease of Experimentation** : The experimenter should be able to get the information he needs, change parameters in various experiments, and have a friendly environment.
  - a. **Data Collection**: The user should be able to measure parameters of interest, and collect the data that he requires. For example, it should be possible for him to determine the delays in sending messages, or the percentage of messages which are lost.
  - b. **Changing System Parameters**: The user should be able to test algorithms under varying system conditions, such as under different load conditions, or by assigning bus priorities. It should be possible to change system parameters also, such as the distribution pattern of time delays in the message passing mechanism.

c. A Well Defined User Interface: This will be the window through which the experimenter can monitor, control and change the system for his experiment. It is important that the interface is convenient to use.

**3. Expandability and Adaptability :** The system should be able to incorporate new features without any great difficulty . It should be possible to adapt Icon suitably for small deviations in experiments for which it was not designed. For example, if we wish to experiment at a different level of Archons, we should be able to do so by just changing some of the primitives which are available, or by including a few more modules in the system.

## 4.4 Analysis of Requirements

This section contains a discussion of the impact of the requirements for Icon as dictated by its application. It describes the features which Icon should have if it is to fulfil its goal.

### 4.4.1 Environment

Before we can attempt to describe the environment of Icon, we must have a fair idea of the environment of Archons. We can then abstract the detailed structure and provide a first order approximation at the level of interest.

#### 4.4.1.1 Archons

The environment of Archons, as visible to the lowest level of the executive kernel, consists of a message passing mechanism, to which the user processors are connected.

The message passing mechanism consists of powerful Communication and Control Units, which are connected together by means of several buses. Message sending over these buses is cheap. The buses can be allocated according to several different policies; e.g., send the message on any available bus, or reserve certain buses for high priority system messages only.

Messages can take one of several alternative paths. This results in different messages taking different amounts of time, and arriving out of the sequence in which they were sent. The time taken by a particular message depends on the state of the system at that time.

Each Communication and Control Unit is connected to a user processor. Each of these processors has its own private memory, which it does not directly share with any other processor. All communication is entirely message based. There are no central tables or shared memory in the system, so that it is not possible for each processor to have the same consistent view of the system.

Let us now look at Archons as visible to the higher levels of the kernel, and to the applications programmer. The system now looks like a set of communicating processes. Each process runs its own code, on its own data, both being local to the process. It does not share any code, data or other information with any other process. All communication is by explicit message passing, and sharing is indirect by sending copies.

The processes can use primitives like *send* or *receive* for their message passing. The actual message passing mechanism is abstracted away, and not visible at this level. However, the effect of the message passing mechanism can be felt: e.g., messages may be randomly delayed or lost.

In addition to the basic message passing primitives, the processes will have access to other primitives, depending on the level at which the system is viewed. For example, at the lowest executive level only *send* and *receive* may be available; but at higher levels, *CreateProcess*, *DeleteProcess* and various synchronization primitives may also be available.

#### 4.4.1.2 Icon

The immediate concern for Icon is to be able to abstract the environment of the lowest level of the executive kernel, so that synchronization and resource management algorithms can be tried out. It would be desirable to be able to abstract higher executive levels at very low incremental cost, so that other algorithms can also be tried out.

In achieving the characteristics of the message passing mechanism of Archons, it is not essential to have an explicit representation for details like the time spent in the CCU queue, or the number of times a message had to be retransmitted. When we are studying the design of management algorithms, the characteristics of the message passing mechanism are needed at a higher level of abstraction.

The Icon message passing system need only be a facility which can take a message from the sender process and deliver it to the receiver process, after having injected a random time delay. It should also randomly prevent some messages from reaching their destination. This time delay will be able to account for various attributes of the message passing environment, such as delayed messages, messages taking different routes and arriving out of sequence, and lost messages.

The message passing facility of Icon makes the primitives *send* and *receive* available to the user. The characteristics of this *send* or *receive* will be very close to that of a message sent on the Archons system.

Icon will also provide a large number of processes, which abstract the user processors seen by the lowest kernel level. This enables the easy extension of the system for use at higher levels of abstraction. These processes will execute their own private code and have private data. They have access to the message passing system by using the primitives *send* and *receive*. These processes will be running the code supplied by the user.

#### **4.4.2 Ease of Implementation**

The environment of Icon should help the user in trying out different algorithms.

##### **4.4.2.1 A Powerful User Interface**

The user should be able to start and stop the experiments. To start, he will need to create processes and provide the code which these processes can execute. He requires the system to provide him with a *create* primitive.

He should have access to the local data of all system and user processes, so that he can determine the state of the system and monitor the experiments. He should be able to access various statistical modules, so that he can change parameters such as load factor or delay. This means that the statistical packages should provide him with primitives for altering the system parameters.

##### **4.4.2.2 Data Collection**

As any experiment proceeds, the system processes should record various statistical data. This should then be available to the user.

##### **4.4.2.3 Statistical Functions**

Statistical functions like random delay generation should be implemented by separate processes. The user should have access to primitives by which he can alter these functions and their parameters.

#### **4.4.3 Expandability and Adaptability**

It should be possible to try out algorithms at different levels of the system executive. This is made possible by giving Icon a very general structure of communicating processes. The only difference from one level to the next would be in terms of the primitives available at that level. In order to incorporate these primitives, more processes can be created at the lowest system level of Icon. These processes, like the message passing system, will implement these primitives. This structure also helps in finding the set of primitives which may be necessary and sufficient at a particular level of the system.

If there is a need to introduce more statistical functions, this can be done by creating processes to achieve that functionality at the system level.

The power of Icon is in the fact that it consists of a large number of processes at all levels, each implementing a specific function. This makes it very modular and consequently adaptable over a wide range of algorithms, by simply adding more processes. It should be possible to use Icon at different stages of the development of Archons, because primitives can be added and changed very easily.

## 4.5 Specific Requirements for Icon

From the preceding discussion, it is clear that the only major requirement in the implementation environment of Icon is the availability of a language which allows the easy implementation of a large number of concurrent processes. It should be possible to have either real concurrent processes or to conveniently simulate this concurrency. There should also be a facility for creating new processes.

## 4.6 System Structure

The structure of Icon is very uniform. It consists of a large number of processes. These processes can be logically divided into three categories:

- **System Processes** : These processes implement the primitives which are used by the user processes and the experimenter process. They also provide the structure of the user processes.
- **User Processes** : User processes run the actual code of the algorithms we wish to experiment with.
- **Experimenter Process** : This is a highly privileged process which provides the interface to the outside world.

The different classes of processes can be pictured as forming concentric layers. The system processes form the core of the system, on which the user processes reside. The experimenter process forms the outermost ring, and is in contact with the outside world.

The detailed functionality of these processes is discussed in the following sections.



#### 4.6.1 System Processes

**User Primitives :** The user processes assume that they have access to a message passing mechanism. The system processes implement the primitives *send* and *receive* used by the user processes.

On a *send* command, the message passing process first accepts the message which is to be sent, and notes the time. It generates a random delay to represent message transition time. At the end of this time interval, it puts the message in the queue for the receiver process.

On getting the *receive* command, the message leaves the queue and is accessible to the user process.

**Experimenter Primitives :** The *create* primitive is used by the experimenter for creating a user process. The purpose of *create* is to make the new process known to the system. The name of the process is entered in the system tables, and a queue is created for it, in which messages for that process will reside.

The experimenter is also provided with various statistical primitives. Each statistical function is implemented by a different process. Each of these processes will provide primitives with which the experimenter can modify the parameters of the system.

**Structure for User Processes :** One of the system processes will define a *template* for a user process. This will provide the very basic structure of the process. Each user process will have a notion only of private code, private data, and the primitives of *send* and *receive*, which give it access to a message passing system for communication purposes. Each process will be associated with a system maintained queue, in which messages for that process are stored.

When the experimenter creates a user process, he will have a process of this class created for him. He can supply the code which each process must use.

#### 4.6.2 User Processes

As already explained, each user process is characterised by :

- Private code
- Private data

- A system queue in which messages for it are kept
- Access to a message passing system by using the primitives *send* and *receive*
- The ability to be created by the system on a command from the experimenter
- The basic structure of the process is provided by the system
- The experimenter provides the precise code or the algorithm which has to be executed
- It can be started and stopped by the experimenter.

#### 4.6.3 Experimenter Process

The experimenter process is a highly privileged process. It has access to all the primitives provided by the system processes. It can monitor an experiment by reading the tables kept in the local data of the system processes. It can start, stop and monitor an experiment. It can create as many user processes as it needs, and provide them with the particular code that it wishes to experiment with.

The different classes of functions performed by the experimenter are :

- Starting and stopping an experiment
- Creating processes and providing them the relevant code
- Monitoring the experiment by reading the local data of the system processes
- Changing system parameters and running experiments under different conditions

#### 4.7 User Functional View

The user has a system available to him which is capable of concurrently running a large number of processes. These processes have only local data, and no shared data. They communicate with each other by passing messages.

The user processes form a *class* in the programming languages sense. They have a *template* which is specified by the system processes. The user provides the code for these processes, by supplying the algorithm which he wants executed. He can create as many such processes as he likes by using the *create* command.

The user has access to several statistical packages, so that he can change various parameters in the system. Since the user is himself a privileged process, he can, at any stage, ask for the state of the

system, including the private data of the rest of the system processes. The user has to start, stop and monitor his experiments. He has access to any information in the system that he may desire.

The user will be able to experiment with different *levels* of Archons, because it is easy to emulate the other higher levels by adding the appropriate primitives for that level.

#### 4.8 Implementation Environment : Cm\* versus Simula

We need to decide whether Icon should be implemented in a multiprocessor environment like Cm\*, or in a system simulation language like Simula. (Note : Simulation as used here is not in the sense of modelling or other performance evaluation type of simulation. It is a functional simulation of a system. It provides a similar environment to a user, so that he can test out some ideas.)

The two environments can be compared on the following scales :

1. **Real concurrency versus simulated concurrency** : Eventually, all our algorithms must run in a truly concurrent environment. If we simulate concurrency, it is possible that we may not account for some problems which are inherent in a truly concurrent environment. Depending on the level of abstraction we are trying to emulate, it may be impossible for some emulation tasks to give any meaningful results on a uniprocessor. However, if the emulation is at a sufficiently high level, then the use of concurrent processes in Simula can still give meaningful results.
2. **Software Tools** : Any task done on Cm\* will have a much longer development time than on the PDP-10, because of a lack of good software tools.

In this project, we decided that the initial experimentation would be done in Simula, as the task becomes considerably easier. Besides, the purpose of Icon is to try out decentralized algorithms, as opposed to obtaining exact timing information, and so the only concept needed is that of concurrent processes, which Simula provides. Information such as the time taken for messages and bus protocols is not needed here and so Simula should be able to provide a functional simulation at this level of abstraction.

The strategy at this point is to develop Icon and to try out some algorithms and ideas on it. After that, we can move over to Cm\*. The considerations in moving over from Simula to Cm\* are the following.

1. We do not wish to spend too much effort on Simula, because we will be working on Cm\* finally. It will be unnecessary to implement the entire system twice.
2. There are fewer development tools on Cm\* and it takes a longer time to try out new ideas

in that environment. It would be beneficial to work in a friendlier environment, while we are still in the process of defining the operating system more clearly.

During the initial stages of the development of Archons, we are more unsure of the functionality of the system. At this stage, we are more likely to want frequent changes in the system environment. We would like to work with Simula in the beginning, as it will be easier to try out new ideas and make frequent changes. As soon as we have some definite ideas that we want to implement, we will move over to Cm\*. Of course, these lines are not very exact. The system on Cm\* will not be fixed for all time to come, but it is hoped that the Cm\* implementation will start when the system design is somewhat more stable.

## 5. ArchOS: An Operating System for Archons

### 5.1 Introduction

ArchOS is an operating system being designed for the Archons distributed computer system. The reader is assumed to be familiar with the highest level system architecture of Archons and with the motivations and objectives for its design.

The design of ArchOS is intimately entwined with the design of Archons. Archons is intended to provide a collection of powerful *mechanisms* to support the *policies* established by ArchOS. The goals and objectives of the ArchOS research effort are discussed in the following section. That section also explains the ways in which the ArchOS and Archons design tasks interact.

ArchOS is an experimental operating system. It is intended to be the vehicle for research in the areas of decentralized control at the operating system level and hardware/software tradeoffs in operating system implementation. Equally important, ArchOS will be an experiment in the design of the user visible machine of a distributed computer system. This report is concerned with this latter aspect of the ArchOS research effort.

An operating system user visible machine actually consists of two distinct but related abstract machines: the operator's abstract machine (OAM), and the program's abstract machine (PAM). The OAM is essentially defined by the command and control language provided by the operating system for loading and executing programs, monitoring, measuring, and manipulating running programs, and so on. The PAM is the abstract, conceptual machine seen by an application program as it executes. The ArchOS research effort is concerned with the design of both the OAMs and the PAMs of distributed computer systems. This report concentrates on the ArchOS PAM design.

In designing the program's abstract machine that is to be supported by an operating system, two distinct but related design activities are involved. First it is necessary to develop and define the concepts embodied in the abstract machine. Then those concepts must be reflected in a set of facilities which are available to the program and make it possible to solve problems using the abstract machine. This report presents some of the initial results of the investigation of the major issues involved in the development of the concepts for the ArchOS PAM. Each design decision is explained in light of the spectrum of choices available on each issue, and is compared with the decisions made in a number of extant operating systems. The development of the ArchOS PAM concepts and the design of the ArchOS facilities reflecting these concepts is still in progress. The complete results of this design effort will be reported at a later time.

## 5.2 Goals and Objectives

The ArchOS research effort is concerned with all aspects of the design of operating systems for distributed computer systems. However, the primary goals and objectives of the project are fourfold:

1. Experiment with decentralized resource management and system control techniques at the operating system level.
2. Investigate the impact of operating system design on computing system architecture. In particular, investigate the ways in which a highly decentralized operating system and the design of the user visible machine affect the choice of mechanisms and features provided by the hardware.
3. Investigate the impact of computing system architecture on operating system design. In particular, investigate the ways in which a distributed computer architecture, having lower system wide signal observability than traditional central shared memory machines, affects the design of the operating system, both at the abstract machine level and in the implementation of the operating system itself.
4. Experiment with the design of user visible machines which provide convenient environments for employing decentralized resource management and control techniques at the application level.

The first objective listed above is in many ways the primary *raison d'être* of ArchOS. The entire Archons project is founded on the hypothesis that various types of decentralized control offer significant potential benefits, such as improved robustness and modularity and perhaps even improved performance, under certain conditions [23]. The design and implementation of ArchOS will be viewed as an experiment in building a highly decentralized operating system. It is expected that portions of ArchOS will be implemented and reimplemented in a variety of different ways. This will provide some feel for the spectrum of decentralized control techniques which are appropriate for implementing operating system facilities. It will also provide a basis for the comparison of these techniques, both among themselves and with respect to more traditional, centralized solutions to control problems.

The second objective listed above deals with one of the ways in which the ArchOS and Archons design activities strongly interact. It deals with the impact of ArchOS design decisions on the choice and design of the mechanisms provided by Archons. The ArchOS design impacts that of Archons at two levels. First, Archons must provide mechanisms to support the decentralized control algorithms used in the implementation of ArchOS. It is expected that the success of many decentralized control techniques will hinge on novel hardware/software tradeoffs in the design of special mechanisms to support those techniques. Since ArchOS implementation will be the subject of active

experimentation, the Archons mechanisms will have to be flexible enough and complete enough to support the wide variety of different decentralized control algorithms applied to each system control problem.

The second way in which the ArchOS design impacts that of Archons is that Archons must provide mechanisms to support the implementation of both the OAM and the PAM portions of the ArchOS user visible machine. As an aid to maintaining the defined semantics of the ArchOS user visible machine when it is implemented on Archons, it is necessary to have a close correspondence between the conceptual machine and the underlying computing system architecture [20]. Thus, the mechanisms provided by Archons must embody the same or very nearly the same concepts found in the ArchOS user visible machine. Since the ArchOS user visible machine is itself highly experimental, the Archons mechanisms must be flexible enough and complete enough to accommodate the evolving concepts and facilities.

One other aspect of this second objective, to investigate the impact of operating system design on computing system architecture, deals with the way in which ArchOS controls the Application Processors (APs) of the distributed computer system. ArchOS is being designed to run on the Control and Communication Units (CCUs) of Archons, which are separate processors from the APs. In other words, ArchOS will attempt, as much as possible, to exert *external* control over the APs. ArchOS will be an experiment to determine the degree to which it is possible for an external operating system to control an AP. The goal will be to determine the architectural characteristics of an AP which make it amenable to external control. It will then be possible to design and build processors which possess those characteristics and hence can more readily be incorporated in distributed computer system architectures.

The third objective of the ArchOS research effort, to investigate the impact of computing system architecture on operating system design, deals with another way in which the ArchOS and Archons design activities strongly interact. It deals with the impact of the Archons architecture and mechanisms on the design of ArchOS, both at the level of the user visible machine and in its implementation. In the implementation of ArchOS, the Archons distributed computer architecture with its lower system wide signal observability and its special mechanisms to support decentralized control techniques has a profound effect. It is expected that the provision of certain support mechanisms will make it possible to use various decentralized control techniques which would otherwise have been impractical. On the other hand, lower signal observability between the distributed components of Archons, as well as the components of ArchOS, will result in a fragmented, inaccurate, and incomplete view of the system state [23]. In this respect the ArchOS research effort

will concern itself with the impact of poor signal observability on the design and implementation of "best effort" system control techniques.

The Archons distributed computing system architecture is designed for improved robustness, modularity, and performance. One of the purposes of the ArchOS research effort will be to investigate the ways in which this architecture and its associated advantages can be reflected in the design of the ArchOS user visible machine. In particular, it is expected that the program's abstract machine design will be strongly affected by the parallel execution of the operating system and the application programs on their respective processors. This fundamental parallelism will change the relative economics of many program level operations. The ArchOS user visible machine should reflect these changes and encourage the user to experiment with new programming and structuring techniques which are commensurate with the new programming economics.

At the same time, the lower signal observability inherent in the Archons architecture will also affect the design of the user visible machine. It is expected that this will primarily manifest itself in the form of restrictions on the possible semantics of proposed interprocess communication mechanisms. The ArchOS research project will be concerned with evaluating the significance and extent of these effects.

One other way in which ArchOS will be the vehicle for investigating the impact of computing system architecture on operating system design is in the area of external control of application processors. Existing application processors were not designed to be controlled externally. Their architectures and control mechanisms will have a strong effect upon the division of labor between the AP resident portions of ArchOS and the CCU resident portions. One goal of the ArchOS implementation will be to minimize the amount of AP resident operating system code, and the amount of work performed by that code. By supporting a variety of extant processors as APs it will be possible to evaluate and compare the various architectures and control mechanisms as to their effect on external control techniques in operating systems.

Finally, the fourth objective of the ArchOS research effort is to experiment with the design of user visible machines which support decentralized resource management and control techniques at the application level. In this regard, the fundamental question to be investigated is whether the *potential* benefits of distributed computer systems can in fact be made available and *realized* at the application level. The ArchOS user visible machine should encourage application programmers to employ decentralized control techniques, rather than penalize them for doing so. The true test of the Archons project hypothesis regarding the potential benefits of decentralized control will be in the



evaluation of the robustness, modularity, and performance of highly decentralized applications running on Archons.

### 5.3 ArchOS PAM Concepts: Issues and Design Decisions

There are many goals and objectives for the ArchOS research effort, as outlined in the preceding section. However, the guiding purpose of the project is, of course, to build an operating system for the Archons distributed computer system. To this end, the design of ArchOS is being developed, as much as possible, in a top down fashion. The design will proceed in two main stages. First, the ArchOS user visible machine will be designed. Then the internal structure and implementation scheme of ArchOS itself will be developed.

The first stage, the design of the ArchOS user visible machine, itself consists of two main subtasks. The first task is to design the program's abstract machine. Then the operator's abstract machine can be designed to allow the user to manipulate, monitor, and measure programs in terms of the concepts and facilities of the PAM. This section deals with the design of the ArchOS PAM. The design of the OAM is still in progress and will be reported at a later time.

The design of the program's abstract machine consists of two phases. First it is necessary to develop and define the *concepts* which are to be embodied in the abstract machine. This section presents some preliminary results of this initial part of the ArchOS PAM design. The second phase is to actually embody the concepts in the design of the *facilities* provided to the programs which are to run on the abstract machine. This part of the ArchOS PAM design is still in progress and will be reported at a later time.

ArchOS is an experimental operating system, and as such will be continually undergoing design and implementation changes. This could have an adverse effect on the top down design methodology outlined above, since often the changes will require reiteration through substantial portions of the design loop. However, most of this experimenting will only begin following an initial design and implementation of ArchOS. Thus, the top down design methodology is quite appropriate for developing the initial version of ArchOS. But bear in mind that this will only be an initial version, to be experimented with, modified, and reimplemented as much as desired.

This section discusses some of the initial results of the investigation of the major issues involved in the development of the concepts in the ArchOS program's abstract machine. The issues are dealt with in essentially top down order, moving from the very general toward the more specific. For each

issue, the spectrum of available choices is presented and illustrated by means of examples from existing operating systems. Each ArchOS design decision is then explained in light of the available choices by weighing the advantages and disadvantages of each.

### 5.3.1 Nature of the Program's Abstract Machine

The first and highest level issue in the design of the ArchOS program's abstract machine concerns the general nature of that machine. What does ArchOS look like, in general terms, to the application programmer at the time of coding an application to run on it? Is the application program restricted to being only a sequential program, or is there some degree of parallelism in the ArchOS machine which can be exploited in the program? Furthermore, does the application programmer see the ArchOS abstract machine as providing, and hence being defined by, a particular application programming language, or is the ArchOS architecture a conceptual, language independent framework within which the programmer is free to code his application in whatever source language he may find convenient?

#### 5.3.1.1 Spectrum of Choices

This issue of the nature of the program's abstract machine primarily involves two distinct questions:

1. How dependent or independent of particular source languages should the ArchOS PAM definition be?
2. What should be the degree and the grain size of the parallelism that can be expressed in an application program?

These two questions, as they are stated above, are orthogonal and can be dealt with separately. But care must be taken to maintain this independence. It would not be acceptable, for example, to answer the first question by choosing a particular language system, since that would determine the answer to the second question at the same time. Such specific design decisions must be postponed until both of the above questions have been examined and there is a clear picture of the available alternatives.

The first question to be addressed is the degree of language dependence/independence of the ArchOS PAM definition. There can be seen to be three alternative positions, two extremes and one somewhere in between:

1. The ArchOS PAM definition is completely language *dependent*. The ArchOS machine can only be programmed in a single language, and the definition of that language completely defines the ArchOS PAM concepts and facilities.
2. The ArchOS PAM definition is completely language *independent*. The concepts and

facilities of the ArchOS PAM define a conceptual, language independent framework within which it is possible to code applications in whatever source language is desired.

3. The ArchOS PAM is somewhat language independent but its concepts and facilities are oriented toward the support of a particular language or class of languages.

The first position, that of complete language dependence, is conceptually equivalent to building a high order language machine. In this case, ArchOS could be regarded as the run time portion of the chosen application implementation language system. For example, one might consider implementing an Ada machine [46]. The concepts and facilities of the Ada programming language, such as task, rendezvous, and so on, would then define the conceptual machine seen by the programmer when coding an application to run on ArchOS. Since the Ada language definition includes a fairly complete description of the run time environment for Ada programs, few, if any, additional operating system facilities would have to be provided. The Ada abstract machine definition is complete in itself. As an alternative to Ada, a totally sequential programming language such as Pascal [18] could be selected, if it was desired that the nature of the program's abstract machine be sequential.

The following is a list of some of the advantages of complete language dependence:

1. The concepts and facilities of the program's abstract machine blend smoothly into the application programming language, since they were designed as part of that language.
2. The application programmer need only learn the concepts and facilities provided by the chosen implementation language to have a complete picture of the program's abstract machine.
3. The consistency, adequacy, and completeness of the PAM is assured if the chosen application language system was carefully designed.
4. The PAM concepts and facilities have already been designed, so that part of the operating system design task can be skipped.

Some of the disadvantages of complete language dependence are the following:

1. The application programmer is largely restricted to using one programming language, although it may be possible to implement other languages using the chosen PAM definition language.
2. Existing software, designed to run on the application processors that are being used, but not written in the same source language as that selected for the PAM definition language, cannot be easily incorporated in new programs without completely rewriting it.
3. The ability to experiment with the design of the user visible machine, to develop a convenient environment for employing decentralized resource management and control techniques at the application level, is severely limited, if not completely eliminated.

The second position on this issue of the language dependence/independence of the ArchOS PAM definition is that of complete language independence. This is the usual goal which most general purpose operating systems strive for in the design of their PAMs. In this case, the operating system PAM defines the runtime environment and provides a general, language independent model for application programs. The various application programming languages are then used to describe the (usually sequential) programs or parts of programs which are to execute within that operating system PAM environment. Each language provides its own interface, through language extensions, subroutine libraries, or otherwise, to the facilities provided by the operating system PAM. As an example, Unix [38] is an operating system that supports a variety of languages, though in all cases a program is regarded as a sequential Unix process, with the operating system facilities provided through the subroutine call mechanism.

Some of the advantages of complete language independence are the following:

1. The application programmer has a choice of source languages in which to code his application.
2. There is a potential for mixing various source languages, as appropriate, within a single program, since in all cases the PAM is identical.
3. Existing code in various source languages can be incorporated in new programs with relative ease, often saving a great deal of labor and allowing code sharing.
4. It is not necessary to count on the existence of a particular source language processor for any application processor which may be selected for the computer system. It should be possible to extend any supplied programming language to provide an interface to the operating system PAM facilities.
5. It is possible to experiment with the design of the program's abstract machine since such experimentation does not modify the semantics of a fixed application programming language.

Some of the disadvantages of complete language independence are the following:

1. It may not be possible in practice to achieve complete language independence in its strictest sense. Many languages, such as Lisp [32], provide and require their own abstract machine definition, regardless of the underlying operating system PAM.
2. The operating system PAM concepts may conflict with the concepts supported by a particular programming language, requiring parts of the programming language to be replaced.
3. It may be difficult to smoothly incorporate the PAM concepts and facilities into existing languages which were not designed with them in mind. For example, strong type checking may be required, but not supported by the application programming language.

4. The application programmer must understand the PAM, as supported by the operating system, as well as the application programming language, before being able to design and code a program.

The third position on the issue of language dependence/independence is somewhat of a compromise. In this case the ArchOS PAM definition is somewhat language independent but the concepts and facilities are oriented toward the support of a particular language or class of languages. This position is the one which most general purpose operating systems achieve in practice. The PAM is designed, as much as possible, independently of programming language considerations. But a particular application programming language, or set of languages, is kept in mind as the most likely one(s) to be supported, and hence must be accommodated in the design.

Some of the advantages of this compromise position are as follows:

1. Most of the advantages listed for the position of complete language independence also hold here, although this position is somewhat more restrictive.
2. There is a good possibility of smoothly incorporating the PAM concepts and facilities at least into those languages that were kept in mind.

Some of the disadvantages of this position are the following:

1. Most of the disadvantages listed for the position of complete language independence also hold here, although with this compromise position the disadvantages are somewhat less severe.
2. The choice of appropriate application implementation languages may be somewhat limited, which also reduces the portability of previously existing code.

The second question to be addressed regarding the nature of the program's abstract machine deals with the degree and the grain size of the parallelism that can be expressed in an application program. Note that varying degrees of parallel processing may be going on "behind the scenes", but this issue only talks about the parallelism that is explicitly seen and expressed in an application program. Once again there are three main alternative positions:

1. The ArchOS PAM is completely sequential, allowing no parallelism. An application program is restricted to being a strictly sequential program.
2. The ArchOS PAM allows parallelism on a procedure grain size. An application program can be written as a set of asynchronously executing, cooperating processes.
3. The ArchOS PAM allows parallelism on a statement grain size. An application program can express a fine grain of parallelism, such as requesting the parallel execution of arbitrary statements.

The first position listed above, that of no parallelism, is the typical abstract machine view provided by most common programming languages such as Fortran, Algol '60, and Pascal. In this case, no facilities are provided through either the application programming language or the operating system for expressing, requesting, and controlling the parallel execution of parts of an application program.

The advantages of a sequential PAM are the following:

1. This is the traditional view of programming. Hence, no retraining of application programmers is needed, and no extensions or modifications are needed for most application programming languages, since no new, foreign concepts of parallel execution have been introduced.
2. A sequential PAM is the simplest type to program, and the programs can be more easily verified.
3. This is the simplest PAM for an operating system to implement and support.

Some of the disadvantages of a sequential PAM are the following:

1. It is incapable of supporting application programs for handling real time monitoring and control environments. In such environments, events occur asynchronously with respect to program execution.
2. Control is centralized in the single, sequential application program, and there is no way to experiment with more decentralized solutions to resource management and control problems.
3. It is impossible for the application programmer to use parallelism to achieve improved performance.

The second position on this issue of the degree of parallelism visible in the PAM is that parallelism is available on a procedure grain size. Such asynchronously executing procedures are usually called processes, though other terms such as tasks and activities are also common. Many operating systems, particularly those for real time control systems and for multiprocessor systems, support parallelism at this level. Some examples of "process oriented" operating systems are Unix [38], Hydra [49], HXDP Executive [7], StarOS [25], Medusa [34], and Thoth [10].

Some of the advantages of a process oriented PAM are the following:

1. It allows the handling of asynchronous events, as required in a real time monitoring and control environment.
2. It allows experimentation with decentralized resource management and control techniques using multiple processes in an application program.

3. Improved performance through the parallel execution of components of a program is possible, assuming that the parallelism is real (as it is in multiple processor machines), as opposed to simulated (as it is in uniprocessors).
4. Process orientation reflects the hardware organization and architecture of a distributed computer system, such as Archons.

Some of the disadvantages of a process oriented PAM are the following:

1. It is more difficult to design a program as a set of cooperating processes than to simply write a sequential program. Of course, in many cases, the entire program could be written as a single process, if the programmer was unconcerned about the potential advantages of multiple processes.
2. It is difficult to verify the correctness of programs unless the possible interactions between processes are carefully restricted.
3. Few existing languages directly support the concept of program structuring using processes. Facilities for expressing, requesting, and controlling the parallel execution of processes must somehow be added into the application programming languages, or otherwise made available through those languages.

The third position on the degree of parallelism issue is that an application program can express a fine grain of parallelism, such as requesting the parallel execution of arbitrary statements. An example of such fine grain parallelism might be the solution of large numerical problems involving vector and matrix operations, where all components of a vector or array can be computed in parallel.

Some advantages of fine grain parallelism are the following:

1. It allows the maximum possible amount of parallelism in a program to be exploited.
2. It includes process structuring as a special case, where procedure invocations are the statements which are executed in parallel. Hence, all of the advantages listed for process oriented PAMs also apply here.

Some of the disadvantages of fine grain parallelism are the following:

1. There are many similar disadvantages to those listed for process oriented PAMs.
2. Special parallel processors are required as the application processors in order to support such a fine grain of parallelism and make it economical.
3. The application programming language must support such a fine grain of parallelism directly, since procedure calls or other techniques would be uneconomical.

### 5.3.1.2 Resolution

The two main questions concerning the general nature of the program's abstract machine have been discussed at length in the preceding section. It is now necessary to make particular design decisions, selected from the spectrum of choices available on each issue.

On the question of how dependent or independent of particular source languages should the ArchOS PAM definition be, the compromise position is chosen. The ArchOS PAM definition will be as language independent as possible, though a couple of languages will be borne in mind during its design. The aim will be to design the concepts and facilities of the PAM so that they define a conceptual, language independent framework within which all applications will be coded. However, when design decisions which may affect the degree of language independence have to be made, they will be resolved by considering the chief languages which are to be supported by ArchOS. A final decision on what these languages will be has yet to be made. However, it is currently expected that ArchOS will support Bliss [48] during its initial development and implementation on the Cm\* multiprocessor [43], and it will support Ada [46] when reimplemented on the Archons distributed computer system.

The compromise position on this issue of language dependence/independence was chosen for the following reasons:

1. One of the goals and objectives of ArchOS is to experiment with the design of the user visible machine, to develop a convenient environment for employing decentralized resource management and control techniques at the application level. Since this is not permitted under the position of complete language dependence, that option must be rejected.
2. Complete language independence may not even be achievable in practice, and since portability of existing software is not as large a consideration in an experimental system such as ArchOS, it is not worth the extra effort needed to try to assure language independence.

On the question of what should be the degree and the grain size of the parallelism that can be expressed in an application program, the position of process or procedure grain size is chosen. It will be possible to write application programs as sets of asynchronously executing, cooperating processes.

A process oriented PAM was chosen for the following reasons:

1. ArchOS is intended to support experimentation with decentralized resource management and control techniques at the application level, and the primary application area is



expected to be that of real time monitoring and control. Since a sequential programming environment is inappropriate for this type of experimentation and application area, the choice of a completely sequential PAM must be rejected.

2. The Archons project is not intended to involve research in the area of fine grain parallelism. The Archons distributed computer architecture is designed to support the process level of parallelism. It is not expected that the application processors will all be special parallel processors capable of supporting fine grain parallelism. Hence, the choice of a PAM supporting fine grain parallelism must be rejected.

### 5.3.2 Nature of Program Structuring

Having decided that the ArchOS PAM will support application programs structured as sets of asynchronously executing, cooperating processes, many issues are raised regarding the nature of those processes and the way they are combined into programs. The first issue to be addressed is the nature of program structuring. How are processes to be combined to form programs? Are the component processes and their interconnections defined statically, or can new processes and new interconnections be added within a program as it executes?

#### 5.3.2.1 Spectrum of Choices

There are two, conceptually distinct aspects of a process structured program which must be considered: the component processes which make up the program, and the process interconnection structure which defines the highest level structure of the program. The process interconnection structure is essentially the network or directed graph defined by the communication and synchronization relationships which hold between the processes. At issue here is whether the structure of a program is determined statically or dynamically. There can be seen to be four alternative positions:

1. The set of processes in a program is statically defined, and the interconnection structure is also static.
2. The set of processes can vary dynamically, but the interconnection structure is statically defined.
3. The set of processes is statically defined, but the interconnection structure can vary dynamically.
4. The set of processes can vary dynamically, and the interconnection structure is also dynamic.

The first position, that of a completely static program structure, conceptually involves the definition and linking of the component processes of a program prior to execution. Once a program is

executing, it is incapable of modifying its own structure. The HXDP executive [7], and Medusa [34] are examples of systems in which application programs are statically structured as sets of processes.

Some of the advantages of completely static program structuring are the following:

1. Facilities for dynamic process creation, destruction, and interconnection are not needed, simplifying the PAM definition and implementation.
2. Programming in the large (interconnecting modules) and programming in the small (defining the modules) are separate tasks, simplifying the design and implementation of application programs.
3. Program verification is easier since the program structure is fixed.
4. Many applications, particularly in real time control, have a relatively simple, fixed form [8], and hence can be readily accommodated by a statically defined program structure.

Some of the disadvantages of completely static program structuring are the following:

1. A program is incapable of reacting to failed processes either by destroying processes that are malfunctioning or replacing those that have been destroyed. *This limits the degree of robustness that a program can achieve.*
2. A program is incapable of reacting to failed processes by adjusting the interconnection structure to allow remaining processes to handle the tasks formerly managed by the failed processes. *This limits the graceful degradation capability of a program, the ability to continue operating, though with reduced performance, following a failure.*
3. A program is incapable of dynamically adjusting the number of processes to react to varying workloads. *Either the maximum workload must be planned for by providing many more processes than is normally needed, or the program must accept far less than optimal performance under a heavy workload.*

The second position on this issue of program structuring is dynamic processes with a static interconnection structure. In this case the process interconnection structure is specified prior to execution and essentially defines a static template for the program. The individual processes within the template structure can be destroyed and replaced at will, without changing the essential structure of the program. The MMBC Executive [2] is an example of a system which supports this form of program structuring.

Some of the advantages of dynamic processes with static interconnection are the following:

1. Most of the same advantages as for completely static program structuring are also enjoyed here. However, process creation and destruction facilities must now be provided, making the PAM definition and implementation somewhat more difficult.

2. A program is able to react to the failure of processes by destroying processes that are malfunctioning or replacing those that have been destroyed. This improves the degree of robustness that a program can achieve.

Some of the disadvantages of dynamic processes with static interconnection are the following:

1. The graceful degradation capability of a program is limited by its inability to adjust the process interconnection structure to allow remaining processes to handle the tasks formerly managed by processes that have since failed.
2. The ability of a program to dynamically adjust to varying workloads by varying the number of processes is limited, since the static interconnection structure determines a fixed maximum number of processes.

The third position on the issue of program structuring is a static set of processes with a dynamic interconnection structure. Such a situation is seldom seen in practice, though there are instances where this form of program structuring can be advantageous. With this type of program structuring, the set of processes is fixed prior to execution. As an example, each process may correspond to a particular hardware device and perform the function of handling that device. The interconnection structure of those processes can then be varied at will to help balance the work load, adjust to failed devices, and so on.

Some of the advantages of static processes with dynamic interconnection are the following:

1. A program can exhibit graceful degradation by adjusting the interconnection structure to allow remaining processes to handle the tasks formerly managed by processes that have since failed.
2. A program can balance the workload among processes by adjusting the interconnection structure to allow idle processes to handle some of the tasks originally intended to be handled by overburdened processes.

Some of the disadvantages of static processes with dynamic interconnection are the following:

1. A program is incapable of reacting to the failure of processes either by destroying processes that are malfunctioning or replacing those that have been destroyed. This limits the degree of robustness that a program can achieve.
2. A program is incapable of dynamically adjusting the number of processes to react to varying workloads since the number of processes is fixed prior to execution.
3. Programming in the large and programming in the small are combined to some extent since the process interconnection structure is dynamically specified at run time, through the program code. This makes the design and implementation of application programs somewhat more difficult.

4. Program verification is somewhat more difficult since the program structure can vary throughout execution.

The fourth position on the issue of program structuring is dynamic processes with a dynamic interconnection structure. In this case the entire program structure, the number of processes and their interconnection, is determined at run time and can vary considerably during execution and from one execution to the next. Thoth [10], Roscoe [41], and Trix [45] are examples of systems in which application programs are dynamically structured as sets of processes.

Some of the advantages of totally dynamic program structuring are the following:

1. The same advantages as those listed for static processes with dynamic interconnection also hold here.
2. A program is able to react to the failure of processes by destroying processes that are malfunctioning or replacing those that have been destroyed. This improves the degree of robustness that a program can achieve.
3. A program is capable of dynamically adjusting the number of processes to react to varying workloads.

Some of the disadvantages of totally dynamic program structuring are the following:

1. Programming in the large and programming in the small are combined to some extent, making the design and implementation of application programs somewhat more difficult.
2. Program verification is somewhat more difficult since the program structure can vary considerably throughout execution.

#### 5.3.2.2 Resolution

On this question of the nature of program structuring, the fourth position discussed above is chosen. The set of component processes of a program can vary dynamically, and the process interconnection structure is also dynamic. The ArchOS PAM will provide facilities for dynamically creating and destroying processes and modifying the interconnections among processes.

Totally dynamic program structuring was chosen for the following reasons:

1. One of the goals of the ArchOS research effort is to investigate the ways in which the Archons distributed computer system architecture and its associated advantages of improved robustness, modularity, and performance can be reflected in the design of the ArchOS user visible machine. Totally dynamic program structuring offers the best opportunity for developing highly robust and flexible, high performance application programs.

2. Another goal of ArchOS is to support decentralized resource management and control techniques at the application level. Very little is currently known about the program structuring requirements for such techniques, and so it is best to provide the most flexible facilities possible. This will provide the maximum freedom when experimenting with new decentralized control algorithms.
3. With totally dynamic program structuring, the various forms of static structuring can be built on top of ArchOS, if they should be found desirable in certain situations. Of course, there is a cost associated with supporting static structuring in this manner, since it is done at the beginning of execution rather than during compilation or linking. But the flexibility of totally dynamic program structuring again facilitates experimentation with these various options.

### 5.3.3 Further Issues in the Design of the ArchOS PAM Concepts

Having decided that the ArchOS program's abstract machine will support totally dynamic structuring of application programs as sets of asynchronously executing, cooperating processes, many new issues come to mind. What is the nature of a process? In particular, how autonomous are processes? What is the nature of the code and the storage space of a process? And what is the nature of process procreation?

Processes must interact through communication and synchronization. Many issues regarding the nature of interprocess communication and synchronization have yet to be addressed. In addition, there must be some means of controlling process procreation and structuring, as well as interprocess interactions.

All of these issues have yet to be dealt with in developing the concepts of the ArchOS PAM. This work is still in progress and will be reported at a later time.

## 5.4 Future Work and Research Directions

The ArchOS decentralized operating system research effort is still in its very early stages. This report has described in detail the ultimate goals and objectives of the project and its intimate relationship with the Archons distributed computer system design effort. It has also provided a glimpse of the ArchOS top down design methodology. Each of the major issues to be addressed is attacked by first surveying the spectrum of possible design choices and evaluating each of them. Only then is a design decision made, bearing in mind the advantages and disadvantages of each of the options.

Current plans are to proceed with the design and implementation of ArchOS in approximately the following manner:

1. Complete the investigation of the issues and the design of the PAM concepts.
2. Design the PAM facilities which embody and reflect those concepts.
3. Design the operator's abstract machine (OAM), which is the other component of the ArchOS user visible machine.
4. Design and experiment with an initial implementation of ArchOS on the Cm\* multiprocessor system.
5. Design and experiment with the implementation of ArchOS on the Archons distributed computer system testbed hardware.

It is expected that the design of the ArchOS user visible machine will be completed by early next summer. The initial implementation on Cm\* should follow one year later. The final implementation on Archons will begin as soon as the hardware becomes functional, probably one year after the Cm\* implementation.

## Acknowledgements

Discussions, advice, criticism, and patience of all the members of the Archons Project is gratefully acknowledged by the authors. In addition, we would like to thank all of the members of the Computer Science and Electrical Engineering Departments for the resources, intellectual and physical, which we used in pursuit of this research.

James W. Wendorf was partially supported by a postgraduate scholarship from the Natural Science and Engineering Research Council of Canada.

## I. Decentralized Resource Management and Control

The cost of processor resources focused the attention of early system software designers on uniprocessors, where they developed the current foundations of traditional operating systems. As processor hardware became less costly, multiple processors were connected to shared primary memory (forming "multiprocessors"), because most of the uniprocessor software concepts and structures could be successfully retained with minimal modification and augmentation. One consequence of this historical evolution is that many of the premises on which these traditional operating system concepts were strongly based are now very often so taken for granted that they have become transparent--they are either not recognized explicitly, or believed to be universally valid. Instances of more or less decentralized resource management have occurred in various aspects of computer system designs, but it appears that almost invariably they have arisen in an ad hoc fashion--particular approaches have been used accidentally or out of convenience or necessity, rather than through consideration of fundamental principles. A systematic method of selecting among management alternatives, based on the issues, will be necessary to meet some of the challenges posed by contemporary application and architectural trends.

Therefore, we begin with a conceptual model [Jensen 1980] to illuminate the spectrum of resource management and control decentralization from minimal (i.e., centralized) to maximal, and to delineate the region of interest to us. It was created to meet four objectives:

- contribute to an improved understanding of the fundamental nature of "decentralization," particularly with respect to control;
- assist in the formulation of a common frame of reference and terminology for discussing control decentralization;
- clarify the perception of relative differences and similarities among specific instances of control;
- facilitate the synthesis of control policies and mechanisms which are as decentralized as desired.

The model was not intended to provide predictions or quantitative evaluations, nor to ascribe attributes (e.g., "better," "more robust") to different points in the management and control spectrum. These endeavors are important but difficult to accomplish without first meeting the model's objectives.



The primitive object in this model is a *resource*, which is an instance of an abstract data type. Resources exist at different levels of abstraction--in computer systems these levels tend to range from the application user interface at the top, to the physical interconnection hardware at the bottom. (A type is implementation-independent, so hardware may appear at any level and is not per se a level of abstraction [Jensen 1980a].) A resource is encapsulated by one or more *managers* (also typed objects) which abstract it and thus themselves become resources at higher levels. This abstraction, which we call *management*, consists of the decision and action activities involved in manipulating, maintaining (e.g., detecting and recovering from faults), and regulating access to (e.g., sharing) the resource.

An activity at a particular level of abstraction is decentralized to a degree determined by the number and relationships of the entities which perform it. Those relationships are a function of the first two factors. We define the "control" of that activity to be the enforcement of a consistent view of its state on the entities participating in it. ("Consistency" is the invariance of predicates defined on the state [Eswaran 1976].) This enforcement is comprised of activities carried out by entities at a lower level--thus, the extent of their *resource management* decentralization governs the *control* decentralization of the higher level activity. An activity (however centralized or decentralized) whose consistency is enforced by a unique lower level entity (e.g., monitor, sequencer, bus arbiter) has totally centralized control (a similar view is expressed in [Le Lann 1979]). When there multiple such entities (e.g., logical clocks [Lamport 1978]), control is decentralized to some extent depending on how they interact. The third alternative is that enforcement does not take place--this generally implies that it is not possible to guarantee activity consistency, although the system or application may allow it to be achieved with some (perhaps very high) probability. The possibility of provable consistency without reliance on one or more underlying enforcement entities is the subject of speculation, but to our knowledge is not yet established.

#### **Management Decentralization Factors**

In this model, decentralization is based on the notion of *multilateral* resource management, which is a function of five major factors. The first three deal with activities on *individual* resources: 1) the extent to which all the responsible managers must contribute to an activity before it is complete; 2) the parity of the managers involved in an activity; and 3) the number of managers associated with an activity. The remaining pair of factors have to do with *system-wide* management: 4) the average percentage of the other managers (at a given level) which each cooperates; and 5) the number of resources in any intersection of managerial scope.

#### **Individual Resource Management**

## Archons

The management decentralization of the activities on a single resource is determined by the number of managers involved, and by the relationships among them.

The most common form of management is autocracy, where a single entity unilaterally makes and carries out all decisions on every resource (at a particular level of abstraction). However, multiple entities may participate in the management of the same resource--examples of diverse ways they may do so include:

- successive, where all activities are performed for a period of time by one manager, and then by another, in some serial sequence;
- partitioned, where each manager performs a different activity, whether consecutively or concurrently;
- democratic, where all managers perform each activity by negotiation and consensus among equals.

Many of the various multilateral management forms exhibit different degrees of decentralization. This model distinguishes them on that basis according to two factors: *consentaneity* and *equipollence*.

In the sense intended here, "consentaneity" is the extent to which a particular activity for a resource is carried out by all its managers together, with or without any real concurrency. Activities may each differ in this regard. The most centralized case is where only one manager performs a particular instance of the activity on the resource, as exemplified by autocracy. The most decentralized case is where every manager responsible for the activity is involved in every instance of it, as with democracy.

"Equipollence" is the degree of equality with which authority and responsibility are distributed across the multiple managers associated with a particular activity. It may be different for each of the various management activities. Equipollence can be visualized as the inverse of the relative dispersion of capability. The maximally centralized case, typified by autocracy and partitioning, is where one manager has all responsibility and authority for the activity and no other manager has any. Succession and democracy are instances of the maximally decentralized case of equipollence, where every manager is equally capable of participating in the activity.

When the first two factors are evaluated together, autocracy, having minimum consentaneity and minimum equipollence, is the maximally centralized case. The maximally decentralized case, maximum consentaneity and maximum equipollence, is exemplified by democracy. Succession is an intermediate case where equipollence is high but consentaneity is low.

The third factor which contributes to the degree of an activity's decentralization is the number of managers it has. The most centralized case is where only one manager is involved in the activity, such as when management is autocratic or partitioned. In principle, decentralization can be increased without limit in this respect by adding managers. Neither consentaneity nor equipollence are affected by the number of participants--however, the most centralized case of the third factor is an exception because when there is only one manager, consentaneity and equipollence are both necessarily minimum.

While the minimally and maximally decentralized cases of management can readily be identified, most instances are more difficult to order solely according to their factors. This is because (at least at the present time) there appears to be nothing intrinsically more decentralized about greater consentaneity than about greater equipollence--the relative significance of each factor depends on the motivations and requirements of a particular system analysis or synthesis effort. In our current view, when the number of managers is greater than one, it is less consequential than the first two factors.

These three factors are defined for individual activities, but they can also represent the management decentralization for a resource, on the basis of some system- or application-dependent activity weighting such as "importance" or frequency of execution. (However, one must resist any temptation to quantify the model beyond its intent and suitability.)

#### **System-Wide Resource Management**

Very often resources are managed not just as separate entities but also collectively in accordance with more global objectives and constraints (i.e., as a system). The three factors above do not account for the extent to which this latter aspect of management is decentralized, even by combining the per-resource results. The precept of multilateral resource management can be applied here as well to derive two factors that determine the degree of *system-wide* decentralization.

The first of these factors (the fourth in our model) is the aggregate (e.g., mean) percentage of all other managers at the same level in the system with which each manager functions multilaterally. The maximally centralized case is where no manager engages in multilateral activities on any resource--the resources are partitioned into disjoint subsets, each of which is governed independently of the others by one manager. The maximally decentralized case is where every manager operates multilaterally with every other in the handling of at least one resource.

The second system-wide factor (number five in the model) is the number of resources involved in

each instance of multilateral management. The maximally centralized case is where no resources are multilaterally managed, and the maximally decentralized case is where all are. (If no managers act multilaterally, then obviously no resources are multilaterally managed, and vice versa.)

Together, the fourth and fifth factors provide a measure of global management decentralization. The maximally centralized case is that no resources are subject to multilateral management; the maximally decentralized case is where every manager participates in the management of every resource. Beyond that, ordering of cases depends on the relative importance ascribed to each of the two factors (as with the first three factors).

These five factors account for the management and control decentralization at any particular level of abstraction, and in general a computer system will exhibit a different degree of decentralization at each level. For example, a computer network could have:

- rather decentralized management at the user interface level, provided by a "network operating system";
- rather centralized management at the executive level, because the host operating systems are autonomous;
- rather decentralized management at the communication subnet level, as a consequence of both the hardware and the routing algorithm designs.

Our research is focused on computers having highly decentralized control, particularly at the system-wide executive levels and down--we term such a machine a "distributed computer".

#### **The Implications of Communication on Management and Control**

Multilateral resource management clearly necessitates that the participants share some information. Some of this information may be static--e.g., an a priori model of other managers' known or expected strategies, tactics, and even algorithms. Other information may be dynamic, including models of observed behavior and estimates of current state of the other managers. Static information can be easily shared through separate copies, but dynamic information sharing requires communication among the managers.

Communication involves two conceptually distinct aspects: the production, and the manifestation, of signals--the relationship between them we term *signal observability*. Poor observability includes the usual memory bit errors and packet switching errors (such as lost and duplicate packets)--these are almost always readily correctable, but at the cost of delays. In fact, *time* is the most important aspect of signal observability--we group its effects into two categories: latency and synchronization.

*Latency* in the signal observability sense is the extent to which a manager can see a signal in time for it to be useful. More specifically, it is the set of probabilities for each manager that it can observe each signal in any particular set of signals any necessary amount of time before some other set of signals occur (e.g., in time to respond or affect which signal is sent next). Some of the probabilities may be conditional on certain aspects of the system state. The best case is that every manager can observe every signal within any arbitrarily small amount of time after it is sent; in other cases, some managers may not be able to observe any signal in the set until after all have been sent. When a probability is so low that the signal is of no value, observability is "incomplete".

*Synchronization* of signals is the extent to which all managers can place them in the same sequence, despite complications such as variable and unknown delays [Le Lann 1977]--more exactly, the probability for each manager in any particular set of managers that it can induce any particular ordering (according to origination time, priorities, atomicity of operations, etc.) on any particular set of signals. As with latency, some of the probabilities may be conditional. Depending on the circumstances, synchronization of signals may be either weak or strong: the former means that the managers are all able to agree on some ordering, but it may *not correspond to the desired criterion*; the latter means that the managers are all able to establish a common sequence corresponding to the chosen criterion--e.g., that in which the signals were sent with respect to each manager's own (independent) time reference. The best case is that every manager can determine whatever ordering is wanted on all signals; the worst case is that no managers can reach accord on any ordering of any signals. Between these lie cases where only partial or probabilistic signal synchronization can be achieved.

Typically, each level of abstraction has different signals, and communication at one level is carried out by the next level down. Signal observability at the lowest system-wide level, interprocessor communication, is particularly important because higher level communication, management, and control depend on it. Tacit assumptions about these dependencies underlie many executive level management mechanisms and policies.

When the physical communication mechanism is shared primary memory (as in uniprocessors and multiprocessors), signal observability is normally very good. The performance, and even correctness, of nearly all conventional centralized executive control techniques rely on that. For example, consistency of any activity is typically enforced by a single synchronizing entity (such as a monitor), based on the premise that there is a high degree of signal observability (e.g., consistency of signal order) between the synchronized entities (e.g., a producer in a critical section) and the synchronizer. A distributed computer could be implemented with multiprocessor hardware by (physically or

logically) partitioning part of the primary memory, and using the shared portion (in a decentralized fashion) only as a communication medium.

But in multicomputers where primary memory is physically partitioned and interprocessor communication is by explicit I/O (e.g., buses), signal observability is usually much worse--communication protocols can help alleviate the "syntactic" part of the problem, but exacerbate rather than remedy the more critical delay parts. It may be neither cost-effective nor technologically realistic to assume that higher levels can virtualize the physical interconnection to have any desired degree of signal observability. Depending on the individual system and application, it may be possible to extend certain centralized control schemes to less favorable signal observability conditions. However, techniques originally intended for shared primary memory can be inappropriate and even counterproductive if naively re-implemented in systems with disjoint primary memories. Quantitative signal observability differences between memory and I/O interprocessor communication, as uninteresting as they may seem, generally necessitate a conceptually significant qualitative difference in management and control (just as different memory technology speeds lead to different management approaches for primary and secondary memories). New control concepts and mechanisms which are explicitly intended to function correctly and efficiently despite incomplete and inaccurate information must differ radically from current approaches by employing algorithms which make "best effort" resource management decisions. (Such generalization of signal observability premises may prove to be beneficial in centralized management environments as well.)

Movement away from centralization in multiprocessor and multicomputer executives has been rather limited--notably absent are instances of decentralized control, and reflection of limited signal observability in the management algorithms. Examples of what is currently taking place include: a) hierarchies of unilateral management (e.g., MicroS [Wittie 1980]); b) partitioning of resources and activities (e.g., StarOS [Jones 1979], Medusa [Ousterhout 1980]); c) round-robin unilateral process binding decisions, and successive pairwise load balancing between neighbors (e.g., Arachne nee' Roscoe [Solomon 1979], [Bryant 1981]); and d) unilateral bidding decisions in DCS [Farber 1972]). Portions of executives also have been the subject of varied decentralization efforts, particularly the management of messages and message repositories for interprocess communication (e.g., [Boebert 1981]), and process/processor binding (e.g., [Casey 1977]). Some of the most relevant steps toward decentralized control in our sense have taken place at application levels above the executive: in artificial intelligence, especially decentralized problem solving ([Lesser 1979], [Smith 1979]); and in distributed data bases, including data sharing systems (e.g., the circulating sequencer [Le Lann 1981]) and concurrency control in multiple-copy data bases (e.g., consensus algorithms [Thomas

1978]). At the physical communication levels below the executive (where the management functions are simpler and less general, and the resources less abstract and less dynamic) there have been notable contributions as well: dynamic routing in packet-switching communication subnetworks [Schwartz 1980]; and state exchange communication protocols [Fletcher 1979]. Some useful notions may be derived from fields other than computer science and engineering, such as stochastic [De Groot 1970] and fuzzy [Gupta 1977] decision making.

Nonetheless, executives which are significantly decentralized according to our criteria are still not possible within the current state of the art. This dramatically impacts multicomputers by preventing most of them from having a singular operating system in the same sense that a uniprocessor or multiprocessor can--that is, one which integrates the processors into a *computer* by attempting to manage all the *executive* level resources in the best interests of the whole system. Instead, they are constrained to being *networks*--for each processor there is a separate operating system which manages a local partition of the system's resources for its own good. In many applications (e.g., resource-sharing networks), such an arrangement may be adequate or even necessary (for technical or nontechnical reasons). But in many others (e.g., real-time control), having only local but no global executive resource management greatly reduces the potential extent to which certain important attributes, such as robustness and modularity, can be provided on a *system-wide* basis. Helping to alleviate this situation is one of the motivations for our research.

#### Appendix Bibliography

Anderson, George A., and E. Douglas Jensen, "Computer Interconnection Structures--Taxonomy, Characteristics, and Examples," *Computing Surveys*, ACM, December 1975.

Barbacci, Mario R., "The ISPS Computer Description Language", *Trans. on Computers*, IEEE, January 1981.

Bell, G. Gordon, and Allen Newell, *Computer Structures*, McGraw-Hill, 1971.

Boebert, W. Earl, Dennis Cornhill, William R. Franta, E. Douglas Jensen, and Richard Y. Kain, "Communications in the HXDP Executive," *Trans. on Software Engineering*, IEEE, to appear.

Boebert, W. Earl, *Distributed Data Processing Rationale: The Program Planning Point of View*, Honeywell Systems and Research Center Report 77SRC66, September 1977.

Bryant, Raymond M., and Raphael A. Finkel, *A Stable Distributed Scheduling Algorithm*, *Proc. Second Int. Conf. on Distributed Computing Systems*, IEEE, April 1981

## Archons

Casey, Liam, and Nick Shelness, "A Domain Structure for Distributed Computer Systems," *Proc. Symp. on Operating System Principles*, ACM, November 1977.

De Groot, Morris H., *Statistical Decisions*, McGraw-Hill, 1970.

Dowson, Mark, Brian Collins, and Brian McBride, "Software Strategy for Multiprocessors," *Microprocessors and Microsystems*, July/August 1979.

Eddington, Donald C., *SEAMOD: Sea Systems Modification and Modernization by Modularity--Data Bus and Combat Systems Integration Study*, U.S. Naval Ocean Systems Center Report Q238, January 1976.

Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of Consistency and Predicate Locks in a Database System," *CACM*, November 1976.

Farber, David C., and Kenneth C. Larson, "The Structure of a Distributed Computer--Software," *Proc. Symp. on Computer-Communications Networks and Teletraffic*, Polytechnic Press, 1972.

Fletcher, John G., *Serial Link Protocol Design: A Critique of the X.25 Standard, Level 2*, Lawrence Livermore Laboratory Report UCRL-83604, November 1979.

Goos, Gerhard, "Hierarchies," *Software Engineering: An Advanced Course*, Springer-Verlag, 1975.

Gupta, Madan M., Saridis, George N., and Gaines, Brian R., *Fuzzy Automata and Decision Processes*, North Holland, 1977.

Hsia, Pei, "A Configurable Distributed Computing System," *Proc. First Int. Conf. on Distributed Computing Systems*, IEEE, 1979.

Jensen, E. Douglas, "The Honeywell Experimental Distributed Processor," *Computer*, IEEE, January 1978.

Jensen, E. Douglas, "Decentralized Control," *Distributed Systems: an Advanced Course*, Springer-Verlag, 1980.

Jensen, E. Douglas, "Hardware/Software Relationships in Distributed Computer Systems," *op. cit.*



Jones, Anita K., Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proc. Symp. on Operating Systems Principles*, ACM, December 1979.

Knuth, Donald E., "Additional Comments on a Problem in Concurrent Programming Control," *CACM*, May 1976.

Lamport, Leslie, "A New Solution of Dijkstra's Concurrent Programming Problem," *CACM*, August 1974.

Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, July 1978.

Le Lann, Gerard, "Distributed Systems -- Toward A Formal Approach", *Proc. IFIP Information Processing Congress*, North Holland, 1977.

Le Lann, Gerard, "An Analysis of Different Approaches to Distributed Computing," *Proc. First Int. Conf. on Distributed Computing Systems*, IEEE, 1979.

Le Lann, Gerard, "A Distributed System for Real-Time Transaction Processing," *Proc. Hawaii International Conf. on System Science*, Western Periodicals, January 1981.

Lesser, Victor R., and Daniel D. Corkill, "The Application of Artificial Intelligence Techniques to Cooperative Distributed Processing," *Proc. Int. Joint Conf. on Artificial Intelligence*, August 1979.

Luczak, Edward C., "Global Bus Computer Communication Techniques," *Proc. Computer Networking Symp.*, IEEE, 1978.

Ousterhout, John K., Donald A. Scelza, and Pradeep S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *CACM*, February 1980.

Ramseyer, Richard R., Robert G. Arnold, Hugh L. Applewhite, and Robert O. Berg, "The Modular Missile Borne Computer Architecture from the Requirements and Constraints Point of View: An Overview," *Proc. First Int. Conf. on Distributed Computing Systems*, IEEE, 1979.

Saltzer, Jerome H., David P. Reed, and David D. Clark, "End-to-End Arguments in System Design), *Second Int. Conf. on Distributed Computing Systems*, IEEE, April 1981.

Schwartz, Mischa, and Stern, T.E., "Routing Techniques Used in Computer Communication Networks," *Transactions on Communications*, IEEE, April 1980.

Smith, Reid G., "The Contract Net Protocol," *Proc. First Int. Conf. on Distributed Computing Systems*, IEEE, 1979.

Solomon, Marvin H., and Raphael A. Finkel, "The Roscoe Distributed Operating System," *Proc. Symp. on Operating System Principles*, ACM, December 1979.

Swan, Richard, "Cm\*--A Modular Multi-Microprocessor," *Proc. NCC, AFIPS*, June 1977.

Tobagi, Frank A., "Multiaccess Protocols in Packet Communication Systems," *Trans. on Communications*, Vol. COM-28, No. 4, IEEE, April 1980.

Thomas, Robert H., "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases," *Proc. COMPCON/Spring*, IEEE, 1978.

Wittie, Larry D., "A Distributed Operating System for a Reconfigurable Network Computer," *Trans. on Computers*, IEEE, 1980.

Zimmerman, Hubert, "OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection," *Trans. on Comm.*, IEEE, April 1980.

## References

- [1] Anderson, George A., and E. Douglas Jensen.  
Computer Interconnection Structures.  
*Computing Surveys*, December, 1975.
- [2] Applewhite, H.L., Arnold, R.G., Gorman, T.J., Gouda, M.G., and Marks, C.P.  
Modular Missile Borne Computer (MMBC) Software Structure and Implementation.  
In *Proc. 1st Int. Conf. on Distributed Computing Systems*, pages 725-735. IEEE, October, 1979.
- [3] Atkinson, R., and Hewitt C.  
Synchronization in Actor Systems.  
*Fourth SIGPLAN - SIGACT Symp. on Principle of Programing Language*, Jan 1977.
- [4] Barbacci, Mario R.  
*The ISPS Computer Description Language*.  
Technical Report, Carnegie-Mellon University Computer Science Department, 1977.
- [5] Baskett, Forrest, and J. H. Howard.  
Task Communication in Demos.  
In *Proc. of the Sixth Symposium on Operating Systems Principles*. November, 1979.
- [6] Bloom, T.  
*Synchronization Mechanisms for Modular Programming Languages*.  
Technical Report, MIT/LC/TR-211, 1979.
- [7] Boebert, W.E., Franta, W.R., Jensen, E.D., and Kain, R.Y.  
Kernel Primitives of the HXDP Executive.  
In *Proc. COMPSAC '78*, pages 595-600. IEEE, November, 1978.
- [8] Boebert, W.E., Franta, W.R., Jensen, E.D., and Kain, R.Y.  
Decentralized Executive Control in Distributed Computer Systems.  
In *Proc. COMPSAC '78*, pages 254-258. IEEE, November, 1978.
- [9] Bryant, Raymond M., and Raphael A. Finkel.  
*A Stable Distributed Scheduling Algorithm*.  
Technical Report, University of Wisconsin, September, 1980.
- [10] Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R.  
Thoth, a Portable Real-Time Operating System.  
*Communications of the ACM* 22(2):105-115, February, 1979.
- [11] Dahl, O. J., Dijkstra, E. W., Hoare, C. A. R. .  
*Structured Programing*.  
Academic Press New York., 1972.
- [12] Dijkstra, E. W.  
Co-operating Sequential Processes.  
In F. Genuys (editor), *Programing Languages*, . Academic Press, New York, 1968.

- [13] Eddington, Donald C.  
*Sea Systems Modification and Modernization by Modularity -- Data Bus and Combat Systems Integration Study.*  
Technical Report, U.S. Naval Ocean System Center, 1976.  
Report Q238.
- [14] Farber, David C., and Kenneth C. Larson.  
The Distributed Computer System.  
In *Proc. Symp. on Computer-Communications Networks and Teletraffic*. Polytechnic Press, 1972.
- [15] Halstead Robert H.  
*Reference Tree Networks: Virtual Machine and Implementation.*  
PhD thesis, MIT, 1979.
- [16] Halstead, Robert H., and Stephen A. Ward.  
MuNet: A Network - Oriented Operating System.  
In *Proc Seventh Architecture Symposium* . 1980.
- [17] Hansen, P. B.  
Experience with Modular Concurrent Programing.  
*IEEE Transaction on Software Engineering* , March 1977.
- [18] Jensen, K. and Wirth, N.  
*Pascal User Manual and Report, 2nd ed.*  
Springer-Verlag, 1974.
- [19] Jensen, E.D.  
The Honeywell Experimental Distributed Processor - An Overview.  
*IEEE Computer* 11(1):28-38, January, 1978.
- [20] Jensen, E.D.  
The Honeywell Experimental Distributed Processor - An Overview.  
*IEEE Computer* 11(1):28-38, January, 1978.
- [21] Jensen, E. D.  
Decentralized Control.  
In *Distributed Systems: An Advanced Course*, . Springer-Verlag, 1980.  
(to appear).
- [22] Jensen, E. D.  
Decentralized Control.  
In *Distributed Systems: An Advanced Course*, . Springer-Verlag, 1980.  
(to appear).
- [23] Jensen, E.D.  
Decentralized Control.  
In *Distributed Systems: An Advanced Course*, . Springer-Verlag, 1980.

- [24] Jones, A.K., Chansler, R.J., Durham, I., Schwans, K., and Vegdahl, S.R.  
StarOS, a Multiprocessor Operating System for the Support of Task Forces.  
*In Proc. 7th Symp. on Operating System Principles*, pages 117-127. ACM, December, 1979.
- [25] Jones, A.K., Chansler, R.J., Durham, I., Schwans, K., and Vegdahl, S.R.  
StarOS, a Multiprocessor Operating System for the Support of Task Forces.  
*In Proc. 7th Symp. on Operating System Principles*, pages 117-127. ACM, December, 1979.
- [26] Lamport, L.  
Time, Clocks, and The Ordering of Events in a Distributed System.  
*CACM*, July 1978.
- [27] Le Lann, G.  
Distributed Systems -- Towards a Formal Approach.  
*In Proc. IFIP Information Processing Congress*. North Holland, 1977.
- [28] Le Lann, G.  
An Analysis of Different Approaches to Distributed Computing.  
*Proc. First Int. Conf. on Distributed Computing Systems*, 1979.
- [29] LeLann, G.  
Delta - An Experimental Distributed Data-Sharing System.  
*INRIA, Project Sirius. BP 105, 78 150 Le Chesnay France.*, 1981.
- [30] Lister, A. M.  
The Problem of Nested Monitor Calls.  
*Operating System Review*, July 1977.
- [31] Luczak, Edward C.  
Global Bus Computer Communication Techniques.  
*In Proc. Computer Networking Symposium*. IEEE, 1978.
- [32] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I.  
*Lisp 1.5 Programmer's Manual*.  
MIT Press, 1962.
- [33] Ousterhout, J.K., Scelza, D.A., and Sindhu, P.S.  
Medusa: An Experiment in Distributed Operating System Structure.  
*Communications of the ACM* 23(2):92-105, February, 1980.
- [34] Ousterhout, J.K., Scelza, D.A., and Sindhu, P.S.  
Medusa: An Experiment in Distributed Operating System Structure.  
*Communications of the ACM* 23(2):92-105, February, 1980.
- [35] Ramseyer, Richard R., Robert G. Arnold, Hugh L. Applewhite, and Robert O. Berg.  
The Modular Missile Borne Computer Architecture from the Requirements and Constraints  
Point of View: An Overview.  
*In Proc. First Int. Conf. on Distributed Computing Systems*. IEEE, 1979.

- [36] Reed, D. P. and Kanodia, R. K.  
Eventcounts and Sequencers.  
*CACM*, Feb. 1979.
- [37] Ritchie, D.M. and Thompson, K.  
The UNIX Time-Sharing System.  
*Communications of the ACM* 17(7):365-375, July, 1974.
- [38] Ritchie, D.M. and Thompson, K.  
The UNIX Time-Sharing System.  
*Communications of the ACM* 17(7):365-375, July, 1974.
- [39] Smith, Reid G.  
  
In *The Contract Net Protocol*. IEEE, 1979.
- [40] Solomon, M.H. and Finkel, R.A.  
The Roscoe Distributed Operating System.  
In *Proc. 7th Symp. on Operating System Principles*, pages 108-114. ACM, December, 1979.
- [41] Solomon, M.H. and Finkel, R.A.  
The Roscoe Distributed Operating System.  
In *Proc. 7th Symp. on Operating System Principles*, pages 108-114. ACM, December, 1979.
- [42] Swan, R.J., Fuller, S.H., and Siewiorek, D.P.  
Cm\*: A Modular, Multi-Microprocessor.  
In *Proc. National Computer Conference*, pages 637-644. AFIPS, 1977.
- [43] Swan, R.J., Fuller, S.H., and Siewiorek, D.P.  
Cm\*: A Modular, Multi-Microprocessor.  
In *Proc. National Computer Conference*, pages 637-644. AFIPS, 1977.
- [44] Tobagi, Frank A.  
Multiaccess Protocols in Packet Communication Systems.  
*Trans. on Communications* COM-28(4), April, 1980.
- [45] Ward, S.A.  
*TRIX: A Network-Oriented Operating System*.  
Technical Report, MIT, December, 1979.
- [46] Wegner, P.  
*Programming with Ada: An Introduction by Means of Graduated Examples*.  
Prentice-Hall, 1980.
- [47] Wittie, Larry D.  
A Distributed Operating System for a Reconfigurable Network Computer.  
*Trans. on Computers*, 1980.
- [48] Wulf, W.A., Russell, D.B., and Habermann, A.N.  
BLISS: A Language for Systems Programming.  
*Communications of the ACM* 14(12):780-790, December, 1971.

- [49] Wulf, W.A., Cohen, E., Corwin, W., Jones, A.K., Levin, R., Pierson, C., and Pollack, F.  
HYDRA: The Kernel of a Multiprocessor Operating System.  
*Communications of the ACM* 17(6):337-345, June, 1974.
- [50] Zimmerman, Hubert.  
OSI Reference Model -- The ISO Model of Architecture for Open Systems Interconnection.  
*Trans. on Communications* , April, 1980.