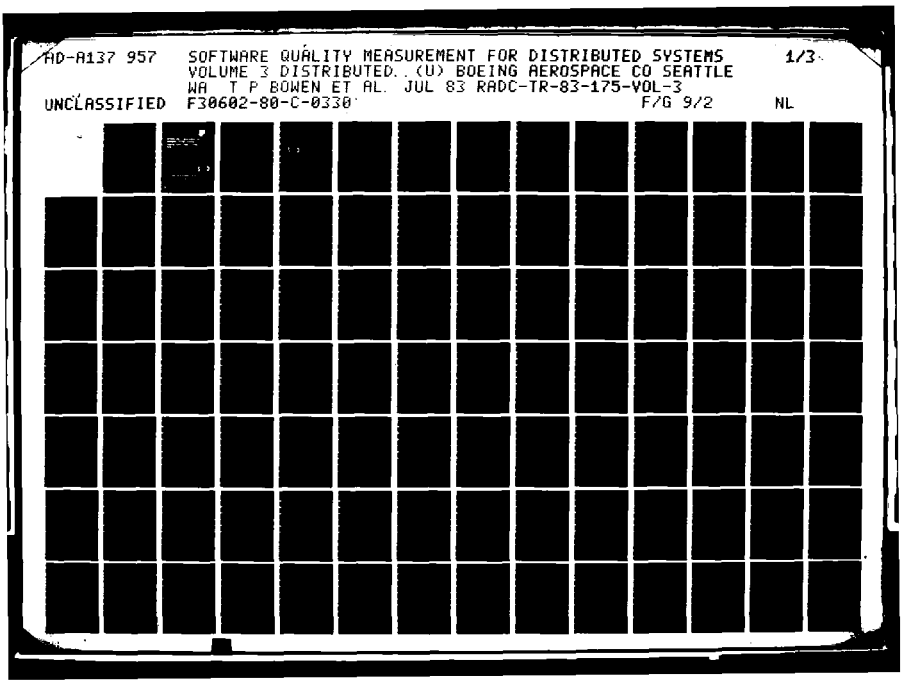
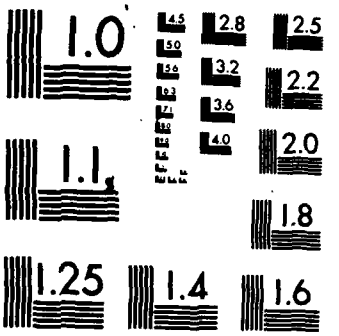

AD-A137 957 SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS 1/3
VOLUME 3 DISTRIBUTED..(U) BOEING AEROSPACE CO SEATTLE
WA T P BOWEN ET AL. JUL 83 RADC-TR-83-175-VOL-3
UNCLASSIFIED F30602-80-C-0330 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

RADC-TR-83-175, Vol III (of three)
Final Technical Report
July 1983

AD A137957

SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS Distributed Computing Systems: Impact on Software Quality

Boeing Aerospace Company

Thomas P. Bowen, Jonathan V. Post, Juitien Tsai, P. Edward Presson and Robert L. Schmidt

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-83-175, Vol III (of three)	2. GOVT ACCESSION NO.: AD-A137957
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS Distributed Computing Systems: Impact on Software Quality	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report October 80 - March 83
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Thomas P. Bowen, P. Edward Presson, Jonathan V. Post, Robert L. Schmidt, Juitien Tsai	8. CONTRACT OR GRANT NUMBER(s): F30602-80-C-0330
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Boeing Aerospace Company, PO Box 3999, Seattle WA 98124	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55812030
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COEE), Griffiss AFB NY 13441	12. REPORT DATE: July 1983
	13. NUMBER OF PAGES: 228
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: Joseph P. Cavano (COEE)

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

Software Quality
Software Metrics
Software Measurement
Distributed Computing Systems

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

Software metrics (or measurements) which are used to indicate and predict levels of software quality were extended from previous research to include considerations for distributed computing systems. Aspects of the products of software life-cycle activities which could affect the quality levels of software, and metrics to measure them, were identified. Two new quality factors, survivability and expandability, were validated. A Guidebook for Software Quality Measurement was produced to aid is setting quality goals,

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

applying metric measurements, and making quality level assessments. New metrics for interoperability and reusability were also included in the guidebook.

PREFACE

This document is Volume III of the final technical report (CDRL A003) for the Quality Metrics for Distributed Systems contract, number F30602-80-C-0330. The contract was performed for Rome Air Development Center (RADC) to provide methodology and technical guidance on software quality metrics to Air Force software acquisitions managers.

This report consists of three volumes as follows:

Volume I Software Quality Measurement for Distributed Systems - Final Report
Volume II Guidebook for Software Quality Measurement
Volume III Distributed Computing Systems: Impact on Software Quality

The objective of this contract was to conduct exploratory development of techniques to measure system quality with a perspective on both software and hardware from a life cycle viewpoint. The effort was expected to develop and validate metrics for software quality on networked computers and distributed systems; i.e., systems whose functions may be tightly distributed over microprocessors or specialized devices such as data base machines. At the same time, the effects hardware has on software was to be studied, as well as the trade-offs between hardware, firmware, and software. The results of this research are reported in Volume I.

Volume II describes the application of quality metrics to distributed systems and provides guidance for AF acquisition managers. The guidebook provides guidance for specifying and measuring the desired level of quality in a software product.

This volume
~~Volume III~~ describes a qualitative study of distributed system characteristics, reasons for selection, design strategies, topologies, scenarios, and trade-offs. These analyses led to the changes in the Framework shown in Volume I, and to the validation of models.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (cont'd)

LIST OF TABLES

SECTION 1
INTRODUCTION

1.1 OBJECTIVES OF RESEARCH

This work was performed under a research contract (F30602-80-C-0330) for Rome Air Development Center, Griffiss AFB, NY. The object of this effort was to develop techniques which can be used to measure distributed system software quality. The study looks at both hardware and software from a life cycle viewpoint, studies the effect that hardware or operating environments, i.e., distributed systems, have on software quality, and studies tradeoffs between hardware, firmware and software. This study was conducted to develop and validate proposed metrics for software quality on networked computers and distributed systems. This effort was also conducted to expand and refine the software quality measurement framework defined in prior Government (RADC) contracted work; Factors in Software Quality, F30602-76-C-0417 and Software Quality Metrics Enhancement F30602-78-C-0216.

The approach chosen to evaluate distributed systems is the software quality metrics methodology, which has been fruitfully applied to the study of a broad range of uniprocessor computers and embedded computer systems. In the 1970's, reliability was a factor closely identified with software and system quality. McCall and others identified eleven software quality factors and developed a system of metrics to predict and assess the degree of presence of these factors. Each factor is composed of a number of criteria which are further broken down into quantitative metrics. The eleven factors identified: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability. This approach has been extended to distributed systems in this research contract. The research has concentrated on identifying unique characteristics of distributed systems, and on the definition or redefinition of factors and criteria which can measure these characteristics. Three new software factors, four new system factors, and twelve new criteria have been described, and the factor of testability has been generalized into the factor of verifiability.

The results of this study are expected to result in a methodology for AF software acquisition managers to determine software quality requirements over a projects life

cycle in terms of its software quality factors and to specify or describe those requirements to contractors.

1.2 SCOPE OF VOLUME III

1.2.1 Relationship Between Volumes I, II, III

The work of this research contract (F30602-80-C-0330) is described in three volumes. Volume I, entitled "Software Quality Measurement for Distributed Systems - Final Report", includes an executive summary of the research effort, a description of how the software quality framework was enhanced to cover distributed systems, a detailed presentation of the current factor/criterion/metric framework, notes on enhancements which were considered but dropped from implementation, the validation of the quality model as applied to Expandability and Survivability, recommendations for future research, and several appendices. Volume III relates to Volume I as the background of study which led to the results reported in Volume I.

Volume II, entitled "Guidebook for Software Quality Measurement", describes the application of quality metrics to distributed systems and provides guidance for specifying and measuring the desired level of quality in a software product. Volume III may be used in conjunction with Volume II, as a set of qualitative analyses of distributed systems, one or more of which may be instructively similar to the particular software product under study.

1.2.2 Summary of Volume III

This volume discusses the transition from the old quality metrics framework to the new framework for distributed systems. Distributed computer systems have been variously defined, and these definitions are compared. An historical overview is provided, as well as the relationship to the current DoD software initiative.

Over 50 rationales are given for the selection of a distributed system rather than a uniprocessor system. These rationales are grouped into 9 reasons, where each reason is a high-level system acquisition goal. A matrix relates these rationales and reasons to those

quality factors most impacted. Within each reason there are tradeoffs, and tables illustrate the tradeoffs.

There are three areas which must be addressed in the design of a distributed system. These areas are: the distribution of control and processing, the structure and distribution of the data base, and the strategy for communication among the elements of the system. Successful design strategies in each area are outlined, with an emphasis on quality factors.

Distributed system topology is a term referring to the physical or logical pattern of interconnection of system components.

Aspects of distributed system topology discussed in this volume include:

* topology impact - how and why topology is related to system quality factors
* communications strategies - differing approaches to routing messages between nodes
* distributed system layers - the ISO Reference Model for interprocessor communication protocols
* distributed system architecture classification - what topologically different designs are possible, and their relative advantages
* distributed system hardware architecture (topology) impact on system quality
* distributed system hardware (topology) impact on software quality

Fourteen particular topologies are illustrated, from the familiar (loop, ring, bus) to the more exotic (cube-connected cycles, binary hypercube, shuffle-exchange).

A number of scenarios were developed that collectively cover many of the major system and software quality allocation issues that arise in distributed systems. Scenarios include: distributed command and control, distributed communications, distributed database, distributed avionics, distributed functional testing, distributed space systems, distributed virtual topology, distributed optoelectronics, and distributed microcircuit multiprocessor.

A tentative classification is introduced for the decisions available to software, hardware,

and system acquisition managers. The concept of a distributed system life cycle is outlined.

A number of firmware issues are explored. These include the difficulties of procurement, an example of Ada implementation, and the quality factors involved in hardware/firmware/software tradeoffs. A classification is introduced which emphasizes hardware/firmware/software tradeoff opportunities in memory management, operational management, I/O management, error monitoring, and special algorithmic capabilities.

Efficiency, virtuality, adaptation, performance, and reliability are discussed in terms of direct and indirect effects of hardware architecture on system quality. Quality factor emphasis and methodology as a whole is discussed, as well as qualitative relationships between correctness and reliability.

Collectively, these qualitative studies of distributed computing systems and their impacts on quality raised a number of issues which were not considered in the previous (uniprocessor) framework. New factors, criteria, and metrics were developed in order to quantitatively assess these issues. There is not a one-to-one mapping between the discussions and scenarios in this volume and the new factors, criteria, and metrics described in Volume I. Instead, this material illustrates the learning process of researchers in this contract which enabled the quality metrics framework to be extended to the more complex domain of distributed computing systems.

1.3 RELATED RESEARCH CONTRACTS

Rome Air Development Center (RADC) has been pursuing a program intended to achieve better control of software quality since 1976. This program has been seeking to identify the key issues and provide a valid methodology for specifying and measuring software quality requirements for software developed for major AF weapon systems.

In 1976 RADC and the Electronic Systems Division (ESD) sponsored an effort which defined a set of eleven user-oriented characteristics or quality factors (correctness, efficiency, integrity, usability, testability, flexibility, reusability, maintainability, reliability, portability, and interoperability) which extended throughout the software life-cycle. This effort established a hierarchical software quality measurement framework as

shown in Figure 1.3-1. The user-oriented factors, for use by acquisition managers to specify quality requirements, are at the top level. The software oriented criteria (attributes which indicate quality) and software metrics (quantitative measures of attributes) are at the second and third levels respectively. The metrics represent the most detailed level of the framework and completely define quality in terms of measurable elements. Taken collectively, this hierarchy formed the basis of a model for predicting and controlling software quality. This research was performed under contract F30602-76-C-0417, and was described in technical report RADC-TR-77-369, Factors in Software Quality.

In 1978, RADC and the US Army Computer Systems Command sponsored additional research to enhance this framework under contract F30602-78-C-0216, Software Quality Metrics Enhancement. The results of this effort were reported in technical report RADC-TR-80-109, Volume I, Software Quality Metrics Enhancement and Volume II, Software Quality Measurement Manual. The manual provides methodology to assist the AF acquisition manager in describing to a contractor what quality factors the manager considers the most important.

In 1979, RADC and the US Army Computer Systems Command issued contract F30602-79-C-0267 to develop an Automated Quality Measurement Tool for the H6180/GSOS Computer system. The purpose of this tool was to automate the collection of specific metric data, provide quality measurement assessments. This tool was delivered to the Air Force in September 81.

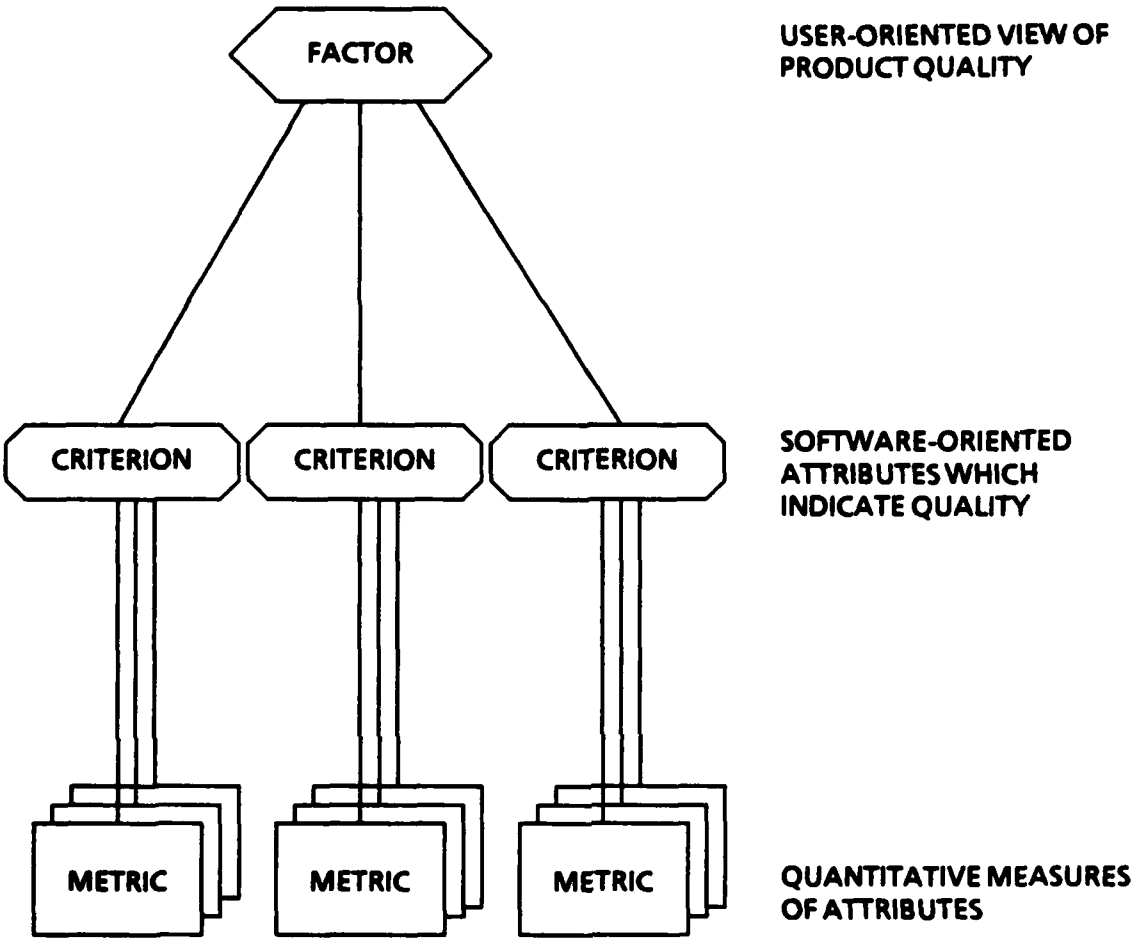


Figure 1.3-1 Software Quality Model

In 1980, RADC sponsored further research of the software factors of interoperability and reusability (Contract F30602-80-C-0265). The objective was to enhance the Software Metric Model by incorporating new findings for these two factors which had not been extensively studied in prior research contracts. The contract resulted in adding new criteria and metrics for these factors. The results of this effort are incorporated in Volume II of this report (Software Quality Measurements Manual).

In 1980, RADC also sponsored this research contract to extend the quality measurement framework to distributed systems and to transition the information into a form useful to the AF software acquisition manager. The prior research focused primarily on the software subsystem and largely ignored the total system aspects such as the computing hardware, operating system and communications network. The increasing demand and importance of distributed systems in future defense systems created a need to extend the framework to address system concerns which affect the emphasis and meaning of software quality. In moving from a uniprocessor system to a distributed system the system quality factors change the emphasis of a software quality factor, criterion or metric and required modifications and additions to the software quality framework. What is not a distributed system? One term for a non-distributed system is a "uniprocessor system". Another term is "monolithic". Yet another is "standalone".

It will be increasingly important to understand distributed computer systems. Some of their characteristics will emerge more extensively in future configurations. One characteristic peculiar to distributed systems, and of importance in the 80's, is geographic dispersion. The extent to which computers within a distributed system can be physically displaced from each other, range from the centimeter to the multi-thousand-kilometer. Computers will indeed be "tightly-coupled" over intercontinental distances by fiber-optics technology currently under research. Interconnection of even a very small percentage of available computers will be able to form distributed systems of complexity beyond those of today, since by 1999 there will be on the order of one billion computers in the world.

1.4 RELATIONSHIP TO THE DOD SOFTWARE INITIATIVE

The "Strategy for a DOD Software Initiative" (DSI 82) includes a section, Appendix II.6, on distributed systems. The definition used is: "In a distributed system, functions are apportioned among several processors that cooperate to complete a task." The

Department of Defense has played a major role in the evolution of distributed systems, particularly with the development of the ARPANET and its support of the National Software Works. This document emphasizes that "usabie results in this area cannot be expected in the short term; too little is known about the costs, risks and behavior of distributed systems.... The topic of distributed systems is so important, however, the opportunities in a number of areas should not be missed." Quality Metrics for Distributed Systems offers an approach to evaluating these "risks and behavior" in a quantitative way, so as to enhance the opportunities for further research and development.

SECTION 2
DISTRIBUTED SYSTEM CHARACTERISTICS

2.1 CHARACTERISTICS

In order to develop quality metrics for distributed systems, it is first necessary to define distributed systems. It is also necessary to identify the characteristics that distinguish them from uniprocessors or centralized processors, for which the original Quality Measurement Framework was developed.

The term "distributed system" is used loosely to refer to systems consisting of spatially separated components. The careless use of the term "distributed system" has led some authors, notably Enslow (ENS), to attempt to give a precise definition of the term. Enslow's basic approach is to require the system to meet a threshold of compliance with five criteria. The system should have:

1) a multiplicity of general purpose, dynamically assignable resources
2) physical distribution of the components of the system
3) a high-level operating system unifying the components
4) system transparency
5) cooperative autonomy

The intent of this definition (ENS) is to levy significant requirements on a system before it can be properly called "distributed". In this way, the term "distributed system" can be reserved for systems which have some promise of fulfilling the claims of improved system performance and functionality which are frequently made for distributed systems.

LeLann (LEL) takes an approach which focuses on the main characteristic of a distributed system, rather than on the requirements for a "good" distributed system. LeLann defines a system to be a collection of entities participating in the performance of system functions. A function is decentralized or distributed if none of the entities participating in the function knows the state of all the other participating entities. Distribution in the sense of LeLann becomes an unavoidable consequence rather than a choice if the system is large, because even moderate physical distribution can

introduce communications delays, which are significant on the scale of computer cycle times. Even for physically small systems, distribution may be chosen to avoid the overhead of maintaining global state information, or because quasi-autonomous operation of system components yields a superior approach to meeting system requirements.

Current technology is available to interconnect a variety of processing components. Standalone processing systems and sophisticated components can be interconnected in ways which were not fully considered during the component design process. There is a need to interconnect these systems and components into distributed systems which will provide functions and capabilities which are not available with single monolithic systems. For these systems, formed by the interconnection of component subsystems, state and control information must be distributed. This distribution is required not only for efficiency, but because the components of the systems are designed to manage and control local resources, and cannot efficiently function without some degree of autonomy.

For purposes of this report a distributed system is defined as a system formed by the interconnection of potentially autonomous systems to accomplish system functions cooperatively. This definition is consistent with the definition given by LeLann, and describes an area of significant interest. This definition is less restrictive than that given by Enslow, because it does not levy strict requirements on the forms of interconnection, cooperation, or on the functional qualities and capabilities of the system. A system need not be a "good" system to be distributed.

2.2 TERMS AND DEFINITIONS

Several terms associated with distributed systems lack the preciseness required to develop a theory for assessing the relationship between distributed systems, system quality factors, and software quality factors. These terms include: distributed, centralized, decentralized, tightly coupled, and loosely coupled. Definitions of these and other relevant terms have been developed to the extent necessary to formulate the relationship between distributed system characteristics and the software quality framework. These terms are defined in Appendix B. Approximately 80 of the terms referenced in "The DACS Glossary" (DACS) were deemed relevant to distributed systems. They are appended, and some are defined in greater depth; all appear in Appendix B.

2.3 EXAMPLES

2.3.1 The First Distributed Computer

The first distributed computer was built by John Mauchley and Presper Eckert. This was the BINAC, designed circa 1947 to be the first stellar navigation guidance system for intercontinental ballistic missiles. BINAC, which was also the first computer to use magnetic tape, was little known to the computer community, due to security considerations. In modern terminology, BINAC was clearly a distributed computer. It consisted of two CPUs, each capable of autonomous operation, tightly coupled for the purpose of redundancy, reliability, and testability. Mauchley acknowledges that the structure of the human brain with its two cerebral hemispheres was a guiding metaphor in the design of BINAC. (MAU 78)

2.3.2 The Brain is a Distributed Computer

Dr. Roger Sperry, in experiments performed at Caltech, established that the brain is a distributed computer. In Sperry's classic "split brain" experiments, apes and humans both showed this property. Each hemisphere of the brain has most of the capabilities of a entire brain. There are significant differences in the way each hemisphere processes information. Normally, the two hemispheres are tightly coupled by a nerve bundle called the Corpus Callosum. When this communications link is cut, by scalpel or by traumatic injury, the two halves of the brain begin to operate autonomously. Sometimes, this degrades performance. One hemisphere may direct an arm to move some object, and the other hemisphere doesn't know what the object is. Literally, "the left hand knows not what the right hand is doing." The two hemispheres process written and spoken language differently, but each has some linguistic competency. There may be benefit in decoupling the brain hemispheres. For some epileptic patients, the operation of last resort (cutting the corpus callosum) can prevent electrical seizures from spreading across the entire brain. This research is extremely important to research and development in the field of Computer Science. To pioneers in the field – Turing, Weiner, Von Neumann – the computer was a metaphor for the human brain. Better understanding of the brain's distributed functioning may help us to design more effectively functioning distributed computer systems.

2.3.3 The First Multiple Computer System

The first multiple computer system was created in 1954 when researchers at the National Bureau of Standards interconnected the two computers SEAC and DYSEAC. It was claimed (COD62) that this dual computer system could efficiently solve problems beyond the capabilities of the two individual computers. This led directly to the first proposal to be published on the construction of a multiple computer system, PILOT (LEI), (CUR).

PILOT, a project which was never completed, consisted of a main computer, a "clerical" computer, and a processor dedicated to I/O. "These computers intercommunicate in a way that permits all three to work together concurrently on a common problem. The system can be used in conjunction with other digital computer facilities forming an interconnected communication network in which all the machines can work together collaboratively on large-scale problems that are beyond the reach of any single machine."

The BINAC and PILOT projects illustrate that the concerns of distributed computing systems have been with us for over 30 years, but that the problems of design and implementation of distributed computers have been, at times, unmanageable. Modern management methodology precludes two computers from being casually linked together (as with SEAC and DYSEAC), but also lessens the likelihood of major innovations floundering (as with PILOT). The Quality Metrics methodology extended to distributed computing systems, gives additional prediction and control over distributed system design and implementation, thereby heeding the lessons of history.

2.4 LEXICOGRAPHY OF 'DISTRIBUTED'

'Distribute', from the Latin 'dis' + 'tribuere' (to allot), has six meanings classified by Webster's. Each has special significance to the new computer system usage of the word. The total meaning is probably an agglutination of these specialized meanings, modified by the contexts in which the term originally appeared in the technical literature. The original denotations are:

(1) To divide among several or many; to deal out; allot. This is the main

denotation in the computer context, but there are shades of meaning, connotations, to the synonyms. 'Deal' implies the delivery of a suitable portion to each member of a group. 'Dole' implies scantiness and periodicity. 'Deal' is more appropriate for multitasking, and 'dole' to timesharing.

(2) To dispense or administer, as justice. The implication here is that of carefully weighed or measured portions, and this is the sense of optimized resource allocation. Systems with a central authority, which resolves conflicts in priority between satellites, distributes the processing load in this sense.

(3) To spread out so as to cover a surface; as, to distribute fertilizer. This is the meaning most applicable to networking, distributed sensors, and systems with high geographic dispersion.

(4) To divide or separate, as into classes; to classify. This is the connotation most suitable for problem partitioning, frontend processing, or backend storage.

(5) (Logic) To use a term so as to convey information about every member of the class which it names; thus, the proposition 'All men are mortal' distributes the term 'man' but does not distribute 'mortal." This captures the sense of object-oriented processors, vector- and array-processors, relational data bases, and languages with the 'class' construct, such as Smalltalk.

(6) (Printing) To separate type matter that has been used and return the pieces to their compartments in the case. This connotation emphasizes the reusability of software, and the reconfigurability of identical or interperable hardware components.

In summary, 'distributed' has a number of shades of meaning in common English usage. Each of these is particularly relevent to some portion of the technical computer usage. Taken together, the term 'distributed' unites a number of otherwise separate concepts. 'Distributed' is itself distributed.

2.5 DISTRIBUTED SYSTEM HISTORICAL OVERVIEW

The distribution of computing resources to improve performance has been an accepted approach since the first enhancements to sequential computer architectures appeared. Most such early enhancements were initially based on overlapping of input/output functions with the arithmetic operations of sequential architectures. This same concept of overlapping and concurrency has been used to exploit the natural concurrencies and parallelisms inherent in the applications, exploitations which have emerged as major factors in achieving improved system performance. As the physical limits of performance were reached for the Von Neumann computer architecture, new architectures have emerged based on the exploitations of internal concurrency and parallelisms of the instruction and data streams. Such architectural concepts have served as the basis for the supercomputers of today, including the SDC/Burroughs PEPE, the Texas Instruments ASC, the Cray-1, the CDC STAR 100, the CDC 7600, and the Amdahl 47 V/6.

Distribution now permits utilization of several smaller, less complex processing components (minicomputers, microprocessors, and special-purpose processors) instead of a single, large-scale, centralized system. No longer do the long development/procurement cycles of monolithic super computers mandate early hardware commitment. Instead, the reduced component complexity associated with distribution significantly shortens development time and therefore reduces the requirement for early hardware commitment. No longer is the design approach based on fitting the application to the hardware—hardware which has been selected based on preliminary sizing estimates and dominant algorithms within the applications. These estimates and algorithms often change during the design and development process, owing to changing requirements or technology breakthroughs. Now the flexibility afforded by distributed systems allows the designer the freedom to fit the architecture to the applications; system tailoring is becoming the most effective design approach.

Using small, inexpensive, custom-designed computing units has led to numerous claims of benefits. However, the inherent complexity of the design problem is greatly increased by the distribution of functions, data base, and control. Distribution also adds a new dimension to system responsiveness—that is, interprocess communication. The fundamental centralized data processing assumption of instantaneous interprocess

communication response is no longer valid. As a result, process interactions become major factors in the design of distributed systems that then affect not only the responsiveness but also the control effectiveness and the overhead of the distributed system. These considerations reflect the need for a design approach for distributed systems that differ significantly from today's current ad hoc methods.

SECTION 3
REASONS FOR SELECTION OF DISTRIBUTED SYSTEMS

Quality factors that receive more emphasis in distributed systems may be identified by analyzing why a distributed system architecture is selected. An analysis was conducted in terms of reasons (high-level system acquisition goals) and rationales (potential benefits) for selection, which have been identified through a literature search and Boeing Aerospace Company experience. Nine important reasons for selecting a distributed system architecture are listed below; examples, in terms of rationales and their impact on system quality factors are discussed for each reason.

The relationship between reasons, rationales, and the distributed system quality factors are identified in the matrix of Table 3.0-1. The approach to using this matrix is described as follows. If a set of rationales has been provided for the selection of a distributed system, based on some set of specifications or requirements, then the number of times each system quality factor is identified in total gives an indication of the relative priorities for the quality factors. If only preliminary and top-level specifications or requirements exist, then the basis for selection of a distributed system may be less specific: a reason or set of reasons. The breakdown of reasons into rationales, guided by the emphasis of particular quality factors, can allow the top-level set of reasons to be resolved into a more specific and detailed set of rationales. On the other hand, if a particular set of quality factors has been identified as top priority within specifications or requirements, then the rationales which impact those quality factors need to be taken into account.

3.1 REASON #1: IMPROVE RESPONSE TIME

Distributed processing systems can utilize decentralized resources to provide parallelism and load balancing. The net effect should be improved performance of the application program. This is predicated on being able to partition applications into components which run in parallel and on developing algorithms to solve the dynamic load balancing problem. There are a number of new criteria and metrics to add when distributed systems are considered. These reflect the rationales of enhanced processing and data parallelism. All of these impact flexibility, efficiency, and usability.

Reasons and Rationales for Selection of Distributed Systems / Software Quality Factors	Correctness	Maintainability	Reliability	Flexibility	Verifiability	Portability	Reuseability	Efficiency	Useability	Integrity	Interoperability	Survivability	Expandability
1. Improve Response Time													
o Concurrency of Diagnosis with Normal Operation	X	X	X		X							X	
o Enhanced Data Parallelism				X				X	X				
o Minimize Memory/Processor Communication Time								X					
o Allow Optimal Partitioning of Workload				X			X						
o Load Leveling				X				X	X				
o Real-Time Coordination of Multiple Subsystems												X	
2. Improve Processing and Accessing Capabilities													
o Automatic Job Segmenting				X			X	X					X
o Partitioning of Functionality	X	X		X		X		X			X	X	X
o Increased Variety of Processing Modes				X					X		X		X
o Resource Uniformity	X	X			X	X	X						
o Specialized Hardware: Database Machine					X			X					
o Interoperability with Existing Systems						X	X						
3. Reduce Cost													
o Lower Cost to Upgrade			X	X			X						X
o Local Administrative Approval of Components									X		X		
o New Topological Configurations on Demand		X	X	X					X			X	
o Lower Initial Cost								X	X				
o Increased Procurability									X				
o Increased Deployability		X		X								X	
o Lower Total Weight		X		X					X				
o Lower Total Power Consumption		X		X					X			X	
o Network Topology Optimization			X	X					X			X	
o Resource Sharing			X	X			X		X		X		
4. Reduce Vulnerability to Hardware Error													
o Redundancy at Each Node		X	X										
o Tolerance to Node Failure	X	X	X		X				X	X		X	
o Tolerance to Communications Link Failure		X	X	X	X					X		X	
o Capability for Isolating Failed Components		X	X		X					X		X	
o Diagnosis of Failure to Least Replaceable Unit		X	X		X								
o Repair Without Interruption			X	X				X	X	X		X	

Table 3.0-1: Relationship Between Reasons/Rationales and Software Quality Factors (Page 1 of 3)

Reasons and Rationales for Selection of Distributed Systems / Software Quality Factors	Correctness	Maintainability	Reliability	Flexibility	Verifiability	Portability	Reuseability	Efficiency	Useability	Integrity	Interoperability	Survivability	Expandability
5. Replace Hardwired Logic with Microprocessor													
o Resource Uniformity	X	X			X	X	X						X
o Reconfigurability		X	X	X							X	X	
o Machine Independence	X					X	X				X		X
o Delayed Commitment to Specific Node Hardware		X		X							X		
o Multiplicity of Vendors		X	X	X							X		X
o Reconfigurability Through Low-Cost Hardware				X				X	X				
6. Improve Thruput													
o Distribute Jobs to Several Nodes Concurrently				X				X					X
o Exploitation of Uniform Interchange Media			X			X	X				X		X
o Enhanced Data Parallelism								X	X				
o Enhanced Computational Parallelism				X				X					X
o Optimal Partitioning of Workload			X					X	X				
o Reduce Load on Host			X	X				X				X	
o Distributed Operating System				X								X	X
o Eliminate Multiprogramming		X			X			X	X				
7. Improve Survivability													
o Security on Hierarchical Network									X	X		X	
o System Protection from Overload		X	X		X					X		X	
o Backup Redundancy	X		X	X								X	X
o Restoration/Recovery	X	X	X		X			X		X		X	
o Endurance/Hardening		X	X							X		X	
8. Improve Sensor Performance													
o Distributed Sensors	X	X	X	X					X			X	X
o Distributed Effectors			X	X					X			X	X
o Intelligent Sensor Clusters		X		X									
o Deployable Sensor Arrays			X	X	X							X	
o Concurrent Multi-Spectral Scanning			X	X				X					

Table 3.0-1: Relationship Between Reasons/Rationales and Software Quality Factors (Page 2 of 3)

Reasons and Rationales for Selection of Distributed Systems / Software Quality Factors	Correctness	Maintainability	Reliability	Flexibility	Verifiability	Portability	Reuseability	Efficiency	Useability	Integrity	Interoperability	Survivability	Expandability
9. Improve Geographic Disperson													
o User Distribution				X			X	X	X		X		X
o Gateway to National/International Network						X			X	X	X		X
o Global C^3I Applications	X		X						X	X	X	X	X
o Space Systems Networks		X	X	X						X	X	X	X
o Need for Mobile Nodes		X	X	X					X				
o Need for Distributed Database Management	X						X	X	X	X			X
o Adaptive Routing						X			X	X		X	X
NUMBER OF OCCURRENCES	11	24	27	32	12	8	11	22	24	13	15	26	26

Table 3.0-1: Relationship Between Reasons/Rationales and Software Quality Factors (Page 3 of 3)

The response time for a distributed system depends on a number of variables. Two variables are particularly important in considering tradeoffs between architectures and strategies. As shown in Table 3.1-1, the number of processors and memories, and the amount of interprocessor communication, serve to distinguish between a number of operating regimes.

If there is only one processor, and interprocessor communication is low (cards, tape, or disk hand-carried) then we have the classical uniprocessor situation, which is outside the scope of this study. If the degree of interprocessor communication is high (communication between two uniprocessors does not imply a true distributed system) then the quality factor of interoperability is of paramount consideration.

If the number of processors and memories is low (2-50) then there are two approaches to consider to improve response time. If interprocessor communication is low, then response time depends on static and quasi-static allocation of resources. Response time can be improved by appropriate partitioning of functionality, so that each processor has the data and programs to solve a specifically isolated subproblem. This requires careful matching of functional decomposition to architectural configuration, and is a static allocation problem (parameters do not during system operation). The quasi-static and dynamic analogy of this approach is to make real-time changes in the allocation of hardware resources to software components, on the basis of run-time measurements. This is the approach known as load balancing, and is jointly the responsibility of the operating system and the system operator.

If the number of processors and memories is low, but a high degree of interprocessor communication is essential, then there are at least two approaches to improving response time. First, the use of sophisticated multiprocessor-multiprogramming operating systems. Second, there are cases when a data base machine can improve response time. In the data base machine, including the relational data base machine types, the functionality of data base query, data base access, data base maintenance is handled by special-purpose hardware and software, leaving the remaining components of the distributed system to transform the data in application-dependent ways.

Table 3.1-1 Improve Response Time

Interprocessor Communication	Number of processors/memories		
	1	Low	High
Low	classical uniprocessor	partitioning, load-balancing	Array processors, pipelining
High	interoperability concerns	multiprocessor-multiprogramming, database machine	Cross-bar switching, *banyan networks, *omega networks

***Note:** Cross-point switching too expensive for over 50 processors (See IEEE Spectrum "Computing at the Speed-limit", July '82, p.27

If the number of processors is high (greater than 50) and interprocessor communication is low, then the two main approaches to improving response time are pipelining and array processing. In pipelining, data passes in a stream through a series of processors. Each processor performs a specific function on the data stream in a manner similar to assembly-line operation. The situation in array processing is topologically unique. Each processor is connected to a fixed number of other processors. For example, to the four processors immediately to the North, South, East and West. The rectangular matrix of processors so connected is uniquely adapted to solve problems in which the data itself is arranged in a rectangular matrix, and in which processing at each point depends mostly on data at the neighboring points.

When the number of processors and memories is high, and the degree of interprocessor communication is high, the greatest demands are placed on the system, and the tradeoffs for improving response time are most critical. Cross-bar switching, in which each processor can be dynamically interconnected to any other processor or memory, becomes too expensive for much beyond 50 processors. As the IEEE Spectrum article "Computing at the Speed-Limit" (July '82, p. 27) surveys, special configurations such as banyan networks and omega networks become the optimal structures for distributed systems.

An additional rationale for the selection of a distributed system is that diagnosis and maintenance of hardware and software components can frequently continue concurrently with the normal operation of the system. The remaining components operate, perhaps in a slightly degraded response time, whicle the isolated component or subsystem are inspected by automatic test equipment (ATE) or built-in test equipment (BITE). This is more likely if the maintainability, reliability, and testability factors were given high priority early in the system life cycle, and the concurrent diagnosis rationale was designed into the distributed system. As an added benefit to this type of design, the Survivability of the system can be predicted to be significantly higher.

3.2 REASON #2: IMPROVE PROCESSING & ACCESSING CAPABILITIES

In addition to providing potential performance improvements, distributed processing systems may provide a more natural computer achitecture for applications such as artificial intelligence, real time command and control, and data base management.

These applications are large and complex and are characterized by many processes which can run in parallel, for example, as a pipeline. Making software modules correspond (nearly) one to one to (dedicated) hardware modules (partitioning of functionality) may provide both easier problem decomposition and ultimately less software complexity. For example, most multiprogramming may be eliminated. The elimination of multiprogramming in favor of multiprocessing can improve processing capability by reducing the operating system's overhead in allocating an expanded system even if computers from a new vendor are added. The rationale of partitioning of functionality impacts several system quality factors. For example: correctness and maintainability may both be emphasized, if complex software which is difficult to verify, validate, and maintain, is replaced by hard-wired units; flexibility may be enhanced, but can also be degraded if the decomposition is an awkward one; and efficiency is enhanced by special-purpose fast hardware.

As seen in Table 3.2-1, systems may be classified according to whether the number of processes is one, low, or high, and on whether the number of processors is one, low, or high.

If there is only one processor and only one process, the conventional uniprocessor-uniprocessing regime applies, and is outside the scope of this study. Similarly, if one processor has a low number of processes to execute concurrently, the known approaches of uniprocessor-multiprogramming suffice. When the one processor has a high number of processes, some type of pipelining or time-sharing becomes feasible in some instances.

If a low number of processors must cooperate on a single process, then the proper partitioning of functionality is required to maximize the automony of each processor, and to reduce interprocessor communication overhead. If the low number of processors handle a low number of processes, then a centralized multiprocessor-multiprogramming executive, or a resource-sharing operating system is required to improve processing and accessing capabilities. The case of a low number of processors handling a high number of processes is typical of the distributed artificial intelligence viewpoint, which is discussed in Section 4.1.2 and in Reference (DAI).

If one process is attacked simultaneously by a high number of processors, then the

Table 3.2-1 Improve Processing and Accessing Capabilities

Number of Processors	Number of Processes		
	1	Low	High
1	uniprocessor-uniprocessing	uniprocessor-multiprogramming	time-sharing, pipelining
Low	functionality partitioning	multiprocessor-multiprogramming, resource sharing	distributed artificial intelligence
High	array machine, Staran, Illiac IV FMP	functionality-partitioning + multi-microprocessors, CHIP,TRAC, data flow systolic arrays	prohibitive communication overhead, banyan networks

FMP: Flow Model Processor
Burroughs/NASA-Ames

TRAC: Texas Reconfigurable
Array Computer,
W. Texas - Austin

CHIP: Configurable Highly
Parallel multiprocessor,
Purdue

array machine approach (exemplified by the Illiac IV and the Goodyear Staran) may improve processing and accessing capabilities. Another approach is the FMP, or Flow Model Processor, being developed by Burroughs for NASA Ames Research Center. If a low number of processes are resident in a high number of processors, then the application may determine one of a number of approaches, including functionality partitioning, multi-micprocessors, the CHIP (Configurable Highly Parallel) multiprocessor at Purdue, the TRAC (Texas Reconfigurable Array Computer) at the University of Texas-Austin, or the so-called dataflow systolic arrays.

When a high number of processors and a high number of processes are both requirements, there may be prohibitive communications overhead, or pressure to adopt a banyan network topology may trade off against logical constraints on network topology.

Correctness is a quality issue when functionality is partitioned or when resources are not uniform (i.e., from different vendors, or of low interoperability). If the partitioning is not done well, then the interfaces between the separate processors (each performing its own partition of the total functionality) become complex, confusing, and difficult to verify. If an unnecessary multiplicity of vendors is involved, the maintainability of the system may also suffer.

To some users, automatic job segmenting, partitioning of functionality, and increasing the variety of processing modes are all desirable approaches because of the way the flexibility of the distributed system can be improved. Other users discover that on systems of uniform resources (high interoperability) the same design features that partition functionality also add to the reusability of the system (at least of one or more partitions).

3.3 REASON #3: REDUCE COST

Several aspects of cost reduction impact system quality factors. Technological changes which enable distributed systems have added uncertainty to the size/cost relationship. Grosch's law is a rule of thumb relating computer size and cost. One statement of Grosch's law is that computing power goes up as the square of cost. For uniprocessor systems, this makes attractive the policy of putting maximum investment in one large mainframe. For distributed systems, however, this tendency towards centralization is opposed by a differing price/performance curve than for uniprocessors. Grosch's law for centralized systems is no longer valid. We should now be able to tie together a multiplicity of inexpensive components to achieve the performance we need, and can afford to downgrade the importance of processor utilization to concentrate on other quality goals such as system reliability, flexibility and expandability and "Reduce Load on Host." An especially important rationale, (reducing cost) is "resource sharing." This is one of the most frequently mentioned cost-reduction rationales for the selection of a distributed system in those cases where pre-existing hardware consisting of a number of stand-alone centralized systems, which evolve through the distributed system life cycle into a local network, tied together by new system software and interface hardware.

As shown in Figure 3.3-1, Grosch's law does not apply directly to distributed systems. The cost of multiple processors, as isolated units, may actually rise more slowly than linearly as the computing power increases, because of supply economics and the possibility of discounts for ordering multiple units simultaneously. The problem comes in considering the cost of interprocessor communication. This tends to increase faster than linearly with the number of processors, unless special-purpose hardware (such as the Boeing Aerospace Companies Transition-Machine) handles the interface. The total cost of the distributed system may be higher than that of a uniprocessor system up to a point, but beyond that point, as Grosch's law takes hold, the uniprocessor costs more for increasing computing power, and the distributed system costs do not rise as fast. Therefore, there may be a situation where the distributed system is automatically cheaper than the uniprocessor system for the same level of computing power.

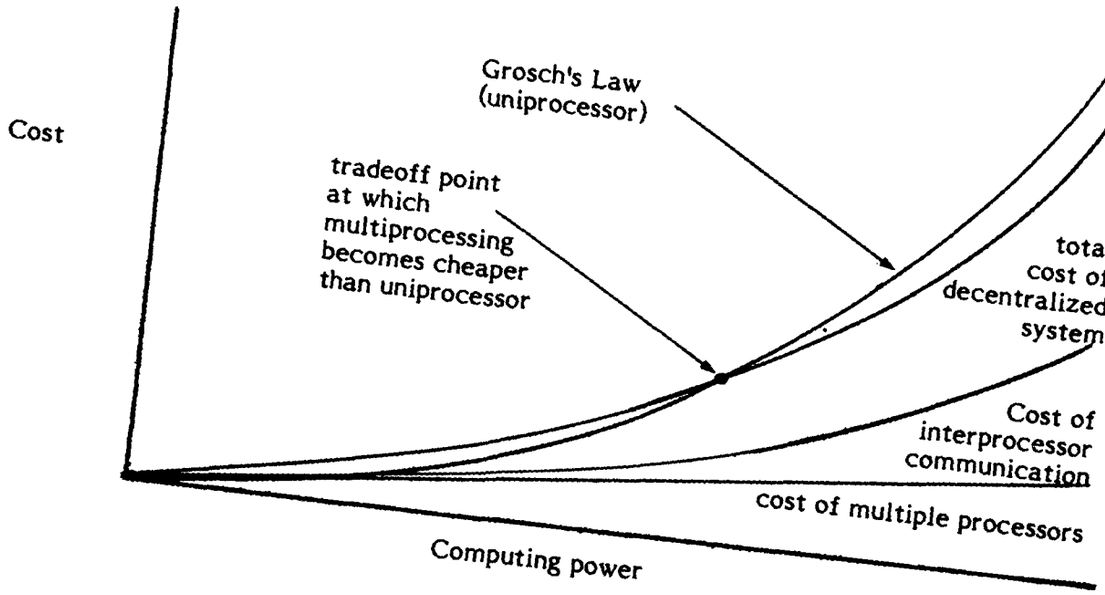


Figure 3.3-1 Reduce Cost: Grosch's Law

When organizations own a large, rapidly increasing number of computer resources and wish to tie them together for resource sharing, the problem is getting the decentralized components to cooperate and function as a harmonious whole. With multiple processors tied together as a single system, resource sharing will be a normal mode of operation for distributed systems. Computer networks also provide resource sharing, but usually require considerable applications programming effort, knowledge about how the resources are distributed, and limitations to certain kinds of resources (e.g., files). Therefore, to achieve resource sharing, the system quality factors most emphasized (as seen in Table 3.0-1) are flexibility (in the use of available hardware and software, and applications programming), reusability (of both hardware and software), usability (use of existing interfaces to users, minimal retraining), interoperability (emphasis on interfacing heterogeneous hardware and other resources), and reliability (since a properly configured system of shared resources can be more reliable than its parts).

As seen in Table 3.0-1, there are additional interactions between the 10 rationales for cost reduction and the 13 quality factors. Maintainability, for example, is improved if the system is dynamically reconfigurable (able to support new topological configurations on demand). The same upgrades in hardware (to reduce weight and power consumption, to shrink in size for deployment, microminiaturization) can also improve the efficiency of the system (the smaller devices can sometimes run faster) and can improve the system survivability (because smaller, cheaper parts can be hardened more cost-effectively).

3.4 REASON #4: REDUCE VULNERABILITY TO HARDWARE ERROR

The decentralized control of a multiplicity of elements provides the potential for higher reliability than is now possible in systems which have more vulnerable central control and/or central state tables. Decentralizing control in and of itself is not sufficient to ensure reliability; many other things must be considered, such as error detection, error categorization, error confinement, and system reconfiguration, restart, and recovery after an error. This is split into two reasons, reduce vulnerability to hardware error, and improve survivability, plus a number of rationales.

As seen in Table 3.4-1, we consider both centralized and decentralized systems composed of both high reliability and low reliability components. In a centralized system with high-reliability components, it is usually possible to improve system reliability at reasonable cost only by improving either the "weakest link" in the system, or by improving the most common component.

A centralized system with low-reliability components is rendered reliable, maintainable, and correct by means of redundancy (either hot standby or majority logic), the use of special error-detection hardware and software, or by built-in-testing. Flexibility and testability then become important factors, and the level of survivability may depend upon these factors having sufficiently high values.

In a decentralized system with high reliability components, the effort to reduce vulnerability to hardware error leads to concentration on software reliability and system maintainability. To build a decentralized system from low reliability components, the system may demand dynamic reconfigurability, so that on component failure, the system may reallocate functionality to undamaged components. Alternatively, there may be redundancy at each node, and redundant communication channels between nodes. The costs of reconfigurability (mostly software) or redundancy (mostly hardware) must be traded off against the costs of upgrading to higher reliability components.

Table 3.4-1 Reduce Vulnerability to Hardware Error

System	High Reliability	Low Reliability
Centralized	improve reliability of "weakest link" or most common component	redundancy, error detection, built-in-test
Decentralized	concentrate on software reliability and system maintainability	dynamic reconfiguration, redundancy at each node, redundant communications

Other factors besides Reliability are related to the rationales of reducing vulnerability to hardware error. Maintainability, in particular, is highly correlated with reliability. The issues of tolerance to node and link failures, redundancy, the locatability of failed components (i.e. can automatic test equipment (ATE) or built-in test equipment (BITE) isolate failures to line-replaceable unit (LRU), to a particular processor, to a particular board, to a single chip or connector) are all crucial issues for predicting or improving the level of maintainability. The survivability of the system is also highly sensitive to these considerations. Further, the integrity of the system is also sensitive to these issues, as may be seen by replacing the term "hardware error" by the term "deliberate or accidental intervention at the system level by personnel."

3.5 REASON #5: REPLACE HARDWIRED LOGIC WITH MICROPROCESSOR

Resource uniformity is a rationale that emphasizes the general purpose applicability of a microprocessor over the special purpose complexities of hard-wired systems. The system quality factor maintainability is enhanced, as is testability, because all microprocessors of a given type can be tested and repaired similarly. Portability and reusability are enhanced, as software written for one microprocessor in one application can be transported to another, which is not the case for hard-wired systems. There is a tradeoff between the lost cost and high efficiency of hardware, and the flexibility of software in microprocessors within a distributed system. As the functions become more clearly defined, much that is now in software can be efficiently implemented in special purpose hardware. During early phases of a project, however, the lower efficiency of a general purpose microprocessor is more than compensated by the usability and flexibility of the more easily reconfigurable system. Machine independence, which is also a criterion of reusability, is sometimes given as a rationale for the replacement of hardwired logic by a microprocessor.

The critical factor in these tradeoffs is flexibility. Table 3.5-1 shows the categorization according to hardware being of low or high flexibility, and software being of low or high flexibility. If both hardware and software are at a low level of flexibility, we may have a hard-wired special-purpose system. This type of system is difficult or expensive to change. But this type of system also has the highest potential benefits for total replacement by microprocessor equipment.

Table 3.5-1 Replace Hardwired Logic with Microprocessor

SOFTWARE	HARDWARE	
	low flexibility	high flexibility
low flexibility	hard-wired special-purpose system: very difficult or expensive to change, greatest potential improvement by replacement with microprocessor	general-purpose microprocessor w/ specialized software- high potential evolvability via S/W
high flexibility	advanced software development system + downloading to specialized target processor	distributed system of microprocessors

An example of low flexibility software and high flexibility hardware is the general-purpose microprocessor running specialized applications software. This system may have exceptionally high evolvability by upgrading the software alone. The case of high flexibility software and low flexibility hardware may be seen when an advanced software development system (which allows software to be generated at high productivity levels) is used to develop software, compile it, and down-load the assembly language code to a specialized target processor. Although the hardware is low flexibility, the system as a whole has high flexibility, evolvability, and expandability, because a new compiler can allow the same (or slightly modified) high-level language code to be retargeted to a new target processor.

A distributed system of microprocessors has high flexibility software and hardware. Therefore such systems have the greatest degree of variety, and lessons learned from developing one such system may have little or no application to developing another such system.

Besides the flexibility/efficiency tradeoffs discussed above, and the maintainability, testability, and reusability issues mentioned in the first paragraph of this subsection on replacing hardwired logic with microprocessors, there are other system quality factors to be considered. As shown in Figure 3.0-1, interoperability is strongly related to the machine-independence of the design (a fully machine-independent module of software can be brought up in a great variety of processors with no software changes at all) and to the multiplicity of vendors (each vendor can guarantee interoperability between compatible components of their own manufacture). These same rationales (machine-independence and resource uniformity) relate to the portability of the system, because software written for one class of microprocessor can easily be run on another microprocessor based on the same chip, or supporting the same set of operations, or interface compatible (MIL-STD-1553).

If the component processors are implemented on chips, one way to compose them is to wire them together. This solution is inflexible because the components are dedicated to a particular problem and cannot be used for another problem. Another compositional scheme is to join the processors to a bus as "peripherals". This is more flexible since a processor can be used in different phases, but the bus becomes a bottleneck and time is wasted in interphase data movement.

Still another approach is to replace the dedicated processing elements with more general microprocessors and simply program the algorithmically specialized processing function. This solution is much more flexible because different components can use the same devices by changing programs, provided the interconnection pattern is the same. The bus bottleneck is eliminated. There is a loss in performance with this polymorphism, however, since circuit implementation of the primitive actions is replaced by the slower process of instruction execution.

3.6 REASON #6: IMPROVE THRUPUT

Distributed processing systems, because of their decentralization of control algorithms and state information, have the potential for easy and cheap extensibility (a criterion of flexibility), both physically and logically. The emphasis is on reducing cost (lower cost to upgrade), and improving processing and accessing capabilities by expanding the current system (interoperability with existing systems). In such systems we should be able to add new computing elements without changing any of the algorithms, essentially having only their control information automatically updated. Multiprocessors and computer networks are physically extensible but the former usually have a small upper bound on the number of processors. Furthermore, in practice they often require more substantial modifications for logical extensibility. For distributed systems, we therefore need to measure logical extensibility as a new metric in the criterion of extensibility. There is also the potential impact of specialized hardware. For example, prototype hardware at Boeing (the Transition Machine) (VAU) enables tight coupling between an arbitrary number of processors and memories, with control information handled by hardware rather than being limited by software system overhead. The extensibility of networks depends in part upon communications protocols, not currently evaluated in terms of quality metrics. Lower cost to upgrade is impacted by reliability (does the system continue to provide reliable service when expanded?), flexibility (was the future growth of the system specified and encouraged by flexible design), and reusability (since a maximum of existing hardware, software, and other resources must continue in use). Interoperability with existing systems is enhanced if software portability is high, as this allows existing software to be used in multiple processes to available resources.

Table 3.6-1 shows how this approach depends on whether there are few or many processes, and whether there is a small or large amount of data. If there is a small amount of data, and few processes, then thruput as such is no problem. This is known as the SISD (single-instruction stream, single data-stream) situation. Integrity, reusability, portability, testability, reliability, maintainability, and correctness are all expected to be high in such a system. The reason is that these "Von Neumann" architectures are the best understood and have the greatest degree of experience associated with their use. The tradeoff in using such a system is typically seen in flexibility, efficiency, survivability, and usability. The system may not be as flexible in working with inherently parallel data. It will be less efficient (in terms of operations per

Table 3.6-1 Improve Thruput

Data	Processes	
	Few	Many
small	thruput no problem: SISD (single-instruction stream, single data stream)	distributed microprocessors with replicated database, local area net
large	SIMD (single-instruction stream, multi-data stream) Centralized system may be adequate; consider special database machine	distributed system with distributed (not replicated) database; MIMD (multi-instruction multi-data; high cost of network overhead

Note: "Faster components made with gallium arsenide or Josephson Junction devices can increase computing speed 10 times if current uniprocessor architectures are used; however, with the new architectures, there is hope of increasing speed 100 to 1000 times."

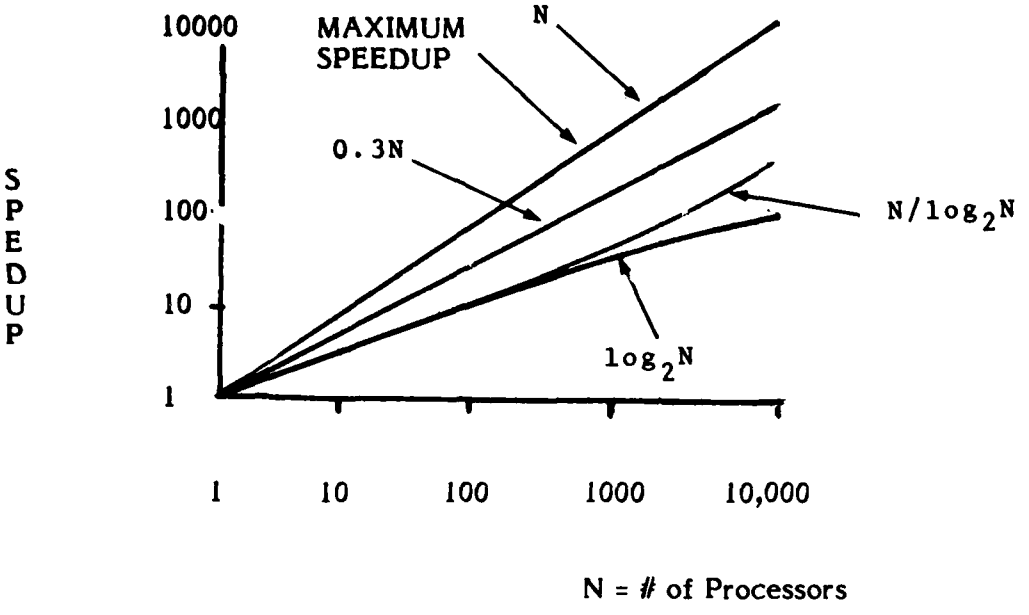
second per processor). It will be lower in survivability, as either the single data stream or the single instruction stream, if interrupted, will reduce system performance to zero.

If there is a large amount of data which is driving only a few processes, then a centralized system may be adequate. A special database machine may effectively divide the data-acquisition function from the data-processing function, and thereby improve thruput. In general, this is the SIMD (Single-instruction stream, multidata stream) situation. This improves efficiency and usability. Efficiency in terms of transactions per second is increased because of the concurrent operation of database machine and mainframe. Usability is also improved, as the user does not need to know where the data physically resides, just what operations are performed on the data.

If there is a small amount of data, but many processes executing concurrently, then a local area net may be appropriate. Alternatively, it may be effective to maintain a replicated database, with the same data at each node of the system, and software used to ensure that the data remains consistent from node to node.

The most difficult case occurs when there is a large amount of data and many processes. It is no longer feasible to replicate the database. Some sort of distributed database is essential, involving, at the least, data segmentation or partitioning. There is usually a high network overhead cost. This is the MIMD (multi-instruction multi-data) situation.

The issue of maximum speedup as a function of number of processes becomes important. Figure 3.6-1 shows four different theories, between which considerable debate continues at this time.



(a) Computer with both high-speed and low-speed modes will be dominated by low-speed mode, even if high-speed is infinitely fast.

(b) S = computer speed $\sim \frac{1}{F_h T_h + F_l T_l}$

F_h, F_l: fraction of results generated in high, low modes

T_h, T_l: time to generate a single useful result in high, low modes

(c) Worlton's corollary: multiprocessor Amdahl's law,

$$S = \frac{N}{\sum_{i=1}^{N} \frac{F_i}{i}} \sim$$ where F_i: fraction of results by i^{th} processor

(d) Equal distribution: if $F_i = \frac{1}{N}$, $S = \frac{N}{N \sum_{i=1}^{N} \frac{1}{i}} \xrightarrow[N \to \infty]{} \frac{N}{\log N}$

Figure 3.6-1 Amdahl's Law

3.7 REASON #7: IMPROVE SURVIVABILITY

In the analysis of distributed computing systems, both nodes (processors) and links (communication lines) are considered. For the quality factor of survivability, both nodes and links must be taken into account. This is shown in Table 3.7-1.

The nodes may be soft, or the nodes may be hardened. The links may be soft, or the links may be hardened. Depending on which of the four combinations of hardware approaches have been selected, the strategy for increasing the system survivability may vary, and may vary in the application of hardware and software.

If nodes are soft and links are soft, then high survivability for the distributed system can be achieved primarily through high redundancy (multiple processors at each node, as many nodes as feasible) and by high interconnectedness (as many ways as feasible to allow for communication between any node and any other). Only by such redundancy can the system be survivable while each node or link is vulnerable. This redundancy is summarized in the rationale "backup redundancy".

If the nodes are hardened but the links are soft, then survivability is related to expandability of the technology for the links, to bring their hardness up to the level of the nodes. This includes such approaches an anti-jamming, frequency-hop, and fiber optic cables. Software for such a system should be oriented towards high autonomy. That is, maximum use should be made of single processors operating on their own as much as possible. Software should attempt to limit dependancy on interprocessor communication over the vulnerable links. The rationale "endurance/hardening" applies, but primarily to the links. The rationale "restoration/recovery" applies, but in the sense of restoration and recovery of the network, assuming that most nodes remain operational and it is primarily links which need to be re-established or rerouted. In a distributed system of hardened node and soft links, it may be a mis-allocation of effort to further harden the nodes, and may be most valuable to make the links hard.

If the nodes are soft but the links are hardened, then the survivability of the system is most closely related to improving node reliability. Improving survivability is attained by improving the extent to which the system can determine the status of the individual nodes, and by being able to exploit that information. Rather than acquiring large

Table 3.7-1 Improve Survivability

Links (Communication Lines)	Nodes (Processors)	
	Soft	Hardened
Soft	very high redundancy, very high interconnectedness	high autonomy, anti-jamming, frequency-hop, fiber optics
Hardened	built-in-test, dynamic reconfigurability	most-expensive systems; concern shifts from hardware to software survivability

amounts of software for testing the status of links, which are presumed to be hardened, resources may be dedicated to testing node status. This means an increased emphasis on built-in-test (BITE) and on the ability for diagnostic equipment or automatic test euipment at a remote site to test a potentially damaged node. This utilizes the links to improve system measurement of nodes. This includes the rationales "backup redundancy" and "restoration/recovery." The latter rationale includes the emphasis on dynamic reconfigurability. In this strategy, the software is specified to be able to allow for new topologies of interconnection to be established, in real-time or near-real-time, so that nodes which remain operational can best be connected across the hardened links. Dynamic reconfigurability, as a strategy for restoration/recovery, allows the system to bypass inoperational nodes, and make maximum use of undamaged processors and hard-to-damage links.

The strategy of hardening both the nodes and the links, while ideal from the point of view of survivability alone, is the most expensive of all the options described. In such an environment, concern for system survivability shifts from hardware to software survivability. Improvements in the hardness of either nodes or links will have only marginal effect on total survivability. The methods of concern involve the increase in sophistication of the software at each node. One problem which does not go away as nodes and links are hardened is the problem of security. The integrity of the system, to both accidental and deliberate access and modification of software, is not improved by the hardening of nodes and links to systemic attack in a tactical environment. Indeed, hardening may actually increase the problems of security. This is do to, firstly, the psychological effect of a hardened system -- users tend to stop looking for potential problems when disruption from outside is minimized, and this can lower their attention to the problem of disruption from inside the system. Secondly, the mechanisms for increasing the hardness of the system may provide additional points of vulnerability to accidental and deliberate violations of integrity. For example, allowing for reconfigurability may allow for unexpected interconnection of nodes which are thought to be isolated from each other. In a system where users are less concerned by a node becoming temporarily inoperational, because they expect it to recover and restart automatically, there is the danger of integrity being breached in the interval between the node going down and the node coming back up. This sort of interplay between integrity and survivability deserves further consideration. It is indicated by the rationale "security on hierarchical network."

3.8 REASON #8: IMPROVE SENSOR PERFORMANCE

One of the reasons for the selection of distributed computing systems is motivated by the strategies for the development of sensors. The number of sensors in a system may be increased over time, and this places burdens on the computational resources needed to process sensor information. Sensors may also evolve to increased "smartness", and this changes the requirements for computation. The very fact that sensors may be geographically dispersed over a wide area leads to considerations of dispersing the computing system itself to cover the same area. These two issues of number of sensors and sensor intelligence are shown in Table 3.8-1.

When there are few sensors, and the sensors are "dumb" (i.e., they deliver raw data to the computer without performing any pre-processing themselves) then the system performance is typically low, and the reliability of the system is a major concern. With few sensors, the loss of even one sensor may degrade the system performance to an unacceptable degree. With dumb sensors, changes in the environment cannot be compensated for (recalibration) and must be corrected at the computational level, which may be an unacceptable load on the computational resources, or may slow system response time. The great advantage of such a system is low cost. One use for software is to calculate the optimal position for sensor deployment.

Where there are few sensors, but each sensor is "smart", then it is possible for most processing to be done at the sensor sites themselves, and there is minimal need for centralized computational facilities. Because smart sensors cost more than dumb sensors, there is a tradeoff between number of sensors and smartness of sensors. One example of a compromise solution to this tradeoff is indicated by the rationale "intelligent sensor clusters". In this case, there is one microprocessor for each of a cluster of closely distributed sensors, and these clusters themselves may be widely dispersed.

If there are many sensors, with each sensor dumb, the distributed microprocessors each serving multiple sensors is an appropriate strategy. Centralized processing may also be employed, but analog-digital conversion may be a bottleneck.

Where there are many sensors, each one smart, there is the highest degree of system

Table 3.8-1 Improve Sensor Performance

Number of Sensors	Sensor Intelligence	
	Dumb	Smart
Few	low system performance, low cost, reliability concerns	most processing done at sensor sites,
Many	centralized processing or distributed micros each serving multiple sensors	highest performance, highest cost, try to distribute sensor processing

performance. However, this approach also produces the highest cost. To improve sensor performance, the data processing of sensor data should be distributed as highly as possible.

All the above arguments apply as well to distributed effectors, as well as distributed sensors. One example of distributed effectors is the flap and slat controls in an aircraft. Boeing's approach on the 757 and 767 (see section 6.1.4) is to have a distributed effector network, with a microprocessor for each effector/sensor pair. In general, this limits weight and power requirements, while delivering high reliability and performance.

The problems of reducing data from a distributed array of sensors have traditionally been related to processing speed of a centralized data analysis computer. In recent years it has become feasible to deploy "smart" sensors, each of which processes some of its own data with low cost microelectronics. The factor reliability applies in four ways: (1) error tolerance of the system to malfunctioning sensors, (2) consistency of data from independent sensors, (3) accuracy of estimation of sensor location or calibration, and (4) the trade-off of sensor simplicity versus sensor smartness. The factor of maintainability applies, especially if the sensors are in inaccessible locations. Survivability is an issue if the sensors are operational in a target-rich environment. The proposed MX basing project provided examples of distributed sensors, as each of 4600 missile shelters is protected by intruder security alarms. The factors of reliablity and integrity are impacted, since each sensor must perform some filtering operations to reduce the number of false alarms in the network to which they are connected. MX also provided an example of distributed effectors, as each of 200 missile transporters is automatically maneuvered from roadbed to shelter by a redundant set of Z8000 and MIL-STD-1750 microprocessors, operating without human intervention. Reliability is of primary importance, as unreliable hardware or software can do serious physical damage. In general, reliability increases in importance for mission-critical and life-critical operations.

3.9 REASON #9: IMPROVE GEOGRAPHIC DISPERSION

The term "distributed" does not mean physically spread out over a large area. This characteristic has been termed "geographic dispersion." For some systems, the

personnel are geographically dispersed (user dispersion), and the communications function is of highest importance in the distributed computer system which connects them. This applies to national networks such as ARPANET and international networks such as WWMCCS, as well as to smaller systems for tactical operations in battlefield or search-and-rescue. Geographic dispersion is closely related to the problems of mobile nodes, from hand-held units to global C3 applications such as the E-3A AWACS and the E-4B Command Post aircrafts. Additional complications apply for space systems, where tracking, attenuation, transmission delay, weight, EMP hardening, and other considerations apply in design and operations. A further example of geographic dispersion occurs when databases are scattered over a large area, long distances from the locations where the data needs to be processed. The system quality factors most heavily impacted in such cases are integrity (especially the criterion of access control), survivability, flexibility (especially the criterion of expandability), interoperability (in the case of networks), and usability.

One reason for the selection of a distributed system is to improve geographic dispersion. The rationales associated with this reason all involve the actual physical dispersement of nodes. As shown in Figure 3.9-1, there are three classes of geographic dispersion, by distance. These are, in order of increasing dispersion, I/O Buses, Local Nets, and Long-Haul Networks. There are areas of overlap between these regimes. For example, a hundred meters (0.1 kilometers) may be the separation of two processors tightly coupled through an I/O bus, or may represent the distance between two distinct nodes of a local net.

Local computer networks were defined to be those networks serving a more limited geographic area. However, the extent of this "more limited" area may still vary considerably, with substantial impact on the suitability of different technologies. The following rough categories, each differing, as they do, by an order of magnitude probably serve to identify the major differences in this requirement:

Same component - distances measured in 0.1 meters
Same system - distances measured in meters
Same room - distances measured in 10's of meters
Same building - distances measured in 100's of meters

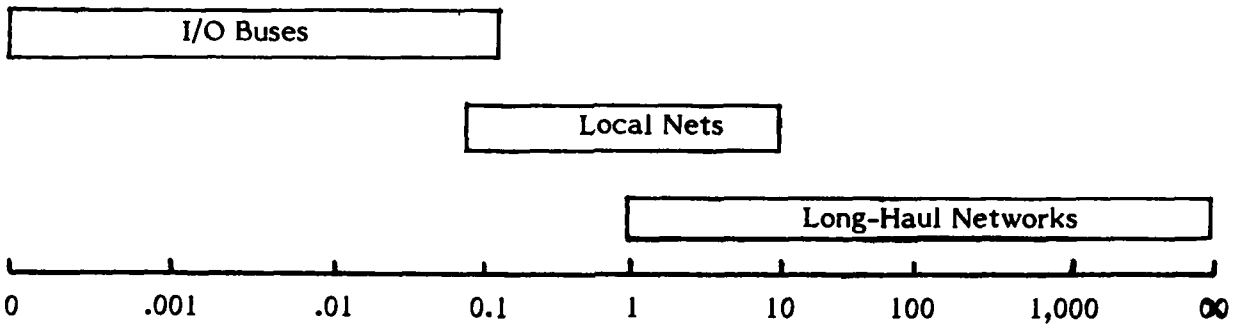


Figure 3.9-1 Extent of Geographic Dispersion

Same plant	- distances measured in kilometers
Same city	- distances measured in 10's of kilometers
Same region	- distances measured in 100's of kilometers

As shown in Table 3.9-1, the distinction between centralized and dispersed may be drawn either between users or between processors. If the users and the processors are both centralized, this is a conventional uniprocessor configuration, and not within the scope of this report.

If the users are centralized but the processors are geographically dispersed, we may have a system which is distributed. For example, a surveillance system with smart sensors may have sensors geographically dispersed on a global scale, yet have all personnel at a central command site. A distributed avionics system may have microprocessors at a number of sites throughout an aircraft, including flight controls, radar, and weapon fire control, but the users are centrally deployed in the cockpit.

The requirements for distributed avionics system will lead to a new generation of multiplexed data busses. The Society of Automotive Engineers (SAE) High Speed Data Bus subcommittee, a successor to the SAE task group that developed th MIL-STD-1553B data bus standard, describes the primary characteristics as follows (AW&ST, 23 Aug 82, p. 77):

* Capability to integrate several avionics subsystems that share common sensor systems.

* Capability for total airframe/avionics integration.

* Distributed bus control: increase Reliability and Maintainability by achieving maximum fault isolation, while providing high Flexibility and Expandability.

* Data rate of 20 megabits/second (Expandability to VHSIC technology)

* Split channel system -- data transferred on one channel and bus control signals on other channel to maximize bandwidth usage (Efficiency)

Table 3.9-1 Improve Geographic Dispersion

Users	Processors	
	Centralized	Dispersed
Centralized	conventional uniprocessor	surveillance with smart sensors, distributed avionics
Dispersed	E-3A AWACS E-4B Command Post P-3B UDACS B-1B Avionics	ARPANET, WWMCCS, MPRT

* Fiber optic or coaxial cable -- single bus can handle 20 audio intercom stations as well as data -- Interoperability

* Bus messages of up to 4K 16-bit words -- Efficiency

* Bus sizes up to 64 terminals -- Expandability

A space system network is an even more extreme example of this. In this case, sensors may be dispersed over orbits of tens of thousands of kilometers, while personnel are at a central, hardened site.

Processors may be centralized while users are dispersed. This is the case for conventional time-sharing systems. This also applies to flight systems such as E-3A AWACS, E-4B Command Post, P-3B UDACS, or B-1B Avionics, where users are distributed between cockpit, airborne stations, and ground stations, while the computational behavior of a single centralized processor is being accessed.

Both users and processors may be dispersed. Networks such as ARPANET and WWMCCS for this description. The Boeing Morgantown Personal Rapid Transit system (MPRT) has users dispersed through a set of people-mover cars, and processors dispersed, 2 at each station (one backup).

Another extreme example of processor dispersion and user dispersion in the Navstar Global Positioning System (GPS) discussed in section 6.1.6. Each of a number of satellites has its own processor, and users with backpack receivers represent highly dispersed mobile ground stations in the field. Each user, regardless of geographical position, can determine latitude and longitude to great precision. This is a system designed with the ultimate in geographic dispersion and usability.

The quality factor correctness is closely related to the rationale "need for distributed database management." A discussion of the correctness issue for replicated databases may be found in section 4.3.

Maintainability is particularly important for Space Systems Networks, because of the extreme difficulty of repairing an in-orbit component, and the difficulty of maintaining

space system software from ground-based sites. Maintainability is also cruicial for the rationale "need for mobile nodes", because of the additional logistical complexity of repair and update of mobile units in the field.

Reliability is significant for the same rationales as maintainability, as well as for "Global C3I applications." The latter correlation is mentioned by users who consider the generally slower response time of global systems than local networks, and the increased risk associated with the propagation of actions from a defective component in a global system, with the WWMCCS false alarm problem being mentioned as a cautionary example.

Usability is correlated with almost all of the rationales within the reason "improve geographic dispersion." The cause of this is that the typical user need not be appraised of the actual geographic topology of a dispersed distributed system, but will find the system more usable if a simplified "virtual" system is represented by the human interface. This is discussed in greater depth in the comments on virtuality, in section 8.1.2.

Integrity is a major issue for the rationales "Gateway to National/International Network", "Global C3I Application", "Space Systems Network", "Need for Distributed Database Management", and "Adaptive Routing". This is discussed in the last paragraph of 3.7, above.

SECTION 4
DISTRIBUTED SYSTEM DESIGN STRATEGIES

Three strategies that are commonly used in the design of distributed systems are distributed intelligence, distributed resources and distributed data bases. These strategies are summarized in Table 4.0-1. Within these areas there are other related considerations that must be addressed; distribution of control and processing, structure and control of the data base and communication among the elements of the system. For example:

- A strategy for the distribution of control and processing is necessary to achieve the goal of higher throughput by parallel processing. Design decisions for the distribution of control and processing can directly affect other quality factors such as: availability, reliability and survivability.

- A strategy organizing the database, and controlling access to the database is necessary to provide access to a common database by multiple processors. Design decisions for databases can have a direct impact on quality factors such as reliability, availability, flexibility, survivability and efficiency. A major tradeoff in distributed database design is between providing high access bandwidth to the data and controlling the consistency and concurrency of the database.

- A strategy for communication among the various elements of a distributed system is required to provide access to common data as well as to pass control information among the elements of the system. The choice of communication strategies can directly affect quality factors such as correctness, integrity and reliability.

Control, database, and communication strategies are all closely related. For example, many consistency and concurrency control strategies for databases assume highly reliable communication. However, the system correctness and integrity for some of these strategies deteriorates rapidly as communications reliability deminishes. Similarly, some process synchronization and control strategies may require highly reliable communication. The system quality factors also depend upon the interrelationships among

Table 4.0-1 DISTRIBUTED SYSTEM DESIGN STRATEGIES

DISTRIBUTED INTELLIGENCE	DISTRIBUTED RESOURCES	DISTRIBUTED DATABASE
Control Activities	Peer Communication	Multiple Data
Processing Activities	Local Network	Monitor
Scheduling	Backend Storage	Concurrent Access
Planning	Problem Partitioning	Segmenting
Inference	Frontend Processing	Network Topology
Problem Solving		Protocols
Focus of Computation		
Knowledge Systems		
Task Cooperation		
Sensors		
Execution Monitoring		

control, data, and communication strategies as well as on the qualities of each area separately. The significant issues impacting quality factors are discussed in the following sections for each of the three design strategies.

4.1 DISTRIBUTED INTELLIGENCE

A distributed intelligence system emphasizes the process aspects of a distributed system. The system is represented as a set of functions that are distributed over a number of processors. Transactions or events occur at one or more processors and a response to the transaction or event requires execution of functions located throughout the system. In this type of system, response time is usually a performance or system efficiency objective.

Factors involved in the design of the system at the processor level include speed of communication between processors, the degree of cohesion between functions performed at different processors, and the degree of coupling between functions at different processors. Table 4.1-1 shows the effect of different combinations of these factors on distributed intelligence system efficiency. If the hardware technology supports the application, the system efficiency objectives will not require a high degree of software efficiency. If the hardware technology does not support the efficiency objective, allocation of efficiency is often made to software or trades with other system quality and software quality factors such as flexibility, reliability, maintainability will be made.

Various "intelligent" aspects of a computer system may be distributed. These include control activities, processing activities, scheduling, planning, inference, and so on. In 4.1.1 the system quality aspects of distributed control, task coordination, and monitoring are summarized. In 4.1.2 system quality aspects of distributed problem solving, distributed focus of computation, distributed knowledge systems, distributed planning, distributed task cooperation, distributed sensors, distributed execution monitoring, and other concerns collectively known as distributed artificial intelligence (DAI) are summarized.

Table 4.1-1 Effect of Design Factors on Distributed Intelligence System Efficiency

SOFTWARE		HARDWARE	
Function	Balance	Tightly Coupled	Loosely Coupled
Tightly Coupled (Cohesion High)	Compute Bound (Coupling low)	Unacceptable because 1) overkill 2) potential memory conflict	Acceptable because I/O delays are transparent to processing
	I/O Bound (Coupling High)	Ideal application for distributed systems	Unacceptable because I/O delays further reduce efficiency
Loosely Coupled (Cohesion Low)	Compute Bound (Coupling Low)	Unacceptable because 1) overkill 2) potential memory conflict	Ideal application for distributed system
	I/O Bound (Coupling High)	Acceptable because I/O delays	Unacceptable because I/O delays further reduce efficiency

4.1.1 Distributed Control Strategies

A system with more than one processing element requires a strategy for coordinating processing activity and for distributing the processing load among the elements of the system. The execution of tasks must be coordinated to ensure the correct time order of dependent tasks. When parallel processing is required to meet the processing load, processing must be coordinated so that tasks can be executed concurrently. When control is centralized, tasks are scheduled and distributed based on information about the state of the system and the queue of pending tasks. Since centralized information about the state of the system must be collected and maintained, a centralized control strategy requires suitable communication and database strategies to maintain state information about the system. When control is decentralized, task coordination is based on local or regional information. The required information about the system state is more easily maintained. The correct function of the global system must be achieved by structural aspects of the system design. Alternatively, some centralized control must exist to monitor the system and direct the system when abnormal functioning is detected.

One of the major errors made during system design is improper allocation to system components. A proper framework for relating system quality to software quality should address this problem. This requires an analysis of the allocation of quality factors to hardware, firmware, operating system, and application software. Furthermore, such an analysis must compare and contrast allocation strategies for single processor systems and distributed systems.

How is an understanding of system quality factors applicable to the process of designing high--quality distributed systems? System quality evaluation has several applications to systems design. Theories and models are developed to represent the behavior of computer systems --and new systems invoke the need for new theories, while new theories may suggest the design of new computer systems. Evaluation techniques are developed for comparing models and theories, and for determining which models best fit the available data. Special purpose instrumentation is built to gather data on computer systems, and this instrumentation affects and is affected by system design philosophy.

The gathering, analysis, and representation of performance and quality data plays a major role in the functional enhancement of existing systems -- in terms of incremental design, upgrading, retrofitting, and fine-tuning. Finally, the quality of planned systems may be predicted or estimated by these methods, so the very existence of future systems can depend on the evaluation of quality during the conceptual and validation phases of the system life cycle.

4.1.2 Distributed Artificial Intelligence

Distributed Artificial Intelligence (DAI) is concerned with those problems for which a single problem solver, single machine, or single locus of computation seems inappropriate. DAI involves multiple, distinct problem solvers each embodied in its own system. On 9-11 June 1980 a group of 22 people gathered at an MIT conference center for the first workshop on DAI. (see ACM SIGART Newsletter, October 1980). Some of the main points raised, and their relevance to distributed system quality, include:

* Is it easier to coordinate the activities of 20 machines than to build one machine 20 times as large? (Interoperability, Efficiency)

* Is a clear functional decomposition possible, so that one function may be allocated to each machine? (Modularity)

* If a large problem is spread over multiple distinct machines, then each machine only knows part of the problem. How can we ensure effective problem solving in the face of incomplete information? (Completeness, Effectiveness)

* Are there approaches to problem solving that are less sensitive to inconsistency? (Correctness, Flexibility)

* Given a collection of relatively autonomous problem solvers, how do we ensure coherent behavior? How can we prevent the system from working at cross purposes? (Correctness)

* With current technology, it is much cheaper to compute than to communicate; bandwidth is thus the limiting resource. When can tightly coupled systems be

replaced by loosely coupled systems which spend more time computing and less time communicating? (Efficiency)

* Distributed processing system often have their origin in an attempt to synthesize a network of machines capable of carrying out a number of widely disparate tasks. This often leads to a concern with issues such as access control and protection, and results in viewing cooperation as a form of compromise between potentially conflicting views and desires at the level of system design and configuration. In distributed problem solving, on the other hand, there is a single task envisioned for the system; the resources to be applied have no other predefined roles to carry out. This means that we view cooperation in terms of benevolent problem solving behavior, i.e., how can systems that are perfectly willing to accomodate one another act so as to be an effective team? (Flexibility, Integrity)

  Our concerns are thus with developing frameworks for cooperative behavior between willing entities, rather than frameworks for enforcing cooperation as a form of compromise between potentially incompatible entities.

* To be sufficiently "intelligent", a system may have to be so complex and may have to contain so much knowledge that it will be able to function efficiently only if it is partitioned into many loosely coupled subsystems. (Efficiency)

* Work in DAI helps sharpen our intuitions and techniques for explicit reasoning about knowledge, actions, deductions, and planning. (Understandability)

* Methods used by one DAI system for reasoning about the actions of other DAI systems will also be useful for reasoning about other dynamic processes in the environment. (Reusability)

* Hierarchical planning: Techniques that enable a system to incorporate the effects of the actions of other systems in its plan can also be used to incorporate the effects of low-level actions in higher levels of the plan. (Modularity)

* Information gathering: Techniques that enable a system to ask another system for information can also be used to acquire information through sensors. (Interoperability)

* Execution monitoring: Techniques that enable a system to check the performance of other systems can be used in monitoring the execution of its own plans. (Correctness)

* In DAI, component agents are themselves rather complex AI systems that can generate and execute plans, make inferences, and communicate with each other. (Modularity)

* Advantages of DAI could include: graceful (fail-soft) degradation characteristics (no single agent needs to be indispensable), upwards extensibility (new agents can be added without requiring major system redesign), and communication efficiency (a message-sending agent can plan its "communication acts" carefully taking into account the planning and inference abilities of the receiving agents). (Survivability, Expandability, Efficiency)

* An example: The CMU Distributed Sensor Net is a testbed project designed to explore the issues and problems of distributed signal understanding. The basic system will locate, identify, and track a single remotely controlled model vehicle in a simulated terrestrial environment monitored by an array of acoustic sensors. The long-term goal of the system is to have it serve as an intelligent assistant which can make decisions in the presence of uncertainty, conflict, delay and lack of a priori hypotheses.

The work of Victor Lesser at the University of Massachusetts at Amherst is a particularly promising attempt to provide a synthesis of artifical intelligence and distributed processing methodologies.

See also (SMI 77), (SMI 79), (LES 79), (LES 80)

4.2 DISTRIBUTED RESOURCES

A distributed resources system emphasizes hardware and software resources that are available to a user. The system often contains heterogeneous computers with widely varying capabilities (e.g. amount of mass storage, line printer capacity, special purpose simulators, etc.). There may be heterogeneous operating systems even among homogeneous computers.

The task of merging a set of heterogeneous components into a distributed resource network is primarily systems engineering. The task involves defining a set of interfaces - hardware, communication protocol, data translators, and uniform command language, file structure and data base structure, to provide a uniform interface for all users regardless of the particular processor or terminal being used.

In such a system the most important quality factors are

1. interoperability - to enable computers and operating system to communicate
2. portability - to enable an application program to have the greatest local accessibility
3. integrity - to protect unauthorized access to proprietary data and programs
4. usability - to isolate the user from peculiarities of a particular operating system, and
5. flexibility - to enable growth with minimum impact on system software.

Distributed resource systems also require increased functionality. The increased functionality can be implemented in hardware, firmware, or software. Examples of this increased functionality are: addition of a new node to a network and updating of network addressing tables; systems performance optimization; error detection and recovery; and, data base update or restoration.

There are quantitative methods for trading off system resources. The scheduling of resources for a single multiprogrammed CPU is a well-understood theory. While there is some similar analysis for task decomposition and allocation in parallel deterministic multiprocessors, the theory is less refined and little applied (COF). Much work needs

to be done in the modeling of multiprocessor system scheduling with intermediate-speed interfaces, processor heterogeneity, and job-transfer overhead (FAY).

4.2.1 System Designs

In practice, the design strategy of a distributed system is usually organized by system functions. The functional organization of a system is a large factor in determining constraints for the performance of the system with respect to system quality.

4.2.2 Peer Commmunication

Several autonomous standalone processors can be interconnected into a network for the purpose of peer communication. Each processor is an equal (peer). Generally, any processor can initiate activity on the network. Each processor has control over local participation in transactions. A node in a peer communication system may function independently of other nodes; communication takes place through message exchange. The ARPA network is a classic example of a peer communication network.

In a peer communication network major design issues include: method of interconnection, costs associated with interconnection, recovery from temporary node unavailability, standardized data communication formats and protocols, communications congestion, delay, and failure recovery. The quality impacts of the distributed aspect of a peer communication are limited to the impacts of the reliability and availability of other nodes and of the communications resource. In many respects, a peer communication network is very similar to a multiuser environment on a single processor from the point of view of each node (user).

4.2.3 Peer Communication in a Local Network

Local networks are networks with high bandwidth communications at interactive rates. Typically the communications facility is under the direct control of the organization using it. Peer communication in a local network can be used to build systems of processes using interprocess communication. These systems are similar to a multiprocess system on a single computer. Because the processes can be physically distributed, resources at several sites can be integrated into a cohesive system.

For a local network, the coupling between processes can be high. Because several processors can be active simultaneously, timing effects can have a significant impact on system quality factors. For example, verification of the system using simulation techniques can be difficult because failures cannot be dependably repeated. Many of these effects are present in multiple process systems on a single processor. See Figure 4.2-1 for examples of local area networks.

4.2.4 Backend Storage

In a backend storage system, the major function is to provide access to 'remote' storage devices. Several processors access or share space on the same storage peripherals. Two strategies can be identified: (1) partitioned sharing on a large or expensive peripheral (e.g. disk), (2) common access to a shared database.

In partitioned sharing, access to the device is shared but access to the data is not shared. In this case, a scheme in hardware or firmware is needed to allow multiple access to the device. Access bandwidth to the device may be reduced by contention and resource waits. Partitioned sharing is functionally similar to multiuser computer systems where several users share the same disk but operate (generally) on different files.

When backend storage is used to provide access to a shared database, the design strategy must include methods for concurrency control and coordinated access. While access bandwidth to a remote peripheral may be high, access latency may require that data be transfered in relatively large blocks. Access via large blocks makes concurrency control easier because access is less frequent; however access bandwidth may be reduced because access reservations for large blocks are more likely to be in conflict. The quality impact is to efficiency.

4.2.5 Problem Partitioning

When the computational workload exceeds single-processor technology, a system may be designed to partition the workload among several processors. This approach is

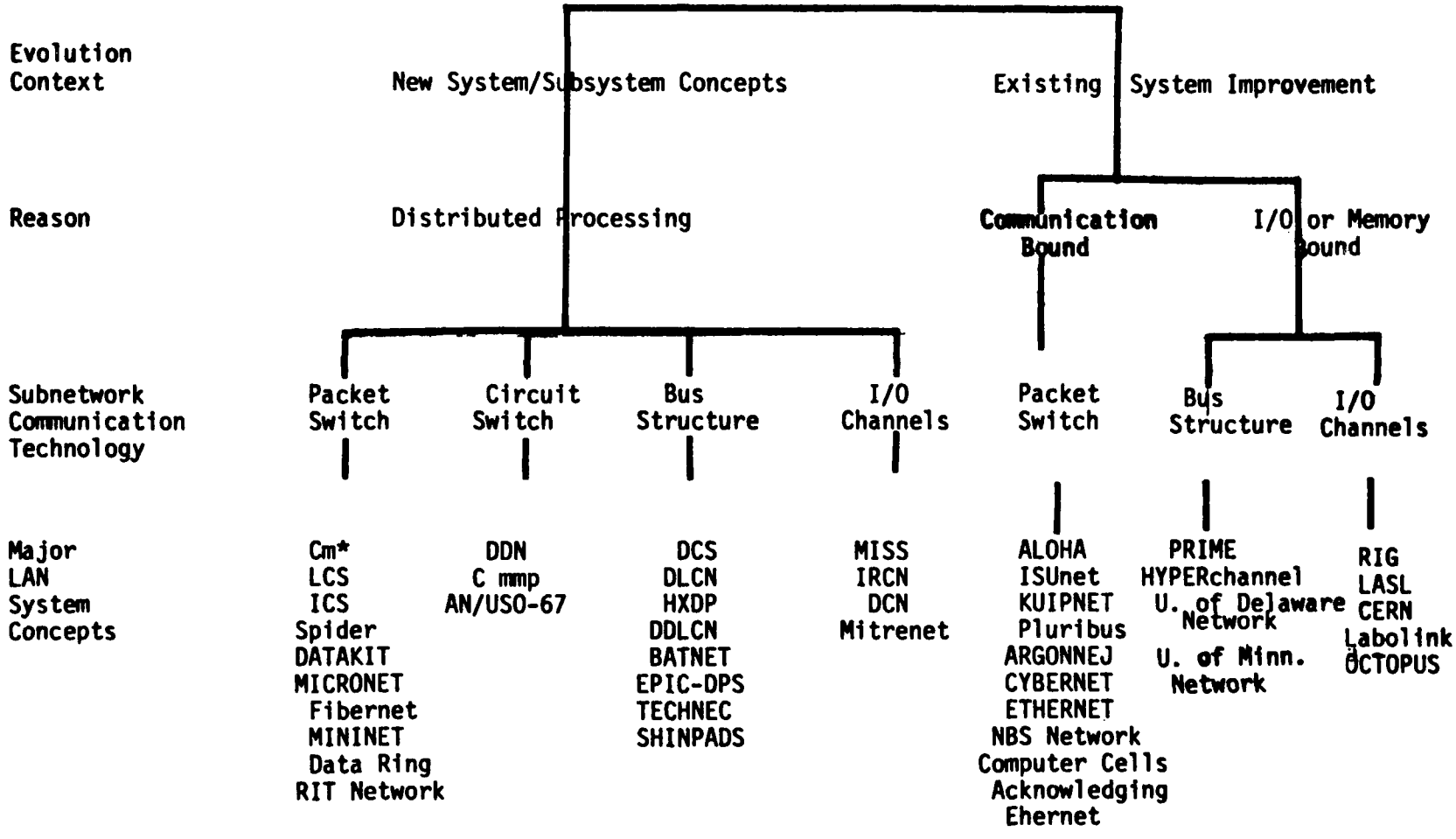


Figure 4.2-1: Taxonomy and Examples of Local Area Networks

easiest to implement when the workload can be functionally partitioned at a high level. For example, the navigation and the target detection functions on a missile can operate on separate but communicating processors. A more complicated approach is to partition the workload in realtime by matching the workload to available processing resources dynamically.

When a problem is partitioned, communications, control and database strategies are necessary to retain the degree of coordination available in a single processor solution. High bandwidth, low latency communications without high level protocols may be necessary. Similarly, control methods such as synchronization to a single hardware clock may be necessary. Database concurrency and consistency may require access scheduling to be built into the design. High coupling between processors at the data and control level adversely affects software quality by reducing modularity, testability, and maintainability.

4.2.6 Frontend Processing

A special form of problem partitioning is frontend processing. To efficiently use a large central resource such as a mainframe, frontend processors can be used. Front-end processing can be used to package and schedule jobs for the main processor, to provide I/O services for the mainframe, and to preprocess and package data.

When frontend processors are used for scheduling, the goal usually is to increase the efficient utilization of the main processor. For example, the CRAY mainframe is most efficiently used to perform computations such as vector operations. Mini-computers are used as frontends to present the workload to the CRAY mainframe, offloading functions which do not make efficient use of the high speed main processor.

Frontend processors are also used to manage I/O devices. For example, on the CDC CYBER, there are several 'peripheral processing units' which are used to provide interface processing for peripheral devices. These simplify the scheduling of the central CPU and so reduce the overhead processing in the CPU.

To usefully process high bandwidth data streams frontend processors can be used to perform formatting, data reduction, buffering, encryption, and error detection and correction functions. Examples of such data streams include imagery data, video, telemetry data from satellites, and bulk encrypted telecommunications streams.

Frontend processing provides a natural and efficient way to use processing resources in parallel. Functions can be partitioned at design time among the available resources. The design can be modularized with clean interfaces between the processors. The relative simplicity of these designs has led some authors to classify these systems as not distributed.

With frontend processing, the survivability criterion of reconfigurability may be low unless redundant hardware elements are available. Testability is high because the functions of each processor can be tested with simulated input, without all the elements of the system available.

Although the terminology of "frontend processor" and "backend processor" implies a static assignment of function to particular processors, this is not necessarily the case. Two or more of the forms of communications can occur between any two processors in the system as different aspects of the application programs are activated.

4.3 DISTRIBUTED DATABASE

A distributed database is a system in which data is located in various sites of a network or distributed system. Usually, the database is partitioned into a number of components, some of which are replicated and occur at several nodes. Replication increases the reliability of the system, because data items are not lost if access is lost to one copy at one node. Replication also increases reliability, as fast access is possible to data items available at the node which needs them. Partitioning is done, when possible, so that most transactions involve data items within a single partition.

The problem is that of establishing, maintaining, and verifying consistency among the copies, since an updated data item must be updated at each of the nodes which host that partition. Table 4.3-1 indicates some of the other issues involved in distributed

Table 4.3-1 ISSUES IN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

SYSTEM PARAMETER	DESIGN ISSUES IMPACTED
DISTRIBUTEDNESS	Degree of Distribution versus Integration, Hardware Distribution, Program Distribution, Data Distribution, User Distribution
USABILITY	Data Acquisition, Storage, Security, Privacy, Manipulation, Risk Reduction, Service Flexibility, Service Evolution, User Input, User Output
HOMOGENEITY	Degree of Homogeneity of Hardware Components, Degree of Homogeneity of Requirements, Multiplicity of Vendors
DBMS DESIGN	Architecture, Top-down vs Bottom-up, Database Description, Database Manipulation, Distributed Control, Distributed Executive
TRANSMISSION	Transmission Media, Transmission Control, Resource Allocation, Distributed Control, Consistency, Reliability, Protocols, Rates
INTEGRITY	Locking Strategies, Synchronization, Granularity, Deadlock Elimination, Inconsistency Elimination, Multiple Copy Data, Crashes and Restarts, Active Transaction Backup, Restoration/Recovery, Audit Trail

database management systems. Paragraph 4.3.1 discusses the problem of multiple (replicated) data.

A distributed system can be built around a centralized data base. This usually occurs when a number of computers are connected to a local network via a high speed bus or in geographically distributed systems when system data rates are compatible with communication link capacities. The central data base may be controlled by one of the processors or by a special purpose database management machine.

When the data base is physically distributed there are several issues that must be considered. For example the database may be partitioned either horizontally (segregated data base) or vertically (hierarchic data base). It may also be distributed redundantly (i.e. multiple copies). A particular function's access to the data may be data coupled (function asks for data by name), access coupled (function asks for location of data and then directly retrieves the data), or fully coupled (function maintains directory of the data and directly retrieves the data on one access).

Table 4.3-2 summarized how the data base coupling choice may affect system and software quality factors.

Table 4.3-2 Impact of Database Coupling on System and Software Quality Factors

	Impact on Quality Factors		
Database Coupling	Access Efficiency	Vulnerability to error	Flexibility
Data	low	low	high
Access	medium	medium	medium
Fully	high	high	high

4.3.1 Multiple Data

The same data elements may occur in multiple copies, spread over multiple sites in the system. Mechanisms must be invoked to correctly update these multiple data elements, in order that consistency and concurrency be preserved. This process must continue to operate when one or more processors in the system are inoperative. This includes the need for crash recovery - both for handling the failure of single processors and for communication failures which partition the system into disconnected subsets. Local transactions which were committed before the crash must be eventually merged into the data base. Algorithms are known to solve these problems under certain sets of assumptions, but the general problem must be dealt with on any distributed data system (STO).

4.3.2 Database Strategies

Multiple access can be achieved by coordinating access through some control process such as a monitor, through schemes for allowing concurrent access to a single copy of the data (e.g. multiporting of physical memory), by providing multiple copies of the data, or through structural schemes (e.g. locks) which prevent conflicting access. When centralized control of task scheduling is used, database access can be controlled by coordinating task execution. Access strategies which allow for simultaneous multiple access allow tasks to be scheduled efficiently without database resource conflicts. Strategies which affect task execution (e.g. locking) avoid the consistency and concurrency problems of simultaneous multiple access. A major tradeoff in database design is between database integrity and consistency, and between database availability and efficiency. Some database integrity can be traded for increased access and efficiency for data which is tolerant to some degradation (e.g. video imagery). Physical database redundancy almost certainly requires multiple copies of segments of the database. Keeping these copies concurrent is a major problem in database design. Communications costs and delays can severely restrict the data access bandwidth, and communications errors can adversely affect the integrity of the system. See Figure 4.3-1 for a classification of these and other issues of distributed database systems.

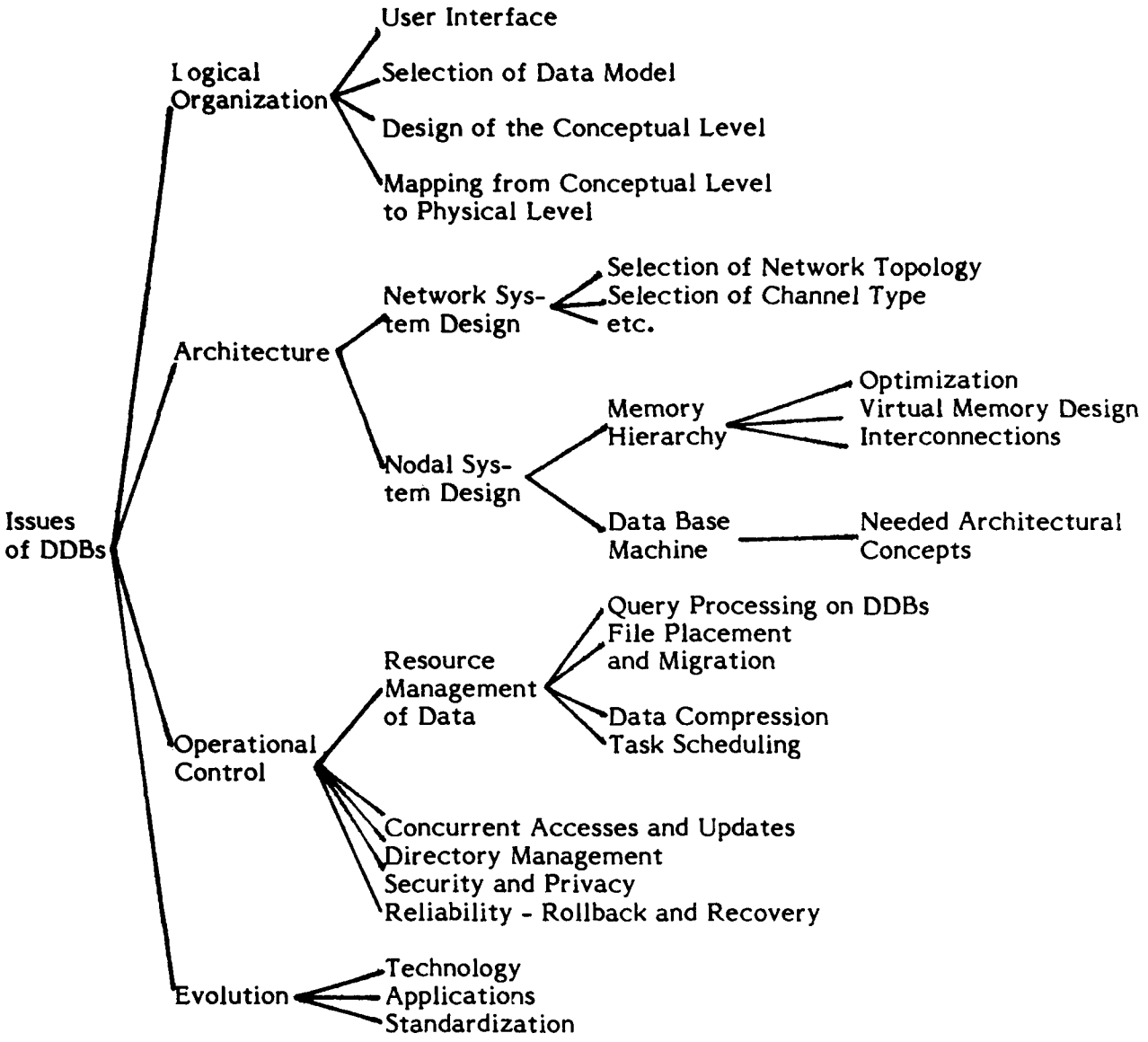


Figure 4.3-1 Classification of Issues in Distributed Database Systems

4.3.3 Topology Impact

The topology of a communications network is the pattern of physical interconnection. Physical interconnection or topology refers to the interconnection of physical communications facilities such as processors and links. The strategy of physical interconnection has a direct impact on system issues such as reliability, load and failure tolerance, communications cost and delay, and reconfigurability. See Table 4.3-3 for examples of system topology impact on system quality and software quality.

Network topology is closely related to logical organization, which is the pattern of interconnection established at the software level. The logical organization of a network can be more restricted than the physical organization. For example in a fully connected network where each pair of nodes can communicate directly, the logical organization might nevertheless require that all communication take place through a "central" controlling node. If the central node fails, the network can be reconfigured by designating another node as "central", because the underlying physical network is fully connected. The term "network topology" will be used to refer to those aspects of the interconnection pattern which are relatively fixed and not subject to automatic reconfiguration. Principal issues include the impact of node and link failure, which will disconnect the network, on probability of a failure and the adequacy of the communications facilities for projected communication traffic patterns. Specialized techniques are required to identify the interaction between the requirements of the system and the utilization of communications resources.

It has been claimed that sophisticated communications processes in a distributed system can overcome the problems of unreliable exchange of messages. Nonetheless, some protocol problems which involve trying to reach agreement in real time over an unreliable communications network do not possess a solution. This is discussed in (YEM). The domain of discussion involves the following assumptions: (1) The processes advance asynchronously with their individual computations; there is no global clocking mechanism. (2) Interaction between processes is limited to exchange of messages. (3) There is no global monitoring facility to guarantee an orderly processing (among the different processes) and centralized resource management. (4) The communication medium is unreliable; messages can be lost, duplicated, or delivered out of order because of random delays. The medium does have data integrity though,

Table 4.3-3 Topology Impact on Quality

System Topology	System Quality Impact	Software Quality Impact
Ring & String	Homologous Control Structure Lacks Flexibility	Ideal for sequential processing Increased Complexity for Hierarchic control structure
Hierarchic	Hierarchic Redundancy to improve reliability High communications overhead	Efficiency to overcome communication delays
Lattice & Fully Connected	Reliability difficult to assess Survivability	Testability is extremely difficult
Star	Centralized Data Base Single Point System Failure	Testability is difficult
Bank	Increased Thruput Increased Reliability	Synchronization and Order control
Bus	High connectivity Communication Contention	Testability is difficult
Master/Slave	Increased Thruput	Synchronization

messages do not contain errors. (5) The communication resources (bandwidth and buffering) are limited and need to be dynamically shared.

4.3.4 Communications Strategies

In a distributed system, communication among the processing elements is required to coordinate processing and to maintain and communicate shared data. Strategies may be needed to provide the required communications bandwidth, to minimize communication overhead and delay, to provide communications reliability, and to enable the interconnection network to be reconfigured in the event of node or link failure.

The basic mode of communication is a network, consisting of nodes (e.g.processors) connected by links (e.g. data busses). Direct strategies provide for communication between nodes without the services of intermediate nodes. With a direct strategy, the communication resources are locally controlled and the links or transmission media are generally passive. An example of a direct strategy is communication over a broadcast medium where each node has its own transceiver (e.g. radio). When communication between two nodes requires the active cooperation of other nodes the strategy is called indirect. A message between two nodes is routed through the network from node to node until it reaches its destination. Issues involved in the choice of routing strategy include communication delay, reconfigurability, network modularity and flexibility, congestion avoidance, and node failure tolerance. The quality factors impacted are Efficiency, Survivability, Flexibility, and Expandability. Table 4.3-4 compares communication strategies for six architectures or access layer types: master/slave, hub, bus arbiter, TDMA, CSMA/CD, and deterministic. Each architecture is rated according to distance, number of stations, speed, etc.

4.3.5 Local Net Architecture Protocols

To connect resources such as computers, terminals, and databases, all components must communicate with each other by means of a standardized behavioral convention known as a protocol. The International Standards Organization and the ANSI X3 Committee both propose a seven-layered structure of protocols for Open Systems Interconnection (OSI). These proposals (the latter is adopted by the National Bureau of Standards) are essentially the same.

Table 4.3-4 Comparison of the characteristics of various access layer types

Type	Distance	# of stations	Speed	Prioritization	Thruput	Deterministic
Master/slave multidrop (centralized)	With modems, varies from miles to thousands of miles	Usually less than 64	300 b/s to 10 Mb/s	Fair or prioritized distribution depending on master program	Little control of store-and-forward, no real-time store-and-forward	Yes (master/slave)
Hub (polling central switch) (centralized)	With modems, varies from miles to thousands of miles	Thousands	Varies	Fair or prioritized distributinn depending on hub program	Some control of store-and-forward, no real-time store-and-forward	Can be, depending on the functionality of the hub
Bus arbiter peer) centralized)	3 km is typical, can be greater	Usually less than 32, more than this becomes a burden	Varies to 1 Mb/s	Fair to prioritized distribution depending on arbiter program	Good control of throughput and real-time performance	Can be, depending on the functionality of the arbiter
Bus arbiter (TDMA)● (distributed)	20 miles is typical	From hundreds to thousands	Varies to 14 Mb/s	Fair distribution; it is difficult to dynamically adjust priority	Good control of throughput and real-time performance	Can be, depending on the functionality of the arbiter
Statistical (CSMA/CD) ● ● (distributed)	500-meter cable segments (Ethernet), to 1.5 km	Less than 100 per segment	10 Mb/s	Fair distribution, cannot prioritize	Good control of throughput and real-time performance	Statistical
Deterministic (token passing) (distributed)	5 miles	256 for wire, 10,000 for CATV	Varies to 1.544 Mb/s	Fair or prioritized depending on data	Good control of throughput, repeatable control of real-time performance	Yes

● Time-division multiple access
● ● Carrier-sensed multiple access with collision detection

Table 4.3-4 (cont'd)

Type	Access Levels	Survivability	Anomaly Management	Reliability	Verifiability	System Issues
Master/slave multidrop (centralized)	Slave only	Single-master failure is concern	Good, immediate control for last response	Good	Good	Requires store-and-forward for slaves to communicate, which creates risk of system failure with master failure, delay, and data corruption
Hub (polling central switch) (centralized)	Masters, slaves (peers)	Single-master failure is concern	Good, immediate control for fast response	Good	Good	Requires store-and-forward for slaves to communicate, which creates risk of system failure with master failure, delay, and data corruption
Bus arbiter (peer) (centralized)	Masters (peers)	Single-master failure as well as station failure are concerns	Good, immediate control for fast response	Good	More difficult but diagnosable	If arbiter fails, stations can no longer communicate; fail-safe mechanism should be incorporated in every station
Bus arbiter (TDMA)● (distributed)	Slot owners (masters, slaves, peers)	Single-master failure as well as station failure are concerns	Good, immediate control for fast response	Timing change could create collisions and cause difficulties	More difficult but diagnosable	If central administrator and timer fail, stations can no longer communicate; fail-safe mechanism should be incorporated in every station
Statistical (CSMA/CD)●● (distributed)	Masters (data generators)	Modem media connector to detect collisions could cause system failure if it fails	Fair, no immediate response can be assumed so errors take longer to isolate	Good	Poor-system generates errors that are hard to distinguish from real errors	Connection between distance, speed, and minimum message size is a problem; system is potentially unstable under continuous 100-percent load
Deterministic (token passing) (distributed)	Masters, slaves (respond), demanders, peers	All stations should have station-failure cutoff	Good, immediate resonse and control available	Proper handling of token passing is important	Good	Adequate initialization and station-failure procedures are important to robustness

● Time-division multiple access
●● Carrier-sensed multiple access with collision detection

In large computer networks, the variety of processor architectures makes it necessary to establish protocols. An example of the levels of control that are necessary to link two dissimilar computers with dissimilar operating systems is shown in Figure 4.3-3. The protocol overhead reduces efficiency. Also if an application is to be executed on several computers in the network, a high degree of portability is desirable.

Figure 4.3-2, shows the seven layers of protocol. Layers 1-4 are the transport service, specifying the means for moving messages from node to node in the network. Layers 5-7 are user layers, allowing users to interface to the network. The overriding concept is the layered division of functions, as defined below:

* Level 1: Physical, specifies electrical and mechanical parameters of the lines between nodes, in standards such as EIA's RS-422, 232C, 449, and CCITT's X.21.

* Level 2: Data Link, describes the passing of data packets over the communications line, their addressing and decoding, error detection and correction. Examples include ISO's HDLC and IBM's SDLC.

* Level 3: Network, describes message switching and routing, as in CCITT's X.25.

* Level 4: Transport, describes message transportation between end-users, such as computers, in a way which relieves the end-users from concern with the details of data transfer. IFIP proposed INWG 96.1 for this.

* Level 5: Session, describes a structured, logical exchange of messages between nodes with a session administration service and a session dialogue service. No standard currently exists, however, NBS has a draft standard.

* Level 6: Presentation, describes management of exchange, display, control, and transformation of messages between different computers, terminals, and database formats, codes, and languages. Some virtual terminal and virtual file protocols in this layer are proposed.

* Level 7: Application or Process, describes the distributed information service appropriate to an application, its management, and system management. These

COMPARISON OF COMMUNICATIONS ARCHITECTURE MODELS

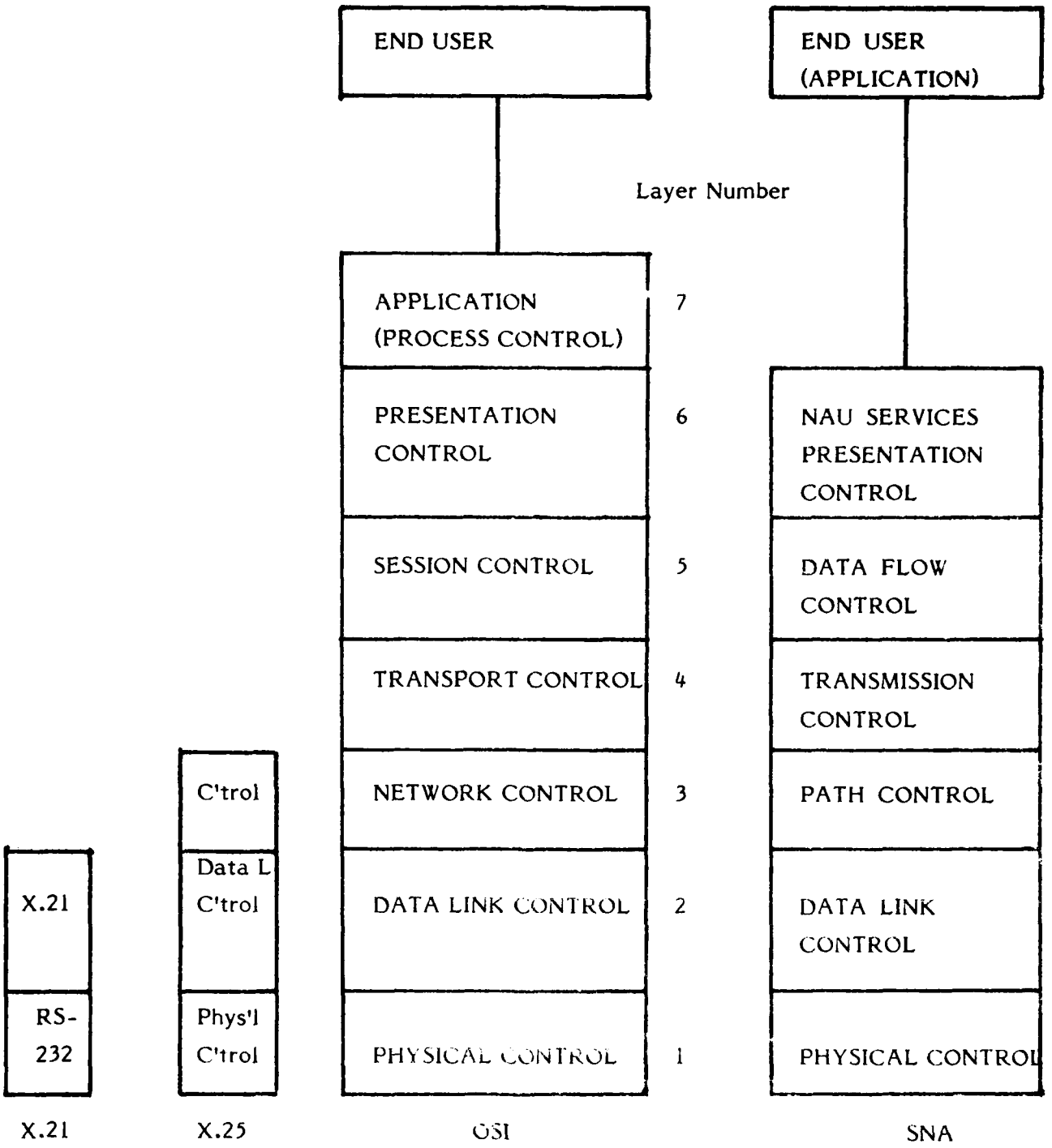


Figure 4.3-2 Distributed System Layers

functions initiate, maintain, terminate, and record data on data transfer connections between application processes. The other layers operate to make this layer possible, in a way which is not transparent to the end-user.

Of particular interest to the system quality approach are the 13 design principles by which the seven-layered architecture was defined. (ZIM). The impacted quality factors are as shown in parentheses.

1) Do not create so many layers as to make difficult the system engineering task describing and integrating these layers. (Usability)

2) Create a boundary at a point where the services description can be small and the number of interactions across the boundary is minimized. (Verifiability)

3) Create separate layers to handle functions which are manifestly different in the process performed or the technology involved. (Maintainability)

4) Collect similar functions into the same layer. (Maintainability)

5) Select boundaries at a point which past experience has demonstrated to be successful. (Reusability)

6) Create a layer of easily localized functions so that the layer could be totally redesigned and its protocols changed in a major way to take advantages of new advances in architectural, hardware, or software technology without changing the services and interfaces with the adjacent layers. (Expandability)

7) Create a boundary where it may be useful at some point in time to have the corresponding interface standardized. (Interoperability)

8) Create a layer when there is a need for a different level of abstraction in the handling of data, e.g., morphology, syntax, semantics. (Correctness)

9) Enable changes of functions or protocols within a layer without affecting the other layers. (Flexibility)

10) Create for each layer interfaces with its upper and lower layer only. (Efficiency)

11) Create further subgrouping and organization of functions to form sublayers within a layer in cases where distinct communication services need it. (Usability)

12) Create, where needed, two or more sublayers with a common, and therefore minimum, functionality to allow interface operation with adjacent layers. (Efficiency)

13) Allow bypassing of sublayers. (Efficiency)

SECTION 5
DISTRIBUTED SYSTEM TOPOLOGY

Distributed system topology is a term referring to the physical or logical pattern of interconnection of system components. Terminology for this is often taken from the mathematical subject of graph theory. Graph theory treats the properties of nodes (points) interconnected by links (lines) with an emphasis on the pattern and its sub-patterns, rather than the classical geometry of distances and angles.

Aspects of distributed system topology discussed below include:

* topology impact - how and why topology is related to system quality factors
* communications strategies - differing approaches to routing messages between nodes
* distributed system layers - the ISO Reference Model for interprocessor communication protocols
* distributed system architecture classification - what topologically different designs are possible, and their relative advantages

* distributed system hardware architecture (topology) impact on system quality

* distributed system hardware (topology) impact on software quality

The architecture of a distributed system is driven by the reasons for the selection of a distributed approach to meet the system requirements. There are two broad categories of reasons for the selection of a distributed design. One category consists of rationales based on the idea that system performance and quality goals can be more easily met with a distributed design. The other category of rationales are those related to required physical separation of the elements of the system.

Perhaps the most popular is the category of performance and quality rationales. Two frequently cited example rationales are: (1) increasing throughput by using several processing elements concurrently and (2) increasing the system qualities of reliability and survivability by distributing system functions over redundant elements. To date,

the results of applying distributed designs to meet performance and quality goals have been mixed.

Distributed architectures can be the only available solution when the components of the system must be physically separated. This can occur when the investment in existing facilities is large and new applications require these existing facilities to be integrated into a single system. Physical separation may also be required when the various parts of the system must be located in incompatible environments.

5.1 COMPONENTS OF DISTRIBUTED SYSTEMS

A computer based system consists of three kinds of components: processors, databases, and external interfaces (displays, sensors, etc.). These components must communicate in order to form an integrated system. In a distributed system, communications issues become so visible that the communications facilities become a fourth component of the system.

As the number of asynchronous, concurrently operating elements in a system increases, the communications requirements can explode as the timely delivery of control and data messages becomes crucial to the performance of the system. The communications requirements may lead the designer to consider the configuration of a communications network to be the primary focus of the design effort. When the design is driven by a requirement for the physical separation of components, most of the system requirements may be communications requirements. However, where the design is driven by performance and quality goals, effective communications facilities are required to support other more important system requirements.

5.2 SYSTEMS TOPOLOGY AND ARCHITECTURE

Physical, communications, and database architectures all interact to form the system topology. Because the speed of signals is limited by the speed of light, component separation of more than a few feet results in communications delays which are significant compared to processor or memory cycle times. Thus there is a strong interaction between the physical placement of system components and the communications architecture of a distributed system. Database architecture interacts

with physical and communications architecture through data access, data placement, and data communications requirements and capabilities. Processing architectures and system topology interact through interprocessor communications requirements and database access requirements. The distribution of processing activity among the elements of the system requires timely communications of control information. Access to source code, executable modules, and datasets residing on remote elements may be required by processing elements. Datastreams of intermediate results may flow from one processing element to another for further processing. A feasible system design must be based on a consideration of the interaction of processing architectures and system topology.

Processing architecture and system topology both drive the system control architecture. Control is needed both to schedule and synchronize multiple concurrent processing elements as well as to manage and allocate distributed resources. The control architecture can become a critical focus in the design of a distributed system. A system configured with adequate processing, communications, and database resources to meet system requirements still requires a control structure to effectively function. Care must be taken to accurately estimate the complexity, scope, and feasibility of the control system required to integrate system resources into a functional system.

5.3 PHYSICAL ARCHITECTURE

The physical architecture of a distributed system can be classified by intercomponent distance. When intercomponent distances are only a few feet, communications delays may be short enough so that a high degree of intercomponent synchronization can be achieved. Communications may be carried on with low level hardware protocols, using technology similar to that used in traditional mainframes. Communication can take place in units as small as a single memory word, or perhaps even a single bit. Local communication can be made highly reliable with current technology. Moreover, uncertainty concerning message delays can be reduced so that higher level software can regard communications as instantaneous. An extreme example of small interconnect distances is given by the case of two processors which share common memory. When the first processor writes a value in the memory, there is little doubt the second processor will see that value if it accesses the memory after the value is written but before any subsequent modification. This contrasts strongly with a longer distance

situation where the value sent by the first processor may not arrive or may arrive at some uncertain future time.

When components are separated by less than a few thousand feet, processing elements can communicate interactively. For example, intelligent peripherals can be accessed by a processor on an interactive basis where the data transactions are in 'pages': units on the order of a thousand words. Two processes may likewise interact where for example one process interactively controls a datastream generated by another. From a system perspective, the design and control of these systems can be similar to that of systems of several processes on a single multiprocessor. Communications delays can however be variable due to competition for communications resources. This uncertainty, among others, generally requires more complicated 'handshaking' to achieve the required synchronization of processing elements.

As intercomponent distances increase, minimum message delays increase to time periods during which processing elements can execute several tens of thousands of instructions or more. This reduces message exchange to a rate where interaction between processing components is infrequent compared to significant processing events. Where data on secondary storage could be accessed in pages on smaller scale systems, in large scale systems data on remote secondary storage is accessed in larger units such as files. Communicating processors must schedule activities so that relatively long periods between interaction do not adversely affect system performance. On very large scale systems, the style of interaction and execution is similar to a 'batch' system in many ways.

5.4 FUNCTIONAL SEPARATION

Distributed systems can be classified according to the "degree of physical separation of functions".

In small scale systems, the system can consist of heterogeneous clusters of homogeneous components. For example, a cluster of processors may have common access to a cluster of secondary storage devices. A design of this type requires a control strategy to manage the access to these devices. This is an example of resources which are traditionally managed within an integrated host system with a single central

processor. To effectively use these resources in a distributed system, new control strategies must be devised. Requirements for physical security of databases can mandate physical separation of processing from data storage. For example, the data may be stored underground or in limited access areas whereas the processing equipment may be located for more convenient user access.

As the distance between components increases, separation of components by function becomes less practical because of communications delay. In this case, the system tends to be configured as homogeneous clusters of heterogeneous components. Each cluster forms a standalone subsystem which communicates with other clusters. A good example of this architecture is a long haul network like ARPANET, where each node is a self-contained computing facility.

5.5 STANDARDIZATION AND MODULARITY

Physical architectures can be classified according to their degree of standardization and modularity. Distributed systems which are configured by replicating a standard subsystem can have a high degree of reconfigurability and hardware maintainability. Software maintainability and reliability is increased because experience gained with one subsystem can be transfered in part to other subsystems. The tradeoff is that standardization requires compromise and locally suboptimal design for the sake of global system performance and quality.

When a large system is modularized to achieve the benefits of replication, physical distribution is required only because the subsystems are physically separate. Inter-component distances in this case may tend to be small. However, replication can also be beneficial in physically dispersed systems. For example, in ARPANET the communications processors are all virtually identical.

From a lifecycle perspective, a distributed system consisting of a few standard component modules may be significantly cheaper than a custom integrated system with less initial cost. This is particularly true if the additional cost of the modular system is due to hardware expense. Additionally, if the system design can also be modularized, the modular system may have a lower development cost than a centralized system. However, if the additional design problems due to distribution are not

recognized and effectively solved, reliability and maintainability of the system may be sacrificed due to the arcane complexity of the interaction among the elements of the system. Failure to recognize the scope of the control problems associated with a distributed design could lead to costly development failure.

5.6 COMMUNICATIONS ARCHITECTURE

Much attention has been focused on communications architecture as the backbone of the architecture of a distributed system. Work has been done in areas such as communications link placement, routing algorithms, traffic analysis, and bandwidth utilization.

Interconnection architecture is sometimes intimately related to system control architecture. For example, processing elements may be arranged in a loop. Messages are passed, usually in one direction, around the loop through intermediate nodes from source to destination. Special properties of this configuration can be used for synchronization and control. For example, if a node in the system originates a control message which is intended for all the other nodes, when the message returns to the originator the originator knows each other node has seen the message. Furthermore, successful error free transmission can be easily verified by comparing the return message with the original. This control scheme is simple compared to other schemes which might require separate independent handshakes with each other node. See Table 5.6-1 for a taxonomy of communication types.

Table 5.6-1 Taxonomy of Communication Types in Distributed Data Processing

Form of communication	Content of communication	Examples
Batch (asynchronous)	Input messages	Remote Job Entry (input and output) Conversational Remote Job Entry
	Output messages	Transaction tape to be processed against data base
	Data requests	Changed-record tape to update data base copy
	Data records	Requests to send a copy of a data set
	Application defined	Partially preprocessed bulk input data from data entry application Summary statistics (daily, hourly) for remote operation forwarded to central system
Interactive (synchronous)	Input messages	Message routing (front-end processor to multiple destinations) Input message checking and assembly
	Output messages	Processing output variables from application programs using locally stored formats and fixed data
	Data requests	Application requests (GET/PUT) sent to remote node for data access
	Data records	Communications required to maintain synchronization of two identical data bases in two nodes
	Application defined	Multinode transactions, e.g., searches in multiple locations

5.7 LONG HAUL NETWORKS

In long haul networks where internode links are leased from common carriers, few node pairs have a dedicated link between them. Instead, message traffic is routed through other nodes on its way from source to destination. This leads to a requirement for some kind of routing algorithm. That is, when a node receives a message destined for a distant node, a nearby node (or nodes) must be chosen to receive the message for forwarding.

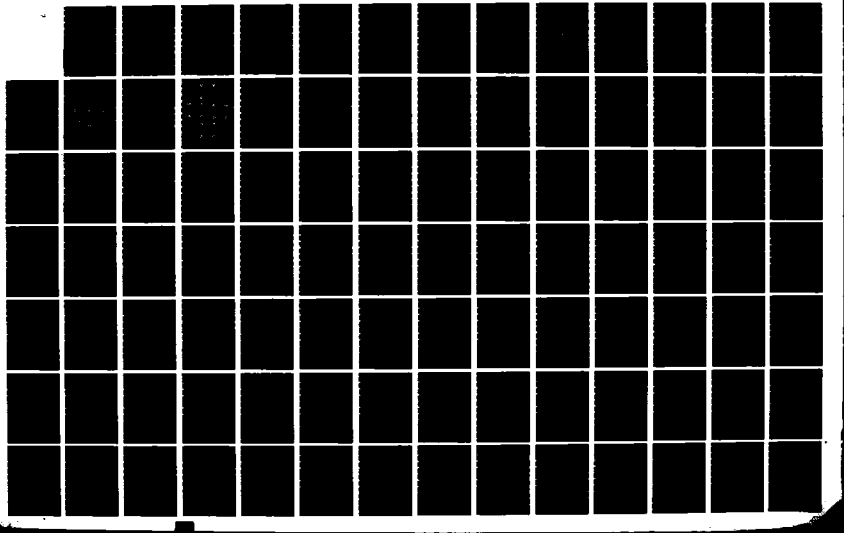
Since the number of node pairs in a network grows as n(n-1)/2, where n is the number of nodes, it is generally impractical to connect every pair with a dedicated link for complexity reasons alone. Additionally in long haul networks, where communications links are leased, the lease costs of the links can be a major part of the total operating costs. In this case, placement of links is generally determined by economic considerations using sophisticated techniques. The problem is complicated by the fact that the capacity of the links can be chosen and the costs of the links is related to capacity. Analysis of traffic patterns is necessary to determine points of congestion or over capacity within the network.

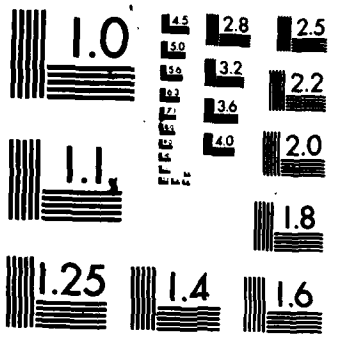
5.8 *INTERCONNECTION ISSUES*

Viewing the architecture as a network of nodes and links, a basic issue is how the individual nodes are linked together. The pattern of interconnection affects system parameters such as the probability of disconnection, maximum traffic capacity, tolerance to node failure, and expected message delay.

Probability of disconnection is closely related to the connectivity of the network. If just two nodes are connected by a single link, the probability of disconnection is just the probability of link failure. In a network of multiple nodes where intermediate nodes may forward messages, the calculation of the probability any two nodes will be disconnected requires more sophisticated techniques. When nodes play an active part in forwarding messages, node failures can also disconnect a network. In a well designed network, typically a single link or node failure will not cause communications to be lost between any two nodes of the network. In order to meet system reliability requirements, a careful analysis of interconnection patterns is required to reduce the probability the network will be disconnected by failures in either links or nodes.

AD-A137 957 SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS 2/3
VOLUME 3 DISTRIBUTED..(U) BOEING AEROSPACE CO SEATTLE
WA T P BOWEN ET AL. JUL 83 RADC-TR-83-175-VOL-3
UNCLASSIFIED F30602-80-C-0330 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The maximum traffic capacity between any two nodes in the network can be calculated from the interconnection pattern. The algorithm commonly used is due to Ford and Fulkerson (F&F). This algorithm can be used as well to find collections of links whose failure can disconnect the network.

For networks with complex or unknown traffic patterns, it would be desirable to have some definition and techniques for evaluating the capacity of the network. Various analytic modeling and simulation techniques have been used to determine the capacity of a network, behavior under high loads, congestion points, and characteristics of message delay. These techniques may be developed into metrics for network capacity.

5.9 ROUTING TECHNIQUES FOR PACKET SWITCHED NETWORKS

In networks where messages travel indirectly through nodes other than the source and destination, some technique must be provided for routing messages. There are basically three alternative strategies: (1) flooding, (2) random routing, and (3) table driven routing.

With flooding strategies, when a node receives a message which is destined for another node, the message is retransmitted to all the neighboring nodes. This strategy has the advantage that the message which reaches the destination first arrives in the minimum possible time. If the message contains a record of the path it took, information can be gathered concerning optimal routes. Additionally this method can be used to find a route in the absence of routing information.

Some limiting strategy must be employed to limit the life of a message in the network which uses flooding. One strategy is to time-stamp each message and have each node destroy any message which is stale when it arrives. Another strategy is to enable each node to recognize messages it has previously handled and limit the number of times the same message is handled.

The disadvantages of flooding routing strategies are related to the fact that extra traffic is generated on the network. The number of duplicate messages generated by a flooding strategy grows exponentially with the diameter (maximum number of links between any pairs of nodes) of the network. These 'extra' messages can cause congestion in the network, and many duplicate messages can load the protocol processing in the receiving

node. For small networks however, the extra traffic generated by flooding may not be a problem.

A more economical route finding method is random routing. Whenever a node receives a message for forwarding, the message is sent to a neighboring node at random. The path of the message resembles that of a dumb mouse in a maze. This method can be made smarter by incorporating some provision at each node to remember information about the network learned from previous messages. Random routing like flooding provides a convenient way to route messages in a poorly understood network. For example, if a network is reconfigured to recover from node and link failures, these methods can be used to reestablish the network.

With directory routing, each node maintains a routing table which indicates the preferred paths for messages according to the final destination. These routing tables may be fixed, or may dynamically change using methods which use information about the current state of the network. For example, when a node or link fails, the routing tables of all neighboring nodes may be changed in a predetermined way. More dynamic techniques may use randomly routed periodic test messages to gather information about the network.

5.10 EXAMPLES OF DISTRIBUTED SYSTEM TOPOLOGIES:

5.10.1 Loop Architecture

The nodes of a loop or ring network are connected to form a continuous ring or loop. Any two nodes can communicate by passing messages clockwise or counter clockwise around the loop. If any node fails, backup communications links (dotted lines in Figure 5.10-1) can be used to reconfigure the network. The advantages of this network topology are its simplicity and modularity -- the addition (or loss) of a node does not alter the basic communications strategy and can be transparent to other nodes. This topology is also compatible with control architectures where each node must be notified or must approve current transactions, since all the nodes can see any message which is passed all the way around the loop even if the message originator does not know which (or even how many) nodes are currently on the network.

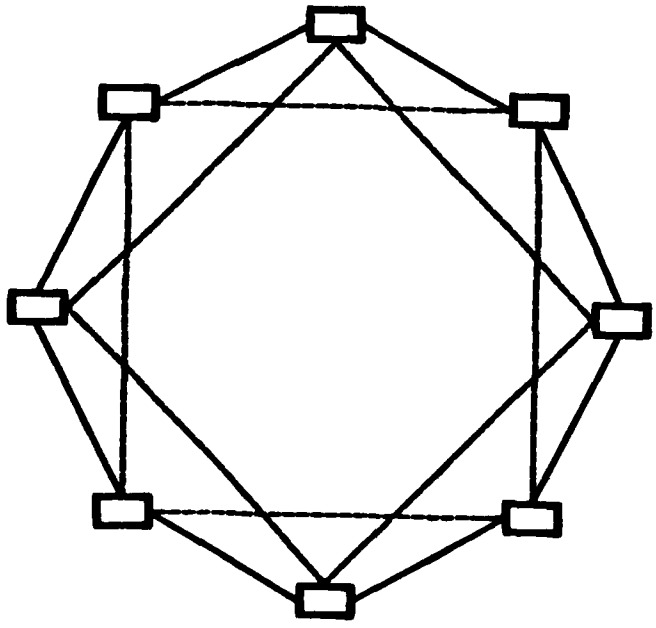


Figure 5.10-1 Loop Architecture (dotted line links extend to Chordal Loop Architecture)

The disadvantages of a loop architecture are mainly related to performance. If the number of nodes on the network is large, messages may be delayed because the path is likely to contain many nodes. Additionally, any node failure can affect communications to other nodes.

One solution to the problem of node failure is the "chordal loop" architecture. Each processor N is connected to processors N-1 and N+1 in the loop. In the chordal loop, the number of connections are doubled. Each processor N is now also connected to processors N-2 and N+2. In this architecture, the loss of a single processor does not disrupt the system connectivity. The loop has had a degree of communication redundancy added, which increases Reliability and Survivability.

5.10.2 String Architecture

A string architecture (Figure 5.10-2) is very similar to a loop architecture with respect to routing and configuration issues. To communicate with another node in the absence of prior information about the node, a message must be passed in both directions. A loop with a single node or link failure can be 'reconfigured' to a string with no physical modifications by simply removing a node or link. The string is also known as the linear array.

Figure 5.10-2 String Architecture

5.10.3 Star Architecture

A star configuration (Figure 5.10-3) consists of a central node to which all the other nodes are connected. The routing strategy is simple -- each message is sent to the central node which forwards it directly to the destination. This configuration is highly modular. A node can be added by adding one additional link to the central node and the routing strategy remains fixed. The failure of any node besides the central node has little impact on other nodes. The obvious disadvantage of a star configuration is that failure of the central node causes all communications to be interrupted. Additionally, in reconfiguring such a network using another node as central, at most one of the links of the original configuration can be used. The star is also known as the hub architecture.

The centralized star switch tends to have a substantial fixed cost component that discourages its use in small configurations that might later grow. If a single switch is used for a local network, no routing software is required, but the network is critically dependent on the reliability of the switch. The configuration of a central switch system is that of a star, with the switch at the central point and all the terminal access lines radiating from it. A hierarchical approach may also be taken where terminals are first fed into multiplexers and/or concentrators which then feed into the central switch. Multidrop lines may also originate at the central switch or at one of the concentrators.

The transmission system of a star is point-to-point, allowing the use of simpler analog technology than the Ethernet in which every receiver must be able to reliably hear transmissions from every transmitter.

5.10.4 Hierarchical Architecture

In a hierarchical network (Figure 5.10-4), the nodes are arranged in levels. Each node communications with one of the higher level nodes and possibly several lower level nodes. This network organization is particularly suited for a distributed system where the control strategy is also hierarchical. The loss of any node can disrupt communications to all the lower level nodes 'attached' to it. The hierarchical network is also known as the tree.

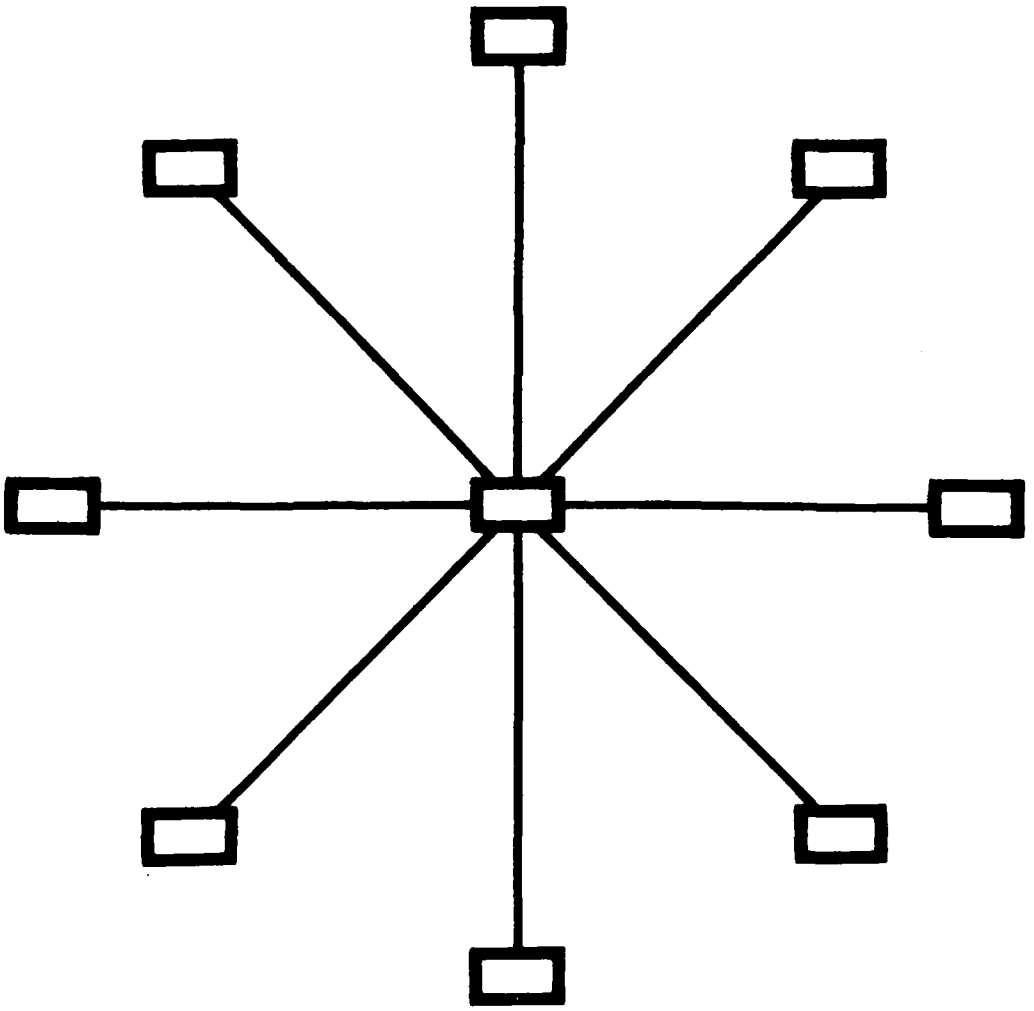


Figure 5.10-3 Star Architecture

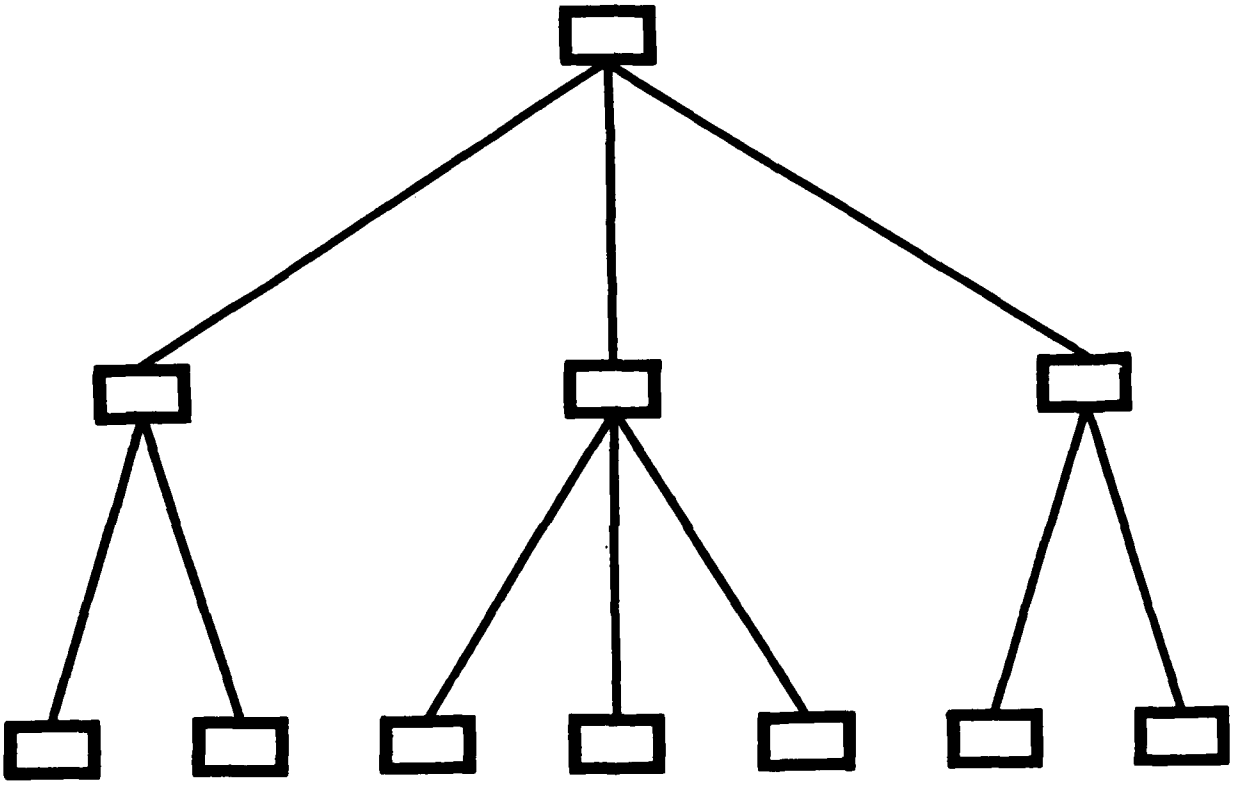


Figure 5.10-4 Hierarchical Architecture

A popular approach to constructing simple local networks for modest numbers of devices is to connect them in a hierarchical fashion. In this approach, there may be intermediate nodes between the two parties to a connection, these intermediate nodes may operate in a store-and-forward fashion, but ordinarily there will be no switching performed by the intermediate nodes, since there will be but a single path between the two communicating parties.

This approach is most useful when there are a number of devices needing accesses to resources which can either be provided by a single central source or sometimes by the intermediate node itself. For example, mini and microcomputers in process control and laboratory applications are often connected in this way, where the intermediate nodes are medium scale machines that can provide some support to the attached devices but that also can forward service requests to a large central machine when required.

One of the hierarchical systems already being funded by ARPA is the Non-Von (for non-von Neumann), whose principal designer is Dr. David Elliot Shaw, a professor at Columbia University in New York. Non-Von is to be composed of as many as a million microprocessors arranged in a binary tree (each element in the tree is connected to two elements below it) and will handle traditional commercial data processing tasks as well as the numerical processing usually reserved for supercomputers, according to Shaw.

5.10.5 Bus Architectures

With a bus architecture (Figure 5.10-5), all the nodes are connected to a single communications medium. To send a message, a node 'gains control' of the medium and sends a message. There are many bus control schemes. The underlying medium may take many forms from twisted pairs to radio frequency broadcast. The use of a fiber optic bus promises very high reliability and survivability.

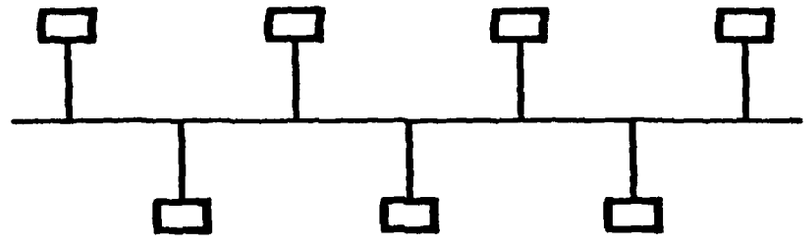


Figure 5.10-5: Bus Architecture

One family of distributed control mechanisms is based on contention detection and resolution. Good examples are CSMA (carrier sensed multiple access) bus protocols such as those used by Xerox's ETHERNET and NSC's HYPERchannel. Nodes on these networks operate asynchronously unless a collision (contention) is detected. Because of short distances used in these local networks (a few thousand feet), collisions can be used to synchronize subsequent recovery activity.

Centralized control mechanisms are also used. A 'bus master' or controller may be used as a centralized clearing house for requests to use the bus. Multiplexing techniques such as time division multiplexing can also be used. Some satellite communications systems can be viewed as using a multiple shared bus architecture with some form of time division multiplexing.

5.10.6 Generalized Interconnection Architecture

There may be no specific pattern used for the interconnection of nodes in the network (Figure 5.10-6). For example, in geographically large networks or general communications networks, nodes may be interconnected with links where physically feasible or convenient. Different communications technologies may be used for different links. This can be especially appropriate for a distributed system whose mission and composition may change significantly over time, since more specific architectures can limit future development.

The most basic approach that can be considered for a local area computer network is simply to interconnect all the devices that need to communicate by means of dedicated point-to-point links. Each link can utilize a different transmission medium and different interfaces, depending on the nature of the devices communicating over that link. Such an approach may be quite satisfactory for a local network with a limited need to communicate with different destinations. With many devices all needing to communicate alternatively with each other, such an approach can rapidly get out of hand due to the large number of links required and means for selecting them at each node.

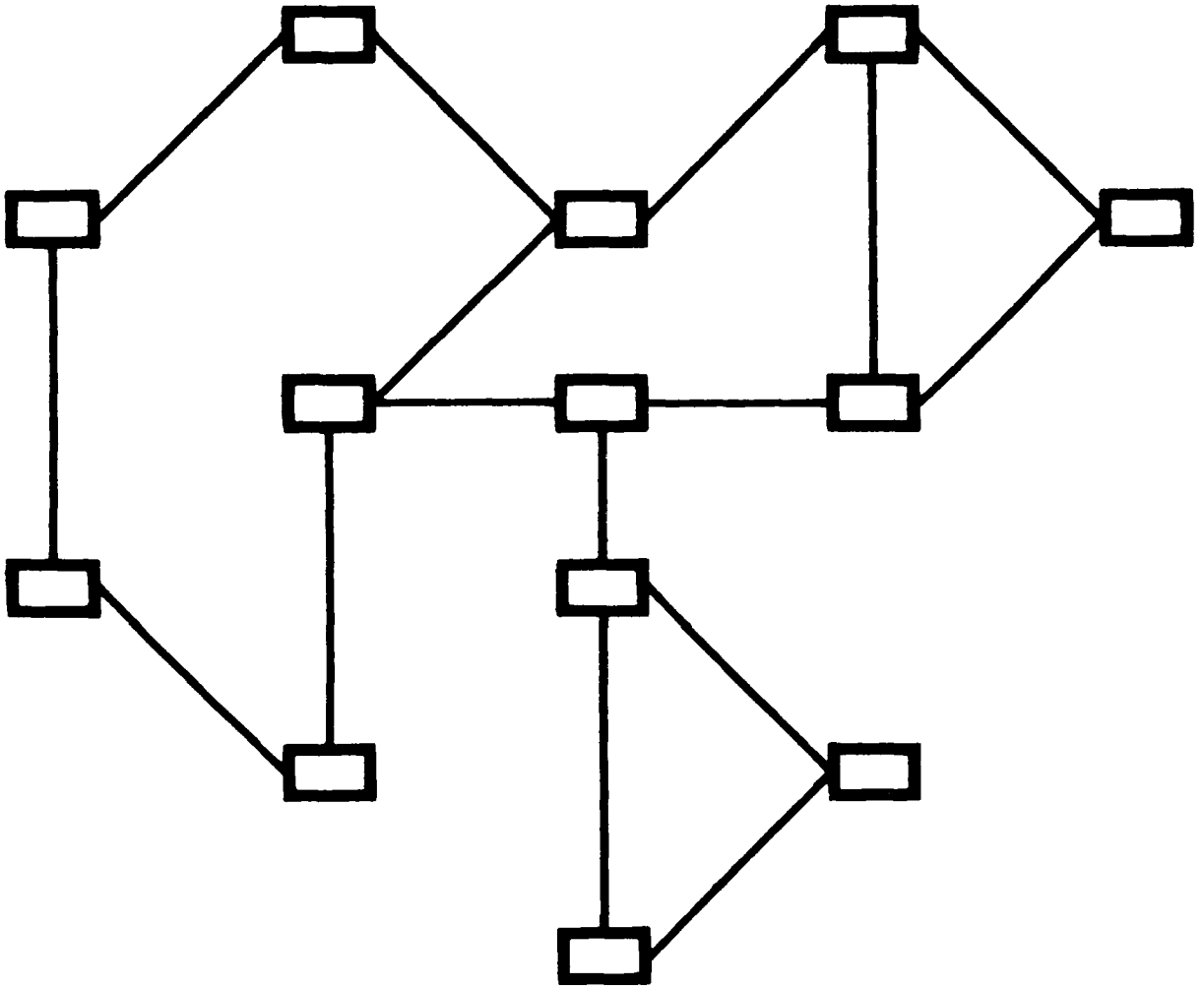


Figure 5.10-6 Generalized Interconnection Architecture

5.10.7 Complete Interconnection Architecture

In a complete interconnection architecture (Figure 5.10-7), there is a direct link between each two nodes of the network. Traditionally, this has been regarded as impractical in general, because the number of links grows as n(n-1)/2 where n is the number of nodes. However using broadcast media, any two nodes can communicate directly. As a result, complete interconnection on a functional level is available using shared bus architectures. All the earth stations communicating with a single telecommunications satellite can be considered completely interconnected. Using a shared media communications approach to achieve complete interconnection, a transparent level of intelligence in the underlying communications facility may be required. This intelligence may be either distributed or centralized.

A local non-switched network may be fully connected for those stations that need to communicate with each other. In this case, each station has as many channels connected to it as there are other stations with which it wishes to communicate; stations not so connected with a dedicated channel are unable to communicate. At the extreme, if there are N devices, each of which needs to communicate with all others, then the number of links required is N(N-1)/2.

Each link may be selected in consideration of the intercommunication requirements of the two stations it serves. Each of these links may operate over a different medium, at a different data rate, and with a different control protocol than all the other links. Multiple links available for use by a single device may be manually selectable (as by a switch box) or may be connected to different interfaces into the device, so that the various links may be alternately (or simultaneously) selected by the device itself. Of course, the device might also have to switch the operating protocol when it switches lines.

Each link is exclusively dedicated to the traffic of the two users it serves to interconnect. Since the link is permanently established (e.g., by a hardwired circuit or leased line), its total capacity is wasted when the stations are not actively communicating. Even when the stations are actively communicating, their actual data rates, and even their maximum data rates, may be substantially less than the capacity of the link. Such unused capacity is lost.

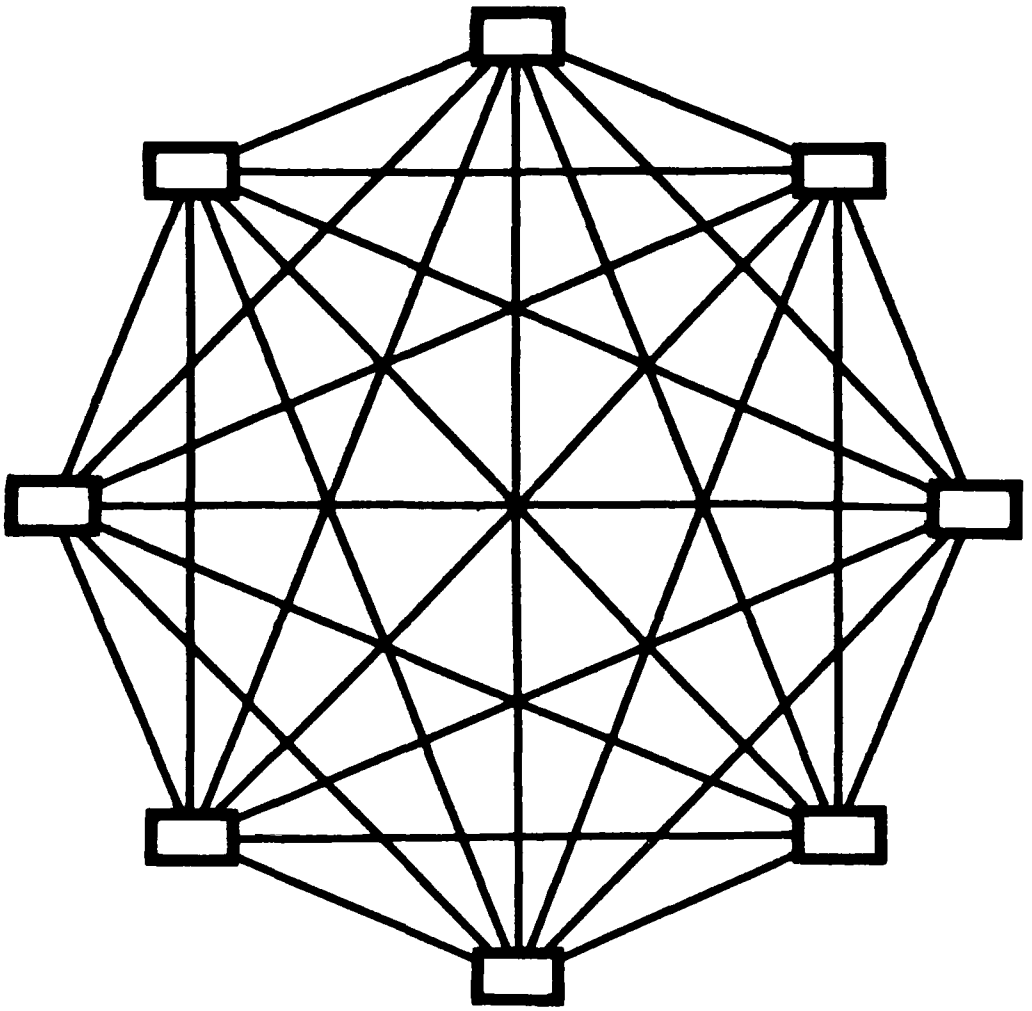


Figure 5.10-7 Complete Interconnection Architecture

The nature of the interfaces employed on local point-to-point links vary according to the speed, distance apart and nature of the communicating stations. If two compatible computers are connected over short distances, high speed parallel channel interfaces may be suitable. Over longer distances, serial communications-type interfaces are required. If the distance and/or data rate is not too great, modems may not be required; for longer distances and higher data rates, matching modems are required at each station's interface.

No speed or code conversion is provided on point-to-point links. The devices communicating over such links must do so at a common speed and with a common code. No flow control can be exercised by the network since each link serves to tightly couple the two stations it serves.

5.10.8 Binary Hypercube Architecture

A binary hypercube (Figure 5.10-8) is a generalization of the ordinary 3-dimensional cube with 8 corners (nodes) and 12 edges (links). This cube can be represented by labeling each node with a triple of zeros and ones (*,*,*) and connecting two nodes with a link if the triples differ in one position. For example, the node (0,1,0) is connected to (0,1,1), (1,1,0) and (0,0,0) but to no others. This pattern of interconnection can be generalized to 16 nodes by using quadruples (*,*,*,*). In general, if n is a power of 2 (2,4,8,16,32 ...), there is a binary k-cube which interconnects n nodes with $k = \log_2 n$.

There are several advantages for this form of interconnection. The number of links required to interconnect n nodes is small -- equal to 1/2*n*log(n). For example, only 80 links are necessary to connect 32 nodes. Communications may remain available even when many links or nodes are destroyed. For example, in a configuration of 16 nodes and 32 links, there is a set of 18 links which can be lost without disconnecting the network. Due to the regular structure of the binary hypercube, there are relatively simple routing strategies for this network.

The most frequently mentioned disadvantage of this configuration is the regular structure which limits the choices available when designing the network. For example, a complete binary hypercube must have a number of nodes equal to a power of 2. However, because

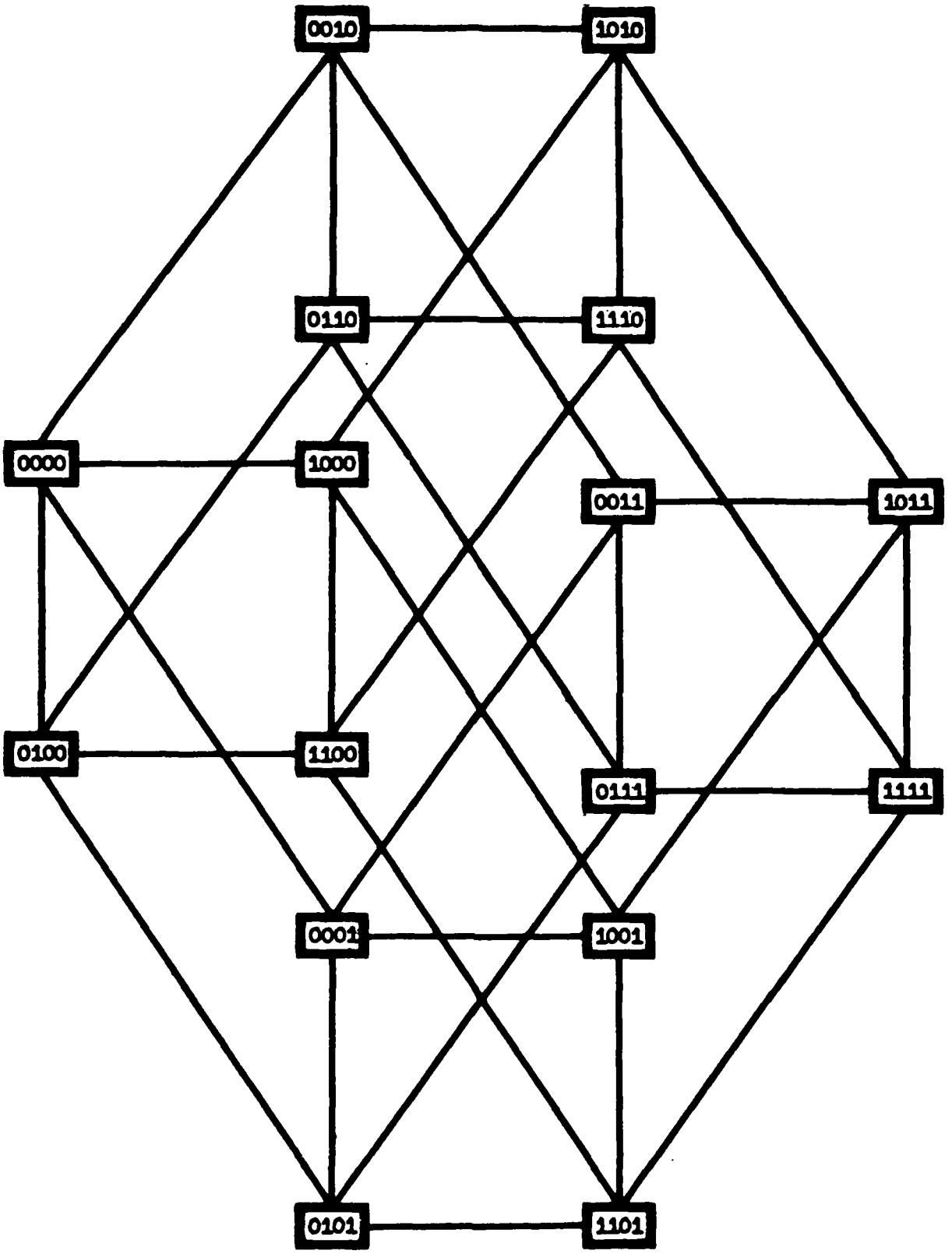


Figure 5.10-8 Binary Hypercube Architecture

the hypercube can function with missing links and nodes, less regular networks can be configured as partial hypercubes.

The generalization of the hypercube to other dimensions is the k-cube. The binary k-cube connects $n=2^k$ processors. If the processors are viewed as the corners of a cube in k-dimension space, the connections are the edges of the cube. More formally, if the processors are numbered from 0 to $2^k - 1$, processors whose binary numbering differs in exactly one position are connected.

For the binary hypercube, k=4, $2^4=16$ nodes. The k-cube contains a rich collection of data paths and is suitable for applications such as sorting. These applications require communication among distant nodes and are expensive to implement in nearest-neighbor-connected planar networks. A planar network is a 2-dimensional topology which can be diagrammed as a graph with no intersecting links. Tradeoff: A less-flexible planar network can be cheaply built on a one-layer circuit board.

A possible difficulty with the k-cube is that it requires k=log n connections per processor. This means that at some point ($n>n_o$) expansion of the k-cube becomes impossible because of wiring problems. However, $n_o=1000$ is feasible (BNL). Very recent work by Seitz and Hewett (STZ) argues that $n_o=64{,}000$ is practical, although there might be a problem with the length of the wires involved.

5.10.9 Cube-Connected Cycles

The next two schemes, cube-connected cycles and the perfect shuffle, can be viewed as implementations of or emulators for the k-cube. They have the advantage, however, of a constant number of wires per processor. To a large extent, these networks draw their power from their ability to efficiently implement divide and conquer algorithmic strategies. If a problem can be recursively broken into smaller ones, the smaller problems solved, and the solutions economically combined into a solution for the large problem, this divide-and-conquer algorithm substantially reduces the computational cost of a solution.

Consider the interconnection scheme depicted in Figure 5.10-9, which consists of 2^k processors arranged in 2^{k-r} rings of size 2^r at the corners of a (k-r) cube. In the figure, k=5 and r=2. As r becomes larger or smaller, the number of connections per processor

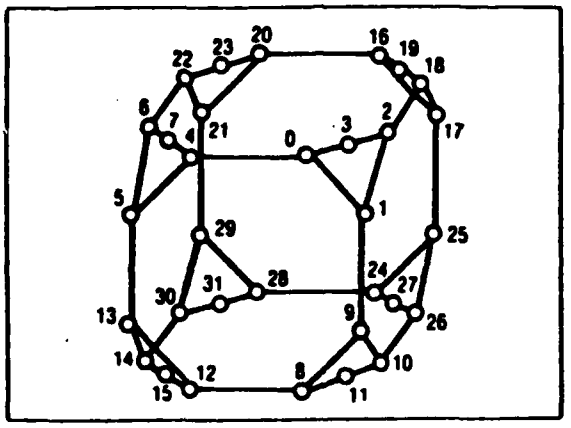


Figure 5.10-9 Cube-Connected Cycles

decreases or increases, and the scheme becomes more ring-like or cube-like. More specifically, performance analysis of divide-and-conquer yields $0(k-r) + 0(2^r)$ running time. The m-cube performs this computation in 0(m) steps; thus, the 0(k-r) term is the cube term and the $0(2^r)$ is the ring term. The trick is to choose r so that cube performance is retained, but interconnection complexity is reduced. If we choose k and r so that $(r-1) + 2^{r-1}$ equal or less than k equal or less than $r+2^r$, the first inequality retains cube performance 0(k); the second assures that enough processors are on each ring, so that one connection per ring processor suffices to connect the ring to the cube. Thus, the number of connections per processor is not more than three. With this choice of k and r, the scheme is called cube-connected cycles, in the sense used by (PRE).

5.10.10 Lattice Networks

From Figure 5.10-10 it is clear that lattices can differ in several ways. The PE degree (Processing Element), like the switch degree, is the number of incident data paths. PE = fan-in + fan-out. Most algorithms of interest use PEs of degree eight or less. Larger degrees are probably not necessary since they can be achieved either by multiplexing data paths or by logically coupling processing elements (e.g., two degree-four PEs could be coupled to form a degree-six PE where one serves only as a buffer). The latter method leads to some loss in PE utilization, however.

We can call the number of switches that separate two adjacent PEs the corridor width, w. This is perhaps the most significant parameter of a lattice, since it influences the efficiency of PE utilization, the convenience of interconnection pattern embeddings, and the overhead required for the polymorphism.

To see the impact of corridor width, let us embrace graph embedding parlance and say that a switch lattice "hosts" a PE interconnection pattern. In theory, even the simplest lattice (like the one in Figure 5.10-10A) can host an arbitrary interconnection pattern, but to do so may require the PEs to be underutilized. There are two reasons for this: First, PEs may be coupled to achieve high PE degree, as mentioned at the beginning of this section. Second, and more important, adjacent PEs in the (logical) guest interconnection pattern may have to be assigned to widely spaced PEs in the hosting lattice (i.e., separated by unused PEs) in order to provide enough data paths for the edges. Increasing

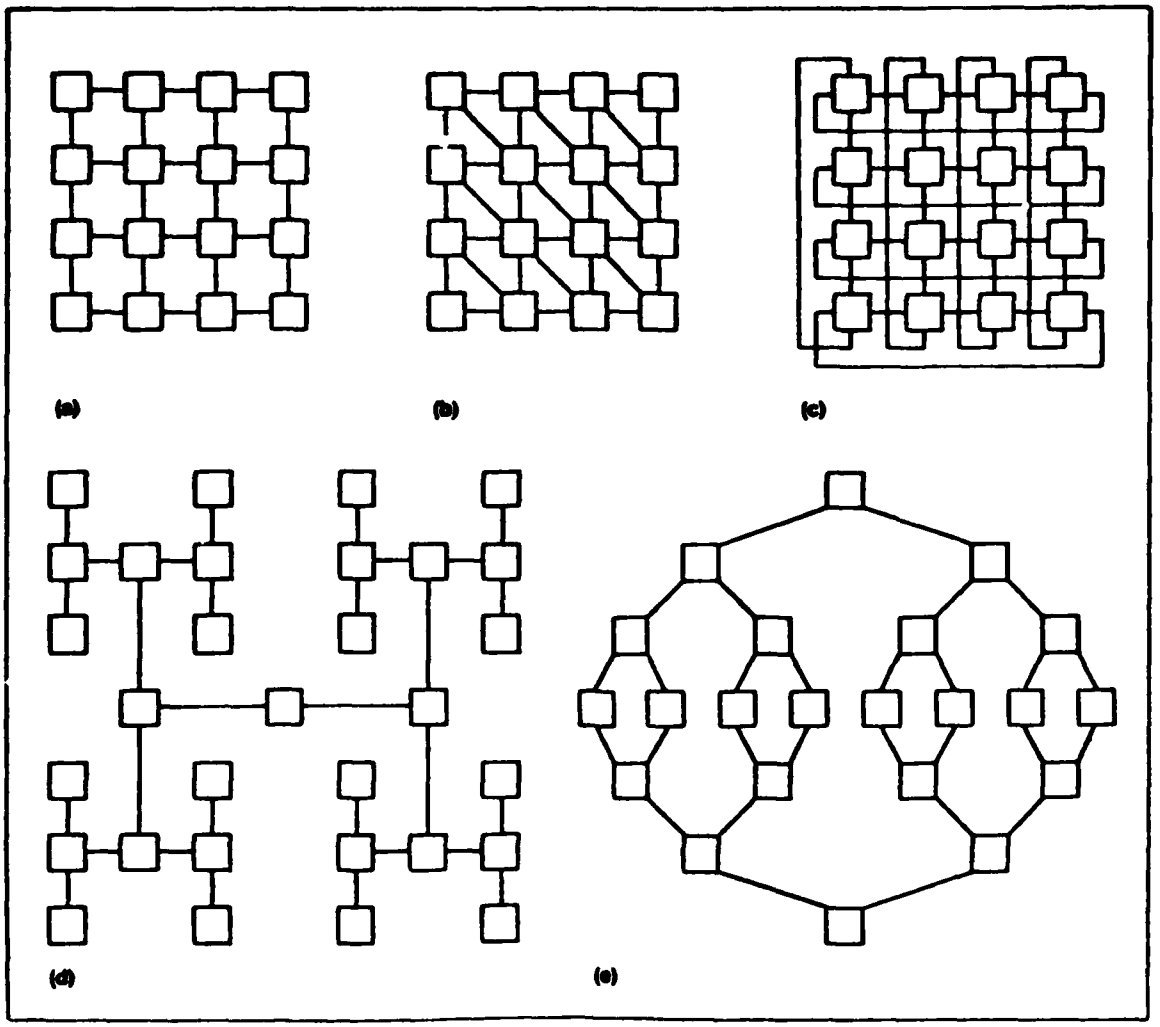


Figure 5.10-10 Lattice Networks

(a) Mesh, used for dynamic programming (GUI)
(b) Hexagonally connected mesh used for LU decomposition (KUN)
(c) Torus used for transitive closure (GUI)
(d) Binary tree used for sorting (BRO)
(e) Double tree used for searching (BEN)

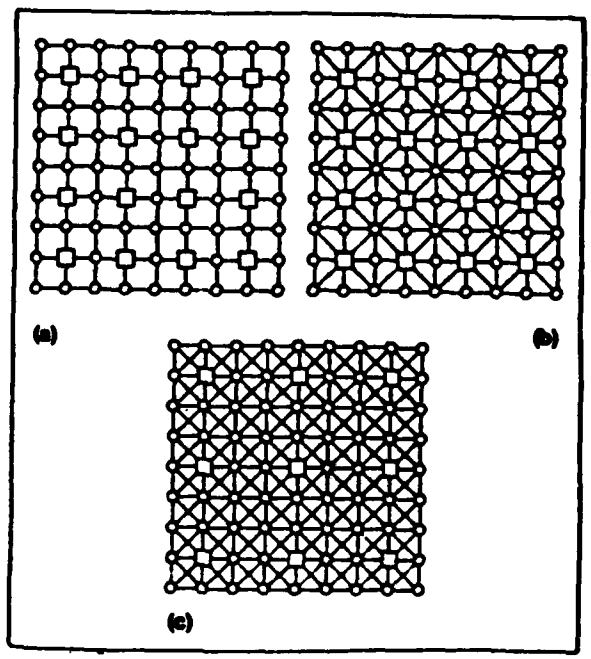


Figure 5.10-10A Three Switch Lattice Structure

- Circles represent switches
- Squares represent PEs

corridor width improves processor utilization when complex interconnection patterns must be embedded because it provides more data paths per unit area.

How wide should corridors be? It depends on which interconnection patterns are likely to be hosted and how economically necessary it is to maximize PE utilization.

For most of the algorithmically specialized processors developed for VLSI implementation, a corridor width of two suffices to achieve optimal or near optimal PE utilization. However, to be sure of hosting all planar interconnection patterns of n nodes with reasonably complete processor utilization, a width proportional to log n suffices and may in fact be necessary. (VAL) To host patterns such as the shuffle-exchange graph (Figure 5.10-11) efficiently will require even wider corridors; on the average w must be at least proportional to n/log n. (THO)

Selecting a corridor width is difficult, especially if it is a nonconstant width. The benefit, in some cases, is higher PE utilization; the cost is a loss of some locality (in all cases), more area overhead, and increased problems with pin limitations. Preliminary evidence indicates that w equal or less than 4 provides a reasonable cost/benefit trade-off, but further experimentation and analysis are required. (SNY 81), (SNY 82)

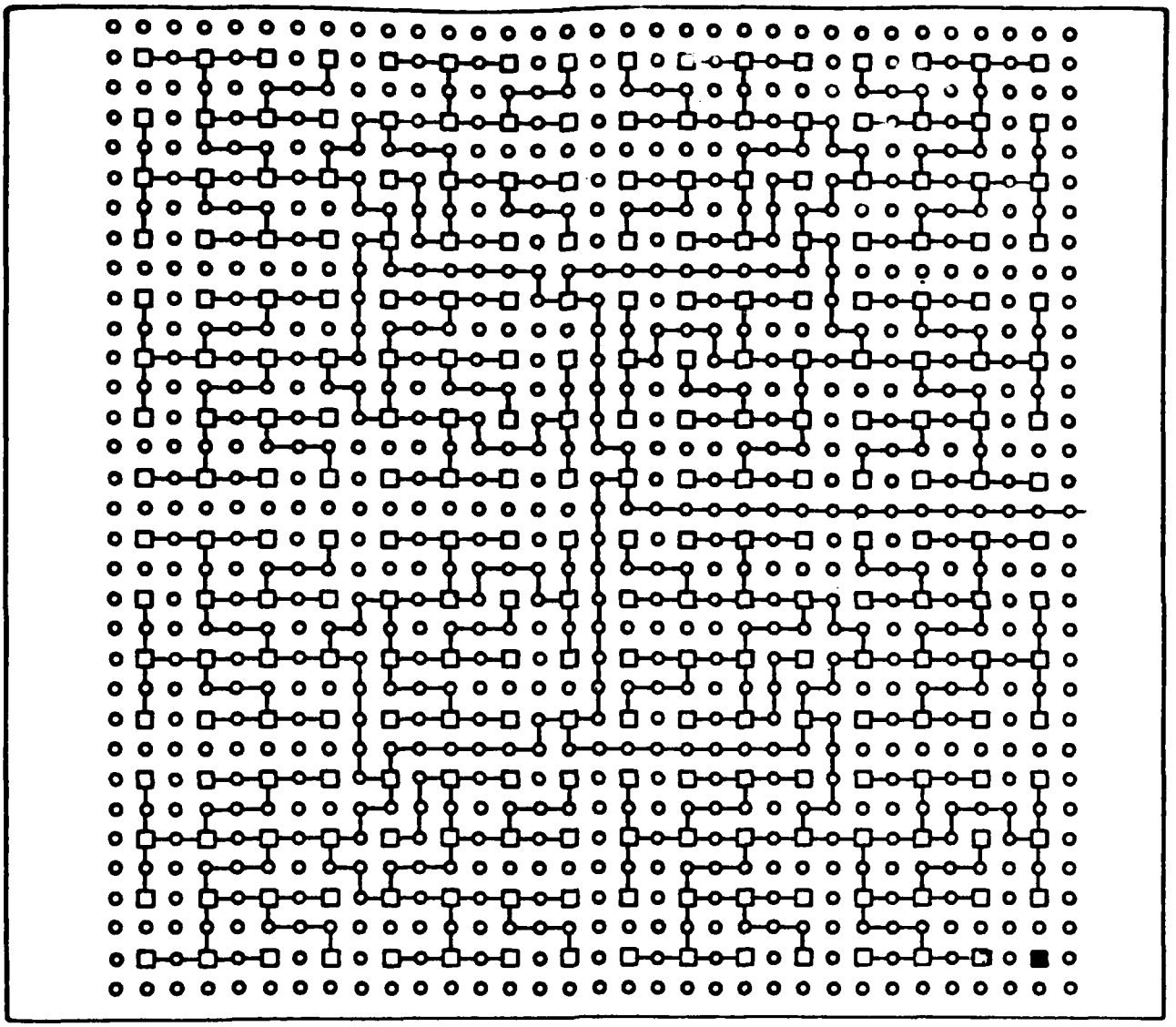


Figure 5.10-10B Planar Embedding of a 255-node Complete Binary Tree into the Lattice of Figure 5.10-10A

5.10.11 Shuffle Exchange Network

The shuffle-exchange network, first suggested by Stone, (STN) connects $n=2^k$ processors, as shown for k=3 in Figure 5.10-11. It takes its name from the fact that cutting the "deck" of processors between numbers 3 and 4 results in perfect interleaving of the two halves of the data deck. The shuffle-exchange network has been around for a long time and performs well in sorting, fast Fourier transform, matrix transposition, and linear recurrence evaluation. Siegel (SIE) shows that a composition of k shuffle-exchange networks (called an omega network) is functionally equivalent to the k-cube. Alternatively, one can simulate k-cube communications by cycling through a single shuffle-exchange network k times.

5.10.12 Banyan Network

A Banyan network (GOK) can be roughly described as a partially ordered graph divided into distinct levels. Each node has a set number of edges fanning into it (called spread) and a set number of edges fanning out from it (called fanout) toward the nodes in the next level below. Figure 5.10-12 shows a simple Banyan network with two lines coming into each switch (spread = 2), two lines emanating from each switch (fanout = 2), and three levels. The levels could be used, for example, to interconnect eight processors and eight memories or I/O devices. The attraction of the Banyan is that it provides complete interconnection of n devices at a cost in switching circuitry that grows as n log n. A crossbar, by contrast, grows as n^2. The binary k-cube is a graph homomorphic image of a Banyan (LIP)—as are perfect shuffle and tree networks. Here, homomorphic means that the image k-cube is obtained by identifying (combining) communications lines in the Banyan. Thus, the homomorphic images have lower hardware cost, but greater possibility of contention and reduced efficiency.

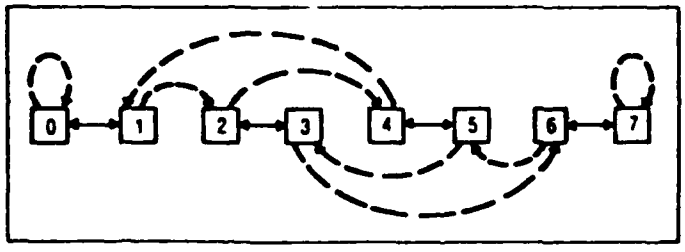


Figure 5.10-11 Shuffle-Exchange Network. (Solid line is exchange, dashed line is shuffle.)

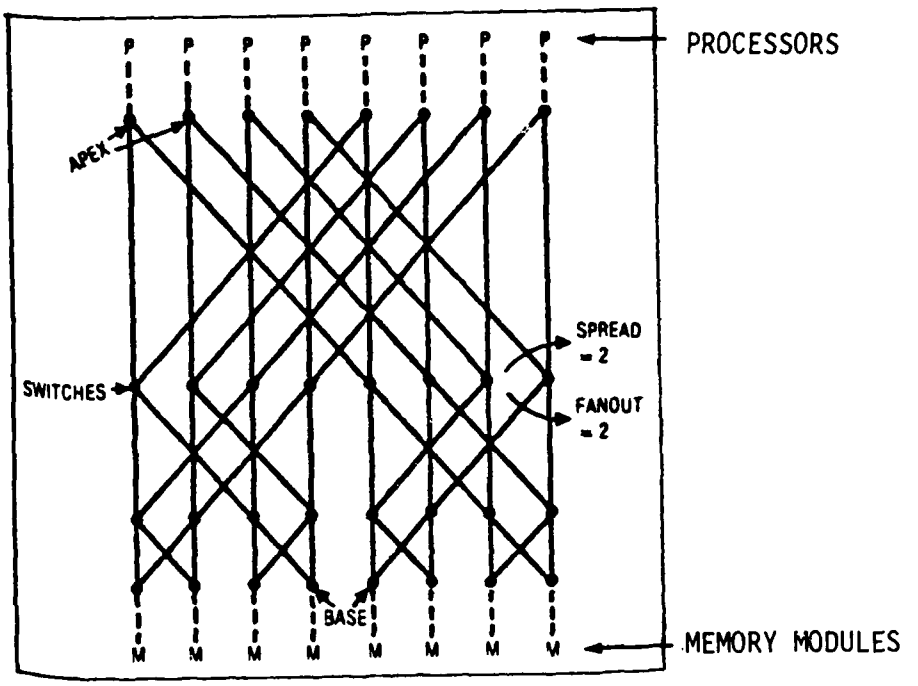


Figure 5.10-12 Banyan Interconnection

5.10.13 Cross-point Switch

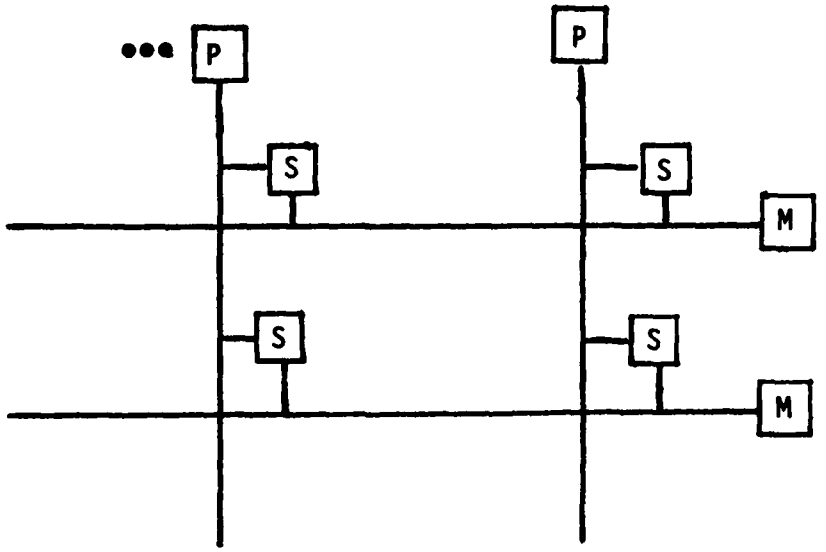
The most general and flexible interconnection scheme is the fully connected structure, or cross-point switch, with every processor logically connected to every other. The scheme is implemented by using n^2 switches to connect n processors and n memories, as in Figure 5.10-13. There is never contention for communications resources, although there might be contention for memory. No calculation, other than address translation, is required to establish a route. For sufficiently large arrays, this structure becomes infeasible because of the number of switches required. (For those wishing to implement a cross-point architecture, the recently announced Intel 432 system will include cross-point elements available on an intelligent chip called the bus interface unit, or BIU.) In the structure illustrated in Figure 5.10-13, one can expand the connections arbitrarily, populating as many cross points as desired, to limits imposed by cost and hardware reliability considerations.

5.10.14 Ring Architecture

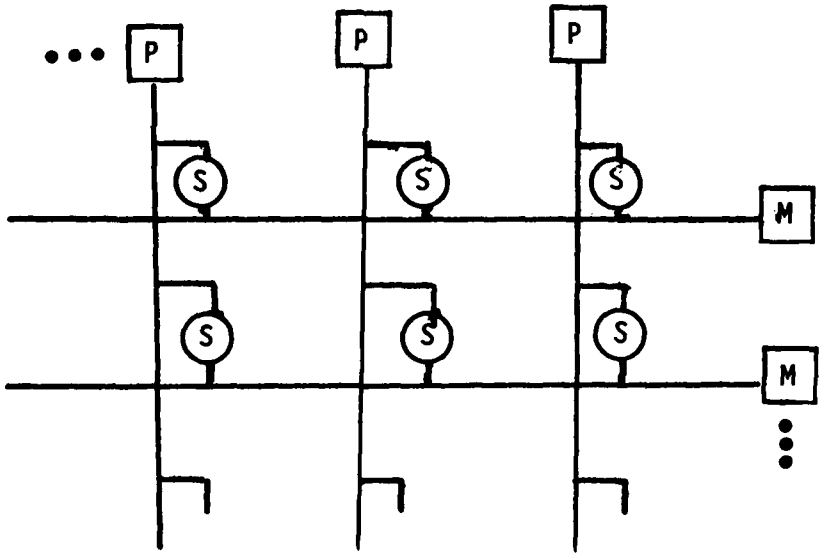
In the ring network, n processors are connected on a circular bus. In terms of both hardware simplicity and logical simplicity of the resulting architecture, the ring network represents an extreme point. Only 1/n of the bus bandwidth is available for each of n processors. Therefore, this architecture is suitable only where communications requirements are very small or where the processors can be kept busy on tasks that do not require cooperation while they are waiting to receive or transmit data.

Proposers of ring networks usually draw pictures as in Figure 5.10-14, with the implication that the cables interconnecting individual repeaters follow any convenient, reasonably direct route from one repeater to the next. Installing a ring network with that approach could be expected to expose the following problems:

1) Cable Vulnerability. The physical ring trails widely through the building, and is vulnerable at every point to accidents. If a link is accidentally severed or shorted, the entire ring is inoperable until the problem can be isolated and a new cable installed.



(a) 2x2 Cross-Point Switch



(b) 3x2 Cross-Point Switch

Figure 5.10-13 Cross-Point Switch (also known as Cross-bar switch)

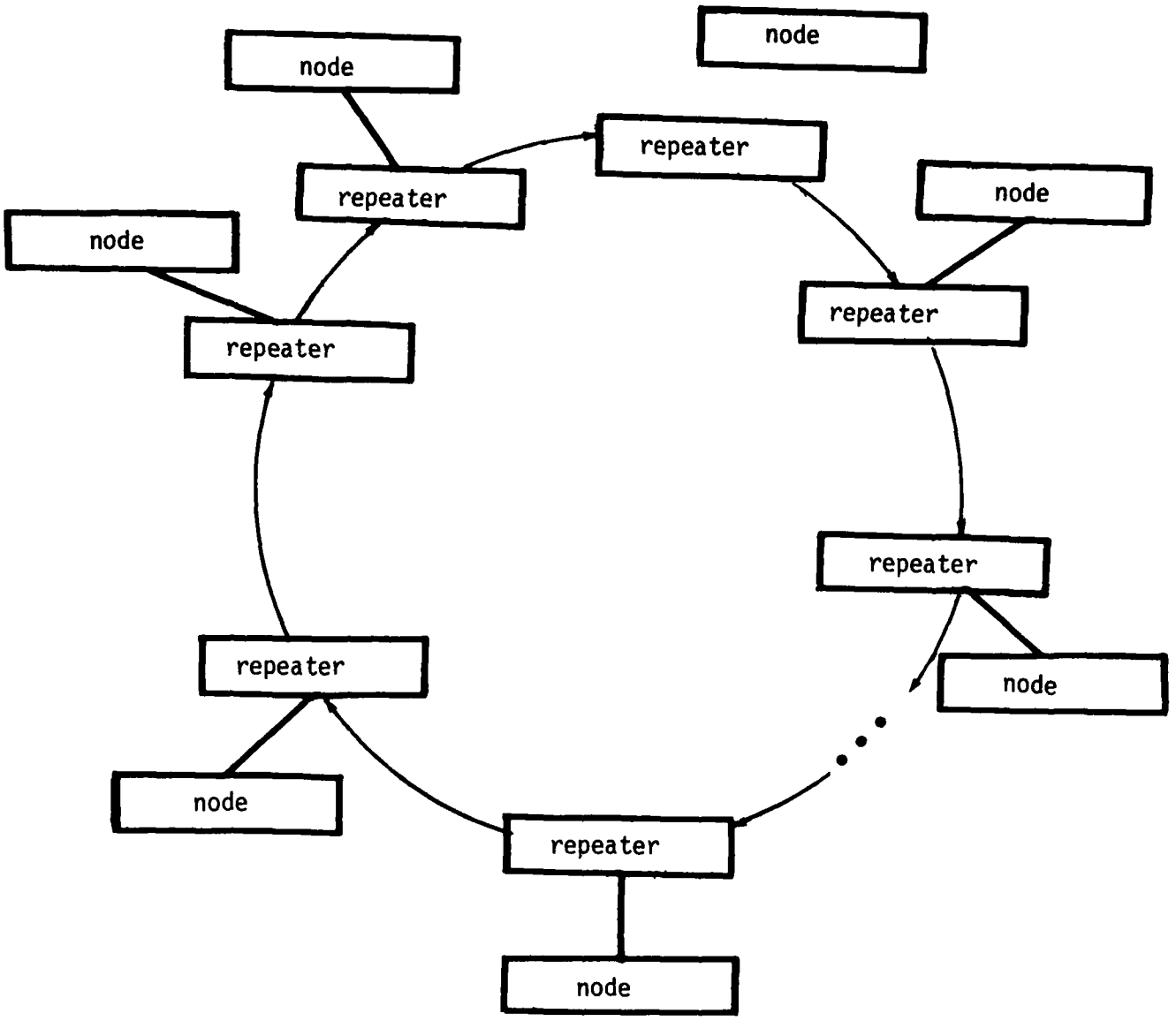


Figure 5.10-14 Ring Network

2) Repeater Failure. Since the repeaters are all in series, failure of any repeater would make the ring inoperable. For the kind of office environment for which our ring is intended, it will be common for many of the nodes not to be in operation at any time. But the repeaters must always operate.

3) Perambulation. When either a repeater or a cable linking two repeaters fails, locating the failure requires perambulation of the ring, and thus access to all offices containing repeaters and wire runs containing cables. Portable test equipment is also required.

4) Installation Headaches. Installation of a new repeater requires selecting two repeaters that are supposed to be directly linked, verifying that they actually are linked.

This architecture topology is not to be confused with the "virtual ring" described in section 6.1.7. The "virtual ring" is a distributed system design with a high degree of the software quality criterion "virtuality" (see 8.1.2). In a "virtual ring", the user on a system interacts with the system as if (from his point of view) it was simply a ring network. In fact, the underlying architecture may be Ethernet, generalized interconnect, or any other topology which allows intercommunication of messages called "tokens". The "virtual ring" design can have substantially higher values of reliability, survivability, and reconfigurability while keeping the logical simplicity of the ring.

5.11 TAXONOMY OF ARCHITECTURES

Another way of classifying distributed system architectures is given in (A&J). As shown in Figure 5.11-1, architectures can be contracted according to communications transfer strategy, transfer control method, transfer path structure (topology), and system architecture.

5.12 COMPLEXITY VS. GENERALITY OF ARCHITECTURES

Figure 5.12-1 shows how 16 of the topologies of 5.10 are rated in terms of interconnection complexity, cost, and specialization. These relative rankings for complexity, etc., indicate relative differences in topology impact on quality factors. For example, the

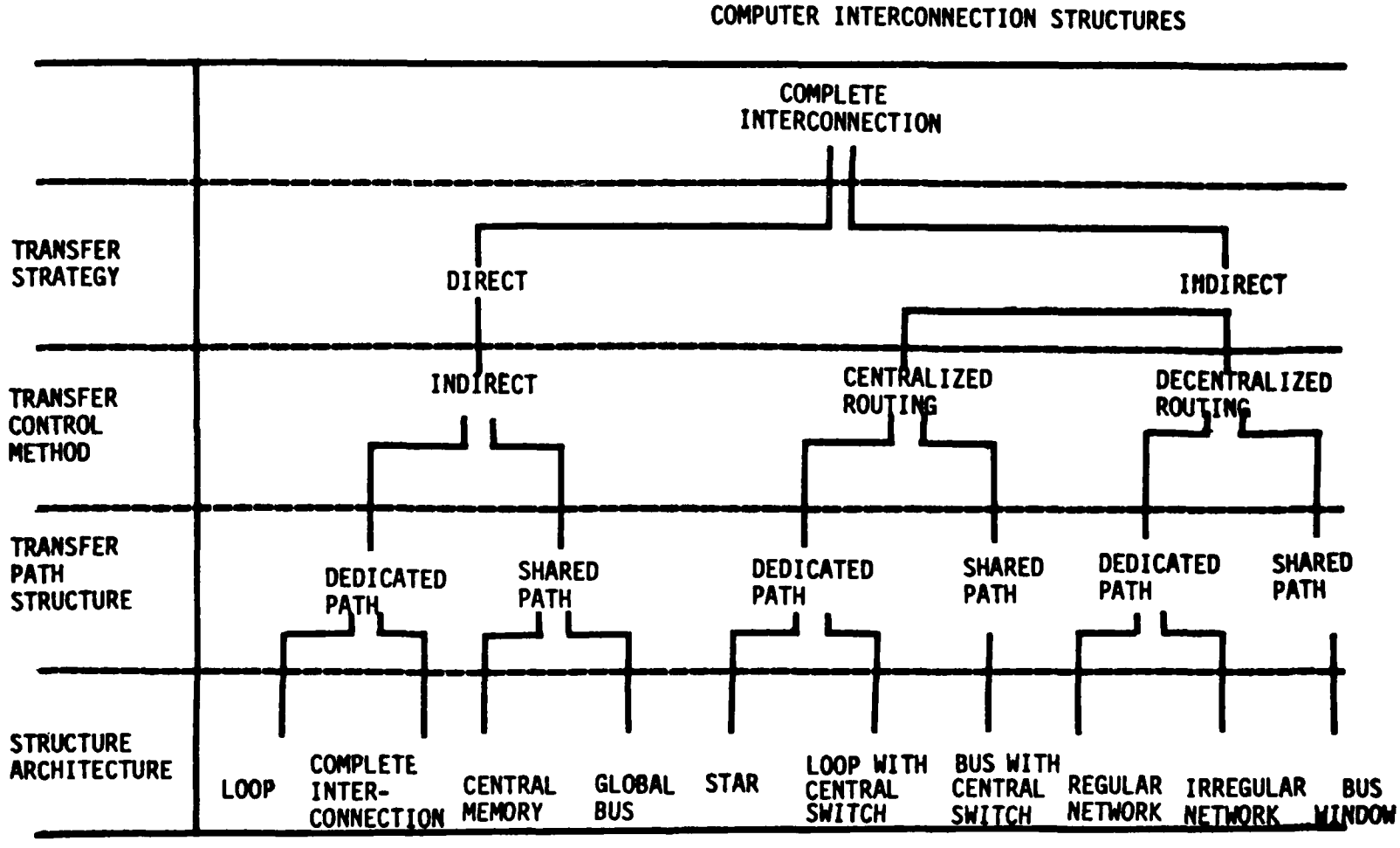


Figure 5.11-1: A Taxonomoy of 10 Multiple-Computer Architectures

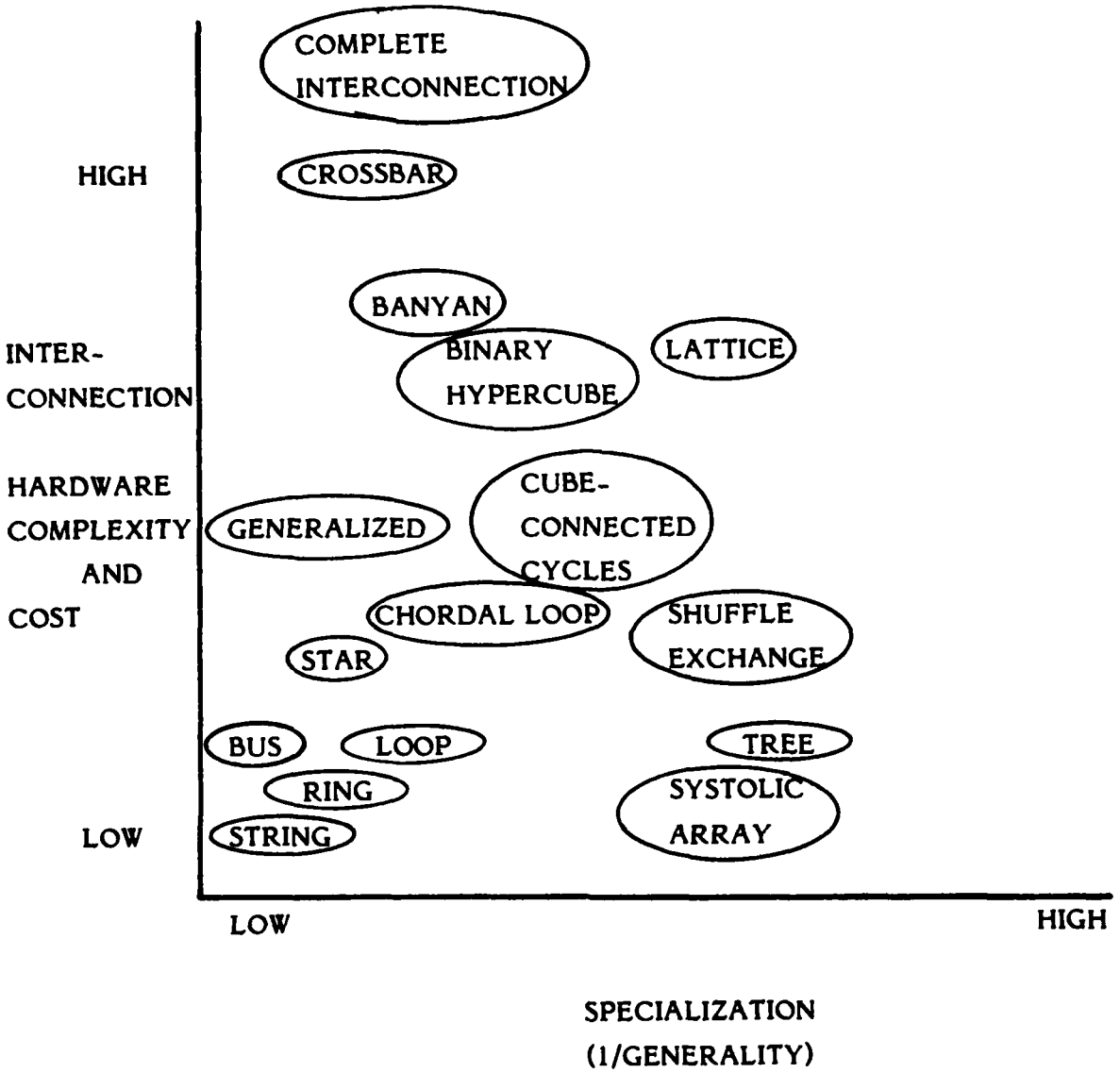


Figure 5.12-1 Topology Ratings

complete interconnection topology is higher in cost and complexity than the string topology, but this is traded off against the higher survivability of complete interconnection. The systolic array topology is more specialized than the star topology, and so we expect star topologies to be applicable to a greater number of embedded distributed computer systems.

5.13 IMPACT OF TOPOLOGY ON QUALITY FACTORS

Table 5.13-1 shows, for each of 12 quality factors, the impact (high or low) of choosing any of the 14 topologies described in section 5.1.10.

5.14 SYNCHRONIZATION OBJECTIVES

(LEL 79) clarifies some basic concepts of distributed computing. It includes a description of the objectives of distributed interprocess synchronization techniques, a classification of them, and a list of criteria useful to assess them. Two particular techniques are compared against the criteria of throughput and response time. Multiple interprocess synchronization techniques include: single physical clock systems, multiple physical clock systems, circulating tokens, shared variables, synchronized counters (logical clocks), independent counters, static sequencer, and circulating sequencer. Evaluation criteria include: (1) RESPONSE TIME: Any solution should take full advantage of the parallel nature of the system; parallelism and anticipation in processing as well as in communication may result in a good resource utilization ratio. This has a non-negligible impact on system costs and response time. (2) RESILIENCY: Any solution should survive failures. Actually, we need a more precise measurement of such a property which would express the number of simultaneous failures (of producers, of consumers) such a solution may survive. (3) THROUGHPUT: How many operations per time unit may be processed? Techniques which do not take full advantage of the parallelism offered by distributed computing systems will probably achieve low throughput. (4) OVERHEAD: Costs of a given technique may be low, monstrous, or acceptable. It is necessary to evaluate overhead as regards traffic (number and size of additional messages), processing (handling of additional messages, CPU cycles necessary to run synchronization software) and storage (for control information). (5) CONVERGENCE: When conflicts occur, how does a solution lend itself to avoidance of infinite waiting, without resorting to any exotic or ad-hoc mechanism? (6) EXTENSIBILITY: If a solution allows for dynamic system reduction

QUALITY FACTOR	BANYAN	BINARY HYPERCUBE	BUS	COMPLETE INTERCONNECT	CROSSBAR	CUBE-CONNECTED CYCLE	GENERALIZED	LATTICE	LOOP	RING	SHUFFLE EXCHANGE	STAR	STRING	SYSTOLIC ARRAY
Correctness	H		H	L			L		H	H		H	H	
Efficiency	H	H	H	L	L		H		H		H	H	L	H
Expandability	L	L	H	L		L	H	H	H	H	L	H	H	L
Flexibility	L	L	H	L	H	L	H	L	H	H	L	H	H	L
Integrity	L	H	H	L	L		L		H	H		H	H	
Interoperability		L	H	L	H	L	H	L	H	H	L	H	H	L
Maintainability	H	H	H		H		H				L	H	H	L
Portability	L	L	H	H	H	L	H	L	H	H	L	H	H	L
Reliability	L	H	L	H	H		L		L	L	L		L	L
Reusability	L	L	H	H	H	L	H	L	H	H	L	H	H	L
Usability	H	H	H	H	H	L		H	H	H	L	H	H	L
Survivability	L	H	L	H	H		L	L	L	L	L	L	L	H

Figure 5.13-1: Impact of Topology on Quality Factors

H = High

L = Low

(it is resilient), then it is necessary to show how this requirement matches the requirement of dynamic system extension. What this means is that it should be possible to re-insert or to add new processors to the system without disrupting the functioning of the system. (7) DETERMINACY: A technique may be designed so that it always achieves some necessary synchronization. Such a technique may be characterized as being deterministic. Oppositely, a technique will be said to be probabilistic if it achieves some necessary synchronization only most of the time. (8) RECOVERY: How fast and how easy is it to recover from a crash and to install again processes and data in a system when using a given technique? Is some exotic technique needed? (9) CONNECTIVITY: With some techniques it is required that producers be fully connected, i.e. each of them operates with permanent connections to each of the others. This may be expensive, difficult, or impossible to achieve compared to partial connectivity (to only some neighbors). (10) INITIATION: How easy is it to initiate the system and to let know to any process when it is allowed to produce or consume actions? (11) EXPRESSION OF ORDERING RELATIONSHIP: When producers themselves wish to enforce some ordering relationship on a given set of operations, they are supposed to exchange some messages to synchronize themselves. But there are many ways of doing this. For instance, it may be necessary for every producer to wait for the completion of its producer-operation before it can fire the next producer, or it may do so when this is initiated. This may greatly impact response time and throughput. (12) CONSTRAINTS ON UTILIZATION: Is there any artificial constraint due to the utilization of a specific technique? For example, is it possible to achieve any desired synchronization even if producer-operations are dynamic? (13) MUTUAL INDEPENDENCE: A technique should provide for the individual isolation of processes against abnormal situations. For instance, the failure of one producer which was using a number of resources of some consumers should not destroy the consistency of the system, nor should this prohibit another producer from using these same resources. (14) UNDERSTANDABILITY/SIMPLICITY: It is intuitively understood that formal correctness proofs will be easier to establish if a solution is simple. This is important when observing that the number of situations which may arise from failures may become very large. Also, when time has come to implement a given technique, problems such as specifying, debugging, maintaining and modifying the corresponding software become preponderant.

SECTION 6
EFFECT OF HARDWARE AND FIRMWARE ON SOFTWARE QUALITY

The distributed system regime is analyzed from several points of view. These include:

(1) The issues that arise between a system acquisition manager and a software acquisition manager in allocation of quality goals to software.

(2) The issues that arise between a software acquisition manager and a hardware acquisition manager in ensuring that hardware and software implementations of the quality goals are compatible.

((1) and (2) are summarized in 6.3)

(3) the issues that arise when an existing distributed system is being modified or expanded to include new functions. (These issues are discussed in the section on Metrics, Tradeoffs, and the Distributed Life Cycle, 6.4)

(4) the issues that arise in making tradeoffs between firmware, on one hand, and hardware or software. These firmware issues are presented in 7.3.

A number of scenarios were developed that collectively cover the major system and software quality allocation issues that arise in distributed systems. These scenarios are presented in 6.1.

6.1 SCENARIO GENERATION AND ANALYSIS

A simple scenario for each of several application types was generated and analyzed. Scenarios collectively cover the major system and software quality allocation issues that arise in distributed systems.

Scenarios include: Distributed Command and Control, Distributed Communications, Distributed Database, Distributed Avionics, Distributed Functional Testing, Distributed

Space System, Distributed Virtual Topology, Distributed Optoelectronics, and Distributed Microcircuit Multiprocessor.

6.1.1 Scenario for Distributed Command & Control System

LT. Gen. Hillman Dickenson, Director, C3 Systems, JCS, gave his views on C3 requirements, survivability, interoperability, and trends in (DIC 81) (Dickenson, LT. Gen. H., "Improving C3 Systems and Requirements", SIGNAL, May/June 1981, p.67-76). He emphasized the challenge of developing and establishing a new, long-term, DOD emphasis on a "systems approach to identifying requirements ... for C3 systems." It has become clear that the life-cycle development of a C3 system demands greater up-front expenditure of effort in the requirements and preliminary design phases. Already accomplished is an "improved evaluation of our survivability requirements to meet warfighting needs." Besides the improvement in the evaluation of Survivability of distributed C3 systems there is also an improvement in the monitoring and increasing of Interoperability in theater and joint tactical C3 systems. The appropriate scenarios to consider for the development of a distributed C3 system can be grouped into two categories: Lessons from the Past, and Extrapolations for the Future. What distinguishes the two, formally, is a marked change in the regulations.

DOD Regulations *5000.1* and *5000.2* have new provisions which provide for "evolutionary improvement of command and control systems." Further impetus is provided by JCS PUB 19, "Command, Control, and Communications Systems Objectives and Management Plan" which structures that part of the acquisition process which includes the management of objectives identification and requirements. Requirements can be divided into Evolutionary and Revolutionary. Evolutionary requirements derive from evaluation of deficiencies, hardware failures, changing environment, mission changes, procedural changes, or increased threat. Revolutionary requirements may stem from new concepts, new objectives, new strategies, new policies, new weapon systems, or new technology.

Examples of Revolutionary change mentioned in (DIC 81) include:

Microcomputers. "Significant advances are sometimes the result of a radical departure from precedent but sometimes result from the cumulative effects of trends of scale.

Micro-computer technology is undoubtedly one of the significant drivers in revolutionizing C3 systems. The reason is not only because of their ability to provide greater computational capacity in smaller volumes and at lower costs, but for applications in communications processors, various interface devices to improve systems interoperability and ease of packaging for use in mobile systems".

Optical fibers. "We are beginning to use optical fiber transmission lines in place of cable, but their wide bandwidth and physical characteristics make them potentially much more important in terms of entirely new systems concepts." See 4.3.1.8 for further discussion.

EHF/MMW. "EHF radio and millimeter wave technology provide similar promising improvements in bandwidth survivability and security."

Examples of evolutionary change include:

The common user subsystem of WWMCCS (Worldwide Military Command and Control System) was criticized in 1973 by the General Accounting Office (GAO) as being unresponsive to national and local level requirements. As a result, hardware replacements are planned, computer-computer connections are being increased, with the number of networked sites going from 8 to 20. Dual communication lines now link major users. New hardware, software, and firmware are being considered, including "more capable machines in smaller packages, more survivable systems, new programming techniques, internetting techniques, and intelligent terminals. Other examples of evolutionary change are described by the former Assistant Secretary of Defense, C3I, Gerald P. Dineen, in (DIN 80). (Dineen, G.P., "C3I - The Essential Element", NATIONAL DEFENSE, April 1980) "We have, for example, told Congress that WWMCCS must have a BMEWS (Ballistic Missile Early Warning System) upgrade, the fuzzy sevens (FSS 7 : An early offshore radar warning system) need to be replaced by PAVE PAWS (Phased Array Warning System); we want to keep PARCS (Perimeter Acquisition Radar Character-ization System, part of SAFEGUARD ABM defense) and DEW (Distant Early Warning, bomber defense radar) in operation; we must harden the NEACP (National Emergency Airborne Command Post -- Boeing E-4B) and improve our MEECN (Minimum Essential Emergency Communications Network) with more TACAMO (Navy's ballistic missile submarine very low frequency communications) and begin planning to replace

AFSATCOM (Air Force Satellite Communications) with Triple-S (Strategic Satellite System.)"

He gives this jargon-riddled and virtually impenetrable example to illustrate that C3I is "perceived as an extremely complicated milieu of acronyms which are somehow electronically linked together. We communicators have failed to communicate.... we tend to exhibit what I call the 'black box' syndrome; that is, instead of dealing with entire systems, we have broken systems down into subsystems, and these subsystems have been broken down further into ... black boxes." One would describe this as a design flaw, measured by a low level of Communicativeness, Clarity, and Virtuality. One notes that these criteria are quite important, because however well the hardware, software, and firmware are designed, the system may not be politically manageable when "we communicators then go to the Congress to try to defend communications, command, control, and intelligence necessities in terms of these black boxes, assigning each an obscure, unpronounceable acronym."

Indeed, LT. Gen. James W. Stansberry, USAF, Commander of ESD, commented (STA 81) (Stansberry,J.W., "Defense Industrial Base Issues", SIGNAL, July, 1981, p.16) on cumbersome acronyms in a similar vein. "We feel that what we are doing is vitally important, but all these wonderful systems will not be any good unless we know where they are, can talk to them and tell them what to do. We need to communicate this to a lot of people, including Congress. How do you go about this? One way is to change the name of our systems.... Names are important. Names capture the imagination of people and help people understand what it is we are about. Names help get money and funding. As long as we talk to each other in code, the good things are not going to happen."

Cipriano and Cohen of Booz, Allen & Hamilton (CIP 81) (Cipriano, F.L. and Cohen, A., "C3I Covenants -- An Architectural Tool for the 80s", SIGNAL, May/June 1981, p.115-123) claim that the problems of C3I systems are not the fault of systems architects, "but rather the inadequacy of the tools available to them and the organizational climate in which they must manage the design of the system." This suggests criteria for Design Tool Availability and Organizational Supportiveness.

(CIP 81) also suggest that the allocation of functionality can be a problem, and can lead to shifting interface requirements. "The definition of functional boundaries between major C3I systems is often unclear and subject to frequent change. This gives rise to a set of dynamic interdependencies between systems which are uncontrollable by any single architect." We do not see this as a problem peculiar to C3I systems. Rather, we may state this as a metric hypothesis:

The Higher the Distributedness, the Heavier the Dynamics.

The More Distributed the System is, the More Changes will be Distributed.

The most important document for a Distributed System is the Declaration of Interdependence.

Distributed Systems are Globally Dynamic.

Another problem is the "fragmentation of design responsibility among a variety of individuals and organizations who respond to different sets of operational requirements and development objectives." This may be recast as a maxim:

Distribute the system, not the responsibility!
Distributed systems need centralized goals.

An additional source of problems mentioned in (CIP 81) is "the C3I CONTINUUM IMPERATIVE which requires that all new capability be added in such a way that current service is not degraded, duplication of resources is minimalized, and proper respect is shown for the sunk costs of the existing system." The tradeoff here is between high levels of expandability and evolveability on the one hand, and the implementation of a sequence of "quick fixes" on the other hand. Each quick fix may satisfy the C3I continuum imperative when taken by iteslf, but a series of such changes can "paint you into a corner" and preclude necessary architectural changes, provide diminishing returns, and unduly expand life-cycle costs.

(CIP 81) proposes a conceptual model, the "covenant", which extends the notion of layered protocols in distributed processing computing networks into a structured

methodology for all C3I systems and subsystem designers. There is a parallel between the properties of the proposed covenants and the factor/criteria/metric framework, as the two following examples show:

"They must provide for standardized interfaces at these boundaries (between systems, subsystems, elements) to permit MODULARITY."

"They must permit unambiguous definition of the conditions to be met to achieve INTEROPERABILITY, COMPATABILITY and INTERCHANGEABILITY."

It appears plausible, in principle, to apply the ISO (International Standards Organization)/ANSI (American National Standards Institute) "Reference Model for Distributed Systems" to the design of distributed C3I systems. As a conceptual and functional framework for C3I standards development, this should prove useful. What is needed, however, is a consensus within the C3I design community. Such a consensus is difficult to achieve, as opinions within the community are themselves highly distributed.

LT. Gen. Donald R. Keith, Deputy Chief of Staff for Research, Development and Acquisition, Department of the Army, has outlined the necessity and advantages of Distributed C3I. "The most unique, most pervasive battlefield function which offers the greatest potential as an attainable force multiplier is command, control, communications and intelligence (C3I)." (KEI 81) (Keith, LT. Gen. Donald R., "Distributed C3I -- A Force Multiplier for the 90s", SIGNAL, Sep.1981, p.11) Gen. Keith analyzes conventional C3I architecture as based on a centralized minicomputer and peripheral mass storage devices. The deficiencies of this architecture are classified as:

* Critical node -- low survivability

* Little integration with other functional systems

* Low hardware Commonality with other systems

* Requires stable, sheltered environment

* High unit cost

A desirable alternative would be a distributed C3I system with microprocessors replacing the minicomputers. The advantages of this architecture would be:

* Physically portable compared to minicomputer

* Lower unit cost

* Most processing at user node

* Distributed data storage in network

* Survivability -- no critical nodes

* Hardware commonality

* Lower acquisition cost and ILS investment

* Commonality of architecture

* Software flexibility

Gen. Keith adds that "progression from the recent centralized C3I architecture is not without risk.... Present strategy focuses on starting with proven technology in development systems." After noting that this evolution depends upon development of a quickly assimilated man-machine interface, evaluation in a genuine user environment, and survivability through being able to use a repertoire of communications media, he concludes:

"Although the communications technology is being developed, we are still limited by the state-of-the-art in how we can separate C3I nodes. A truly distributed C3I system may not be achievable for another 5 to 10 years. However, unless we direct our attention to this area now, it may be a high technology bubble which will continue to float just beyond our grasp well into the decade of the 90s."

6.1.2 Scenario for Distributed Communications System

The earliest significant reference on Distributed Communications is (BAR) (Baran,P., "On Distributed Communications Networks", IEEE Trans. Comm. Sys., March 1964, p.1-9) This paper evolved from a series of RAND Corporation Memoranda, *and predicted* several approaches which have now become commonplace, but were then presented in a theoretical context. The design goal was, explicitly, "the synthesis of a communication network which will allow several hundred major communications stations to talk to one another after an enemy attack."

The emphasis on survivability requires, for quantitative evaluation, a clear measure of vulnerability to destruction of the system as a function of vulnerability of each station and the topology of the interconnection. In this early paper, the "criterion of survivability" is the "percentage of stations both surviving the physical attack and remaining in electrical connection with the largest single group of surviving stations." This presumes that surviving sub-networks are considered ineffective if not linked to the largest surviving sub-network, and is therefore a conservative measure of the ability of the network to operate after attack as a coherent entity.

The justification for this analysis is given in terms of cost/reliability tradeoffs, which in turn lead to tradeoffs in topology and routing strategies. The description of these tradeoffs is simply stated:

"We will soon (after 1964 but well before 1983) be living in an era in which we cannot guarantee survivability of any single point. However, we can still design systems in which system destruction requires the enemy to pay the price of destroying N of N stations. If N is made sufficiently large, it can be shown that highly survivable system structures can be built, even in the thermonuclear era. In order to build such networks and systems we will have to use a large number of elements. We are interested in knowing how inexpensive these elements can be and still allow the system to operate reliably. There is a strong relationship between element cost and element reliability. To design a system that must anticipate a worst-case destruction of both enemy attack and normal system failures, one can combine the failures experienced by enemy attack together with the failures caused by normal reliability problems, provided the enemy does not know which elements are inoperative. Our future systems design

problem is that of building at lowest cost very reliable systems out of the described set of unreliable elements."

The metric employed to measure connectivity is the "redundancy level." A minimum span network, one formed with the smallest possible number of communications links, has a redundancy level of 1. If twice as many links are used, for each node, the redundancy level is 2. For redundancy levels above 3, the details of which nodes are linked to which become relatively unimportant. There are two key conclusions drawn from Monte Carlo simulations of such systems. Both conclusions are validated by numerous subsequent studies.

Conclusion #1: "extremely survivable networks can be built using a moderately low redundancy of connectivity level. Redundancy levels on the order of only 3 permit the withstanding of extremely heavy level attacks with only negligable additional loss to communications". (see Figure 6.1-1)

Conclusion #2: "the survivability curves have sharp break points. A network of this type will withstand an increasing attack level until a certain point is reached, beyond which the network rapidly deteriorates. Thus, the optimum degree of redundancy can be chosen as a function of the expected level of attack. Further redundancy gains little". (see Figure 6.1-2) See also the definition of Kleitman's algorithm for metric DI.1 in Volume II of Appendix C of this report.

6.1.3 Scenario for Distributed Database System

One of the world's most advanced distributed database management systems is R*, under development at IBM San Jose Research Lab (LIN 81). (Lindsay,B., "Object Naming and Catalog Management for a Distributed Database Manager", DCS2, p.31-40) The design and development of R* shows particularly careful attention has been paid to tradeoff issues on distributed systems. For example:

Users may distribute data and workload among multiple processing sites

-- but --

Users need a "single system image" for which data seems to be in one place

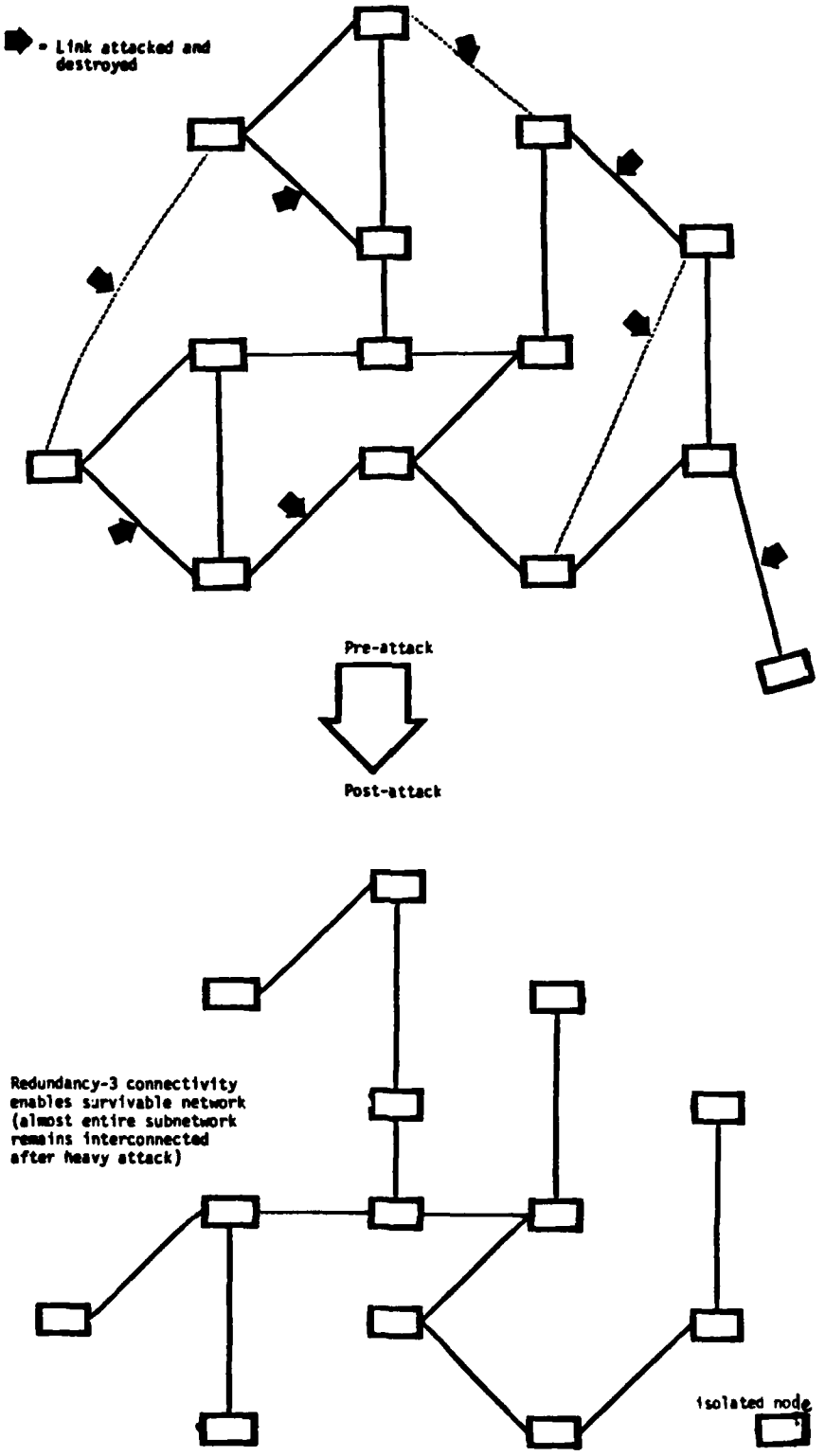


Figure 6.1-1: Survivable Network Withstands Attack

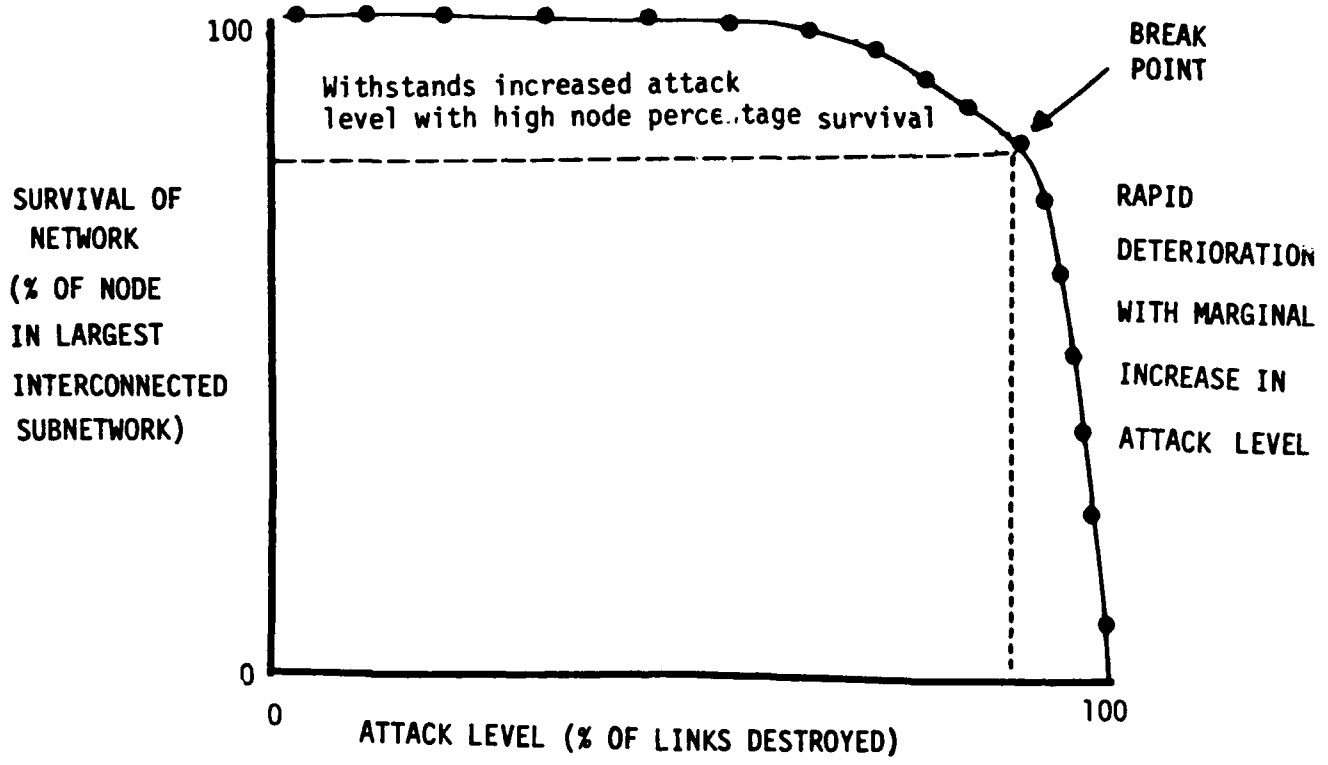


Figure 6.1-2: Network Deteriorates Beyond Attack Level Breakpoint

The system should support "graceful growth" of new sites and data

-- but --

The system should preserve local administrative autonomy and control of access to locally stored data

The system supports relational database management (as in the experimental single-site System R, developed at IBM San Jose)

-- but --

There can be a relational database manager at each site of geographically or locally distributed hardware interconnected by a data communication network.

Some discussion of Relational Database Systems follows, to put this in context. Questions which need to be answered include: (1) What is a relational system? (2) Why does it matter that it's relational? (3) How can you tell if it is a good system? (4) What metrics are applicable to evaluate a distributed relational system? (5) If the relational system rates poorly for a distributed application, how can it be improved? The relational database concept is increasingly recognized as an important advance in the state-of-the-art. It is most useful for big systems (i.e. more than a trillion characters) where the database is shared by several users. Large storage systems such as IBM's AMPEX (a trillion bytes in ten cubic meters) are mostly used for archival storage at present, put can be accessed by a relational front-end.

Writing the software for a relational system, especially a distributed relational system, is a nontrivial task. The prevailing approach is to use an off-the-shelf relational system, and then add whatever special purpose application software is needed for a particular project. The relational system itself is designed to be highly REUSABLE. Otherwise, to develop a new relational system, it is cost effective only if amortized over several usages or several concurrent users. In the latter case, the user interface must be well defined. One usually can't afford to have one user tie up a hundred disk spindles on a global search.

The important software characteristic for a distributed relational system is CONSISTENCY. Access to a record must be such that all other users are excluded while one user is changing that record. The database on the distributed system EVOLVES over time, physically. As a result, the overall constraint on the system is DATA

INDEPENDENCE. DATA INDEPENDENCE is the ability of the program to execute consistently while the physical structure of the data changes. The system must evolve along structures while maintaining data independence. The system must live over time because it is used by people over time.

The problems of consistency number at least four. (1) Dealing with redundancies in the database, (2) Change in one record propagating to a change in another record, (3) the set of all people reached by one pointer must be the same as the set of people reached by another pointer, or else the pointer structure itself causes inconsistencies, (4) you must be able to tell the database what conditions on consistency allow conformity to the database as desired. The problem is not that databases crash, but that they POLLUTE. Bad records remain, and spread, and degrade the consistency and reliability of the entire system.

The relational model of data was developed by E.F.Codd at IBM Research and refined by Michael Stonebraker. The basic idea is easily stated. The proper way to manage databases is to view data as a TABLE (two dimensional array, with labelled rows and columns) with certain structure and certain design and implementation so that the user does not care what that design and implementation are, only that the manipulation of the database involve homomorphisms (mappings which preserve the data).

The requirements for the database, in order to achieve high USABILITY and VIRTUALITY are: (1) The separation of logical and physical structure and function, (2) Extraneous detail should be handled by the system, not thrust at the user, (3) Data independence must be maintained, so that the user need not care about changes in the actual storage structure of the data, (4) the system must be simple, without sacrificing power, (5) The reason that REUSABILITY is emphasized for a relational system is that rewriting programs is difficult when the inventor is gone. Programs depend on a stored ordering. This is, implicitly, order dependent. To have a relational system, you must make the program independent of permutation of order OR give information to generate that order when accessed, but be able to take cognizance of sometimes needing to sort the data. The system must also minimize indexing dependence. In COBOL we recognize syntax changes to handle physical dependence. In this sense, a pointer is an identifier for an item, but not an identifier of where the item is (otherwise there is no data independence). A pointer may be seen, logically, as an implicit representation of

a predicate. The system doesn't know what the pointer means, but only that it is there. The system can vary as long as the predicate is still true. The pointer in a relational system can also be seen as a procedure: the user must be able to get to a data item eventually, but independent of how the system actually finds that item.

Codd summarizes the process of transforming a database system into a relational system: "Flatten all your files." A flattened file has all its records arranged in a table. There are no hierarchies in the database, no tree structures. Once the user activities are limited to dealing with a collection of flat tables, the USABILITY, SIMPLICITY, and UNDERSTANDABILITY take on extremely high values.

6.1.4 Scenario for Distributed Avionics System

Boeing's experience in distributed systems applied to avionics for tactical transports and modern tactical fighters is directly related to the evaluation of software efficiency, reusability, testability and flexibility. Much of this work has been connected with the Air Force Digital Avionics Information System (DAIS) program. A set of processing load and I/O scenarios was developed and implemented via a user-controlled set of applications software. A number of metrics applicable to software efficiency (e.g., interprocessor communications overhead, transmission delay time, and system response time) were developed and measured under this program. In addition to quantitative measurement of efficiency, a number of qualitative metrics were examined during the evaluation of the DAIS executive. These included reusability, flexibility and testability.

Based on the results of the DAIS executive evaluation, a smaller, more efficient executive was developed. The same metrics were applied to the new executive to show that a substantial increase in efficiency had been achieved with no negative impact on the other software characteristics.

6.1.5 Scenario for Distributed Functional Testing

A distributed array of interconnected sensors, servos, digital computers and touch-sensitive video terminals on the Boeing 767 twinjet is revolutionizing the functional

testing of the new airplane. The scenario below is adapted from a Boeing report by Barry Romsberg.

Functional testing is what every airplane must go through in final assembly and on the flight line prior to flight and delivery to the customer. On the 767, automated test systems use state-of-the-art components, both built into the airplane and installed temporarily, to obtain faster, more accurate and more consistent results than test systems used on earlier-model airplanes. The 767 and 757 represent the first introduction on a total scale of the use of digital flight avionics. As a result, we have digital computers that talk from one end of the airplane to the other. They are very sophisticated, and they are very accurate. Since the new digital avionics touch almost every system in the airplane, they introduced a whole new series of functional-test opportunities. Both physical and electronic airplane systems are functionally tested. The physical systems are the airplane's flight controls, engine controls, engine and air system and fueling system. The electronic systems tested are the airplane's avionics. This is the complex instrumentation used for navigation, automatic flight and engine control and physical system monitoring.

Flight Controls

The flight control surfaces of the two new airplanes -- the flaps, ailerons and spoilers on the wings, the elevators and rudders of the tail assembly -- have small transducers built into them that send electrical signals on their positions to the airplane flight deck. The transducer signals in the 767 and 757 are used on the flight deck to indicate the position of the control surfaces. In the case of the spoilers, which are controlled by a hybrid electrical-hydraulic link, rather than a straight hydraulic link as are the other control surfaces, the signals provide feedback to the control loop. The signals are also recorded, for later investigation of any malfunctions. The transducers are already built into the airplane control-surface actuators and are highly accurate. The result is that we will now not only be able to measure much more accurately than before, but more rapidly and with better repeatability.

The test system is attached to the built-in transducers with temporary cables. Microprocessors convert the transducer signals into digital form and send the signals to a

Hewlett Packard HP-9845 computer mounted on a mobile console outside the airplane. Other transducers are temporarily attached to the control wheels, control columns and rudder pedals on the flight deck. Signals from these transducers, showing control movement and force required, are also sent via microprocessors to the HP-9845 computer.

The computer, programmed with a testing sequence based on engineering functional-test-document requirements, compares the signals from the flight deck controls and from the control surfaces with values from the functional test document. In the past, these tests required five or six people to functional-test an airplane's flight controls. Now it takes just one person in the cockpit and another outside as an observer.

The electronic readouts show the same results for the same movement, time after time, and they are accurate. Estimated cost savings from the computerized flight-control functional testing are at $735,000 over the first 200 airplanes.

Engine Controls

The complexity of the engine-control functional testing is suggested by the number of control and indication wires -- some 300 -- that electrically link the engines to the 767 flight deck. There was no way that we could get a test harness in there to test the wiring properly by conventional methods. We had to do a lot of starting and stopping of the engines to troubleshoot the electrical system -- very wasteful. Not only did it use up a lot of fuel, but it wasted man-hours, because when you start running the engines, most of the other work has to shut down. With this type of advanced, computerized equipment, we could assure ourselves that all the wiring that goes from the engines to the cockpit, including the engine instruments, works like it's supposed to, before the engines are installed.

Like the flight-control functional test system, the engine simulator uses an HP-9845 computer outside the airplane, connected as the engine will be to the flight-deck wiring, and a Fluke touch-sensitive terminal temporarily mounted on the flight deck. The computer is programmed to simulate engine response to the flight-deck engine controls and instruments, in a test sequence according to engineering functional-test

document requirements. We wanted no pencils in the cockpit because of the greater possibility of error in a paper system, according to Dean Patterson, lead MR&D engineer for all the 767 physical-system functional tests and responsible engineer for the engine simulator. What we were shooting for on the flight deck was something as natural and foolproof as pointing your finger.

Engine Trim

Once installed, the 767's engines have to be trimmed: another functional test. This means verifying that the engines are adjusted and operate properly so they can achieve their rated take-off thrust and rated thrust at other power settings. The computer is programmed with functional-test requirements, testing sequences and the mathematics to convert engine performance under atmospheric conditions at the time of testing to performance under standard conditions for which the engine specifications are written. In automatic mode, the trim system will take readings on engine operating parameters to determine how well the engines are doing, calculate what adjustments are necessary and perform them. The only piece of equipment the operator will have to interface with will be the Fluke touch-sensitive terminal.

Also, we are trying to get improved accuracy in fuel measurement. Current equipment is plus or minus 2 to 3 percent. We're shooting for plus or minus a few tenths of a percent, to certify the system for 1 percent. Fuel on the 767 is carried in the airplane wings, its quantity measured by 36 capacitance probes (which change in electrical capacitance as the fuel level around them changes) and indicated by gauges beneath the wings and on the flight deck. A densitometer in each wing tank automatically determines the unit weight of the fuel, which the fuel-system computer multiplies by the quantity of fuel on board to determine total fuel weight -- important for calculating airplane gross weight to keep within structural weight limitations and where takeoff distance is weight-critical. Operator responses are made with a light pen on the video control terminal screen and by means of programmable keys on the bottom edge of the terminal. A multicolor pictorial display on the video control terminal shows the critical elements of the fueling system, and such information as switch and valve positions, pumping rates, system pressures, weight of fuel in the tanks and the percentage difference between the amount of fuel pumped and the amount indicated by the airplane's gauges. Each step of the test and the corresponding results are printed

out by the computer to document the accuracy of the fuel system and, in the event of a system abnormality, to provide a detailed list of system component-performance for troubleshooting. One problem was bringing all the computerized stuff into a fueling environment. It has to be explosion-proof.

Analysis in Terms of Distributed System Quality Factors

The quality factors most enhanced by this distributed system are testability and usability. The testability criterion of instrumentation is improved by allowing software to more effectively see and measure hardware errors, by using a distributed sensor array of built-in transducers. The criterion of modularity is improved by the common use of the HP-9845 and test executive software in both physical and electronic functional-test. Usability of the system is extremely high. The criterion of operability is raised by embedding most test procedures within software, rather than in the traditional paper-based approach to functional-test. Training is another criterion within usability which increases as a result of sensors being built-in and pre-wired, rather than having to be installed at functional-test time by specially trained personnel.

Maintainability is another factor which is fundamentally changed. The maintenance of the physical and electronic systems continues the use of the same equipment and software developed for testing during the earlier phase of the aircraft. Up-front costs became high during the development of the system, but major cost savings were realized over the system life cycle. These savings may also be attributed to consistency and accuracy of tests, allowing for the elimination of redundant testing, and to the flexibility and efficiency of the test system. Interoperability and reusability were important goals of the test equipment design, because a maximum of commonality is intended for the 767 and the 757.

The system is designed around the user, who has access to most of the components which need to be tested. The user interfaces with a portable terminal, rather than an unwieldy set of heavy tools operated by a number of people on stands built up off the ground. The terminal is touch-sensitive, eliminating the need for paper, lightpen, or frequent keyboard activity. The interpretation of test results is unambiguous, compared to the subjectivity of the old tape measure and bubble protractor readings. Every attempt was made to increase the effectiveness of the user, and this gave a real

gain in reducing the number of testing personnel required. The same user can [illegible] initiate, and monitor a test sequence, then manipulate engine controls, [illegible] from flight-deck instrumentation, compare values with those in test requirements, and record observations of malfunctions. This amplification of the capabilities of the operator by means of a carefully designed distributed system shows that attention to quality factors, even when not evaluated by numerical metrics, could have a significant effect on performance of an embedded system -- such as a military aircraft incorporating these features of a Boeing 767.

6.1.6 Scenario for Distributed Space System

Space-based communications systems pose special problems for hardware/software/firmware tradeoffs. Estimates of hardware evolution time have varied widely. According to Aviation Week & Space Technology, 20 July 1981, there may be a space-based strategic laser communications system as early as 1987. This contrasts strongly with Carter administration predictions of such a system after the year 2000. One key component is a narrow bandwidth tuneable receiver to be located on manned bombers and fleet ballistic missile submarines. The space shuttle would be used to deploy and maintain the laser satellites in orbit. DARPA has successfully tested a prototype receiver (GTE Sylvania / Lockheed) onboard a submarine, according to AW&ST 18 May 1981, p.15. An experimental prototype laser satellite, P80-1, is scheduled for launch in 1984. The interest in this system surrounds its high bandwidth (on the order of a billion bits/second) and its narrow footprint (on the order of 100 meters). This suggests very high quality criteria of data security.

The concept most important to hardware/software/firmware tradeoffs in such a system is known as "complexity inversion." Complexity inversion, a term coined by Ivan Bekey is a non-standard way of distributing computational functions between in-orbit units and ground support units. The traditional approach is to optimize the satellite for minimum weight, minimum power consumption, minimum functional complexity, and maximum reliability. The ground stations are non-critical in weight, power, and complexity, with less stringent reliability built in. The complexity inversion approach, on the other hand, optimizes the ground stations for minimum weight, minimum power consumption, and minimum functional complexity. The space-based components become larger and more complex. There are several consequences to this shift [illegible]

parameters. Cost, flexibility, survivability, and expandability are all impacted by complexity inversion. When there are many ground stations per in-orbit unit, complexity inversion can actually cut total costs.

For example, the Defense Satellite Communications System (DSCS) supports a wide community of users, including the Worldwide Military Command and Control System (WWMCCS), Defense Communications System (DCS), National Command Authority (NCA), Ground Mobile Forces, Diplomatic Telecommunications Service, and some defense forces of selected allies. Federal communications are handled by commercial carriers, but the need for secure and survivable military command and control links is growing. Complexity inversion is being designed in, to some extent, within the DSCS. The upgrade from DSCS I to DSCS II satellites (first launch in Fall 1981) included major additions to the complexity of the satellite. These new capabilities include: a variety of selectable antenna patterns, nulling of communications jammers, payload control by users, increased number of channels, and higher flexibility. Survivability testing, including hardening against nuclear EMP effects, has received considerable attention through development.

Another explicit example of complexity inversion is the Navstar Global Positioning System (GPS), a space-based radio navigation network for all services and NATO. Earlier generations of navigation satellites, such as Transit and Nova, required large and complex ground stations. GPS will allow contact with ground troops using lightweight backpack receivers. GPS will provide precise three-dimensional position (within 16 m), velocity to within 0.1 m/sec, and time to within a microsecond to users worldwide. Applications include precise weapon system delivery, tactical missile nav updates, enroute navigation for land, sea, air, and space vehicles; geodesy and photocartography; range safety, including search and rescue missions; ariel rendezvous and refueling. Eighteen satellites total are needed for global all-user coverage. But one effect of this complexity inversion is to make these services more attractive and cost effective to civilian users. Hence, GPS may be extended to worldwide use by civilian terminals (in ships and planes) at somewhat degraded accuracy. The use of backpack receivers may also be extendable to civilian services, and may ultimately affect the tracking of cargo delivery on land.

6.1.7 Scenario for Virtual Topology

Anotner significant aspect of the Sirius-Delta project is the careful design of the virtual topology. Although the physical topology may be an Ethernet connection or some other distributed design, the virtual topology is that of a ring (see 5.1.10.14) -- a "virtual ring." To explain this structure, we first identify the elements of the system:

AGENTS are external entities (human users, sensors) that access the system in order to activate transactions.

TRANSACTIONS are sets of elementary actions (delete, read, write, create) that manipulate data objects grouped into files.

PRODUCERS are executive processes which control the execution of transactions, and the firing of actions.

CONSUMERS are executive processes in charge of performing actions.

The goal of the executive design is to achieve high levels of three qualities: Consistency, Robustness, and Extensibility. The meanings of these qualities are described in (LEL 81) Appendix B.

Consistency is the set of constraints on the values of objects, so that all copies of a given object have identical values at all times, through any sequence of updates and potentially interfering transactions. The key to achieving Consistency in this system is the criterion of Atomicity. Each transaction is viewed by producers and consumers as Atomic, indivisible. The goal is to provide a mechanism at the distributed executive level which guarantees Atomicity regardless of transaction concurrency.

Robustness is the set of mechanisms for surviving or recovering from failures of the storage elements, the processing elements that host the producers and consumers, and the communication subsystem. When possible, transactions are completed. An incomplete transaction violates Atomicity, so redundant information must be used to reinstall system files to a Consistent state. Additionally, the communication subsystem may fail and split the global system into several isolated subsystems. One may want to

keep only one subsystem alive, or several, or all (note that this contradicts an assumption of (BAR 64)). In the latter cases, the distributed executive must allow several subsystems to be merged while communication is restored. Extensibility is achieved by a combination of physical modularity and executive modularity. Physical modularity -- plugging modular hardware in and out -- is not enough to reach a dynamically extensible system. The executive itself should be modular: a set of autonomous cooperative local executives which embed a decentralized control mechanism which achieves mutual independence among various system elements. "Extensibility and efficiency are related issues. For example, it is possible to improve the performance of a given system -- such as its response time -- by adding more processing and storage elements, provided that the executive mechanisms do not create artificial bottlenecks. If this aspect is overlooked, a computing system, although physically extensible, may not satisfy performance requirements."

Producers are viewed as being in sequence on a virtual ring. Each Producer has a unique permanent identity (an integer value). Each producer has, due to the virtual ring structure, a unique successor and predecessor at each moment. Each producer regularly checks for the existence of its successor, and as this goes around the ring, system status is monitored in a distributed way. If these existence checks ("life messages") fail, the originating producers initiate reconfiguration by passing special messages to the potential successors of the vanished producers. Processors which recover may reinsert their producers into the virtual ring by means of another distributed protocol. The important point here is that all reconfiguration activities are performed in a distributed way, so there is no possibility of single-point failures jeopardizing survivability.

The system also passes messages called "tokens" around the virtual ring, and these tokens carry "cycle numbers" which are incremented for each revolution around the virtual ring, which helps decide which messages have been lost prior to or during reconfiguration. Transactions submitted to producers by agents are time-stamped with a "ticket" message, and this is used to resolve all conflicts in a distributed fashion. Consistency is maintained, although no place in the system contains total information on the system status. This approach is very important in achieving high survivability.

Other designs for distributed database systems which are efficient, extensible, and robust are described in (BER 79), (ELL 77), (GRA 78), (LAM 79), (REE 79), (ROS 78).

6.1.8 Scenario for Distributed Optoelectronics

Glass fibers have been used to transmit information over short distances (meters) for 30 years. Optical communications links on this scale now are under development for embedded computers in weapon systems. Numerous tradeoffs exist with coaxial cable or twisted wire pair technology. The primary effects on systems quality for short-distance optoelectronics are shown in Table 6.1-1.

Table 6.1-1: Optoelectronic Impact on Quality Factors

FACTOR	OPTOELECTRONIC IMPACT
Correctness	Inherently low bit-error rate; redundant paths improve this even further
Maintainability	Only a very few available facilities for optoelectronic maintenance, but extremely long MTBF possible with integrated semiconductor/fiber hybrid technology; reduced installation time, fast deployment
Reliability	Very high reliability indicated, but special applications experience still limited
Flexibility	Optoelectronic devices have increasingly diverse functional applications, however greater flexibility is given by electronic/semiconductor technology at present due to intensive development of the latter; flexible connectivity (connect optimum sites)
Testability	Not accessible to most available Automatic Test Equipment. Special microscopic, optical, multi-gigahertz testing apparatus needed for testing. BITE possible.

Table 6.1-1: (cont'd)

Portability	Insufficient standardization to date for optoelec. communications frequencies, protocols, but much activity starting in this area with RADC/Army document on tactical cable structure submitted to DESC
Reusability	Little compatability at present between different optoelectronic systems
Efficiency	Exceptionally high efficiency in data transmission per unit of power expended. Exceptionally high efficiency in physical density; low-weight thin fibers replace heavier copper cables;
Usability	Currently limited usability, but rapid expansion likely as semiconductor/optical interface technology evolves
Integrity	Physical security of optoelectronic links is high. Almost unjammable; untappable by electrical pickup
Interoperability	Currently low interoperability between special applications; generic 26-pair cable replacement proposed for USAF/Army applications
Survivability	Exceptionally high survivability, due to inherent insensitivity to EMP, jamming, radiation; some degradation (darkening) with heavy radiation

The reason for restriction to low-distance light transmission has been, for decades, due to enormous losses. Early light fibers had losses over 1000 decibels per kilometer. Amplitude of data-carrying light pulses could be cut in half within a meter. By 1970 losses in Corning fibers were down to 20 db/km. 1980's fiber technology approaches ultra-low losses such as 0.1 db/km. This is near the quantum limit, which is due to photon Raleigh scattering. The scattering is no longer determined by fabrication impurities, but by thermodynamically generated in homogenieties of the glass. Glass fibers are anisotropic; only crystal fibers could decrease scattering further. Another approach to decreased scattering and increased transmission distance is the (currently experimental) use of "super long wavelength media." Super long wavelength operation would be in the near- to mid-infrared range of the spectrum, with fluorozirconate fibers replacing silica glass fibers, lead selenium telluride sources and mercury cadmium telluride detectors. Super long wavelength fiber optic cables would also need to be cryogenically refrigerated to limit thermal noise generation within the cable itself. Intermediate between super long wavelength cables, which will be available in the late 1980's, and today's short wavelength cables are the long wavelength cables now emerging from the laboratories. These use special silica glass fibers which transmit at 1.3 micrometers, use indium gallium arsenide phosphorus source and germanium detectors, and double or triple transmission distances. Injection lasers, instead of LEDs, also increase distances.

All of this change has one major impact, but this is an extremely important impact:

> Optoelectronic links can be hundreds of kilometers long. With repeater stations, optoelectronic networks can be world-wide in extent.

This can be combined with the previous major impact of optoelectronics.

> Extremely high bandwidth: ability to transmit very short optical pulses (picosecond) allows a glass fiber a few microns in diameter to simultaneously carry on the order of a million low-speed telephone lines.

Fibers can handle gigabits per second, which is significantly faster than other modes of communication. With higher power and larger losses, similar data-transmission rates may be possible for laser propagation through the atmosphere or earth-to-space.

The effect of combining high rate (gigabit/sec) with long distance (hundreds or thousands of kilometers) is dramatic:

> Geographically distant components can be linked into a tightly-coupled distributed system.

One limitation to this so far is the experimental nature of optoelectronic interfaces. The development of the interface technology is important, as it links the crucial three components of the system:

> semiconductor diode laser
> VLSI semiconductor electronics
> low-loss light fibers

Future components of such a system include:

> optical memory
> optical computation

Fiber optics military development areas are shown in Table 6.1-2.

Table 6.1-2 Fiber Optics Military Development Areas

ADDCS	I/O modules development
Air-layable Cable	Intrusion resistance
AN/GRC-206	LHOST
AN/GYQ-21(V)	Long Haul FO System
AN/PPS-18 links	MIL-STD-188
AN/TPS-32	MIL-STD-1553/FO
AN/TPS-43	Mine countermeasures analysis
AN/TTC-39	Missile test link (Pt. Mugu)
AN/URN-23	NAVALEX fiber optics courses
ASDC	NORAD system
ASW aircraft video links	NSA testbed
ATACS	NTDS link
Audio Demo Kits	Nuclear EMP program
A-6E analysis	PTTI link (Wahiawa)
A-7/ALOFT	SDMS analysis
A-7 EMI/HUD	Shipboard study
CMPS	SLCM/GLCM
COMMSTA analysis	SPS-48 analysis
COMPASS EARS	Submarine Standards/Specs
DEB analysis	TACS C2
DIGITAC	TAOC
ELCS	Towed Array link/IOs
ESSEX Program	Tri-service
FAC-1 (FT. Meade, MD)	TTCP
FAC-2 (Wahiawa, HI; Guam)	TV demo system
FAC-2A (Offut AFB, Ramstein AFB, Norfolk Naval Station, Hickam AFB)	USS Kitty Hawk TV link
	USS Little Rock phone system
FO Crypto bypass	USS Plunger sonar link
FORTE	Video dist analysis
FO guided missile	YA-V8B flight test program
FO technology	600 VF/VO link
ICF analysis	2000-ton SES analysis
I/O modules concept	

Space Operations Center Application

NASA Johnson Space Center has funded research in phased array RF beam applications. Their immediate interest was in using fiber optics to distribute phase information in large space-based antennas for beaming energy to Earth from solar power satellites. A number of fiber cables, for stability under temperature variation, have been developed and tested. A one kilometer 980 GHz fiber optic link, transmitter, and receiver was built and is now being modified for potential application to the manned Space Operations Center.

6.1.9 Scenario for Distributed Microcircuit Multiprocessor

Examples of system design strategies are provided by the S-1 family of embedded distributed microcircuit multiprocessors. (See Table 6.1-3). Originally developed by Lawrence Livermore Laboratories, under Navy direction, the S-1 is being extended to second-generation devices which emphasize various design strategies and quality factors. Boeing Aerospace Company has benchmarked the S-1 for potential space applications for USAF, including a space-based radar system and mosaic infrared star sensors. Original Navy design emhasized reusability, efficiency, and reliability. Reusability was enhanced through the development of a uniprocessor instruction set called the S-1 native mode. This ensures that throughout the distributed system life cycle, the accumulated software will be minimally impacted by changes in hardware. Such changes are anticipated, including faster microcircuits, change from logic gate to gate array technology, and VLSI reduction to single-chip configuration. Efficiency was emphasized through the design of a common hardware structure for cost-effective high-performance S-1 uniprocessors in each of several types of applications, including interconnection to accomplish multiprocessing. Efficiency for multiprocessing depends on a distributed architecture of 16 uniprocessors (each comparable to a Cray-1) which share a common main memory. Efficiency in manufacture was also a goal, so the first-generation Mk.1 version of the S-1 processor used 5500 commercially available ECL 10,000 series integrated circuits, and was built in under 30 man-months with automatic wirewrap making 60,000 single wire connections in two weeks.

Table 6.1-3 The S-1 Multiprocessor

S-1 design	Mk.1	Mk.2A	Mk.3
Integrated Circuits	5,500	11,000	450,000 transistors
Type of IC	ECL 10,000	ECL 100,000	VLSI
Speed	20 megaflops	400 megaflops	1 gigaflops
Configuration	uniprocessor	multiprocessor: 16 CPUS + memory	multiprocessor
Companies involved	Lawrence Livermore	Boeing Cray GE IBM Norden Sanders	
Software	Standard instruction set microstore emulation Fast Fourier Transform paged virtual memory	SCALD (CAD-CAM) AMBER operating system high survivability multilevel security	
Applications	Lawrence Livermore (Navy/DOE development) Navy (ocean surveillance, point defense) Army (distributed mobile BMD system) Boeing Aerospace (USAF/space applications) . mosaic infrared starer sensors . space-based radar system		

Reliability was a quality factor of primary importance in certain applications. As discussed below, when geographically dispersed systems are built around an S-1 system, the factor of survivability predominates.

One of the potential users which has evaluated the S-1 is the Army Ballistic Missile Division Command (BMD). BMD emphasized efficiency, interoperability, and survivability. BMD sought an inexpensive computer with high bandwidth, and wanted enhanced efficiency through VLSI technology, multiprocessing, and concurrent processing techniques (vector instructions, digital filter, and fast Fourier transform). BMD (in Spring, 1981) tested the second generation S-1 device known as the Mk.2A. The Mk.2A is expected to achieve higher levels of efficiency throught the use of higher parallelism (11,000 IC's) and faster microcircuitry (the ECL 100,000 series). Testing is done on about 400 lines of code for missile intercept algorithms. BMD design strategy emphasizes the quality factor of interoperability. The S-1 is planned to interface as effectively as possible with existing BMD equipment for C2 and phased-array radar. The S-1 interoperability should allow it to provide aperture synthesis, with a large aperture for accurate range, range rate, and angular measurement essential to the interceptor missile function. Interoperability is also improved by the use of microcoded instruction decoding which allows the S-1 to emulate older machines. Survivability is enhanced through geographic dispersion, hardened communication links, mobility, distributed sensors, failure tolerence, and reconfigurability. Components of the system would be geographically dispersed, with components linked to small phased-array radars. The communication links would be coaxial cable, secure VHF, secure UHF, or fiber optics, in order to minimize jamming, EMP, and other challenges to the quality factor of integrity. Mobility of the system should also make it harder to destroy. The sensors would be distributed: many small radars instead of one large radar site. The S-1 multiprocessor architecture is designed so that single-module failure of a uniprocessor, memory bank, or crossbar switch, will not degrade system performance. The operating system (AMBER) is designed to detect hardware failures and automatically reconfigure the system to replace damaged functions from a bank of reserve modules.

Software plays an important role in the flexibility of the S-1 system. Expandability is expected to be enhanced through the use of the SCALD package of programs (AW&ST, 26 Feb 1979, p.67). SCALD is a computer-aided design package which reduces the number of man-months of design by using hierarchical structured techniques originally used in

software design. SCALD detects certain classes of design errors before prototyping. SCALD is a full CAD/CAM system since it drives wire-wrap equipment, reducing errors in circuit board manufacture. Several companies have requested further information on SCALD and its use in designing the Mk.2A, including Boeing, Cray, IBM Yorktown, General Electric, Norden, and Sanders. SCALD also has a timing verifier that optimizes performance within a given arithmetic logic family. Advances in semiconductor technology are expected to increase logic density (the next generation Mk.3 should have almost 500,000 transistors on a single chip) but only slightly improve speed, so increased throughput should be attained through increased parallelism and optimized design. Usability receives attention in S-1 design. The processor's native instruction set includes three-operand instructions, which enhance usability of compilers for high level languages. Paged virtual memory makes the logical address space machine independent. To aid in the operation of time-sharing, which scores high in usability, the software architecture has concentric rings of protection, also improving integrity. Techniques are under development to optimize the partitioning of functionality by segmenting conventional scientific application programs into modules distributed among multiple processors, for minimum-time execution. The program would be prefaced with a direction set to aid in this partitioning, and which would minimize program redesign.

6.2 INTEGRATION OF ANALYSIS RESULTS

In the terminology of "Software Metrics" (Perlis, Sayward, and Shaw, ed., MIT Press 1981), the central issues in Software Metrics revolve around the question:

> Is it possible to identify or define indices of merit that can support quantitative comparisons and evaluations of software and of the processes associated with its design, development, use, maintenance, and evolution?

In the context of the Boeing Aerospace Company study, a similar question is implicit in the framework of the research:

> Is it possible to identify or define a theoretical/empirical model which includes a hierarchical framework of Factors, Criteria, and Metrics for the quantitative comparisons, predictions, and evaluations of Distributed systems, the tradeoffs between software, hardware, and firmware, and of the processes associated with the

design, development, deployment, maintenance, and evolution of these Distribution Systems?

Despite the numerous references and definitions in the interim reports of research to date, it is clear that only fragmentary evidence is available in the current literature, but that these seemingly isolated partial results suggest the urgent need for and promising value of a scientific basis for analyzing and evaluating Distributed Systems.

6.2.1 Quality Factor Emphasis

The distinctive feature of our approach is the quantitative evaluation of distributed computer systems in terms of the quality factors/criteria/metrics methodology. This may be seen as either extending the quality factors approach from the original uniprocessor domain to the distributed systems domain, or as the classification of distributed system issues in terms of the quality factors nomenclature. Gonzalez has emphasized the need for techniques that permit an evaluation of distributed structures (GON 78). Gonzalez suggests that such an approach requires four elements: (1) a set of attributes or descriptors, (2) a measure of how much of each of these descriptors is needed for a particular problem, (3) an indication of the extent to which a distributed structure possesses each of these attributes for that problem, and (4) an indication of the relative importance of each attribute to the problem being addressed.

The quality factors approach addresses each of these elements as follows: (1) The factor/criterion/metric structure provides a hierarchy of attributes, which are identified independently of any specific architecture. All candidate architectures are subjected to the requirements of each of these attributes. (2) The measure of need in a particular problem is determined either directly or indirectly. The Acquisition Engineer may directly specify the quality factors and criteria which receive top priority in system design strategy. Or, indirectly, the set of stated rationales for the selection of a distributed system may be analyzed as to the factors most heavily impacted. The matrix given in Figure 3.0-1 may be used for such analysis. (3) The quantitative measure of extent of attribute possession is accomplished through computation of the associated metrics. Since the metrics are most easily calculated by direct measurement of an existing system, they are most accurate in this case. When metrics are applied to a hypothetical candidate system, accuracy is decreased and is dependent on the applic-

ability of the model. (4) The relative importance of each attribute to the problem being addressed depends on the priorities mentioned in (2) as well as the interrelationships of quality factors. Qualitative and quantitative relationships among quality factors are discussed in Vol. I.

Gonzalez has suggested that "decisions are often made by placing a disproportionate amount of emphasis on a small number of evaluation parameters or descriptors while neglecting others that are more important and should have more of an impact on the final choice." This error is avoided in the quality factors approach, which attempts to be all-inclusive in the identification and measurement of parameters. Nonetheless, the quality factors approach is subject to abuse if too few quality factors or criteria are emphasized in the design strategy, as this may leave too many design decisions unconstrained. The opposite error is also possible. If too many quality factors are emphasized, the design may become overconstrained, and there may be no design at all which satisfies a contradictory set of requirements. It is expected that continued use of the quality factors approach should produce lessons learned as to the appropriate number of parameters to specify and the degree of freedom left to the system designers.

Another common practice is to base a current design on a variation of a previously successful design even though the requirements that apply in one situation are only marginally applicable in another. This may be avoided through the use of the quality factors approach, as distinct requirements will generate distinct quality factor emphases. Different rationales for the selection of a distributed system will characteristically lead to different design approaches.

Gonzalez expanded his approach in a follow-up paper: (GON 79). This paper includes mathematical means for assigning weights to attributes, selecting the number of attributes, evaluating the degree of confidence, and globally optimizing the design. There are comparisons yet to be drawn between this approach and the one currently under study.

Hardware/software tradeoffs in the design of distributed systems are shown in Figure 6.2-1.

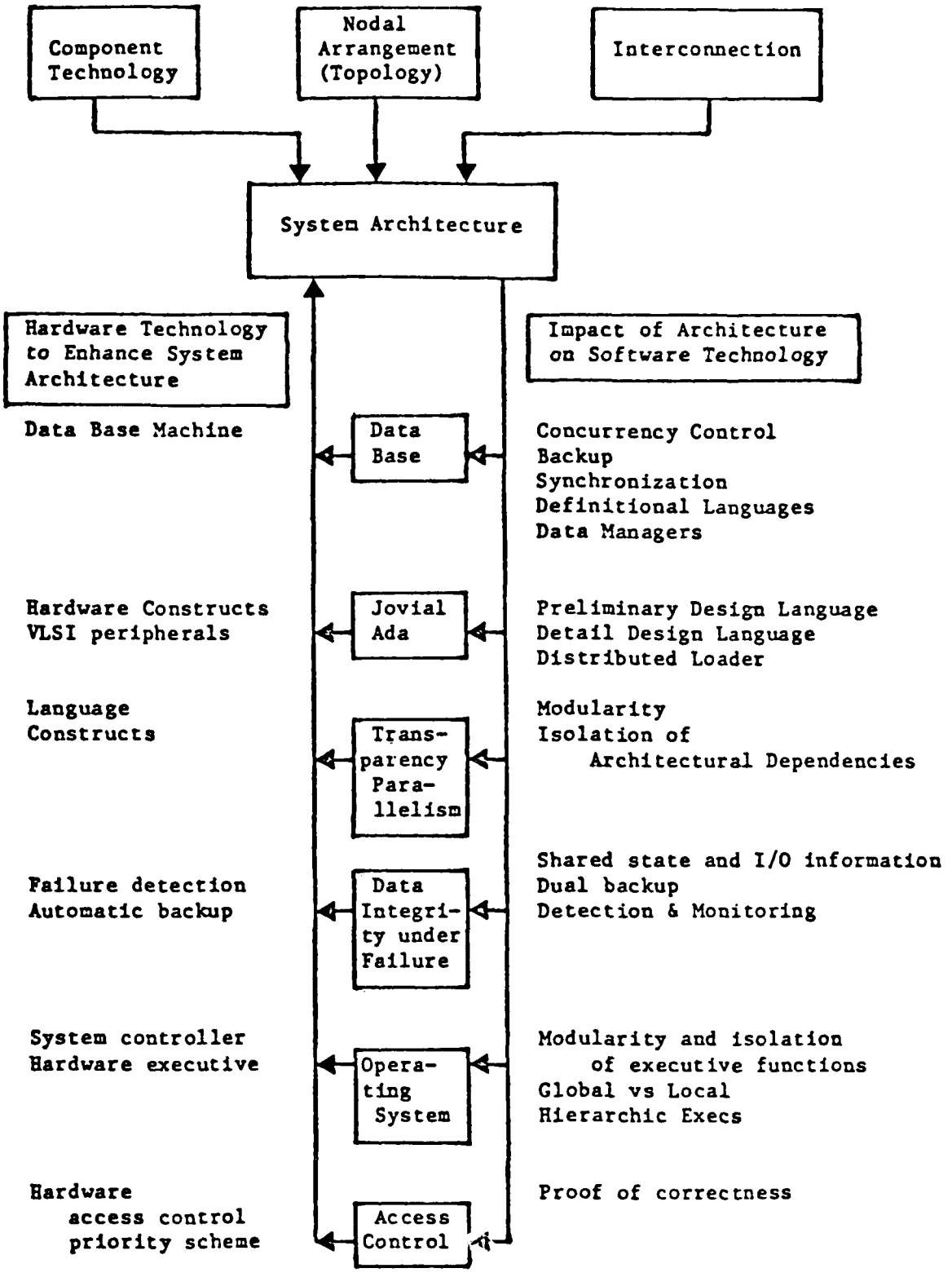


Figure 6.2-1: Hardware/Software Tradeoffs in the Design of Distributed Architecture

6.2.2 The Stuff of the System

"Scientists certainly believe in quantification: to measure things is a part of their trade sine qua non. The question is: what should we measure and what measuring rod should we use? I shall call whatever answers these questions THE METRIC. My advice remains and is quite general: always seek a metric that handles the stuff of the system." (BEE 75) (Beer, Stafford, "Platform for Change", John Wiley & Sons. 1975 p. 95).

6.3 DISTRIBUTED SYSTEM TRADEOFF MODELS

From a management standpoint, the value of distributed system metrics is to help make decisions. Some such decisions are classified in Table 6.3-1 as the concerns of the software, hardware, or system acquisition manager.

Table 6.3-1 Management Decisions and Distributed System Metrics

Software	Hardware	System	Management Decisions
		x	Is it time to move ahead to the next phase of the system?
		x	Is a return to an earlier phase recommended?
x	x	x	Does part of the system demand rework or special action?
x	x		Are the software/hardware/firmware tradeoffs being made effectively?

Table 6.3-1 (Con't)

Software	Hardware	System	Management Decisions
	x		Is changing technology introducing new cost-effective implementations?
		x	Does the current system meet requirements and specifications?
		x	Are production costs appropriate in this phase?
x	x	x	Will more memory (or other resource) need to be added?
		x	Is the system developing fast enough?
		x	Are components needing to be replaced more often than expected?
		x	Does manpower loading put people and effort where needed?
	x		Is the originally designed machine configuration still appropriate?
		x	Will the project be done only when it is no longer needed?
x			Is there pressure to change language (i.e. to Ada) or protocol?

Table 6.3-1 (Con't)

Software	Hardware	System	Management Decisions
	x	x	Is there new work elsewhere which can be incorporated in the system?
x			Does the availability (or non-availability) of a software tool matter?
x	x	x	Is the system running near its limits? Will it need to?
x	x	x	Are new enhancements being demanded or dreamed up?
		x	Does the system have a meaningful all-encompassing conceptual structure, or is it becoming a grab-bag of options?
		x	Should the rest of the system be built all at once, or should some prototype be demonstrated first?
x			Is the programming discipline and standards enforcement having a positive effect on the project?
		x	In what sense is the project on schedule? For a distributed system, it is always possible to "prove" it is ahead AND behind schedule.

Table 6.3-1 (Con't)

Software	Hardware	System	Management Decisions
		x	Is the system within budget? Does the budget accurately reflect the peculiar nature of the system?
		x	How close is the system to meeting specifications? 50%, 85%, 99% ?
x	x	x	Is top down or bottom up testing and integration a better plan?
		x	How can the best match be achieved between system architecture and system test methodology?
x		x	How well tested is the system, module by module and as integrated s/w?
	x	x	How well tested is the hardware and firmware, component by component, facility by facility, link by link, and as an integrated system?
x		x	Numerically, how close is the simulated environment to the operational environment?
		x	Are progress reports, error reports, technical changes, and contract changes being effectively handled?

Table 6.3-1 (Con't)

Software	Hardware	System	Management Decisions
x		x	Can the documentation, configuration management. design, or testing processes be enhanced by the use of automated software tools?
		x	Is this phase of the life cycle a "post-release augmentation of the system specifications to meet unforseen demands" or a "let's patch the worst bugs first so they won't know how bad we screwed up?"
		x	Never mind what this deviation/enhancement/retrofit costs, what does it do, and why is it worthwhile?
	x		Will the new hardware last longer than the old hardware?
x			Is each successive version of the software really an improvement, or is the system evolving in random directions as if unmanaged?
x	x		Has the hardware outgrown the software?
x	x		Has the software outgrown the hardware?
		x	Have the system requirements grown beyond the need for the system?

Table 6.3-1 (Con't)

Software	Hardware	System	Management Decisions
		x	Has the system become obsolete before going operational?
		x	Is the underavailability of some resource holding up the whole system?
		x	Are there any clear reasons for rebuilding the system from the ground up instead of iteratively enhancing it?
x	x	x	When the system is to be replaced/cancelled/enhanced what hardware, software, and firmware can be reused, at what total savings?
x			Is there a necessity of changing the language of the system (i.e. assembler to HOL, BASIC to ATLAS, Jovial to Ada), and, if so, how much of the code should be translated?
x			Is the system drowning in data?
x		x	Is the system understood by the users (besides being evaluated as "understandable")?
		x	Does the system appear to be competently administered, or are little problems becoming big problems by mis-action?

Table 6.3-1 (Con't)

Software	Hardware	System	Management Decisions
x			Is the size of the software becoming a problem?
		x	Are system specifications/requirements/designs varying unusually much, or too fast to be adapted to?

6.4 METRICS, TRADEOFFS, AND THE DISTRIBUTED SYSTEM LIFE CYCLE

In this section, the concept of a distributed system life cycle is outlined. We believe that a life cycle model is essential in understanding the behavior of distributed systems of embedded computers. Just as "software" is a dynamic entity whose properties cannot be accounted for measuring some static piece of "source code", so also is "distributed system software" a slippery entity whose properties cannot be evaluated by examining any fixed configuration of hardware and information.

When we talk about a distributed system, we are often using the term as a generic for all the chronological stages of managing a system to solve a changing problem to meet an evolving need. It is widely acknowledge, although the terminology varies, that all software passes through several phases of its life cycle. It is also plausible to model distributed systems which include software to pass through various phases of a life cycle. Since system specification is always imprecise (and more imprecise for larger systems) and since the user demands on a system vary over time, there are often backtracks to earlier phases of the life cycle, or stages to be going on interdependently, or subsets of the system at different locations in space-time undergo different sequences of life cycle phases.

One approach to modeling the distributed system life cycle is to divide procedures and concerns along the lines of Activities. The three major activities are (in the terminology of Vol. I of this study): operation, revision, transition. Software/hardware/firmware tradeoffs may also be grouped according to activities in the distributed system life cycle. Examples are shown in Table 6.4-1.

Table 6.4-1 Metrics, Tradeoffs, and the Distributed System Life Cycle

OPERATIONS	Correctness	Does it do what I want?	Proven h/w or proven s/w
	Reliability	What confidence for use?	Redundancy of h/w or s/w
	Efficiency	Uses resources well?	S/w or f/w controller
	Integrity	How secure is it?	s/w or f/w encryption
	Usability	Is it easy to use?	smart or dumb terminals
	Survivability	Can a subset run alone?	fixed or adaptive routing
REVISION	Maintainability	Can I fix it?	BITE or ATE
	Verifiability	Does performance verify?	s/w or h/w monitoring
	Flexibility	Can I change it?	ROM or PROM
TRANSITION	Portability	Can I move the s/w?	standard HOL or Assembler
	Reusability	Can I use some later?	relocatable code or not
	Interoperability	Interface to other system?	standard protocol or not
	Expandability	Can capability grow?	Off-the-shelf CPU or not
	Evolveability	Can use new technology?	Optical fiber or not

The life cycle approach to the management of large systems is proving effective in USAF experience. The administration of the development of major weapon systems is coordinated through the system life cycle model, which relates decisions, documentation, resource allocation, transfer of responsibility, and so on. Within this structure, management of the software component has been enhanced through the formalization of the software life cycle. There is reason to believe that distributed systems have characteristics distinct from uniprocessor systems, which make it difficult to shoehorn them into the system or software life cycle model. Consequently, there may be advantages to defining and formalizing a distributed system life cycle.

The Distributed System Life Cycle (DSLC) has structure similar to, but is not identical to "System Life Cycle" or "Software Life Cycle." DSLC is more complex than uniprocessor life cycles, in the number of possible development scenarios and designs. This is because a distributed system can EVOLVE in more different directions. For example, a system can evolve from uniprogramming to multiprogramming to tightly coupled multiprocessor to loosely coupled network. Or a system can evolve as the convergence of separately developed subsystems (flight system, test system, support system, weapon system) which are never all coupled together at once, or for long periods of time (in this case, the system is distributed in time rather than space). Or a system can go through several changes in virtuality without any physical reconfiguration. Or physical reconfiguration may be hidden from users, who need not know that new, cheap, distributed hardware has replaced, emulated, and extended existing systems or applications software. Or through several cycles of damage and recovery, survivability is achieved despite reductions in operational hardware components. Depending on what quality factors are emphasized, and what unique operational scenarios ensue, the DSLC can be very different from one application to another. This is a reason why the quality factors approach is valuable for the design of distributed systems: the evaluations of system quality is in terms of a large number of pre-defined metrics, some subset of which always apply to a particular application, regardless of how different its requirements and development management methodology may be.

This discussion led to the proposal of a quality factor Evolvability, which was later dropped. (see Vol. I, Section 4.1)

SECTION 7
FIRMWARE ISSUES

7.1 FIRMWARE AND PROCUREMENT

One difficulty in procuring firmware, as opposed to software, is the documentation scheme which limits military procurement. In particular, MIL-M-38510 is intended to minimize the use of "nonstandard" single-source parts in miltary systems, as these are difficult to replace. To have a part promoted from the ranks of the "nonstandard", the commercial vendor submits it to an agency such as RADC for assessment as a procurement item. After a period of time, usually in excess of one year, it is added to the QPL (Qualified Parts List). During this period, it is possible for numerous suppliers to release parts which are faster, cheaper, smaller, or more powerful, but the items on the QPL always get preference. In the fast-paced technological evolution of computers, this delay typically prevents the best tradeoffs from being made.

One way of speeding the procurement of nonstandard parts is for the user to get an interim purchase agreement from DESC (Defense Electric Supply Center), but this procedure takes almost a year itself. The break in this logjam may be Revision G of MIL-M-38510. Revision G recognizes "custom hybrids", the first such recognition in military procurement regulations, and therefore (once approved) will let many more technologically sophisticated parts into the embedded computer system arena. This should allow better tradeoffs to be arrived at in firmware/software situations, by decreasing the time-lag between new part vendor release and new part user availability.

7.2 ADA MICROENGINE SCENARIO

An extreme example of software/firmware tradeoffs appeared at Western Digital (WD), in Newport Beach, California. WD was designing a microcomputer to run Ada. As originally planned, WD would use a set of to-be-developed chips which used Ada as their native code (as in the Intel iAPX). According to Electronic Engineering Times, 6 July 1981, p.4, this approach was abandoned after several months of effort. Instead, existing chips in the 16-bit WD9000 series, which run in Pascal p-code, would run a software-only Ada compiler written in Pascal, in a total of 128 kilobytes of RAM. The

project now splits into two phases. First, complete the software development of the Pascal p-code Ada compiler, and accumulate operational statistics on the WD9000-based system. Second, before 1983, use the accumulated data to help a WD design team develop an optimized Ada microengine. An intermediate step, not actually performed, would be the reduction of the Ada-Pascal compiler to firmware, in ROM, reducing the amount of RAM needed. This scenario exemplifies the use of one HOL (High Order Language) to bootstrap another HOL onto a microcomputer, and then use that to design a High Order Language architecture. The difficulty of making firmware/software tradeoffs is demonstrated by the need for substantial redirection of this WD project. The subtle conclusion is that the tradeoff is not necessarily an either/or decision, but may be an evolutionary shift. The latter is in keeping with the concept of the Distributed System Life Cycle.

7.3 FIRMWARE TRADEOFF ISSUES

Many of the points discussed below are based on arguments presented in (HEN) Hardware/Firmware/Software Tradeoff Methodology: A Design Guide, J.Henrick, Boeing Document D180-20230-3, 1977. In the 1960's and early 70's, computers came in fairly standard hardware configurations, and user-unique requirements were met through applications software. Even capabilities such as floating-point arithmetic, for which hardware was available, was usually done in software due to cost considerations.

In that era, any operations beyond those standard in hardware were done in software, often at the expense of slow turnaround and increased complexity.

With the rise of LSI and then VLSI technology, the situation changed. The cost of hardware dropped dramatically, the relative cost of software increased, and applications software became a systems bottleneck. LSI and VLSI also allowed the use of novel architectures, and the configurations for arithmetic/logic units, control units, and memory management proliferated. Fast, cheap, compact, non-volatile ROM (read-only memory) began to be used to store tables, constants, and microprograms: this was the birth of firmware.

Firmware at the time consisted of microprograms, which are sequences of instructions in low-level language which directly access and manipulate internal hardware at the

register level. This allows faster and more efficient operations than assembly language or high-level language software. Firmware requires more skill than high-level language to develop, as it necessitates detailed understanding of the underlying hardware. That is, firmware has very low Virtuality.

Firmware requires special protection from casual or inadvertent changes, because it operates on hardware not normally available to other software. This is why it is normally stored on ROM: for the sake of Integrity.

Correctness is often low for firmware, as proof-of-correctness requires a very sophisticated model of low-level hardware device interactions. But once a piece of firmware is developed and tested, it will typically have a higher degree of correctness than equivalent software. To this extent, firmware may be standardized almost to the extent of hardware. Computer-aided design and other software tools are increasing the efficiency of the design of hardware and firmware.

In terms of Efficiency, hardware is usually high, software is low, and firmware falls somewhere in between, usually closer to the values for hardware. Software is typically fetched from memory one instruction at a time, decoded, and then executed by hardware. This fetching and decoding requires additional hardware operations, regardless of the efficiency of the software code itself. Firmware is already in memory, reducing fetch time, and requires less decoding than high-level language, hence it can be processed faster (more efficiently) than conventional software. Firmware still involves a less direct sequence of operations than a pure hardware approach.

Flexibility is highest for software, lowest for hardware, and in between for firmware. Software has a clear advantage in flexibility. Hardware devices are difficult to change, especially after deployment and installation. Firmware takes considerably more effort to revise and install into an operational system than does software. On the other hand, a ROM can be removed and replaced by another ROM with little trouble if this need has been anticipated. A ROM, by definition, cannot have its contents changed (written) but only accessed (read). However a PROM (Programmable ROM) is a ROM which can be erased by exposure to ultraviolet light and then have new firmware installed by a "PROM burner." EPROMs (Electronically Programmable

ROMs) have even higher flexibility than ROMs or PROMs, and are only slightly less flexible than software.

In terms of Reliability, hardware is highest, software is lowest, and firmware falls somewhere in between. Hardware reliability is higher than ever at the device level, due to parity checks, circuit redundancy, consensus logic, and so on, but the increasing complexity of embedded computer systems mitigates some of this gain. Software reliability is still a major problem due to the lack of a universally accepted methodology and due to the inherently large human role in design, development, production, checkout, and maintenance. Firmware shares some of the advantages of hardware reliability, particularly from nonvolatility. Firmware is, however, software on silicon, and suffers all the disadvantages of software in its development.

Maintainability involves a different set of concepts for hardware than for software. Maintainability for firmware involves both sets of concepts , those relating to MTBF of the hardware substrate, and those relating to the structuredness of the microcode. This makes firmware maintainability exceptionally tricky to evaluate.

Survivability is a particularly important metric for firmware. Due to the high flexibility and accessability of software compared to firmware, software is vulnerable to unintentional modifications, both of itself and of the data it manipulates. When a processing task is considered too critical to risk the accidental occurrence of software problems such as overwriting or improper manipulation of shared data, then a hardware or firmware implementation is desired. Putting critical functions into hardware is justified in many cases, but usually adds high development costs. Hence it is frequently useful to place critical functions in firmware. The non-volatile nature of firmware also gives it a survivability advantage over software. For endurance and radiation hardening, the tradeoffs must consider the use of magnetic bubble memory, which is more resistant to ionizing radiation than other memory media.

7.4 KEY AREAS FOR FIRMWARE TRADEOFFS

Whenever an application involves a significant degree of concern with speed or frequency of execution, dependability of performance, or potential for change, then this is an area where hardware/software/firmware tradeoffs should be considered.

Functionally, these opportunities arise most often in five categories of computer activities, which are considered in greater detail below: Memory Management, Operational Management, I/O Management, Error Monitoring, Special Algorithmic Capabilities. These functional areas include a significant variety of computer activities. In the subcategories listed below, the need to perform hardware/software/firmware trades is the rule, not the exception.

Memory Management:

- Allocation of storage
- Garbage collection
- Address generation
- Indirect addressing
- Virtual memory
- Indexing
- Paging
- Segmentation
- Cache control
- Data protection
- Data typing
- Stack operation

Operational Management:

- Program binding and linking
- Program loading
- Program relocation
- Data structuring
- Data relocation
- Character string manipulation
- Format checking
- Data type conversion
- Task dispatching
- Queue control
- Compiling
- Assembling
- Emulation

I/O Management:

- Analog-digital conversion
- Interrupt handling
- Peripheral data transfer
- Error detection/correction
- Synchronization
- Buffering
- Channel control
- Protocol

Error Monitoring:

- Parity Checks
- Error-control coding/decoding
- Automatic fault diagnosis
- Self-checking
- Reconfiguration

Algorithmic Capabilities:

- Floating point arithmetic
- Multiple precision arithmetic
- Sorting
- Merging
- Loop execution
- Mathematical functions (sin, log....)
- Table look-up
- Coordinate transformations
- graphics (hidden line, windowing)
- Correlation and autocorrelation
- Fast Fourier Transform
- Number Theoretic Transform
- Vector processing
- Matrix multiplication

SECTION 8
IMPACTS ON QUALITY

The issues discussed in previous sections fit naturally into the system quality framework. This indicates the positive value of analyzing the complex world of distributed systems in terms of quality metrics. The definitions, reasons for selection, design strategies, and topology of distributed systems have been identified.

Examples of new quality factors and criteria based on the analyses in previous sections are provided in 3.1 and 3.2 of Volume I. The quality factors methodology and emphasis is justified for distributed systems. Some comments on qualitative and quantitative relationships among quality factors discussed in 2.5.2 and 2.5.3 remain to be expanded as research continues. An example of distributed system design strategy impact on quality factors, the S-1 family of embedded microcircuit multiprocessors, is presented in 6.1.9. A summary of the new concept of the Distributed System Life Cycle is included in section 6.4.

8.1 DISTRIBUTED SYSTEM HARDWARE ARCHITECTURE IMPACT ON SYSTEM QUALITY

In a distributed system, the hardware architecture has direct and indirect effect on system quality. This is discussed from a number of viewpoints:

* Efficiency - this involves a broader range of issues for distributed than for uniprocessor systems

  Optimum performance - not easily determined

  Performance measurement - systems quality from users' standpoint depends on network aspects of hardware architecture. ARPANET is a prime example

  The theoretical mathematics of Ford and Fulkerson (F&F) - this underlies much of the analysis of network topology as it relates to data flow

* Virtuality - paradoxically, the impact of hardware architecture on system quality may be greatest if the architecture is kept hidden from the user, who is given a deliberately designed illusion as to the system structure.

8.1.1 Efficiency

Efficiency is a measure of actual performance compared to a measure of optimum performance. Various factors inhibit attainment of the optimum. For a centralized system, the allocation of computing resources required to execute a program, the size of the code, and the time needed to run the program are all contributors to efficiency measurement. One can run several programs on the same centralized computer, with essentially the same allocation of resources, and thus compare the efficiency of the programs as a measure of software alone. With a distributed system, the allocation of resources may be expected to change with time in a nondeterministic event-driven manner. Therefore, for the distributed system, software efficiency is not measured against a static background. Rather, efficiency on a distributed system is measured by a quality factor of software and a quality factor of hardware and firmware and communications servicing. Efficiency for a distributed system may also need to be measured over a range of services that can be provided by interprocess communication. Efficiency should be determined for point-to-point terminal service, for broadcast messaging, for distributed control, for packet transmission, for tactical message system priority handling, for file transfer management, for process management, for system I/O, for protocol overhead, for encryption algorithms/processors, for network control, and so on.

It is difficult to compare system performance to optimum performance for two reasons. First of all, it is difficult to accurately monitor actual system performance. Secondly, the optimum strategy for providing distributed processing is rarely known, and the structure of one distributed system is so different from another that inter-system comparison depends on unverified assumptions.

The development of distributed systems has broadened the range of resources available to computer users. Therefore the range of functions to be evaluated is also broadened. The system quality as perceived by the user may be strongly influenced by communications network behavior. To measure system quality, then, requires analysis of software

quality in the context of network performance measurements. Some work has been done in this area for the ARPANET, for protocols, for directory structures, and for network flow (traffic) dynamics.

The performance of the ARPANET is well-established. This is due, in part, to measurement facilities built into IMP software, and to programs for network performance measurement and modeling. For example, throughput for file transport in ARPANET has been analyzed and compared to on-line measurements (KLE 75). General surveys also exist which relate the interactions of protocols and network performance. (SUN) Directory structure and location affects the efficiency of network information management (CHU). The ARPA Network Measurements Center has performed extensive analyses of similar problems (Wood, ... 1975) and the National Bureau of Standards models the dynamics of network traffic in distributed systems (ROS). Much of the theoretical mathematics underlying the analysis of data flow, routing strategy, file transportation, and network topology may be traced to the work of Ford and Fulkerson (F & F)

8.1.2 Virtuality

A proposed criterion for usability, particularly important to distributed systems, is VIRTUALITY. Virtuality is a measure of the extent to which the system appears to the user as it is intended to appear to the user. The user (or users) of a system is not expected or intended to see the system's logical, topological, or physical structure. Instead, an abstract "virtual" system is designed. The "real" system supports, emulates, and embodies the designed appearance of the virtual system.

Theodor H. Nelson, (NEL), explains the relationship between virtuality (which he has emphasized in several applications) and other criteria of usability, such as conceptual simplicity, machine independence, and network file accessibility. "Our approach to a computer design we call 'the design of virtuality.' By virtuality we mean the seeming of an object or system, its conceptual structure, its atmospherics and its feel.... What counts is effects, not techniques.... The design of an interactive computer environment, similarly, should not be based on particular hardware, or a particular display device, or a programming technique.... the systems analysis for an interactive system should deal with the mental space of the user's experience. There is one other key constraint in

system design: conceptual simplicity. If any but highly trained people are to use a system, it must be extremely simple.... easy and focussed control by the user on what he means to do, is not merely compatible with simplicity, it requires it.

T.H. Nelson applies this approach to distributed database design by relating virtuality to portability and to hiding absolute addresses from the user. "All of storage near and far becomes a united whole—what is now called a distributed data base. Actual locations are essentially invisible to the user; or, in that traditional phrase, 'you don't care where it's stored.' The documents and their links unite into what is essentially a swirling complex of equi-accessible unity." Other analyses by Nelson relate virtuality to the problems of file backup and update, extensibility, integrity, and generality.

J. Codd, inventor of relational databases, explains virtuality in terms of "presentation structure." (COD)

> "That is presentation structure, which should be distinguished very strongly from data base structure, just as input record structure should also be. Input schemas for your input structures should probably be different from your data base structures. By making the data base as simple as possible and as powerful as possible, you can factor out that input complexity from the data base manipulation and factor out the output complexity also. You have things called report generators for the latter".
>
> "Obviously, in formulating queries and other transactions, people have to deal with a conceptual view of the data that's stored. And that's what I was really referring to when I talked about the data base engine – the logical data base engine. I wasn't referring to the physical manipulation, the physical structures".

Virtuality is of neglected significance in uniprocessor systems. Virtuality cannot be ignored in distributed systems. Reasons for the increased significance of virtuality for distributed systems include:

* The identification of clear-cut interfaces
* Improvement of portability
* Need for uniform data access despite nonuniform memory

* Capability to present a different virtual system to each user
* Capability to make non-modular system appear modular
* Freeing the user's attention from segmentation/distribution/concurrency
* Enhanced interoperability through emulating other system
* Hiding the effect of node/link failure from user
* Reduction in user training necessary
* Security based on hiding aspects of real system

See also (HAL).

The virtual system may itself be distributed, or may be tailored to appear unified and centralized. The virtual system may be designed around analogies, such as:

distributed system	----	human body
hardware component	----	limb or organ
communication link	----	nerve
input	----	sensory data
output	----	action
bus	----	spinal cord
maintainability	----	health
interoperability	----	empathy with others
response time	----	reflex speed
distributed system	----	human organization
hardware component	----	employee
communication link	----	memo,phone,meeting
subnetwork	----	department in organization
protocol	----	company policy
operating system	----	Management

(example suggested by Hewitt & Kornfeld): "The Scientific Community Metaphor" in IEEE Transactions on Systems, Man & Cybernetics, Jan 1981

distributed system	----	scientific community
hardware component	----	scientist or lab assistant
communication link	----	scientific publication

input	----	scientific data
output	----	solution to problem
computing facility	----	laboratory
correctness	----	validity
system constraints	----	natural laws

The emphasis on designing a virtual system is on the interface presented to the user. If we visualize the user in front of a CRT, and the system in back of the CRT, we may refer to this as FRONT-TO-BACK programming (to use Theodor Nelson's term, as distinguished from top-down or bottom-up). Virtuality also measures the subjective component of the user interface. In the special case of flight training simulators, the "feel" of the system has long been recognized as crucial to usability. This is evaluated by expert pilots (superusers). This goes beyond Human Engineering, which concentrates on one display/sensory modality at a time, or on total bits per second. "Feel", and therefore virtuality involves gestalt perception -- with an emphasis on right-brain holistic activity.

8.1.3 Correctness

For distributed systems, the correctness of a program is essentially the same as correctness in a centralized system. The issue of correctness is quite different for data, though, especially when data exists in multiple copies which must all be updated identically. Consistency is also difficult to maintain when the system has gone through error-recovery or crash-recovery procedures, in which some data manipulation transactions were interrupted. It is also more difficult to trace the various versions of a database which is being processed in a distributed manner by a number of different users. Simply to verify that each user has the same data becomes a time-consuming aspect of system overhead.

8.1.4 Reliability

System reliability is a more complex concept than software reliability. System reliability depends on the reliability of hardware components, the reliability of software, the reliability of communication links, and the system structure and environment. Hardware reliability and software reliability are quite different. Hardware reliability is a

well-understood aspect of system engineering, with a metric for Mean Time Between Failures, and a model of system response to the wear and tear of physical components. Software reliability is not related to physical degradation over time, but rather to the match between performance and requirements. For distributed systems, we consider the evaluation of software reliability in the context of hardware and communications reliability, and we also consider the tradeoffs between hardware and software functions.

8.2 DISTRIBUTED SYSTEM HARDWARE IMPACT ON SOFTWARE QUALITY

Distributed system hardware impacts software quality in a number of ways discussed below.

* Adaptation - a set of concerns arise in the comparison of different architectures for reconfigurable busses

* Performance - dynamic hardware architectures can be evaluated in terms of delay times and technological constraints

* Reliability - software simulation studies give new insights into the reliability of different multiprocessor architectures.

(FAR) compares two systems: minimal delay bus and minimal complexity bus. Each dynamic architecture is evaluated in terms of ADAPTATION and PERFORMANCE.

8.2.1 Adaptation

ADAPTATION involves:

* adaptation to program streams: gives the percentage of the resource used to compute a given set of concurrent programs.

* adaptation to program structures: examines executional speed-up arising from the program being computed by a particular instruction set.

* **array adaptation:** shows the percentage of redundant equipment created during array (vector) computations.

* **bit size adaptation:** measures the delay in the execution of the entire task (program) because of selection of a permanent computer size for executing all of its instructions.

8.2.2 Performance

PERFORMANCE involves:

* the time for architectural reconfiguration: (1) the time to search, fetch, and write all the necessary control codes to those modules of the resource that are formed into new computers in the next architectural state, (2) the time to switch the reconfigurable busses that must either separate computers of the next architectural state or provide correct information exchanges between them. Since each time that an algorithm requires an architectural reconfiguration, the computers affected stop, this time must be added to the time for algorithm computation. Hence reconfiguration time should be minimized so as not to offset the advantage gained by dynamic architecture.

* delays introduced into signal propagation by reconfigurable busses: (1) Instruction path, (2) data exchange path, (3) end-around carry path, (4) paths for equality signals.

* time of between-computer communication: (1) processor-memory, (2) processor-processor, (3) memory-memory.

* modular expansion of existing systems with new hardware components: (1) technological constraints on number of new components that can be attached. For instance, the current restriction on pin count per LSI module may restrict the number of new modules addable to an existing system, (2) expansion of an existing system with new resources may lead to a serious increase in delays introduced by reconfiguration logic and/or partial or

complete disruption of the flexibility of interconnections among modules. For any given system, there might be a limiting complexity such that any expansion fails to improve throughput, (3) modular expansions must be accompanied by the ability of the diagnostic network to maintain the initial diagnosability of the system. Otherwise, some faulty modules may remain undetected and erode the system's reliability. Initial diagnosability tends to decline when new hardware is added.

* cost of realization: (1) the number of different module types used in assembling the system, (2) the overall percentage of circuits mapped onto LSI modules in logical portion of the system.

8.2.3 Reliability

(JOO) compares the reliability of different architectures for interconnecting redundant microprocessors. Component failure data from Mil Handbook 217-B is combined with combinatorial modeling of two different types of failures. Multimicroprocessor architectures are compared on the basis of equivalent numbers of processors and memory modules, on the basis of equivalent performance, and on the basis of actual existing configurations. This indicates the type of reliability estimation necessary for distributed system evaluation, which is more sophisticated than reliability methods for uniprocessor systems. Several interesting conclusions were drawn, which should be considered in a more general context within this study:

RELIABILITY ir /olves:

* Reliability prediction can be substantially enhanced by more accurate modeling of systems and failure modes.

* If there are a large number of modules of one type, this limits the reliability attainable by making less common modules more redundant.

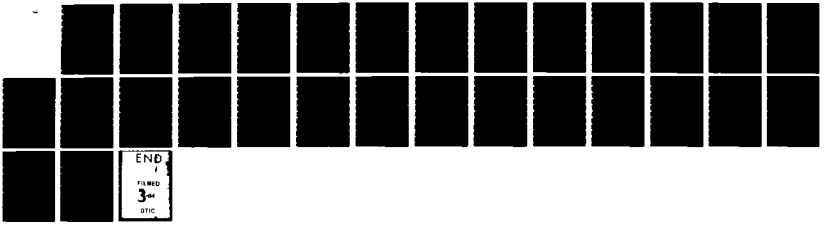
* Increasing redundancy increases system reliability only up to a point of diminishing returns, where further redundancy has only marginal effect.

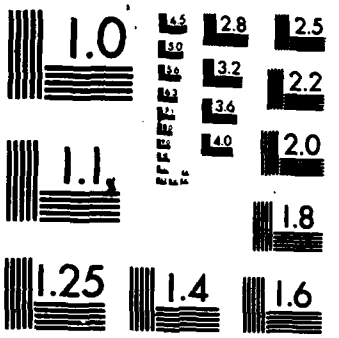
* The impact of high-reliability components (extra screening, burn-in) on system reliability is considerable.

* The amount of resources required can change the optimal system configuration.

APPENDIX A
BIBLIOGRAPHY

AD-A137 957 SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS 3/3
VOLUME 3 DISTRIBUTED (U) BOEING AEROSPACE CO SEATTLE
WA T P BOWEN ET AL. JUL 83 RADC-TR-83-175-VOL-3
UNCLASSIFIED F30602-80-C-0330 F/G 9/2 NL





(ALF), Alford, M., "Requirements for Distributed Data Processing Design", DCS, p. 1-14.

(BAK), Baker, S.L., "JOSS: An Introduction to a Helpful Assistant", Rand Memorandum RM-5058-PR, 1966.

(BAR), (Baran, P., "On Distributed Communications Networks", IEEE Trans. Comm. Sys., March 1964, p.1-9) This paper evolved from a series of RAND Corporation Memoranda.

(BEN), Jon L. Bently and H.T. Kung, "A Tree Machine for Searching Problems", Proc. Int'l. Conf. Parallel Processing 1979, pp. 257-266

(BER), (Bernstein, P.A. and Goodman,N. "Approaches to Concurrency Control in Distributed Database Systems", AFIPS Conf.Proc., 1979 NCC, Vol.48, p.813-20)

(BER), Berra, P. Bruce and Oliver, Ellen, "The Role of Associated Array Processors in Data Base Machine Architecture", Computer, Vol. 12, 3, March 1979.

(BOG), Boggs, R. David and Metcalfe, M. Robert, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM 19, 7, July 1976, 395.

(BRO), Sally Browning, "The Tree Machine: A Highly Concurrent Programming Environment", Ph.D. Thesis, Caltech, Jan. 1980

(CHU), Chu, W.W., "Performance of File Directory Systems for Data Bases in Star and Distributed Networks", Proc. AFIPS NCC 45, 1976.

(CIP) (Cipriano, F.L. and Cohen, A., "C3I Covenants – An Architectural Tool for the 80s", SIGNAL, May/June 1981, p.115-123)

(COD 62), Codd, E.F., "Multiprogramming" in "Advances in Computers", Vol.3, edited by F.L. Alt and M. Rubinoff, New York, Academic Press, 1962, pp. 54-58

(COD 70), Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Comm. ACM, Vol. 13, No. 6, June 1970, p. 377-387.

(COF), Coffman, E.G. Jr., Computer and Job-Shop Scheduling Theory, John Wiley and

Sons, 1976.

(CUR), Curtain, W.A., "Multiple Computer Systems" in "Advances in Computers", Vol.4, edited by F.L. Alt and M. Rubinoff, New York, Academic Press, 1963, pp. 254-303

(DACS), The DACS Glossary, "A Bibliography of Software Engineering Terms", Oct. 1979, Data & Analysis Center for Software, by Shirley A. Gloss-Soler, IIT Research Institute, under contract to RADC.

DCS = Proc. 1st International Conference on Distributed Computing Systems, Huntsville Alabama, 1-5 Oct. 1979, IEEE 79CH1445-6 C.

(DEV), Devy, M. and Diaz, M., "Multilevel Specification and Validation of the Control in Communication Systems", DCS, p. 43-40.

(DIC) (Dickenson, Lt. Gen. H., "Improving C3 Systems and Requirements", SIGNAL, May/June 1981, p.67-76).

(DIN) (Dineen, G.P., "C3I - The Essential Element", NATIONAL DEFENSE, April 1980)

(ELL), (Ellis,C.A., "Consistency and Correctness of Duplicate Database Systems", Proc.6th ACM SIGOPS, Vol.2, No.5, Nov.1977, p.67-84)

(ENS), Enslow, Philip H., "What is a 'Distributed' Data Processing System", Computer, Jan. 1978, p. 13-21.

(FAY), Fayole, G. and Robin, N., "Optimal Queueing Policies in Multiple Processor Computers", Modeling and Performance Evaluation of Computer Systems, Berliner and Gelenbe Ed., North Holland Publishing Co., Amsterdam, 1976.

(FLY), Flynn, M.J. and Hennessy, J.L., "Parallelism and Representation Problems in Distributed Systems", DCS, p. 124-130.

(F&F), Ford, L.R. Jr. and Fulkerson, D.R., Flows in Networks, Princeton University Press, 1962.

(GON 78), Gonzalez. M.J., "Quantitative Evaluation of Distributed Computer Systems", Proc. of 2nd Rocky Mountain Symp. on Microcomputers: Systems, Software, Architecture, IEEE 1978, p 125-30.

(GON 79), Gonzalez, M.J. and Jordan, B.W., "A Framework for the Quantitative Evaluation of Distributed Computer Systems", DCS, p. 156-165.

(GRA), (Gray,J.N., "Notes on Database Operating Systems", in Lecture Notes in Computer Science, ed. Bayer,R. et.al., Springer-Verlag, 1978, p.394-481)

(GUI), Guibas, L.J., Kung, H.T., Thompson, C.D., "Direct VLSI Implementation of Combinatorial Algorithms" Cal. Tech. Conf. VLSI, Pasadena, CA, Jan. 1979

(HAL), Hall, D.E., Scherrer, D.K. and Sventek, J.S., "A Virtual Operating System", Comm. ACM 23, 9, Sept. 1980, 495-502.

(HEV), Hevner, Alan R. and Yao, S. Bing, "Query Processing in Distributed Database Systems", IEEE Trans. Softwaree Eng., May 1979, SE-5, 3, p. 177-187.

(HIL), Hiltz, S.R. and Turoff, M., The Network Nation, Addison Wesley, 1978, p. 348-349.

(HSI), Hsia, P., "A Configurable Distributed Computing System", DCS, p. 172-176.

(IRA), Irani, K.B. and Khabbaz, N.G., "A Model for Combined Communication Network Design and File Allocation for Distributed Databases", DCS, p. 15-21.

(JOO), Joobbani, R. and Siewiorek, D.P., "Reliability Modeling of Multiprocessor Architectures"", DCS, 384-398.

(KAR), Kartashev, S.P. and Kartashev, S.I., "Performance of Reconfigurable Busses for Dynamic Architectures", DCS, p. 261-273.

(KEI) (Keith, Lt. Gen. Donald R., "Distributed C3I — A Force Multiplier for the 90s", SIGNAL, Sep. 1981, p.11)

(KLE 72), Kleinrock, L. Stochastic Message Flow and Delay, Dover, New York, 1972.

(KLE 75), Kleinrock, L. and Opderbeck, H., "Throughput in the ARPANET - Protocols and Measurements", Proc. Fourth Data Comm. Symp., Quebec, October 1975.

(KUN), Kung, H.T., Leiserson C.E., "Systolic Arrays (for VLSI) Tech. Report CS-7?-103, Carnegie-Mellon U., Apr. 1974

(LAM), (Lampson,B.W., "Crash Recovery in a Distributed Data Storage System", Xerox Report, June 1979, later in Comm.ACM)

(LEI), Leiner, A.L., et.al., "PILOT, the NBS multicomputer system", Proc. Eastern Joint Computer Conf., Philadelphia, 1958, pp. 71-75

(LEL 77), LeLann, Gerard, "Distributed Systems - Towards a Formal Approach", 1977 IFIP Congress Proceedings, p. 155-160.

(LEL 79), LeLann, Gerard, "An Analysis of Different Approaches to Distributed Computing", DCS, p. 222-231.

(LEL 81), from page "4.3.1.7".

(LES 79), Lesser, V.R. and Erman, L.D., "An Experiment in Distributed Interpretation", DCS, p. 553-571.

(LIP), Lipovski, G.J. and Malek, M., "A Theory for Multicomputer Interconnection Networks", IEEE Trans. Computers, 1982

(MAH 75), Mahmoud, S.A., "Resource Allocation and File Access Control in Distributed Information Networks", PhD. Dissertation, Systems Engineering Dept., Carlton U., Ottawa, Canada, 1975.

(MAH 76), Mahmoud, Samy and Riordon, J.S., "Optimal Allocation of Resources in Distributed Information Networks", ACM Transactions on Data Base Systems, Vol. 1, No. 1, March 1976, 66-78.

(NEL), Nelson, Theodor H., "Replacing the Printed Word", Information processing 80, Proceedings of the IFIP Congress 80, North-Holland Publishing Co.

(PAL), Palmer, D.F., "Distributed Computing System Design at the Subsystem/Network Level", DCS, p. 22-30.

(POW), Powell, D.R., "RHEA: A Damage and Fault Tolerant Digital Communication Support System for Distributed Avionic Processing", DCS, p. 359-373.

(RDST), Research Directions in Software Technology, Peter Wegner ed., MIT Press, 1979.

(REE), (Reed,D.P., "Implementing Atomic Actions on Decentralized Data", Proc. 8th ACM SIGOPS Symp., Nov.1979, p.66-74)

(ROS), (Rosenkrantz,D.J. "System Level Concurrency Control for Distributed Database Systems", ACM Trans. Database Systems, Vol.3, No.2, June 1978, p.178-98)

(ROS), Rosenthal, R., Rippy, D.E. and Wood, H.M., "The Netword Measurement Machine — A Data Collection Device for Measuring the Performance and Utilization of Computer Networks", NBS Tech Note 912, April 1976.

(SMI 79), Smith, R.G., "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver", DCS, p. 185-192.

(SNY 81), Snyder, L., "Overview of the CHIP Computer", VLSI 81, John Gray (ed.), Academic Press, 1981, pp. 240-9

(SNY 82), Snyder, L., "Introduction to the Configurable, Highly Parallel Computer", IEEE Computer, Jan. 1982

(STA), (Stansberry, J.W., "Defense Industrial Base Issues", SIGNAL, July, 1981, p.16)

(STO), Stonebraker, Michael, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Trans. Software Eng., SE-5, 3, May 1979, p. 188-194.

(SUN), Sunshine, C.A., "Factors in Interprocess Communications Protocol Efficiency for Computer Networks", Proc. AFIPS NCC 45, 1976.

(THO), Thompson, C.D., "A Complexity Theory for VLSI", Ph.D. Thesis, Carnegie-Mellon

University, Pittsburgh, PA.,1980

(VAL), Valiant, L.G., "University Considerations in VLSI Circuits", IEEE Trans. Computers, Vol. C-30, No. 2, Feb. 1981

(VAU), Vaughn and Anastas, "Microprocessor based transition machines", COMPCON 79 Proceedings, IEEE 1979, p. 327-33.

(WEN), Weiner, N., Cybernetics, 1947.

(WIL), Wilkov, R. et.al., "Exact Calculations of Computer Network Reliability", Proc. AFIPS, 1972 FJCC, Vol. 41, Pt. 1, AFIPS Press, Montvale, N.J., p. 49-54.

(WOO), Wood, 1975.

(YAU), Yau, S.S. and Yang, C.C., "An Approach to Distributed Computing System Software Design", DCS, p. 31-42.

(YEM), Yemini, Y. and Cohen, D., "Some Issues in Distributed Process Communication", DCS, p. 100-203.

(ZIM), Zimmerman, H., "OSI Reference Model - the ISO Model of Architecture for Open Systems Interconnection", IEEE Trans. Comm., COM-28, 4, April 1980, p. 424ff.

APPENDIX B
GLOSSARY OF KEY TERMS

DACS references shown as (DAN ###)

* ACCESSIBILITY: (1) Code posesses the characteristic Accessibility to the extent that it facilitates selective use of its parts. (Accessibility is necessary for Efficiency, Testability, and Human Engineering). D.K. Lloyd, M. Lipow, "Reliability Management, Methods, and Mathematics" published by the authors, Redondo Beach, CA 1977 (2) Ease of access to a system. Accessibility is a reflection of the probability of intentional and accidental breaking into a system. Accessibility is nearly synonymous with Security. T. Gilb, "Software Metrics", Winthrop Publishers Inc., 1977 * We clearly use the 1st of these definitions in our study, and should note that.

* ACCURACY: (1) Those attributes of the software which provide the required precision in calculations and outputs (a criterion in this contract). (2) A measure of the degree of freedom from error; the degree of exactness possessed by an approximation or measurement. In this context accuracy is a measure of "design adequacy" rather than "system reliability." Errors which influence this measure are due to the data and the logic for processing that data. Additional error due to hardware failure or logic "bugs" is handled and measured separately under reliability concepts (DAN 781)

* ADAPTABILITY: Adaptability is a measure of the ease with which a program can be altered to fit differing user images and system constraints. (DAN 758) (2) Code posesses the characteristic adaptability to the extent that it can be easily altered to fit differing user images and system contraints. (DAN 239)

* ANOMALY MANAGEMENT: Those attributes of the software which provide for continuity of operations and recovery from non-nominal conditions (a criterion in this research report).

* ATOMICITY: Each software transaction is viewed by users and system as indivisible. (A metric in the criterion Autonomy, studied and dropped)

* ATTACK PROBABILITY: The probability of an attack of a given type on a given system during a specified time interval. (DAN 781)

* ATTACK REPULSION PROBABILITY: Synonymous with Security Probability.

* AUGMENTABILITY: (1) Those attributes of the software which provide for expansion of capability for functions and data (a criterion in this contract). (2) Code possesses the characteristic augmentability to the extent that it can easily accomodate expansion in component computational functions or data storage requirements. This is a necessary characteristic for modifiability. (DAN 239)

* AUTOMATABILITY: "Automatable" means that the human operator not only can control the system completely manually but also can define portions of the system operations as procedures to be performed automatically by the system. (DAN 346)

* AUTONOMY: Those attributes of the software which determine its dependency on interfaces (a criterion in this research report).

* AVAILABILITY: (1) Portion of the total operational time that the system performs or supports critical functions (system quality factor in this contract). (2) Availability is the probability that computer software is "up" or capable of functioning in accordance with requirements at any time. This probability is often measured as the ratio of "up" time to total need time... The computer software may be classified as not available if it is blocked by another user, or if it contains errors and is being corrected. (SET) (3) The ratio of system up-time to the total operating time. (DAN 232) (4) The probability that a system is operating satisfactorily at any point in time, when used under stated conditions. (DAN 781) (5) The probability that a system, subsystem, or component will be functionally ready or operable at some specified point in time. (NASA)

* BEBUGABILITY: "Software Metrics", T.Gilb, Winthrop Publishers Inc., Cambridge, MA, 1977, has a chapter on Maintainability Measurement which emphasizes the technique of "Bebugging". This technique of inserting deliberate errors in order to evaluate debugging effectiveness can itself be evaluated. A highly bebugable system, for example, might allow automation of the bebugging measurement tool. Gilb lists some jobs which could be done automatically by a

bebugging program: (1) bug insertion (2) recording the original state of the source program * restoring the program to its original (pre-bebugged) state (3) determining the number of bugs to insert (based on program size and the accuracy required in the measure) (4) Determining which type of bug to insert (based on statistics of the type of bug found, which could be gathered automatically) (5) evaluation of whether artificial bugs had been corrected (6) statistical reporting on bug-finding rate and type of bug (7) estimation of confidence in the program reliability (8) estimation of mean time to failure of the program (9) estimation of the number of test cases which must be run The more such features, the higher the "bebugability." The DACS Glossary, compiled for RADC by S.A.Gloss-Soler, Oct.1979, quotes these definitions:

* BEBUGGING: Synonymous with Bug Seeding/Tagging * Bug Seeding/Tagging: The process of adding bugs (or errors) to those already assumed to be in a program with the purpose of obtaining an estimate for the number of natural bugs remaining in the program. It is also assumed that the ratio of the number of undiscovered seeded bugs to the total number of bugs seeded can serve as an indication of the degree of "DEBUGGEDNESS" or reliability of the program. M.L.Shooman & H.Ruston, "Software Modeling Studies Summary of Technical Progress", RADC-TR-78-4, Vol.i, Jan. 1978

* BUILT-IN FLEXIBILITY: Built-in flexability is the ability of a system to immediately handle different logical situations. Built-in flexability increases system complexity proportionately. In a well-designed system the initial measure of built-in flexability will be almost equal to the complexity measure. (DAN 781)

* CLARITY: (1) Those attributes of the software which provide non-ambiguous descriptions of functions and implementations (criterion proposed and dropped). (2) Code possesses the characteristic clarity to the extent that it is concise, straightforward (lack of tricky, obscure code), understandable, has clear control structure, uniform style, self-contained with respect to documentation, makes appropriate use of macros and of change levels. (DAN 748)

* COHESION: A relative measure of the strength of relationships among the internal components of a module insofar as they contribute to the variation in

assumptions made by the outside program concerning the role the module plays in the program. Invariant assumptions about a module indicate high strength. (DAN 1153)

* COMMONALITY: Those attributes of the software which provide for the use of interface standards for protocols, routines, and data representations (a criterion in this research report).

* COMMUNICATIVENESS: (1) Those attributes of the software which provide useful inputs and outputs which can be assimilated (a criterion in this contract). (2) Code possesses the characteristic communicativeness to the extent that it facilitates the specification of inputs and provides outputs whose form and content are easy to assimilate and useful. Communicativeness is needed for testability and human engineering. (DAN 239)

* COMPATIBILITY: (1) Those attributes of the software which provide interface protocols and routines that are appropriate to the interface equipment and features (criterion proposed and dropped). (2) Compatibility is the measure of portability that can be expected of systems when they are moved from one given environment to another. By way of contrast, portability is a characteristic of the system; compatibility is a relationship between two environments. (DAN 787)

* COMPLETENESS: Those attributes of the software which provide full implementation of the functions required (a criterion in this research report).

* COMPLEXITY: A measure of the difficulty of implementing a component, independent of the implementor's experience. Easy (or simple) means that any good programmer can write down the correct code with little thought. Hard (or complex) means that much thought is involved in the design. (Compare this with "precise"; e.g. easy and imprecise may mean a vague specification, but once the approach is decided upon, the code is easy to write.) (SEL) (2) A term which can refer to any number of aspects of a computer program: the topology of its control logic, the intricacy of its data structures, the number of computations to reach an answer, the size of the program, the understandability of the documentation, the demonstration effort required to judge correctness, the ease with

which repairs or changes can be effected, etc. (DAN 288) (3) Characteristics of a program which affect complexity include: instruction mix, data reference, structure/control flow, interaction/interconnection. (4) The degree of interactions and dependencies among elements of a computer program. (NASA)

* COMPLIANCE: Those attributes of the software which promote implementations that conform to the requirements (criterion proposed and dropped).

* COMPREHENSIBILITY: Those attributes of the software which enhance understanding of the operation of the software (criterion proposed and dropped).

* COMPRESSION RATIO: The measure of the degree of compression of data as expressed by the fraction: length of original data / length of compressed data. (DAN 781)

* CONCISENESS: (1) Those attributes of the software which provide for implementation of a function with a minimum amount of code (a criterion in this contract). (2) Code possesses the characteristic conciseness to the extent that excessive information is not present. This implies that programs are not excessively fragmented into modules, overlays, functions, and subroutines, nor that the same sequence of code is repeated in numerous places, rather than defining a subroutine or macro, etc. (DAN 239)

* CONFIDENCE LEVEL: Percent probability that a given number is correct. 100% means that the number is known to be correct with absolute certainty. 0% means that the number must be incorrect. (An output of some reliability and error models). (SEL) (2) The probability that a given statement concerning a set of random variables or a segment of a random process will be upheld, if tested. (DAN 1153)

* CONNECTIVITY: The set of assumptions the rest of a program makes about a module (or other program segment). Modules have connections in control, in data, and in services (functions) performed. Connectivity increases with the number, type, and variability of such assumptions. (DAN 1153)

* CONSISTENCY: (1) Those attributes of the software which provide for uniform design and implementation techniques and notation (a criterion used in this contract). (2) The strict and uniform adherence to prescribed symbols, notation, terminology and conventions which tends to foster a quality software product (NASA). (3) A program quality which assures that the results of executing a program are repeatable in a practical sense, in spite of any logical errors which may be present in the program. (DAN 1153)

* CORRECTNESS: (1) Extent to which the software satisfies its specifications and fulfills the user's mission objectives (definition used in this contract). (2) Agreement between a program's total response and the stated response in the functional specification (functional correctness), and/or between the program as coded and the programming specification (algorithmic correctness). (DAN 1153)

* COUPLING: The concept of coupling has been used to describe the interfaces between the program modules which consitute a process. In a distributed system, the concept of coupling is used to describe the interactions of processes and processors. Distributed systems are weakly or strongly coupled according to whether there are few or many interactions among the processing elements. This is a functionally definition of coupling.

  A process segment A is minimally coupled to the system if the effect of the execution of A depends only of the state of the system when the execution of A begins. The funcitional effect of A can be described in terms of the initial state of the structures A accesses (inputs), and the state of these structures after A executes (outputs). The activities of other processors during the execution of A do not affect the execution or functional effect of A.

  The degree of coupling between a processor and the system is inversely proportional to the size of the minimally coupled program segments running on the processor. For example, if each task which runs on a processor is minimally coupled to the system then the processor is weakly coupled to the system. Conversely, the effect of executing an instruction on the processor may depend on events which have occured since the last instruction was executed, then the processor is strongly coupled to the system.

An example of a weakly coupled system is a network of autonomous processors which share information only by file transfers. An extreme example of a closely coupled system would be a system in which the code executing in processor B could be dynamically modified by processor A.

* DETERMINISM: The property of a transformation process that the same outputs are always produced for a given set of inputs. (ANSI-X3H1)

* DIAMETER: Maximum over all nodes of minimum distance between each pair of nodes. (A metric in the criterion of Distributedness, studied and dropped).

* DIFFICULTY: A Measure of how "hard" it is to accomplish a given project. Difficulty expresses the time rate of change of manpower utilization. Difficulty varies directly with the size of the project and inversely as the length of development. D=K/(DT)**2, where D is difficulty, K is size, and (DT)**2 is the square of the development time.

* DISTINCTNESS: Software distinctness is a measure of the failure point independance of a piece of software which is performing the same function as another piece of software. It is analogous to hardware distinctness which is utilized in dual hardware systems, which perform the same tasks and rarely fail at the same instant.

* DISTRIBUTED: See section 2.1 of volume III.

* DISTRIBUTEDNESS: Those attributes of the software which determine the degree to which software functions are geographically or logically separated within the system (a criterion in this research report).

* EFFECTIVENESS: (1) Those attributes of the software which provide for minimum utilization of resources (processing time, storage, operator time) in performing functions (a criterion in this contract). (2) System readiness and design adequacy. Effectiveness is expressed as the probability that the system can successfully meet an operational demand within a given time when operated under specified conditions. (DAN 781)

* EFFICIENCY: (1) Degree of utilization of resources (processing time, storage, communication time) in performing functions (definition used in this contract). (2) Code possesses the characteristic efficiency to the extent that it fulfills its purpose without waste of resources. This implies that the choice of source code constructions are made in order to produce the minimum number of words of object code, or that where alternate algorithms are available, those taking the least time are chosen; or that information-packing density in core is high, etc. Of course, many of the ways of coding efficiently are not necessarily efficient in the sense of being cost-effective, since the process whose end is to increase efficiency is optimization. Efficiency is the ratio of useful work performed to the total energy expended. It can also be expressed as the effectiveness/cost ratio. (DAN 781)

* ERROR SEVERITY: A description (numerical or verbal) of an error's effect upon the results of executing a program in comparison to the expected results.

* EVOLUTION: Evolution is a designed characteristic of a system development which involves gradual stepwise change. A complex system can be implemented in small steps; each step can have a criterion for successful achievement as well as a "retreat possibility" to a previous successful step in the event of failure. (DAN 781)

* EVOLVABILITY: Evolvability is defined as the extent to which the software performance can be enhanced by the incorporation of new technology (e.g., algorithm, compiler). The criteria associated with the quality factor are: virtuality, generality, modularity, specificity, and simplicity. The quality factor evolvability was proposed and dropped.

* EXPANDABILITY: Effort to increase software capability or performance by enhancing current functions or adding new functions/data (a factor in this research report).

* EXPOSURE: The degree of protection which has been provided for an individual object. (DAN 277)

* EXTENSIBILITY: The extent to which software allows new capabilities to be added and existing capabilities to be easily tailored to user needs.

* FAILURE RATIO: Number of failures per calendar intervals divided by total number of runs. (DAN 226)

* FAIRNESS: A queueing system is called fair if, whenever a process is delayed on any queue, there is a possible future state of the system in which that process is on none of the queues. (DAN 420)

* FAULT TOLERANCE: Use of protective redundancy. A system can be designed to be fault tolerant by incorporating additional components and abnormal algorithms which attempt to insure that occurrences of erroneous states do not result in later system failures -- a quantitative prediction of system reliability. (DAN 236)

* FIDELITY: Fidelity is defined as the accuracy with which a given algorithm is mechanized for a given operating system and hardware system. (DAN 781)

* FLEXIBILITY: (1) Effort to extend the software missions, functions, or data to satisfy other requirements (a factor in this research report). (2) The term flexability is usually used to denote the existence of a range of choices available to a programmer or implementer -- the more choices, the greater the flexability. Flexibility is sometimes referred to as "generality." (DAN 470) (3) Flexibility is useful complexity. (DAN 781)

* GENERALITY: (1) Those attributes of the software which provide breadth to the functions performed with respect to the application (a criterion in this research report). (2) Generality is the degree to which a system is applicable in different environments. (DAN 781)

* HARDWARE RELIABILITY: A measure of the success with which the hardware in a system conforms to some authoritative specification of its behavior. A quantitative assessment. (DAN 236)

* HUMAN ENGINEERING: Code possesses the characteristic human engineering to the extent that it fulfills its purpose without wasting the users' time and energy, or degrading their morale. This characteristic implies accessibility, robustness, and communicativeness. (DAN 239)

* IMPLEMENTATION CORRECTNESS: Correctness between design and programmed hardware. (DAN 322)

* INDEPENDENCE: Those attributes of the software which determine its dependency on the software environment (computing system, operating system, utilities, input/output routines, libraries) (a criterion in this research report).

* INPUT DATA SENSITIVITY: Degree to which performance improvements dictated by a program modification under a certain set of input data are preserved under different sets of input data. (DAN 435)

* INTEGRITY: Extent to which unauthorized access to the software or data can be controlled (a factor in this research report).

* INTEGRITY PROBABILITY: A measure of system survival probability. Integrity probability is a function of security probability, and attack probability. System survival is dependent on the frequency of system attack coupled with the ability of the system to make itself secure from a particular type of attack. (DAN 781)

* INTERCHANGEABILITY: Effort to transfer a system component for use in another operating environment (e.g., configuration). (System quality factor in this construct).

* INTEROPERABILITY: (2) Effort to couple the software of one system to the software of another system (a factor in this research report). (3) The term interoperability means that any user of a local system can potentially also operate any interconnected remote system. (DAN 346)

* INTERPRETATION CORRECTNESS: Correctness between requirements and specifications. (DAN 322)

* LEGIBILITY: Code possesses the characteristic legibility to the extent that its function is easily discerned by reading the code. (Example: complex expressions have mnemonic variable names and parentheses even if unnecessary.) Legibility is necessary for understandability. (DAN 239)

* LOGICAL COMPLEXITY: (1) Logical complexity is a measure of the degree of decision-making logic within a system. (DAN 781) (2) Perhaps synonymous with complexity. (3) The degree of decision logic in a computer program. (NASA)

* MAINTAINABILITY: (1) Average effort to locate and fix a software failure (a factor in this research report). (2) Code possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements or to correct deficiencies. This implies that the code is understandable, testable, and modifiable; e.g. comments are used to locate subroutine calls, and entry points, visual search for locations of branching statements, and their targets is facilitated by special formats, or the program is designed to fit into available resources with plenty of margins to avoid major redesign, etc. (DAN 239) (3) Maintainability is the probability that, when maintenance action is initiated under stated conditions, a failed system will be restored to operable condition within a specified time. (DAN 781)

* MICRORELIABILITY MODEL: A reliability model which measures the reliability of the separate modules of a program before the modules are combined into a software system. (DAN 299)

* MODIFIABILITY: Code possesses the characteristic modifiability to the extent that it facilitates the incorporation of changes, once the nature of the desired change has been determined. Note the higher level of abstractness of this characteristic as compared with augmentability. (2) Modifiability implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained. (DAN 109)

* MODULARITY: (1) Those attributes of the software which provide a structure of highly cohesive modules with optimum coupling (a criterion in this research report). (2) modularity is the fragmentation of a program into convenient

discrete pieces called modules.... The main goal of modularizing a program is to make possible the modification of a single module without affecting the other modules. In the context of software engineering, this is considered as a quality characteristic of programming. The crucial elements are: (a) small (the size of the module cannot be quantified and most programmers prefer to follow their own intuitive approach to modularity), (b) self-containment, (c) independence (meaning a program in which any logical portion can be changed without affecting the rest of the system). Also a modular program should have modules that have only one entry point and one exit point. (SET) (3) Modularity deals with how the structure of an object can make the attainment of some purpose easier. Modularity is purposeful structuring. (DAN 109)

* NON-DETERMINISM Converse of determinism (ANSI-X3H1)

* NONSTOPABILITY: Do nodes run on nonstop processors (multiprocessor redundancy with software control program for graceful degradation)? (A metric in the criterion Reconfigurability, studied and dropped).

* OPEN-ENDED FLEXIBILITY: Synonymous with adaptibility. (DAN 781)

* OPERABILITY: Those attributes of the software which determine operations and procedures concerned with the operation of the software (a criterion in this research report).

* OPERATIONAL RELIABILITY: The reliability of the program-as-it-performs as opposed to the reliability of the program-as-it-is. (DAN 245)

* PARALLELISABILITY: (Kuck's metric) (Number of independent processors /10 $\log_{10}$ number of independent processors) (A metric in the criterion Distributedness, studied and dropped).

* PERFORMANCE The evaluation of non-logical properties (i.e. computer run time, resource utilization) of a software system. Performance is measured in terms of the amount of resources required by a software system to produce a result. (DAN 154) (2) A measure of the capacity of an individual or team to build software capabilities in specialized or generalized contexts. Performance distinguishes between work and effort, as it includes productivity as one component of its measure. However, performance also measures quality of work as measured by other criteria as well, as set forth in a prioritized list of "competing characteristics" early in development. (DAN 1153)

* PORTABILITY: (1) Effort to convert a software for use in another operating environment (hardware configuration, software system environment) (a factor in this research report). (2) Portability is the property of a system which permits it to be mapped from one environment to a different environment. (3) "Portability" designates the fact that for many different machines and operating systems, copies of the product can be delivered with uniform operating characteristics, from the user's point of view, any input which is valid on one supported system is valid on any other supported system, and will produce identical output. (DAN 283) (4) Code possesses the characteristic portability to the extent that it can be operated easily and well on computer configurations other than its current one.... This implies that special langage features, not easily available at other facilities, are not used; or that standard library functions and subroutines are selected for universal applicability, etc. (DAN 239) (5) Portability is the property of a system which allows it to be moved to the new environment with relative ease. (DAN 781)

* PRECISION: Precision in software is a measure of the degree to which errors tend to have the same root cause. Software precision could be considered as the ratio of source bugs to the effects they cause. (DAN 781) (2) A measure of the degree of discrimination with which a quantity can be stated, as opposed to accuracy, which states the degree to which that quantity is free from error. (DAN 1153)

* PRODUCTIVITY: Traditionally, the generally accepted description (if not definition) of programming productivity has been "lines-of-code/man-month" (i.e.

quantity of code produced). (2) Instructions/Staff-year for either total resources expended in the software development cycle or real-time software development. (DAN 459) (3) Productivity is the rate of production of computer software. This rate is normally measured in the quantity of code and documentation produced. The definition of productivity may contain at least three additional elements: (a) a qualitative element concerned with the correctness and efficiency of the software; (b) a qualitative element concerned with the difficulty of the applications being implemented including size and complexity, (c) an element concerned with the cost of producing the software. (SET)

* PROGRAM COMPLEXITY: Program complexity is an indicator of program readability. It is a function of the number of execution paths in the program and the difficulty of determining the path for an arbitrary set of input data. (DAN 262) more definitions in (DAN 314)

* PROGRAM CORRECTNESS: A correct program is one that has been proved to meet its specifications. (DAN 237)

* PROGRAM INSTRUMENTATION: A quantitative assessment of how thoroughly a program is exercised by a set of test cases. (DAN 107)

* PROGRAM UNDERSTANDING: Indexing term. Indicates the degree to which a program is comrehensible and can be understood in function and scope (other than by the programmer who wrote it), and also indicates that the document discusses factors which contribute to program understanding.

* PROGRAMMING LANGUAGE COMPLEXITY: An indicator of the readability or understandability of a programming language. (DAN 237) (2) Measured in Software Science by Halstead metrics.

* QUALITY: The degree to which software conforms to quality criteria. Quality criteria include, but are not limited to, CORRECTNESS, RELIABILITY, VALIDITY, RESILIENCE, USEABILITY, CLARITY, MAINTAINABILITY, MODIFIABILITY, GENERALITY, PORTABILITY, TESTABILITY, EFFICIENCY, ECONOMY, INTEGRITY, DOCUMENTATION, UNDERSTANDABILITY, FLEXIBILITY,

INTEROPERABILITY, MODULARITY, REUSABILITY.

* RECONFIGURABILITY: Those attributes of the software which provide for continuity of system operation when one or more processors, storage units, or communications links fails (a criterion in this research report).

* REDUNDANCY: Redundancy is the ratio of the quantity of a particular resource used in a system to the quantity of the resource actually needed to accomplish the systems task. Redundancy may be a desirable characteristic (error detection, error correction fault-tolerance) or undesirable (capacity in excess of what may ever be needed, duplicate data files for processes which could use the same files, etc.). Redundancy may refer to data, code, or hardware devices.

* RELIABILITY: Probability that the software will perform its logical operations in the specified environment without failure (a factor in this research report).

* RELIABILITY INDEX: The probability that a program or device will perform without failure for a specified period of time or amount of usage. (DAN 1153)

* RELIABILITY TREND: Degree of constancy of the failure rate and/or failure ratio of a software product over time. (DAN 226)

* REPAIRABILITY: Repairability is the probability that a failed system(s) will be restored to operable condition within a specified active repair time when maintenance is done under specified conditions. (DAN 781)

* REPEATABILITY A property required of software tests, that each time they are executed, the results will be the same. (DAN 1201)

* RESILIENCY: Resiliency refers to shared systems. A resilient service: (a) is able to detect and recover from a given maximum number of errors; (b) is reliable to a sufficiently high degree that the user of the resilient service can ignore the possibility of service failure, (c) if the service provides perfect detection and recovery from N errors, the (N+1)st error is not catastrophic. A "best effort" is

made to continue service, and (d) the abuse of the service by a single user should have negligible effect on other users of the service. (source unknown)

* REUSABILITY: (1) Effort to convert a software component for use in another application (a factor in this research report). (2) Reusability refers to the extent to which a program can be used in other applications -- related to the packaging and scope of the functions that a program performs. (DAN 512)

* ROBUSTNESS: Code possesses the characteristic robustness to the extent that it can continue to perform despite some violation of the assumptions in its specification. This implies, for example, that the program will properly handle inputs out of range, or in different format or type than defined, without degrading its performance of functions not dependent on the non-standard inputs. (DAN 239)

* SAFETY: Probability that the system will not cause damage or physical injury. (System quality factor in this contract)

* SELF-DESCRIPTIVENESS: Those attributes of the software which provide explanation of the implementation of a function (a criterion in this research report).

* SIMPLICITY: Those attributes of the software which provide for the definition and implementation of functions in the most non-complex and understandable manner (a criterion in this research report).

* SOFTWARE RELIABILITY: Software reliability is the probability that a given software program will operate without failure for a specified time in a specified environment. (2) Software reliability is defined as the probability that a given software program operates for some time period, without an external software error, on the machine for which it was designed, given that it is used within design limits. (DAN 31) (3) (a) System, user, or macroscopic viewpoint: software reliability is the probability that that the use of the software does not result in failure of a system by more than a tolerable frequency. In short, software reliability is the probability that the software is reliable, utilizing a definition of "reliable software." (JVP: an exceptionally circular definition!) (b) Subsystem, developer, or microscopic viewpoint: software reliability is the probability that

the program, routine, or "module" is fault-free.... see also -Reliable Software. (SET) (4) Code possesses the characteristic software reliability to the extent that it can be expected to perform its intended functions in a satisfactory manner. (DAN 239)

* SOFTWARE SYSTEM DEPENDABILITY: The probability that the application program together with its supervisory program, data base and hardware will perform in its intended environment. The environment will include anomalies and failures, such as: (1) Deficiencies in requirements, (2) Software design errors (incorrect algorithms, word length problems, timing problems, etc.), (3) Software failures, (4) Processor errors, (5) Memory errors, (6) Failures in the communications network, (7) failures in peripheral devices, (8) operator mistakes, (9) power failures, (10) Environmental failures, (11) Gradual erosion of the data base, (12) Hardware saturation (CPU, Memory, I/O Channels) (DAN LD4)

* SPECIFICITY: Those attributes of the software which provide singularity in the definition and implementation of functions (a criterion in this research report).

* STABILITY: Stability is a measure of the lack of perceivable change in a system, at a given level of the system, in spite of some occurrence in the system environment which would normally be expected to cause change. (DAN 781)

* STATIC REDUNDANCY: Duplication of part or all of the components (e.g. processor, main and auxiliary memory, and communications equipment) so that in the case of failure by one unit another can be switched in. (DAN 311)

* STRUCTURAL COMPLEXITY: A measure of the degree of simplicity of relationships between subsystems. (DAN 781) (2) See also complexity, logical complexity. (3) Synonymous with modularity. (4) The degree of coupling among modules of a computer program. (NASA)

* STRUCTUREDNESS: Code possesses the characteristic structuredness to the extent that it possesses a definite pattern of organization of its interdependent parts. This implies that evolution of the program design has proceeded in an

orderly and systematic manner, and that standard control structures have been followed in coding the program, etc. (DAN 239)

* SUPPORTABILITY: Those attributes of the software which provide for ease in creation of new software versions (e.g., use of HOL, version update scheme). (criterion proposed and dropped)

* SYSTEM ACCESSIBILITY: Those attributes of the software which provide for control and audit of access to the software and data (a criterion in this research report).

* SYSTEM RELIABILITY: A measure or indication of the success with which the system provides the service specified. (DAN 236)

* SURVIVABILITY: Probability that the software will continue to perform or support critical functions when a portion of the system is inoperable (a factor in this research report).

* SYSTEM SURVIVAL PROBABILITY: Synonymous with Integrity Probability. (DAN 781)

* TESTABILITY: Code posseses the characteristic testability to the extent that it facilitates the establishment of verification criteria and supports evaluation of its performance. This implies that requirements are matched to specific modules, or diagnostic capabilities are provided, etc. (DAN 239)

* TESTABLE: A software product is testable to the extent that it facilitates the establishment of verification criteria and supports evaluation of its performance. ... Some of the characteristics which increase testability are: (a) functional modularity, which facilitates the matching of requirements to specific modules of a program, (b) capability of providing diagnostics, e.g. through use of conditional assembly to invoke macro generation of code for diagnostic printouts, under user control, (c) auxiliary code is used to evaluate certain invariants (e.g., code is added to calculate the total energy for various states of a constant energy system), (d) comments indicating unacceptable values and recommended default

action are placed at a point where intermediate or required output is defined. (SET)

* TESTEDNESS: Testedness is a dynamic measure which indicates the extent to which a particular piece of software has been tested by particular test cases. The measure is defined in such a way that when measuring the extent to which a logical structure has been exercised or tested; the testedness of a node increases as the number of times it is exercised increases, and decreases as the probability of error or the accessability increases. (DAN 766)

* THROUGHPUT: A measure of the amount of work performed by a computing system over a given period of time, e.g., jobs per day. (2) A measure of computer capacity in terms of execution rate and word size. (NASA)

* TOLERANCE: A system's input data tolerance is a measure of the system's ability to accept different forms of the same information as valid, (i.e. without malfunction or rejection). (DAN 781) (2) Nearly synonymous with robustness.

* TRACEABILITY: Those attributes of the software which provide a thread of origin from the implementation to the requirements with respect to the specified development envelope and operational environment (a criterion in this research report).

* TRAINING: Those attributes of the software which provide transition from current operation or provide initial familiarization (a criterion in this research report).

* TRANSPORTABILITY: (1) Effort to physically relocate the system. (System quality factor in this contract). (2) The capability of a program to operate on various different make and model computers with a minimum of modification required. (DAN 1201)

* UNDERSTANDABLE: A software product is understandable to the extent that its purpose is clear to the inspector.... Many techniques have been proposed to increase understandability. Prominent among these are code structuredness which

simplifies logical flow, local commentary, to explain complex coded instructions, and consistenmtly used mnemonics. In addition, references to readily available and up-to-date documents need to be included in source commentary so that the inspector may comprehend more esoteric contents. Inputs, outputs, and assumptions should be stated in the form of glossaries or prose commentary. In general, a coding standard encompassing format of headers and indentation should be followed for all modules so that information can be found where expected.... See also maintainable, modularity. (SET)

* USABILITY: Effort for training and software operation - familiarization, input preparation, execution, output interpretation (a factor in this research report).

* VALIDITY: Those attributes of the software which constrain implementations to a range of acceptable solutions. (criterion proposed and dropped)

* VERACITY: Veracity is defined as the adequacy with which a given algorithm represents the requirements of the physical world. (DAN 781)

* VIABILITY Viability is defined as the adequacy with which a given algorithm meets timing constraints. (DAN 781)

* VERIFIABILITY: Effort to verify the specified software operation and performance (a factor in this research report).

* VIRTUALITY: Those attributes of the software which present a system that does not require user knowledge of the physical, logical, or topological characteristics (e.g., number of processors/disks, storage locations) (a criterion in this research report).

* VISIBILITY: Those attributes of the software which provide status monitoring of the development and operation (e.g., instrumentation) (a criterion in this research report).

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED

3-84

DTIC