

300 015 0097
RADC-TR-88-168, Vol I (of two)
Final Technical Report
August 1988



AD-A206 039

A DESIGNERS' GUIDE TO RELIABLE DISTRIBUTED SYSTEMS Design and Analysis Methods

Honeywell

Anand R. Tripathi, Jonathan Silverman, William T. Wood, Elaine N. Frankowski,
Pong-Sheng Wang, Shiva Azadegan, Shiv Seth, Rita Wu, Helmut K. Berg

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.



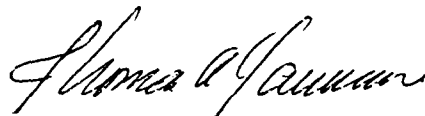
ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

1 89 4 03 093

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

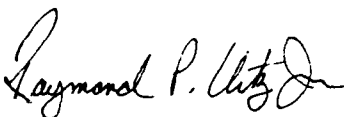
RADC-TR-88-168, Vol I (of two) has been reviewed and is approved for publication.

APPROVED:



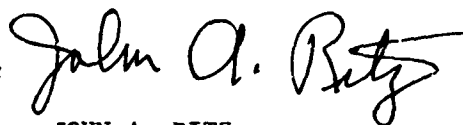
THOMAS F. LAWRENCE
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER:



JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

ADA206039

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-168, Vol I (of two)		
6a. NAME OF PERFORMING ORGANIZATION Honeywell		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)		
6c. ADDRESS (City, State, and ZIP Code) Computer Science Center 10701 Lyndale Ave (South) Bloomington MN 55420			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-82-C-0154		
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 63728F	PROJECT NO. 2530	TASK NO. 01
			WORK UNIT ACCESSION NO. 17		
11. TITLE (Include Security Classification) A DESIGNERS' GUIDE TO RELIABLE DISTRIBUTED SYSTEMS Design and Analysis Methods					
12. PERSONAL AUTHOR(S) Anand R. Tripathi, Jonathan Silverman, William T. Wood, Elaine N. Frankowski, Pong-Sheng Wang, Shiva Azadegan, Shiv Seth, Rita Wu, Helmut K. Berg					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM Sep 82 TO Aug 84		14. DATE OF REPORT (Year, Month, Day) August 1988	
15. PAGE COUNT 366					
16. SUPPLEMENTARY NOTATION Subcontractors: Information Research Associates - Authors: James C. Browne, James Dutton, Vincent Fernandes, Annette Palmer, Raj Kumar Velpuri The University of Texas at Austin - Authors: Donald L. Good, Michael K. Smith (See Reverse)					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Distributed Systems Reliable Systems		
12	07		Performance Evaluation Reliability Evaluation		
			Recovery Mechanisms Atomic Action (See Reverse)		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This report describes an effort to develop a system designers guidebook for designing reliable distributed command and control systems. The guidebook contains a synthesis of reliable system design principles and methods to evaluate distributed system designs for performance, reliability and functional correctness. The approach to developing the system designers guidebook in this effort is example driver. We develop a detailed design of a reliable distributed operating system and evaluate its performance.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence			22b. TELEPHONE (Include Area Code) (315) 330-2158		22c. OFFICE SYMBOL RADC (COTD)

UNCLASSIFIED

Block 16 SUPPLEMENTARY NOTATION (Continued).

Richard M. Cohen, Lawrence Smith, Lawrence Akers, William Bevier, Miren Carranza,
Ann Siebert

Block 18 SUBJECT TERMS (Continued).

Fault-Tolerant Systems	Validation
Verification	Replication
Commit Protocol	Design Methods
Object-Oriented Systems	Formal Specification

UNCLASSIFIED

ACKNOWLEDGEMENTS

This guidebook is a result of a two-years long cooperative effort among three different organizations. During this two-years course several persons from these organizations have directly contributed to this final form of the guidebook. The major parts of this guidebook were typed by Maria Bielinski, Cheryl Bulen, Joan Doyle, Kim Erickson, and Teresa Polski of Honeywell Computer Science Center, and Linda Mixson and Mary Lucas of Information Research Associates. Cheryl Bulen was primarily responsible for integrating and editing the final copy of the guidebook. John Miller undertook the task of making this book clear and pleasantly readable. Considering that this book has been written by about twenty scientists from three different organizations, John's task was a real challenge. The most beautiful contributions to this guidebook have been made by Dave Vogel and Mary Kay Javener from the Graphics Section, Honeywell Computer Science Center. Dave and Mary Kay prepared all the figures for this guidebook. Finally, to make this book more useful, Bill Wood and Diane Ahart developed an emacs-based system for generating index tables.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



VOLUME 1

TABLE OF CONTENTS

	Page
INTRODUCTION	1-1
DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS	2-1
2.1 DESIGN OF FAULT-TOLERANT SYSTEMS	2-1
2.2 RELIABLE SYSTEM REQUIREMENTS	2-4
2.3 ARCHITECTURE OF DISTRIBUTED SYSTEMS	2-6
2.4 DESIGN ISSUES AND TRADEOFFS	2-8
2.5 SPECIFYING RELIABILITY CONSTRAINTS FOR DISTRIBUTED OBJECTS	2-11
2.5.1 Fault Models	2-11
2.5.2 Degrees of Reliability	2-12
2.5.3 An Example Configuration	2-14
2.5.4 An Iterative Approach to Specification	2-17
2.6 AN OVERVIEW OF THE DESIGN STEPS	2-17
DISTRIBUTED COMMAND AND CONTROL SYSTEMS	3-1
3.1 COMMAND AND CONTROL SYSTEM FUNCTIONS	3-2
3.1.1 Operational Environment	3-2
3.2 DISTRIBUTED C2 SYSTEMS	3-4
3.2.1 Required DOS Functions	3-4
3.2.2 DOS Functional Requirements	3-4
3.3 AN EXAMPLE SCENARIO FOR DESIGN EVALUATION	3-8
3.3.1 Organizational Components of a Tactical C2 System	3-9
3.4 CONCLUSIONS	3-12
RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES	4-1
4.1 A DESIGN MODEL FOR RELIABLE DISTRIBUTED SYSTEMS	4-4
4.2 CONSISTENCY MANAGEMENT IN DISTRIBUTED SYSTEMS	4-7
4.2.1 Serialization Consistency	4-7
4.2.1.1 Timestamp Based Protocols	4-8
4.2.1.2 Locking Protocols	4-9
4.2.1.3 Optimistic Concurrency Control	4-12
4.2.1.4 Basic Timestamp Ordering Versus Locking	4-13
4.2.1.5 Deadlock Prevention vs. Deadlock Detection	4-14
4.2.2 Non-Serial Consistency	4-15
4.3 RELIABILITY TECHNIQUES IN DISTRIBUTED SYSTEMS	4-15
4.3.1 Error Detection Techniques	4-16
4.3.2 Error Recovery Techniques	4-17
4.3.2.1 Checkpointing and Rollback	4-17

4.3.2.2 Careful Replacement	4-19
4.3.2.3 Logs/Audit Trail	4-22
4.3.2.4 Commit Protocols and Atomic Actions	4-25
4.3.2.4.1 One-Phase Commit Protocols	4-26
4.3.2.4.2 Two-Phase Commit Protocols	4-27
4.3.2.4.3 Coordinator Selection	4-30
4.3.2.4.4 Comparison of One-Phase and Two-Phase Commit Protocols	4-31
4.3.2.4.5 Presumed Abort and Presumed Commit Protocols	4-33
4.3.2.4.6 Hybrid Commit Protocols	4-36
4.3.2.5 Replication Management in Distributed Systems	4-36
4.3.2.5.1 Centralized Control	4-39
4.3.2.5.2 Voting	4-41
4.3.2.5.2.1 Majority Voting	4-41
4.3.2.5.2.2 Weighted Voting	4-46
4.3.2.5.2.3 Weighted Voting and Directories	4-51
4.3.2.5.3 Operations Under Weak Consistency Requirements	4-52
4.3.2.5.3.1 Fischer and Michael's Scheme	4-53
4.3.2.5.3.2 Allchin's Suite of Algorithms	4-55
4.3.2.5.3.3 Some Performance Measures for Weak Consistency Management Schemes	4-59
4.3.2.5.4 Detecting and Entering Degraded Mode	4-60
4.3.2.6 Network Partitioning and Continued Operations	4-60
4.3.2.6.1 Data Patch	4-61
4.3.2.6.2 Log Transformation	4-62
4.3.2.7 Self-Identifying Objects	4-64
4.3.2.8 Forward Error Recovery	4-64
4.4 INTEGRATION OF RECOVERY MECHANISMS IN THE DESIGN MODEL	4-64
4.4.1 Stable Storage	4-64
4.4.2 Object Management	4-65
4.4.3 Process and Transaction Management	4-68
4.4.4 Remote Procedure Calls	4-75
4.4.5 Distributed Objects and Transactions	4-77
4.5 CONCLUSIONS	4-77

ZEUS DISTRIBUTED OPERATING SYSTEM	5-1
5.1 PRINCIPLES OF DISTRIBUTED OBJECT-ORIENTED DESIGN IN ZEUS	5-5
5.1.1 Structure of Object-Oriented Systems	5-5
5.1.2 Object Naming	5-6
5.1.3 Functions of the Type Managers	5-7
5.1.4 Structure of Type Managers	5-7
5.1.5 Distributed Types	5-9
5.2 STRUCTURE OF THE ZEUS SYSTEM	5-9
5.2.1 Structure of the Zeus Kernel	5-10
5.2.1.1 The Operation Switch	5-11
5.2.1.2 Unique Identifier Generation	5-12
5.2.1.3 Network Handler	5-15
5.2.1.4 Kernel Initiator	5-16
5.2.1.5 Kernel Design	5-16
5.2.1.5.1 The Kernel Interface	5-19

5.2.1.5.1.1 The Remote Procedure Call	5-20
5.2.2 System-Defined Type Managers	5-21
5.2.2.1 Type-Type Manager	5-21
5.2.2.2 Process/Transaction Manager	5-22
5.2.2.3 Principal and Authentication Manager	5-23
5.2.2.4 Symbolic Name Manager	5-24
5.2.2.4.1 Symbolic Name Contexts	5-25
5.2.2.4.2 Context Sharing	5-25
5.2.2.4.3 Reliability Issues	5-26
5.2.2.4.4 Users' View of the SNM	5-26
5.2.2.5 The Program Type Manager	5-28
5.2.2.6 Message Type Manager	5-28
5.2.2.6.1 Reliability of Message Objects	5-28
5.2.2.6.2 Scope of Inter-process Communication	5-28
5.2.2.6.3 Message Operations from the User's Viewpoint	5-29
5.3 DESIGN OF RELIABLE TRANSACTION MANAGEMENT	5-30
5.3.1 Functions for Reliable Application Systems	5-30
5.3.2 Design of Reliable Transaction Management	5-34
5.3.2.1 Overview of the Architecture	5-36
5.3.2.1.1 Process Manager (PM) Structure	5-36
5.3.2.1.2 Object Manager Structure	5-37
5.3.2.2 Transaction Management Design	5-38
5.3.2.2.1 Top-Level Transactions	5-38
5.3.2.2.2 Nested Transactions	5-41
5.4 CONSTRUCTING RELIABLE SYSTEMS	5-48
5.4.1 Site Restart From Crashes	5-48
5.4.2 Reliable Remote Procedure Calls	5-49
5.4.2.1 Duplicate Invocation Messages	5-49
5.4.2.2 Client Transaction Site Crash	5-50
5.4.2.3 Server Object Manager Site Crash	5-50
5.4.3 Replication Management	5-51
5.5 CONCLUSIONS	5-53

COMMUNICATION NETWORK DESIGN METHODS	6-1
6.1 INTRODUCTION	6-1
6.2 MULTITREE STRUCTURED (MTS) GRAPH	6-2
6.3 IMASE AND ITOH CONSTRUCTION	6-4
6.4 MEMMI AND RAILLARD CONSTRUCTION	6-5
6.4.1 Construction C1	6-5
6.4.2 Construction C2	6-6
6.4.3 Graph Construction	6-7
6.4.3.1 T(G,h) Construction	6-7
6.4.3.2 B(G1, G2) Construction	6-8
6.5 SUMMARY OF KNOWN SOLUTIONS	6-9

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS	7-1
7.1 INTRODUCTION	7-1
7.2 PERFORMANCE ANALYSIS METHODOLOGIES	7-2
7.2.1 Performance Measures	7-2

7.2.2 Models and Hierarchical Structuring	7-3
7.2.3 Parts of a Performance Model: System, Environment, Workload	7-5
7.2.4 Techniques for Evaluating Performance Models	7-5
7.2.4.1 Analytic Methods	7-6
7.2.4.2 Simulation Methods	7-6
7.2.4.3 Hybrid Methods	7-7
7.2.5 Performance Measures for Integrity Mechanisms	7-7
7.2.6 Example Metrics for Some Generic Integrity Mechanisms	7-8
7.2.6.1 Transaction Mechanisms	7-9
7.2.6.2 Concurrency Control	7-9
7.2.6.3 Object Replication	7-10
7.3 MODELING TECHNIQUES IN THE ZEUS EXAMPLES	7-10
7.3.1 Introduction to Zeus Modeling Techniques	7-10
7.3.2 Execution Graphs	7-12
7.3.3 Information Processing Graphs	7-13
7.3.4 The PAWS Language and Model	7-17
7.3.5 Automating Model Generation	7-17
7.4 MODELING THE SYSTEM STRUCTURE	7-19
7.4.1 Issues in Modeling Distributed Systems	7-19
7.4.1.1 Low-Level Communications	7-20
7.4.1.2 Remote Procedure Call	7-21
7.4.1.3 Concurrency Control	7-24
7.4.2 Issues in Modeling Reliable Systems	7-25
7.4.2.1 Transactions and Commit Protocols	7-26
7.4.2.2 Object Replication	7-28
7.5 MODELING THE ENVIRONMENT	7-29
7.5.1 Modeling Single Resource Faults	7-30
7.5.2 Modeling Host/Site Faults	7-32
7.6 MODELING THE WORKLOAD	7-34
7.6.1 Instruction Mixes	7-35
7.6.2 Synthetic Jobs	7-37
7.7 EVALUATION OF THE EXAMPLE SYSTEM	7-38
7.8 SUMMARY AND CONCLUSIONS	7-38
 RELIABILITY ANALYSIS TECHNIQUES AND TOOLS	 8-1
8.1 SPECIFICATIONS OF RELIABILITY MEASURES	8-2
8.2 NETWORK-BASED RELIABILITY MODEL	8-4
8.3 NETWORK RELIABILITY COMPUTATION	8-7
8.4 NetRAT USER INTERFACE	8-9
8.5 ANALYSIS OF SOME RELIABILITY TECHNIQUES	8-10
8.5.1 Analysis of Stable Storage	8-10
8.5.2 Analysis of Replication Management	8-15
8.5.3 Analysis of the Broadcast-Bus Network	8-21
8.5.4 Analysis of Fault Tolerance	8-22
8.6 CONCLUSIONS	8-23
 DESIGN VERIFICATION METHODS	 9-1

INTEGRATED DESIGN METHODS	10-1
10.1 INTRODUCTION	10-1
10.2 DESIGN STEPS	10-1
10.3 REQUIREMENTS STATEMENT	10-4
10.3.1 Functional Requirements	10-5
10.3.2 Reliability Requirements	10-6
10.3.3 Performance Requirements	10-6
10.4 CONCEPTUAL DESIGN	10-7
10.5 FUNCTIONAL ARCHITECTURE	10-8
10.5.1 System-Level Type Managers	10-9
10.5.2 Kernel Functions	10-9
10.5.3 Design Analysis and Verification	10-10
10.6 DETAILED DESIGNS	10-10
10.7 SUMMARY	10-12

APPENDIX A PERFORMANCE EVALUATION OF THE EXAMPLE SYSTEM

BIBLIOGRAPHY

INDEX TABLE

CHAPTER 1

INTRODUCTION

The design of a distributed system involves many complex decisions. The purpose of a designers' guidebook is to help a designer in systematically addressing the various design issues and making the most appropriate decisions so that the final design meets the desired requirements. It is important to stress the distinction between a guidebook and a handbook. A guidebook provides a comprehensive set of procedures which can aid a designer in achieving a goal. A handbook provides a comprehensive set of results (e.g., tables) which provide a basis from which a designer may make design decisions for a specific application. It is appropriate to write a handbook if one has the details of a set of applications of interest and the associated system environments. A guidebook is applicable to a larger set of problems and designers because of its orientation to procedures rather than results.

A guidebook describes the steps which take a designer from a set of requirement statements to a detailed system design which would exhibit the desired operational characteristics in a specified implementation base. Each design step refines the design and further defines what are the system's operational attributes. One set of attributes are those associated with the fault tolerance of a system -- availability, reliability, and survivability. An example of the design decisions that must be made are the degree of availability required for a given application and the performance required of a system environment to achieve it. It is a well-established principle that the designs should be subjected to early evaluations before starting any implementations. In fact, the design steps and the evaluation steps should proceed in a closely coupled fashion. This book presents a set of design guidelines for constructing fault tolerant distributed systems and a set of procedures for evaluating the desired operational characteristics of such designs.

The main contribution of this guidebook is a unified presentation of system recovery mechanisms, a framework for their integration, and a set of evaluation techniques. It provides a starting point for the development of a design methodology of fault tolerant distributed systems.

This chapter describes the contents of the guidebook and how it may be used. This guidebook is organized into two volumes. The first volume describes reliability mechanisms, a framework for expressing designs, and techniques for evaluating mechanisms. There are two classes of problems that are not addressed in the reliability mechanism discussion -- security and Byzantine agreement. These problems were deemed outside of the scope of this contract. A framework based on object-oriented design is defined and used for

expressing designs because it motivates the discussion of reliability mechanisms and aids in their integration into a unified design model. An example operating system design is derived from the framework and used as a basis for presenting and demonstrating analysis techniques. Although the mechanisms, techniques, and results are described within the context of an object-oriented design, they are equally applicable to process-oriented designs. Volume II contains the complete details of the example system and the results of its analysis. Some familiarity with Ada [DoD83](1) and the Concurrent System Definition Language [FRAN83a] is required to understand the detailed designs. The definition of the Concurrent System Definition Language is included in the final report for this contract.

Our presentation begins in Chapter 2 by discussing some of the fundamental design issues for reliable distributed systems. Before we discuss any issues, we define a taxonomy for common understanding of various terms related to fault-tolerant systems. This vocabulary introduces notions such as fault, repair, failure, detection, diagnosis, recovery, object-oriented design, and process-oriented design. From here we begin to consider issues. The first issue is what are reliability requirements and how does one specify them. Two approaches to specifying requirements are reviewed. The traditional approach is based on statistical quantities, while an approach that is presented in this guidebook uses discrete measures. Fault models are introduced and a method of iteratively specifying requirements is proposed. This naturally leads into a discussion of the intimate relation between reliability and performance. This is perhaps the most important issue faced by a designer. This chapter concludes by proposing a number of steps that aid in the design of reliable distributed systems.

A discussion of any kind of mechanism is best done in the context of an application and its associated requirements. In Chapter 3 we present an overview of distributed command and control systems. This allows us to elicit requirements for the hardware and software environment. The discussion within this chapter and throughout the entire guidebook is relevant for both strategic and tactical command and control systems. An example scenario based on tactical command and control systems is presented here to serve as a vehicle for evaluating the example system design.

In Chapter 4 we discuss various reliability techniques. The techniques discussed here include atomic actions, checkpointing, rollback, commit protocols, recovery logs, and replication management. The specific problems that reliability mechanisms must solve are described. These include different consistency requirements (e.g., mutual, internal and external) in the presence of concurrent operations and component failures. These techniques are applicable to both process-oriented and object-oriented system designs. The remaining part of this chapter presents an object-oriented design model for integrating these techniques into distributed system designs. The advantages of the object-oriented approach are discussed. This design model shows how to synthesize secure and stable distributed objects that survive crashes of system components and support high availability of functions from unreliable

(1) Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

INTRODUCTION

resources. The remainder and majority of the chapter discusses different mechanisms that can be used to create stable distributed objects. These include mechanisms for concurrency control, commit protocols, forward and backward recovery, replication management, and network partitions.

Chapter 5 presents the principles followed in designing Zeus, an object oriented distributed operating system for integrating recovery mechanisms into the designs of distributed command and control systems. The design is presented in terms of an integrated set of functions by which an application programmer may reliably manage objects in a distributed environment. The functions transparently provide complex recovery mechanisms, commit protocols, concurrency control, and remote object accessing to application programmers. The Zeus design is used as a basis for evaluating reliability mechanisms. A sufficient amount of the design is presented to aid the description of the evaluation techniques. The majority of the detail of the Zeus design is contained in Volume II.

Following the discussion of reliability mechanisms and an example of how they can be integrated into a distributed operating system, we turn our attention to techniques that help us evaluate how reliable a distributed system is. Each of Chapters 6 through 9 provides a different set of evaluation techniques.

The survivability of the communications network of a distributed system is of critical importance. If failures in the network result in partitions, the integrity of the distributed system may be compromised. Chapter 6 presents some design methods and their associated evaluation methods to minimize communication delays and maximize survivability. The methods discussed in this chapter are based on graph theory. The survivability of a network is characterized in terms of connectivity of the network and communications delay in terms of the diameter of the network.

One of the major concerns of a designer is the performance of a system. Chapter 7 discusses performance analysis techniques and tools for reliable distributed systems. A survey of the different metrics and approaches to performance analysis is given with special emphasis placed on those techniques that are applicable to the design phase. A detailed discussion of the representational and simulation tools used for the modeling of the Zeus reliability mechanisms is given. The bulk of the chapter is devoted to discussing the difficulties involved with modeling reliable distributed systems. These issues include low level communication, remote procedure call, concurrency control, transactions and commit protocols, object replication, and fault injection. The discussion is motivated with examples that are drawn from the Zeus models. Finally, a summary of the workload used to drive the Zeus models and of the simulation results are presented. The last part of this section presents some of the performance data obtained from the PAWS models of the Zeus system with workload based on the command and control example scenario. This data illustrates some of the approaches for comparing various options in reliable distributed system designs.

The specification and evaluation of the reliability characteristics of a design are an important process in attaining reliable distributed systems.

Chapter 8 reviews and contrasts the approaches to specifying reliability characteristics. Different measures of fault tolerance attributes are defined. A set of techniques and associated tools that evaluate a design for adherence to a reliability specification are presented. The techniques are based on combinatorial analysis. Examples are given showing the use of the techniques for evaluating the reliability of replicated objects and stable storage.

Another property of reliability mechanisms that may be analyzed is correctness. It involves the formal specification of the desired behavior of a reliability mechanism and the proof/analysis that a model of the implementation exhibits the behavior. Chapter 9 presents three methods for proving or analyzing the fault tolerant properties of distributed system designs. The first method is based on applying program verification techniques to a design expressed in a suitable programming language. The Gypsy programming language and verification environment are described. An example of their application to recovery mechanisms at a single site is given. This method is very good for establishing "black-box" stimulus-response behavior of a process or network of communicating processes. It is, however, difficult to express and verify relationships among state variables. A second method is described that allows the combination of assertions over the states of several local processes to be combined into a global assertion stating a relationship among the variables of several processes. A relatively less rigorous approach that is amenable to manual proofs for small systems is described. It is based on interval logic. An example of its application to a commit protocol is given. A last method of design validation is based on the functional simulation of a design. It uses an executable model to represent the behavior of a system. The model is run with faults being injected and the set of events that are executed is collected. The order and occurrence of events is analyzed with respect to the occurrence of faults. An example showing the functional simulation of transaction atomicity in the presence of site failures is given.

Chapters 2 through 9 introduce and discuss a number of concepts, procedures, tools, and techniques all concerned with the design and development of reliable distributed systems. Chapter 10 integrates these ideas into a single, consistent methodology. The steps of the methodology are based on the waterfall model of software development. The principle extension to the model is the integration of reliability and performance considerations into the requirements specification and design evaluation.

The second volume of this guidebook is devoted entirely to the presentation of the detailed designs of the Zeus system and its performance evaluation data. These designs are presented using CSDL and Ada. A high-level description of these detailed designs is also presented in Chapter 5 of this volume. The performance analysis data presented in the second volume was obtained using the PAWS simulation models. The results presented here contain a more detailed exposition of the results as compared to the presentation of data in Chapter 7 of the first volume.

CHAPTER 2

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

One of the most important characteristics of an effective command and control system is its survivability. Survivability is the ability to support continued operations and meet mission requirements in the event of failures of the system components. Survivability implies a number of system function attributes, such as reliability, availability, and performance; therefore, specifications for a survivable system must include the reliability, performance, and availability characteristics of its functions as well as that functionality. A survivable system must contain recovery mechanisms that deal with abnormal conditions arising from component losses and provide continued operations.

Before actually building a system, the system designs should be analyzed to ensure that they support the intended functionality with acceptable performance and reliability. Such analyses are important if suitable cost/performance and cost/reliability tradeoffs are to be done during the design phase. Modeling techniques play an important role for performing such design analyses. The primary goal of this chapter is to highlight what the principles are for designing fault-tolerant systems, how the requirements for a survivable C2 system can be specified, why distributed systems are important vehicles for building reliable C2 systems, and what steps exist in building reliable distributed C2 systems.

2.1 DESIGN OF FAULT-TOLERANT SYSTEMS

A system is viewed as a collection of objects which denote the physical and logical components comprising the system. The terms "fault", "repair", "failure", and, "recovery" are used as follows.

A "fault" is an event in which an object ceases to function according to its specification. It is assumed that all component faults are clean, which means that the failed components do not actively introduce spurious information in the system. If a faulty component begins to operate correctly at some future time, then it is said to have been "repaired".

Failures in a distributed command and control system can arise due to either loss of components because of hostile actions or software and hardware faults. Generally, the component losses due to hostile actions tend to be geographically isolated. Such actions may cause destruction of one or more sites and communication links in a system. The loss of communication links

may cause the system to be partitioned into subsystems that cannot communicate with one another. A survivable system must include adequate mechanisms to support continued operation under network partitioning conditions. At a site, transient failures can occur within the CPU, primary memory, or secondary memory. Loss of primary memory contents due to transient failures is a major problem that requires critical information to be periodically forced onto secondary storage that is more robust. Secondary storage failures may be the result of events such as head crashes or transient errors during the writing of a page. The design of a reliable distributed system requires appropriate recovery mechanisms to deal with the kind of operating conditions mentioned above.

The general principles of reliable system design consist of incorporating into the designs these three functions: detecting errors, diagnosing fault(s) that caused the error condition, and finally recovering and restarting the activity after isolating or masking the faulty components.

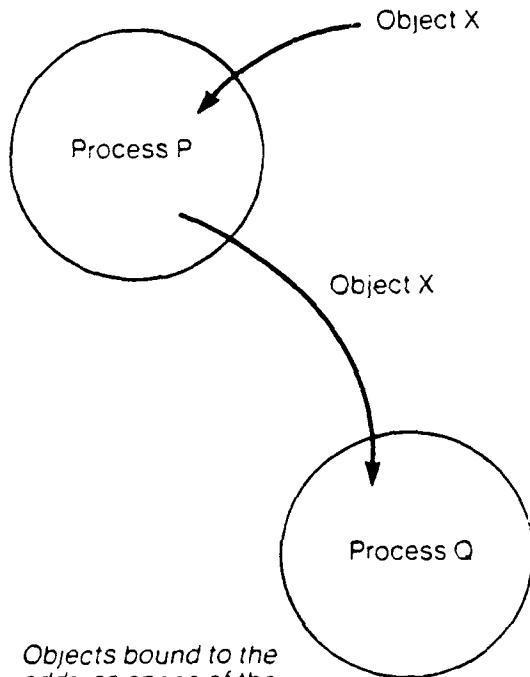
Error detection and fault diagnosis can be quite complex in large systems. System structuring plays an important role in reducing this complexity. There are two distinctly different approaches to reliable system designs: process-oriented and object-oriented. Figure 2-1 presents these two views of system structuring. Traditionally the developers of fault-tolerant systems have taken a process-oriented view of structuring these systems. In this view, resources (data and devices) are bound to the address space of a process during its execution. The process is responsible for recovering its locus of execution along with the resources bound to its address space. The second approach takes an object-oriented view of the system in which each resource is permanently bound to the address space of a process which is called the object manager. Every other process that wants to access this resource does so by invoking the interface procedures supported by the object manager. Each object manager virtually defines a domain that confines the errors. Error recovery and fault-diagnosis are confined to these domain boundaries.

Fault diagnosis can be a very expensive process. In many designs the identity of the component that caused an error is of no concern. In such situations there exist some pre-defined alternate execution paths that either by-pass the faulty component or tend to ignore the effect of the faulty component. A simple example is the atomic transaction facility. If an error occurs while a transaction is under execution, the transaction is aborted and restarted from beginning. In the case the of accessing a replicated object, if one copy is unavailable for access, the requests are directed to some other copy. In this case no efforts are made to find out whether the copy was inaccessible due to site crash, disk crash, or communication network failures.

The designs that avoid extensive and expensive (in terms of time) fault diagnosis for recovery do so by including enough redundancy in the system to provide alternate execution paths when an error condition is encountered. This expedites the recovery process; however, the added redundancy consumes resources such as primary and secondary memory, and CPU cycles during normal processing; thus complex fault diagnosis is avoided at the increased cost in terms of system resources and normal processing requirements.

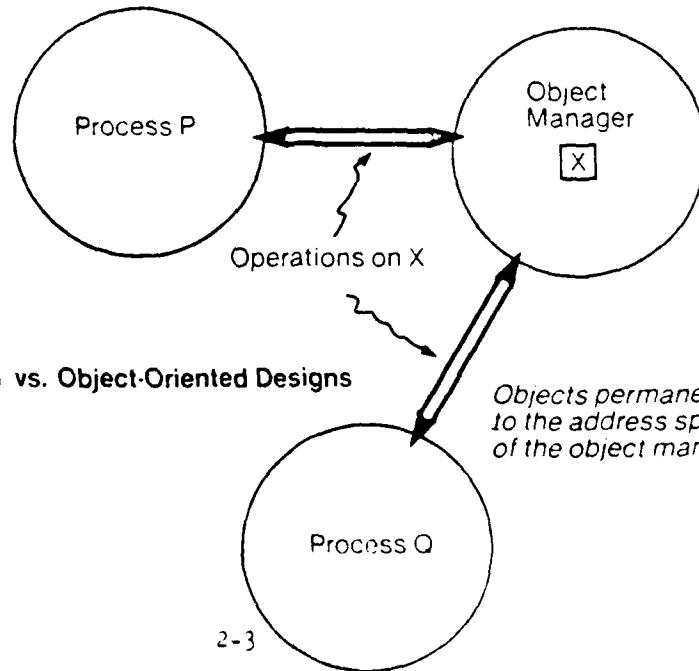
DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

Process-Oriented Design



Objects bound to the address space of the processes operating on them

Object-Oriented Design



Objects permanently bound to the address space of the object manager

Figure 2-1. Process-Oriented vs. Object-Oriented Designs

2.2 RELIABLE SYSTEM REQUIREMENTS

It is recognized that there are multiple ways to decompose a software development lifecycle. We look at the development steps in order to elicit system requirements and understand what modeling tools may be applied at which steps. The system development activity can be divided into a number of phases. For our purposes, we define three phases -- requirements, design, and implementation. The design phase may require several iterations involving design validation, design modifications and/or possible re-definitions of the requirements.

It is inappropriate to assume that every project will develop a system from scratch. Rather, we assume that a system may be expanding its hardware base, moving to a new hardware base, developing new applications, or evolving existing applications. As these activities progress an important question to be asked is "Will the new system meet the applications requirements?". In order to answer this question requirements must be obtained from several key people. The requirements stated below are illustrative of the types of information that are needed. A discussion of modeling tools that use these requirements as input is contained in Part II of the guidebook.

An application may be described as a collection of objects on which operations are executed by users. Some operations may be combined together and executed as a single end-user operation. A requirements statement may be made about the performance and reliability of the operations as described below:

Performance Requirements: In general the performance requirements are specified in terms of the response time and throughput. There are several ways in which these two measures may be specified for a distributed system. Average throughput and average response time for the execution of a given operation on an object can be specified in the following terms:

- (1) For the overall system,
- (2) For some particular sites in the system,
- (3) For each operational mode such as emergency/peace-time modes,
- (4) As a function of available resources (sites) in the system.

In addition to the mean values, upper and lower bounds or variances may be specified for these measures.

Reliability Requirements: Traditionally the reliability requirements for a service or operation are specified in terms of its expected availability and mean-time-to-failure. Like the performance requirements specifications, the reliability requirements can also be specified in terms of the four ways described above. It may also include the types of failures that must be withstood, and the number of failures of a given type that must be withstood.

Similar requirements may be made about groups of operations. In addition, it is assumed that statements are made about what the hardware configuration is and the assignment of objects and operations to sites. From such information we are interested in answering two questions: "What level of

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

reliability does a system provide?", and "What is the extra cost of the reliability?".

Ideally a user should be able to specify the desired reliability requirements for objects, operations, and groups of operations without knowing the details of the implementing mechanisms; tools should then automatically configure a system. More realistically, a system administrator who is knowledgeable about the system's hardware and software will manually make the selections and adjustments needed to achieve the desired level of reliability.

In order to state system requirements and to develop a system that meets them we are still faced with a problem of how to specify the requirements. Traditionally, component reliability is given by statistical quantities such as the mean time to failure (MTTF), mean time to repair (MTTR), and the probability of availability. This suggests that one way for a user to specify the reliability of objects and operations is to give the desired values (e.g., 0.95 MTTF). This is certainly the most accurate way of defining the expected reliability of an object since it takes into account the interrelation among all of the dependent objects and components of a system and their individual characteristics. There are, however, at least three problems with using statistical quantities for user level specifications.

First is the problem of using small quantities to specify values. Should the MTTF be 0.94 or 0.95? Why? Would different users choose different values for similar objects?

Secondly, there are typically many different combinations of parameters, replication strategies, and configurations which will yield the same or roughly the same MTTF and availability for a given object. A simple numerical quantity gives no indication as to which of several possible strategies to choose. Furthermore, without being given additional information, it is difficult for an administrator and probably impossible for the system itself to choose the appropriate solution.

The third problem arises due to the application environment which is being considered in this guidebook. The probabilities of component failures may change unpredictably under military stress conditions. The MTTF and availability of a component completely describe its fault characteristics. There are many circumstances, however, where these metrics are difficult or impossible to evaluate. As an example of such a circumstance, consider a command and control system in a potential combat situation. The definition of a component "fault" in such a system would have to include the destruction or disruption of that component in combat; and thus, this eventuality must be taken into account when calculating the MTTF and availability of the component. Unfortunately, the probability of an attack, or the probability that an attack will ensue in such a way as to affect the performance of a given component of the system, depends on a number of decidedly non-quantifiable factors such as political climate, human factors, recent history, and so on. In such conditions it is more reasonable to ask questions such as "Does this service (function) remain available given that a set of components are unavailable?".

An alternative to using statistical quantities would allow the strategists to define (and redefine as necessary) a number of combat scenarios. Each scenario would correspond to a single fault class in the fault model. Fault class X, for example, would define the set of "faults" which were likely to occur in combat scenario X. Such a fault model would not say anything about the probability that scenario X will occur, it will only reflect the fact that if scenario X is executed, then the faults in class X are likely to manifest themselves. One advantage here is that the behavior of some object with respect to a given fault class is just the reliability of that object in the associated combat scenario.

This alternative to using statistical quantities provides a set of pre-defined reliability levels. Associated with each reliability level is a consistency (integrity) specification. The levels overcome the problems with strictly numerical specifications by associating a boolean-valued consistency requirement. An object is said to belong to a certain reliability level for a given set of faults if the associated consistency specifications are maintained under the presence of those faults. The object is viewed as being "completely immune" to the faults in that set for the associated consistency level. The maximum cardinality of such a set for a given reliability level of an object determines the robustness of that object. Faults may include events such as site failures, link failures, disk failures, and memory failures. Each category can be refined when it is appropriate to do so. For example disk failures can be refined to include single page failures and disk pack failures.

The behavior of an object when it encounters a fault in a given fault class determines the degree of reliability of that object with respect to faults in that class. The possibilities range from complete immunity to complete and permanent failure upon encountering faults in that set. There is a continuum of such degrees of reliability. Later in Section 2.5.2 this point is elaborated and four exemplary reliability classes are introduced. Next we consider what is required to make a system survivable.

2.3 ARCHITECTURE OF DISTRIBUTED SYSTEMS

Redundancy of both hardware and software resources is the most important characteristic of reliable systems that support continued operations despite component losses. The geographical distribution of critical system databases and processing resources is key to the design of survivable systems. Thus, in the event of loss of a particular site in a command and control system, it should be possible to use database copies and processing resources at other sites. The need to maintain and update replicated databases imposes the following requirements for the underlying architecture:

The system must contain appropriate processing resources (CPU, memory, secondary storage) at each site.

There must be communication between the sites as well as with local and remote users of the databases (a) to keep the replicated copies mutually consistent and (b) to provide access to remote users on system reconfiguration.

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

These two requirements make distributed system architectures the most natural candidates for supporting highly survivable C2 systems. Functional redundancy and geographical dispersion enables distributed systems to survive hostile actions and to provide continuous operation. These advantages of distributed systems arise partially from the distribution of system state information. However, effective survivability mechanisms are based upon consistent system state. Distributed operating systems used in this application must incorporate mechanisms to maintain the consistency of the distributed system state information in the presence of concurrent updates and system component failures. This is essential to guarantee correct functioning on reconfiguration and restart; therefore, suitable recovery mechanisms and concurrency control mechanisms are required in the distributed operating system to maintain consistency of distributed state variables.

A distributed system consists of multiple computers interconnected by a communication network that cooperate to complete a computation. The mechanisms that enable the cooperation are implemented by a distributed operating system. A conceptual picture of a distributed operating system is shown in Figure 2-2.

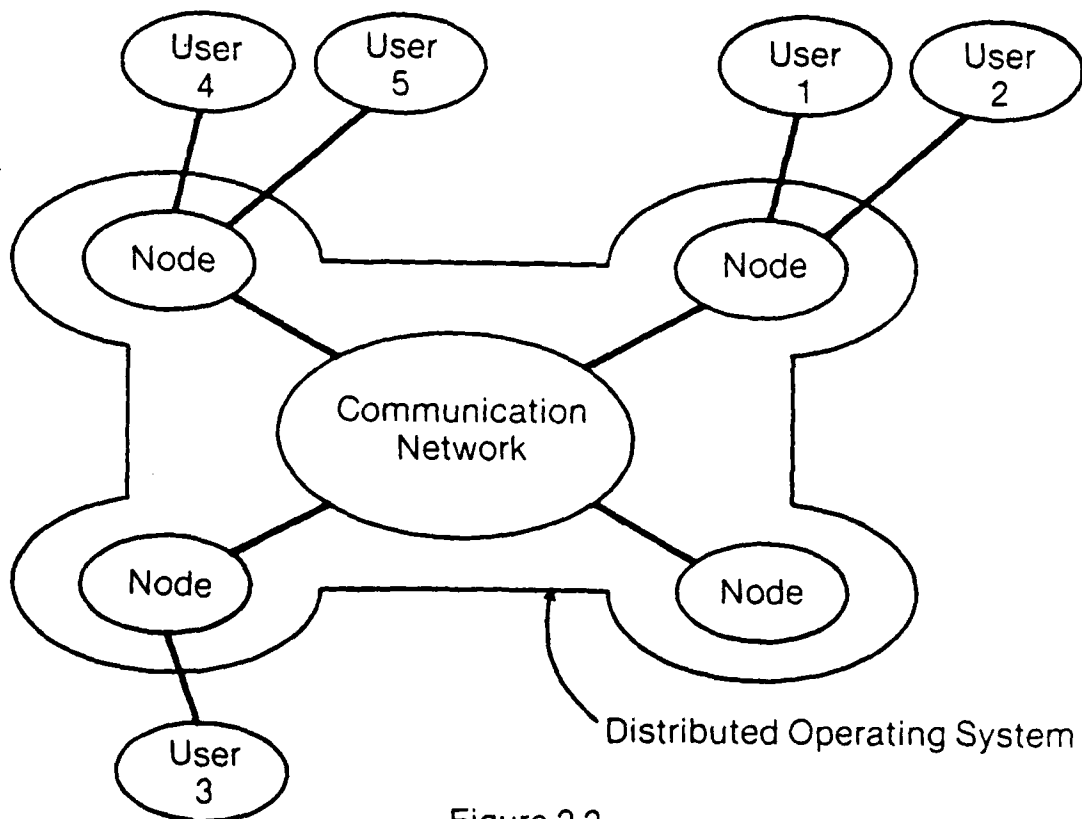


Figure 2-2.

A site consists of at least one physical processor, an operating system kernel, primary and possibly secondary memory, an interface to the communication network, and possibly interfaces to input/output facilities. The sites are physically separated and communication occurs by message exchange rather than by shared memory. Each site has processes and resources which constitute fragments of system processing activities. Since control of these processing activities is distributed among the sites, a single site normally has neither system wide authority nor a complete view of the global system state.

A distributed operating system creates and manages logical (perhaps physically distributed) resources (processes and files) and physical resources (processors and memories). A distributed operating system is based on a set of protocols which govern interaction between sites. The operating system kernel at each site manages its physical resources autonomously and may cooperate with other kernels in the management of its logical resources. The state information may be partitioned and distributed among the operating system kernels. The individual kernels operate concurrently, and possibly asynchronously on the basis of locally available state information. The system interface consists of a set of functions which the distributed operating system provides to the application environment.

A communication system transfers information among the sites in a distributed system. It is used by distributed operating system kernels, system processes, and application processes to convey updates and to gain access to global system state and to utilize resources provided by other sites. Communication systems typically appear in system designs and implementations at a higher level of abstraction because the detailed realization is isolated from the remainder of the system. In the context of a command and control system, a communication system is meant to include an internet, a collection of interconnected networks.

2.4 DESIGN ISSUES AND TRADEOFFS

The previous section stated that exploiting the redundancy of both the hardware and software of a distributed systems is the key to gaining potential benefits. The benefits can include improvements in performance and reliability. The former is possible due to the reduction in communication cost to access an object and the increase in parallelism of operations on an object. The latter is possible because operations can continue despite the loss of system components. For example, if a directory is replicated on every site on a distributed system, the cost of reading it is the cost of accessing a local storage device (e.g., there is no overhead incurred due to communication between two sites). It is possible for users on multiple sites to be simultaneously accessing the directory, further improving a system's performance. Finally, if a site fails, the directory can still be accessed by any of the remaining operating sites.

Unfortunately, increases in reliability and performance do not come for free and in many cases are not simultaneously attainable. This tradeoff in system attributes is often determined by a correctness criteria that describes a relationship between the values of the replicas of a distributed object at any point in time. The correctness criteria must ensure that a replication

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

update algorithm satisfies a mutual consistency property: all replicas of an object converge to the same state and become identical if update operations cease. The convergence time can and will vary for different objects and operations resulting in different consistency constraints.

In order to investigate any performance/reliability tradeoffs we must discuss what failure modes may exist, how failure management impacts a distributed system, and what the fault models are for distributed systems.

Effectiveness of a recovery mechanism can be measured in terms of recovery time and performance overhead. To see why the recovery time and performance overhead are important in evaluating the recovery mechanism, consider the performance of a system under normal and faulty conditions. Assume that throughput (defined as the number of units of work performed per unit of time) is an indication of the system performance. If there were no failures, there would be no need for recovery mechanisms. In this hypothetical situation, under a constant load (e.g., fixed number of jobs running in the system at all times) the throughput stays constant at a level that is referred to as the ideal level of performance. Introducing recovery mechanisms into this system to enable it to deal with failures degrades the performance even when there are no failures. An operating overhead is imposed equal to (1) the processing overhead required to check and maintain information about system state for recovery and (2) a storage overhead equal to the storage required to hold redundant information. It is desirable to choose those recovery mechanisms that have the least performance overhead under normal operation.

When an error condition occurs, certain recovery procedures are initiated. These procedures cause an even higher performance overhead. This is called failure recovery operation overhead. After the fault is eventually cleared and the system is recovered, the performance goes back to the level before the failure. Figure 2-3 depicts this simplified situation. There are two important parameters that have to be considered when a failure occurs. First, how much time does it take for the system to recover from the failure? This period of time is called system recovery time. For the duration of the system recovery time, the performance of the system is at its lowest level. Therefore, a good recovery mechanism has to minimize this time period. Second, how much is the performance of the system degraded for the duration of the system recovery? The performance overhead factor includes both the normal operation overhead and the failure recovery overhead.

A more realistic situation is depicted in Figure 2-4. The system operates normally until a fault occurs and some component of the system becomes inoperative. The system executes recovery procedures and operates at reduced capacity. After the recovery procedures are executed, the performance rises to a level below full system performance. Later, the fault is cleared and the system executes recovery procedures to restore the consistency of global system state. During this time, performance is again degraded. Finally, throughput is restored to the normal level.

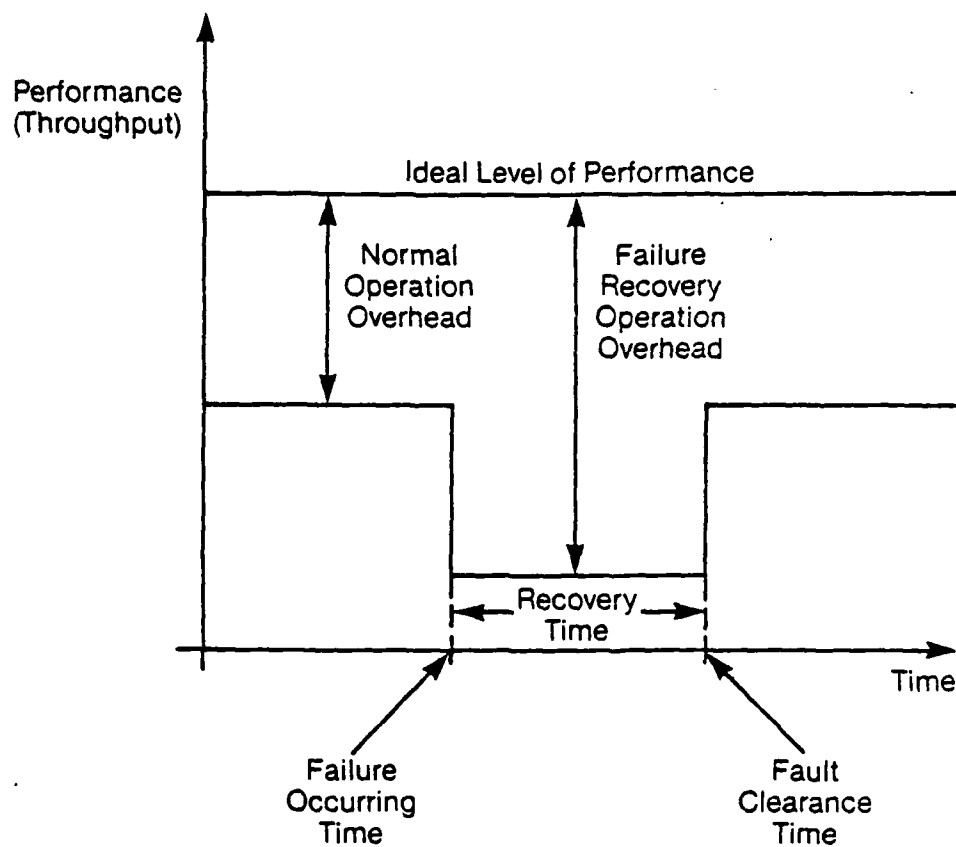


Figure 2-3.

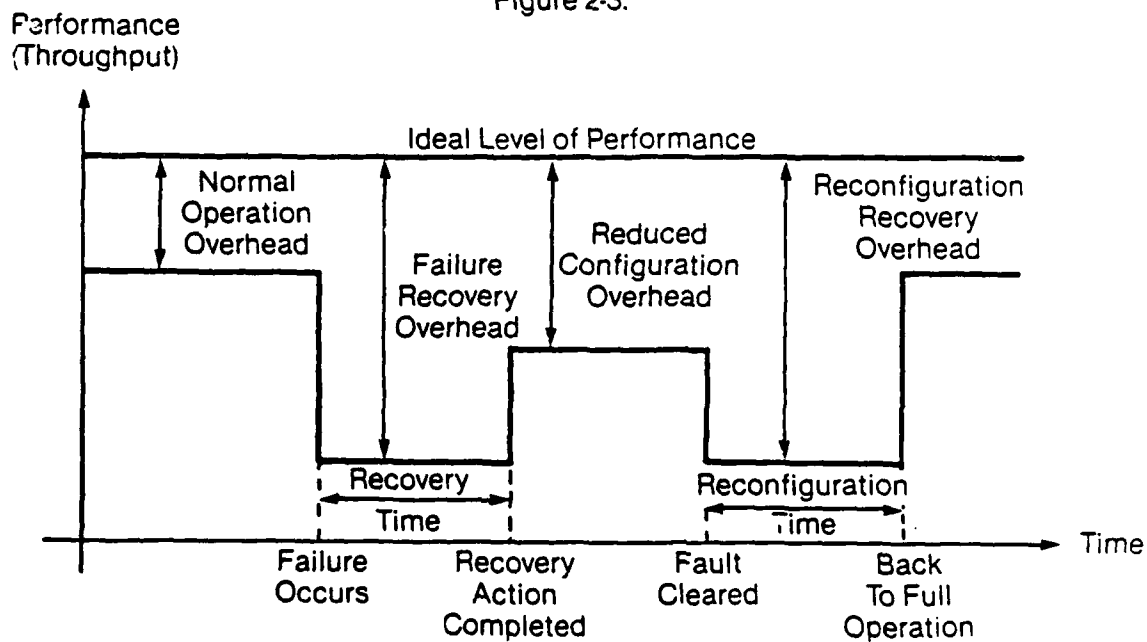


Figure 2-4.

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

This view introduces additional effective measures: Reduced configuration overhead is the difference between ideal performance and performance while part of the system is inoperative. Reconfiguration recovery overhead is the difference between ideal performance and performance while global system state is being restored. Reconfiguration time is the duration of this processing.

Cost is another important factor in deciding which recovery mechanisms should be included in a distributed system. Cost may be measured in terms of the additional hardware resources required to implement a recovery mechanism while maintaining the same level of performance as without the recovery mechanisms. This includes the cost of additional primary and secondary storage and processing power. The memory requirement is derived from the size of the recovery mechanism procedures and the size of any additional data structures. The secondary storage requirements may be further increased if they are required to store multiple copies of objects. The additional processing overhead is derived from the performance overhead previously discussed. Another way to characterize the overhead due to recovery mechanisms is in terms of reduction in response time and throughput.

2.5 SPECIFYING RELIABILITY CONSTRAINTS FOR DISTRIBUTED OBJECTS

Previously we discussed the problems inherent in specifying reliability requirements as statistical values and introduced the notion of using a discrete set of reliability levels as an alternative method. In this section we elaborate this idea by defining a fault model along with a discrete set of reliability levels, followed by an example. Chapter 8 discusses how the reliability specifications based on this model can be analyzed using automated tools.

2.5.1 Fault Models

A fault model of a system is a characterization of the types of component faults which may occur in the system and the object failures which might result. In general, the correspondence between these two is implementation dependent.

An object is considered to be "available" if that object is not currently in a failure state. This means that availability is strictly a local condition in the sense that it depends only on the requirements of the object itself and not on any requirements of its accessor or client process. A more convenient way to express the above conditions in an object oriented system rests on the fact that processes which access objects, including user level jobs, are also considered to be objects as well.

At some point after an object has failed, it may begin to affect the behavior of other objects which are dependent upon it. In particular, the fault may cause one or more of the consistency specifications about the object to be violated. If the object is solely dependent on a faulty resource, for example, then a tacit consistency specification about the object - that it is accessible - may be violated. The situation where an object no longer

satisfies its consistency constraints due to a faulty dependent component will be called a "failure" or an "error condition". At some point after a failure has occurred, a failed object may "recover" from the failure and hence continue operation with all its consistency constraints once again satisfied.

In our model, objects are constructed using five types of resources which may become faulty. These are the volatile memory, secondary storage, I/O devices, the communication links, and the host processors. A set of these components comprise a host node, a cluster or subnetwork of host nodes, and the entire network itself. A resource fault may affect any or all of the objects which depend upon it. A single link fault may have very little effect or cause many failures. A node or host fault normally causes all of the objects (and all of the object managers) associated with it to fail. Failures may, of course, spread to remote objects which are dependent upon those objects. Subnetwork faults are those which cause the network to be partitioned in such a way that all of the objects on all of the nodes in one or more subnetworks fail.

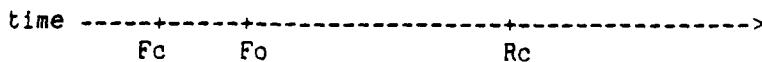
Fault repairability is an additional characterization for faults in each of these five types of components. A resource such as a magnetic disk, for example, may become faulty in a repairable manner such as when the controller breaks or a checksum error occurs, or it may fail in a non-repairable manner such as with a head crash where the data is permanently lost. In command and control systems, the possibility exists that a processor, a link, or some other physical component (including subnetworks and entire networks) may be destroyed thus causing a non-repairable fault of that component.

In addition to listing the types of faults and resulting failures which may occur in a system, a fault model should include data on the relative likelihood of the various faults. This could be done by measuring and recording the statistical behavior of the relevant components in the form of mean-time-to-failure (MTTF) values or similar metrics of component reliability.

2.5.2 Degrees of Reliability

The behavior of an object when it encounters a fault in a given fault class determines the degree of reliability of that object with respect to faults in that class. There is a spectrum of such degrees of reliability. This section presents four points on the spectrum which may be particularly useful for user specifications of object reliability. Along with the definitions of these four reliability classes, an example illustrating these concepts is given.

Volatile objects are unstable with respect to a fault class F and are subject to failure if any of the faults in F should occur. Furthermore, subsequent repair of the offending fault can not repair the object. A volatile object becomes permanently inconsistent whenever a fault in the class F occurs. This situation can be represented graphically on a time line as follows:



DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

The symbols in the diagram have the following meaning:

- Fc - Fault of a component
- Fo - Failure of an object
- Fc - Repair of a component
- Ro - Recovery of the object (not shown here)

The diagram indicates that, with an object which is volatile with respect to a fault class F, a fault in F (Fc) will be followed (after some time) by a failure in the object (Fo). The fault may be repaired at some future time (Rc), but the object will never recover. This definition of volatility corresponds to the conventional notion of volatile memory or RAM. Although volatile objects could be built so that they were stored in non-volatile memory, most frequently, volatile objects will possess at least one component which resides in volatile RAM memory.

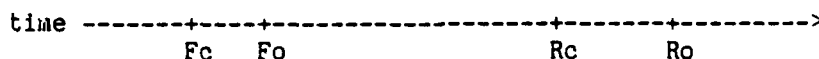
It should be noted that, although most volatile objects have components stored in volatile RAM, it does not follow that all objects which have components residing in RAM will be volatile. In fact, even an object for which all of its dependent objects are volatile can be made to exhibit arbitrarily high reliability by using replication of data and suitable consistency algorithms.

Non-Volatile objects are stable with respect to some fault class F, and may or may not fail after a fault in F has occurred depending on the timing of the fault. Again, the name of this reliability class and its characteristics mirror those of non-volatile memory resources such as magnetic and optical disks, drums, tapes, etc. A non-volatile object will remain intact provided that no faults (in F) occur while some operation is being performed on the object, i.e., while the object is "active". Analogously, a disk record can be expected to survive a repairable fault (in the power supply for instance) only if it was not actually being written upon at the time of the fault.

Resilient objects are guaranteed to be recoverable with respect to a fault class F provided the offending fault is eventually repaired. For objects in this class, if a fault occurs while an operation is in progress, then the operation is completely ignored and the state of the object after recovery is the same as it was before the operation was initiated.

The classical example of a resilient object constructed from two non-volatile objects is the construct described by Lampson and Sturgis in [LAMP81]. In their algorithm, the two component objects are identical and are updated in a strictly serial fashion; one component is updated first and then the other. Any resource fault which happens to occur during an update operation has the potential of perverting the data in only one of the component objects. Assuming that such a perverted object can be identified through a checksum or some similar mechanism, the recovery algorithm simply chooses the non-corrupted component's data to be the state of the composite object after the fault has been repaired. Notice that updates to the resilient object require that both non-volatile components be operational so that, strictly speaking, the object is unavailable for updates and is therefore failed in the event that only one component suffers a fault until

such time as the fault is repaired. A typical timing diagram for a resilient object can be drawn similar to the one for volatile objects.



Here, the object failure also occurs at some time after the fault but, in addition, the object is guaranteed to recover (Ro) at some time after the fault has been repaired (Rc).

Stable objects are the most reliable class of objects. Objects which are stable with respect to a fault class F are guaranteed to eventually recover from a failure without regard to whether the offending fault ever gets repaired. The distinction between resilient and stable objects is that the recovery of resilient objects is contingent upon the eventual repair of the underlying fault while stable objects are guaranteed to recover even if the underlying fault is never repaired. A timing diagram representation of the behavior of stable objects might be as follows:



The repair of the faulty component is shown to occur at some time after the object has recovered although, strictly speaking, it is not necessary that the repair be accomplished at all.

2.5.3 An Example Configuration

Our model of a distributed C2 system is a graph of nodes (representing processors) connected by links (representing communications paths). Associated with each node will be a number of objects and a number of physical resources (such as memory, disks, processors, etc.).

Each object in the model requires access to some other objects and/or some resources. Since we are modeling an object-oriented system, all of the resources which are directly required by an object will be local in the sense that they will be associated with the same node as the object. This, of course, is not the case with conventional data management systems which may require access to remote resources as well. An object may, however, depend on another object which is remote and thus depends in turn on remote (with respect to the first object) resources.

For illustrative purposes, the granularity of reliability requirements is fixed at the object level. That is, reliability constraints are discussed at the level of the objects in the system rather than operations on objects (a finer granularity) or object types (a coarser granularity). Note that processes are objects which are dependent upon access to other objects.

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

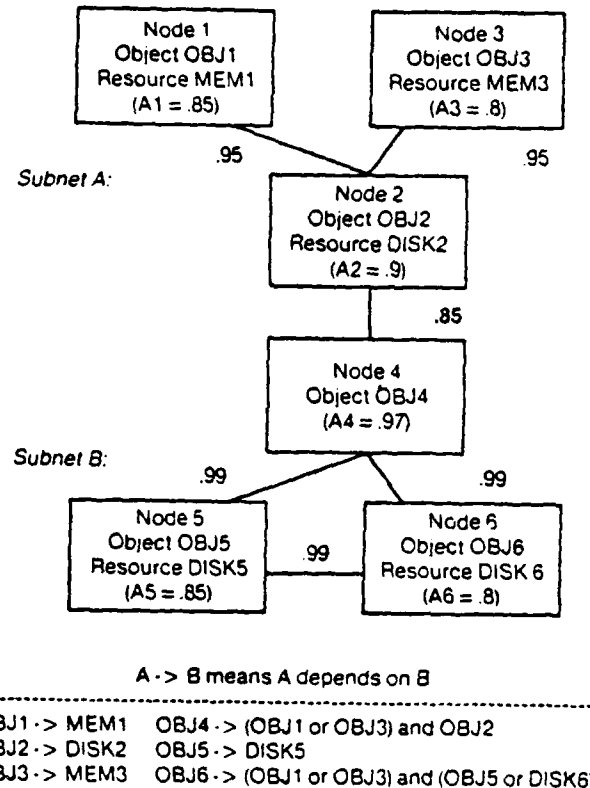


Figure 2.5: Example Model

The system in Figure 2-5 is used as an example throughout the discussion. It consists of six nodes connected as two sub-networks of three nodes each. Subnets A and B are connected by a single communication channel between nodes 2 and 4. Hypothetical probabilities for the availability of each node and each link are given in the figure. For this example, we will define six simple fault classes which will be referred to in the section on specifying reliability constraints. These six fault classes are collections of single, repairable faults of the components in the model of Figure 2-5.

Example fault classes:

- R - Single Resource faults (it assumed that resource faults are more or less equiprobable)
- NA - Single Node faults in subnet A
- NB - Single Node faults in subnet B
- LA - Single Link faults in subnet A (it is assumed that both subnets are local area networks so that a subnet fault means that all the contained links are broken)
- LB - Single Link faults in subnet B
- P - Network partition faults (this is caused by faults in Node 2, Node 4, or the link between them)

Each of the nodes has associated with it a single object called OBJn and some of the nodes also have local resources defined for them. The dependencies of these objects upon other objects and resources are shown at the bottom of the figure. Objects 1, 2, 3, and 5 depend only on local resources. OBJ4 depends on three remote objects while OBJ6 depends on three remote objects as well as a local resource.

In the extended example of Figure 2-5, it is assumed that the resources MEM1 and MEM3 are ordinary RAM memories which therefore have an ordinary volatile nature. Therefore, the objects OBJ1 and OBJ3 which depend on these resources will behave as volatile objects with respect to the fault classes R and NA (Resource faults and Node faults in subnet A). That is to say that any fault in class R, a single resource fault, might cause a permanent failure of either OBJ1 or OBJ3. Likewise, a single node fault within subnet A (i.e., in the class NA) could cause a similar failure.

The resources in the example which are named DISKn are taken to be non-volatile resources; thus, OBJ2, which depends solely on the resource DISK2, can be expected to be classified as a non-volatile object with respect to single resource faults (class R) and to single processor faults in subnet A (fault class NA).

In the Figure 2-5 example, OBJ4 is dependent upon OBJ1, OBJ3, and OBJ2 in a way which can be described with the predicate [(OBJ1 or OBJ3) and OBJ2]. A single resource fault in either MEM1 or MEM3 would therefore have no effect on OBJ4 since it requires access only to one or the other of objects 1 or 3. A resource failure in DISK2, however, would cause OBJ2 to fail and, in turn, would cause the failure of OBJ4. Consequently, OBJ4 must be classified as only non-volatile with respect to fault class R. However, since OBJ4 depends only on non-local objects, it is sensitive to link faults as well as resource and node failures of its component objects. For example, a fault in the link connecting nodes 2 and 4 would prevent OBJ4 from accessing any of its dependent objects, thus resulting in a failure of OBJ4. If and when the link is repaired, then OBJ4 would immediately regain access to the other nodes and thus could recover from the failure; therefore, OBJ4 can be considered to be resilient with respect to fault class P (network partition faults).

The simplest examples of stability with respect to a fault class is the case where an object is dependent only on local resources as in OBJ1 and OBJ3 of Figure 2-5. In cases such as these, faults such as those in classes LA, LB, and P can have no effect on the object; therefore, the object is stable with respect to these classes of faults since the repair of the object can be thought of as occurring immediately.

As a non-trivial example of stability, consider the object OBJ6 in the figure. This object, like OBJ4, is dependent upon remote objects and, in addition, also depends on a local, non-volatile resource, DISK6. The predicate describing this dependency is [(OBJ1 or OBJ3) and (OBJ5 or DISK6)]. Now since a single resource fault can effect only one of MEM1, MEM3, DISK5, or DISK6, and since the dependency predicate is such that OBJ6 will not fail as long as at least three of the four components are available, then OBJ6 can be said to be stable with respect to fault class R, single resource faults. Notice, however, that due to the highly distributed nature of this object,

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

OBJ6 can only be resilient with respect to each of the other defined fault classes.

The following table summarizes the observed reliability classes of each of the six objects defined in Figure 2-5.

	OBJ1	OBJ2	OBJ3	OBJ4	OBJ5	OBJ6
Volatile	R,NA		R,NA			
Non-Volatile		R,NA		R,NA	R,NB	
Resilient				NB, LA, P		LA, LB, P, NA, NB
Stable	LA, LB, P, NB	NB, LA, P, LB	LA, LB, P, NB	LB	NA, LA, P, LB	R

2.5.4 An Iterative Approach to Specification

Both the numerical and the discrete approaches to reliability specification have their advantages and disadvantages. The discrete approach seems to be conceptually more natural and might therefore make it somewhat easier for an object "designer" to use initially. The numerical approach is inherently more accurate and encompasses all the factors relevant to the reliability behavior of objects.

Within the context of a distributed command and control system, the correct way to specify the desired reliability of objects may turn out to be a synthesis of the two methods. In this way, a user could initially estimate the desired class of an object by making essentially qualitative statements such as "object A should be resilient with respect to combat scenario X (and its related fault class) and stable with respect to scenario Y". These specifications could then be formalized and given to an analytical tool such as NetRAT (see Chapter 8) which would convert them to quantitative specifications of MTTF and availability estimates for the object. These numbers could then be fine tuned by the administrator or user and further iterations could be made until the mechanisms and parameters selected for the object could provided the intended level of immunity from the classes of faults defined in the fault model.

Such iterative specification procedures can most likely be carried out only on very high level objects such as the user level functions and procedures. The requirements could then be percolated downward to the lower level objects. The requirements for a number of such high level functions could then define a set of constraints on the lower level objects.

2.6 AN OVERVIEW OF THE DESIGN STEPS

Our approach to designing reliable distributed command and control systems consists of the following steps:

- (1) Characterization of the objects managed by the system for the application environment. This includes their structure and the operations to be performed on them.
- (2) Characterization of the physical (geographic) structure of the system as dictated by the application environment. This includes identification of various sites where the computing resources are to be located based on various factors such as administrative organization, geographic locations, and the military and political situations. Also included, at this point, is the expected workload on the system. The workload characterization consists of identification of the kinds of transactions that are to be executed by the system on the application-defined objects and the rate of transaction requests originating from the various sites in the system.
- (3) Once the system's physical organization and the workload is defined, the next step involves specification of the various performance and reliability requirements. The performance requirements are in general stated in terms of the response time (or throughput) of the application transactions. If required, these statements may be further refined in terms of performance characteristics of operations on certain application-defined objects. These specifications may be further qualified on the basis of the performance requirements relative to a particular site. The reliability requirements can be specified using either the statistical quantities such as MTTF and availability, or in terms of certain reliability levels under a set of faults. These measures can be specified for a given operation on an object. The faults refer to the unavailability of the components in the system architecture identified in step (2) above. The reliability requirements can be specified either for an object or for some operation on an object; the reliability requirements for an operation can be further specified for invocations from some specific sites or from all sites in the entire system.
- (4) The next step involves characterization of faults in the system along with the failure characteristics of the components such as the processors, the disks, and the communication links.
- (5) The next step is to define the system architecture to support the management of the application-level objects. This will introduce some new objects in the system to support the operating system functions. In the most ideal case this architecture will be based on a set of hierarchical functional layers. An example of such a layered approach is presented in Chapter 5. The hierarchical structuring will introduce the notion of object dependency as used in the example in Section 2.5. Also certain decisions regarding the replication or partitioning of an application-defined object will be made at this level. The reliability requirements will have to be suitably extended to specify the mutual consistency among the replicated copies or the partitioned components. The degree of replication will be determined based on the reliability of each copy and the desired level of reliability. The assignment of copies to different sites to optimize performance/reliability is a computationally complex problem. We do not address any optimization techniques in this guidebook; however, once a configuration has been

DESIGN ISSUES FOR RELIABLE DISTRIBUTED SYSTEMS

selected, the techniques presented in Chapters 7 and 8 of this guidebook can be used to evaluate its performance/reliability characteristics.

An integral part of the system architecture is the design of the communication network. Some techniques for designing survivable communications networks are presented in Chapter 6.

- (6) Object replication or partitioning will introduce a need for certain other reliability mechanisms such as atomic actions and commit protocols. Next follows the detailed designs of the object managers. This will require integration of suitable recovery mechanisms such as commit protocols, stable storage management, recovery logs, and intention lists, into the detailed designs. Chapter 4 presents a description of reliability techniques in distributed systems. Also, formal methods are required for the definitions of the detailed designs during this phase.
- (7) Once a detailed design has been defined, it is evaluated for performance, reliability, and functional correctness using the techniques described in Chapters 7, 8, 9 of this guidebook. In many situations these techniques would be applied to some isolated parts of the designs rather than to the whole design together. Performance evaluation requires definition of the workload under which the system is to be tested. Also required are failure characteristics of the system components. A number of iteration of steps 5 through 7 may be required before a satisfactory design is obtained.

The design steps described above are further elaborated in Chapter 10.

The following chapters describe some methods to design and analyze reliable distributed systems. The first part of this volume of the guidebook deals with the architecture of distributed systems and the reliability techniques for such systems. The second part is mainly devoted to formal methods for design specification, analysis, and verification. The integration of all these ideas is presented in Chapter 10.

CHAPTER 3

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

This chapter begins with a summary of our understanding of the functions and operational environment of command and control systems. The summary is based on RADC sponsored research and other published material [THOMP80, MARI80, GRIP79, DINN80, MCM182, MARC82]. This summary is used to define the functional properties and architecture of a distributed abstract machine that meets the needs of users of such systems. This high-level definition is to be the basis of later detailed technical work on the design and evaluation of recovery mechanisms in distributed operating systems for command and control systems.

Any command and control system must support four basic functions: communication, navigation, data collection and decision support. These systems can be divided into two broad categories, strategic and tactical. Systems in these two categories differ in the geographic scope of the system, their functional complexity and the mobility of the system nodes. Strategic command and control systems encompass a relatively large region of operations (roughly 500 to 1,000 miles radius). It maintains large long-lived databases and contains several smaller, and possibly tactical, command and control systems as its constituents. Tactical command and control systems are generally smaller in geographic scope; the distance between nodes is typically 10-200 miles. Nodes of these systems are relatively mobile - they can be moved and installed in a few days. The communications facilities that connect nodes of a tactical system are usually much less reliable than those used to connect the nodes of strategic systems.

Section 3.1 summarizes the major functions of command and control systems.

Section 3.2 summarizes the architectural and functional requirements of distributed operating systems for supporting command and control applications.

Section 3.3 presents a scenario of operation in a hypothetical distributed command and control system. This scenario is later used for evaluating some parts of the distributed operating system presented in Chapter 5.

3.1 COMMAND AND CONTROL SYSTEM FUNCTIONS

The important data processing functions performed by a command and control system are described in this section. The data processing functions support the C2 activities of planning, directing and controlling.

The general goals for the data processing systems in a command and control system are to:

- o Make information available to the users who need it.
- o Improve the response time of time-sensitive operations.
- o Support the database needs of the users.
- o Make available global databases which are needed for planning, coordination, threat assessment, targeting, intelligence production and status monitoring.
- o Provide reliable dissemination of messages carrying requirements, commands, warnings and status information.
- o Provide extensive degraded mode operating capability.
- o Provide enhanced survivability and continuous operation under the loss of C2 system components.
- o Support multi-user multi-level security of information.

Efficient database sharing is the most crucial requirement of command and control systems. A command and control must support global logical objects for the following kinds of information: weather, personnel, logistics, enemy situation, friendly situation, surveillance and identification, warnings and alerts, mission status, tactical air support requests, etc.

The major role of the data processing functions performed by a command and control system are concerned with maintaining this data base and providing timely and accurate reports using the database.

3.1.1 Operational Environment

Because of the evolutionary nature of future distributed C2 systems, it is desirable to adopt an approach which permits relatively easy changes for system expansions, capacity upgrades, functionality upgrades, hardware substitution, and addition of new elements. The approach of modular system design should also help in rapidly configuring new systems.

Instead of designing systems to meet certain specific requirements, it is desirable to provide an architecture which can adapt easily to the long term changing requirements due to the state of the technology and the world-situation, as well as the short-term changes in requirements due to the tactical environment. One can use the following features of distributed command and control systems to characterize their operational environment.

Physical Environment:

A single command and control system can consist of tens of shelters. The distance between the shelters can range from a few miles to a few hundred miles. The geographical dispersion serves to increase the field of view or to

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

provide higher survivability to the command centers by locating them in rear areas. The user population of a TACS is usually hundreds of people.

Communication between the shelters can be implemented with microwave or radio frequency channels. In some cases, where the distances are not too large, coaxial cables or fibre-optics cables may be used. The command centers are high value target, and placing them in the rear areas for survivability reasons will decrease the performance because of communication delays.

Most of the intershelter communication consists of command messages, and database updates and query messages. Most of the other data processing requirements of a shelter will normally be supported by the resources colocated within the shelter.

Most of the important databases critical to the command and control operations are maintained at the command centers. To support continued operations in the event of the loss of a command center, another center must be able to reconstruct the database from the replicated components of the global database.

Threats from opponent espionage operations demand that all the intershelter as well as the intrashelter communication be encrypted. It is desirable to digitize and encrypt voice communication.

Communications Environment:

The communications network for a command and control system must support intrashelter and intershelter communication. The networks supporting the intrashelter communications will be referred to as mininets and those supporting communications between shelters will be called maxinets. Thus a maxinet connects several mininets in a C2 system. The maxinet topology will change dynamically because mobile units will be moved in response to the tactical situation. The length of the mininet communication link can range from a few meters to hundreds of meters. The maxinet links will be up to hundreds of miles long. The communication bandwidth for intershelter communication ranges between 10-50 kb/sec, and that for intra-shelter communication ranges from 1-10 mb/sec. In the following discussions the term "cell" is used to refer to a mininet and the resources connected to it.

This communications network must be connected to external elements such as the World Wide Military Command and Control System (WWMCCS), Intelligence Data Handling System Communications (IDMSC) and the Defense Communications System (DCS).

Some of the biggest problems which will affect the communications system performance are electronic warfare, self-jamming, and the loss of nodes (mininets). Network partitioning, node drop-out, node reunion, network reconfiguration are some of the problems which the designers must address.

3.2 DISTRIBUTED C2 SYSTEMS

3.2.1 Required DOS Functions

A distributed operating system for command and control systems must provide the following functions:

- (1) Directory services to keep track of users, software modules, and data.
- (2) Allocation of resources shared by multiple nodes such as communication services, global data bases, excess capacity (terminals, backup processors, input/output devices).
- (3) Scheduling of tasks involving interprocessor interaction.
- (4) Access to global system software.
- (5) Performance monitoring (processor status, communication service status, interprocessor interaction status, user status, device status, resource utilization).
- (6) Degradation handling and system recovery (error detection, error reporting, fault isolation, restart, and reintroduction of failed unit after repair).
- (7) Interprocessor communication (event occurrence, data mapping, data transmit and acknowledgement, remote terminal access).
- (8) Multi-level data security, access control, release control and audit trail.
- (9) Configuration management.

3.2.2 DOS Functional Requirements

In the previous section the basic required functions of a distributed operating system (DOS) were described. This section presents more detailed requirements for the DOS functions. These functions include:

- A. Interprocess Communication
- B. Resource Management
- C. Security
- D. Configuration Management
- E. Database Management

Interprocess Communication: The DOS interprocess communication must support the following functions:

- o Communication between processes or user task groups located within the same shelter or different shelters.
- o Database query and update across the maxinet.
- o Remote invocation of functions or services across the maxinet.
- o Status query and other status management communication between operating system processes.

In addition, DOS interfaces that facilitate network error handling, and dynamic allocation of bandwidth must be defined.

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

Resource Management:

Resource management functions of the DOS maintain directories of global resources distributed over the maxinet. These directories are used to direct the service requests to appropriate servers.

The resources managed by the DOS include database objects, system service processes, processors, I/O devices, secondary storage devices, and other functional units located within the cell and connected to the mininet. The resource management functions of the DOS must support:

- o Location transparency in accessing the resources.
- o Uniform mechanisms for accessing resources of different types.
- o Prioritized scheduling of tasks and allocation of resources such as processors to tasks
- o Handling of remote service requests within the cell
- o Handling of remote service requests to/from the other cells
- o Detection and handling of error conditions
- o Resolving deadlocks
- o Concurrent operations on shared resources
- o Protection among users and tasks.

The DOS interfaces for resource allocation will direct requests for an external resource to the appropriate DOS functions. Name/directory management is an integral part of resource management and resource sharing in distributed systems. The names are required to identify the resources as well as the users in the system, and to provide appropriate protected environments to the users. The name/directory management supports locating the resources and users in the system and directs the service request to the node where the resources or the user are currently residing.

Security:

Analysis of existing information exchange practices and patterns in the command and control environment shows that there exists a substantial requirement for generation, storage, processing and transfer of information elements of diverse levels of security classification. Further, information must be accessed by users at multiple levels of clearance in compliance with a well defined security policy.

At the global level, the DOS is essentially concerned with direct transfer of information between objects and processes. To provide secure transfer of classified information, the design relies on these basic concepts:

- o Use of a well defined, dynamically reactivated authentication procedure performed by each process to identify other processes, and to eliminate the possibility of impersonation by an unfriendly agent.
- o Transfer of information to other processes using a cryptographic approach dependent solely on the nature of the security classification of information and on the potential existence of a relevant need-to-know by a user of the information-receiving cell.

- o Use of cryptographic schemes that vary dynamically with time so as to assure classified information protection even in the event of enemy capture of a complete cell and its computational resources.

In a command and control system, it must be possible to prevent both cell impersonation by an unfriendly agent and disclosure of information to passive elements able to listen to network transactions between cells. Dynamic reactivation of authentication procedures between cells is a complexity required to minimize secure information loss in the event of enemy takeover.

Configuration Management:

The need to manage a time-varying supply of computing resources has already received Air Force recognition in the form of the development of the concept of a "rolling force package". This concept is specifically aimed at assuring dynamic modification of the command and control system configuration in response to both changes in the specific missions performed and the changes in the tactical as well as strategic situations.

At the information processing level, the changing nature of processing requirements and environment is translated into the need to deploy a system capable of continuous and efficient reconfiguration of its processing, storage and communication resources.

Easy reconfiguration is required at both the cell and global levels. At the cell level, local configurations may require modification to be responsive to dynamic variations in the processing and storage workload. At the global level, cells may join or leave the maxinet as the tactical organizations they support become operational, inoperative or perform physical movements in the tactical field. Further, cells may become connected or disconnected from other cells during processing; therefore, the maxinet configuration perceived by one cell may be radically different from that perceived by others.

The DOS must be concerned with global management of unreliable resources through unreliable, noisy, low capacity channels. It must choose from an extensive set of potential decisions which must be evaluated using incomplete and/or ambiguous data. Decision algorithms should provide a rational basis to perform independent resource allocation, possibly opting to signal intentions or demand additional information from other cells. In the global context, reconfiguration is very much a cell dependent concept which expresses the variations in global resource availability perceived by each cell. At this level, reconfiguration issues are related to the correctness of each cell's view of the world and its consistency with other cell's views (coherence). The nature of the command and control environment makes development of a correct consensus among cells an ideal goal achievable only when perfect environmental conditions occur.

On the other hand, at the local level, it can be assumed that all the resources required to assure correct and consistent perception of cell status (i.e. consensus) by all its processing elements are available (together with all the potential advantages to be gained by their efficient usage). At this level, the design must provide for internal cell reconfiguration with minimal disruption to ongoing processing.

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

At the global level, configuration management activities include:

- o Cell union to maxinet
- o Cell secession from maxinet
- o Maintenance of global resource directories
- o Primary reallocation of data elements to cells
- o Reallocation of backup responsibilities to cells.

At the local level, configuration management activities include:

- o Maintenance of resource directories
- o Local status management (List of nodes in the cell and their status, status management on start-up and shut-down of nodes in the cell, failure detection of nodes, performance status)
- o Interconnection management (routing tables)
- o Primary-backup allocation of processors to functions (tasks)
- o Database to storage device mapping
- o Test and diagnosis of nodes in the cell.

Database Management:

For the tactical as well as strategic command and control systems, the desire for an integrated data base will lead to data being partitioned into multiple files, distributed locally within a local network and distributed geographically across shelters. Information which will be shared across shelters will include status of forces, availability of allied forces, combat plans and frag-orders, in addition to general system status data. The requirements for consistency within each such object must be completely ensured. However, mutual consistency across multiple copies of an object at several shelters does not have to be absolute; i.e., the data may be hours old and still be acceptable as long as the age of the information is known.

The current TACS approach to maintaining replicated information can be visualized as a time stamp (albeit manually via telephones) approach. The major concerns have been the age of the data upon which planning operations are developed and accessibility of data to support crisis C2 operations. Automation of some of the data collection and dissemination offers one mechanism for enhancing the speed and accuracy of data collection. While distribution and automation of the local data base maintained at each cell offers potential for improving the timely access of information for planning and crisis management, the distribution of data across the shelters offers the potential for higher survivability.

The low capacity, unreliable characteristics of communication through the maxinet precludes the use of approaches requiring the definition of a "control consensus" between the different cells before an actual allocation decision is made. In these approaches, extensive coordination messages must be exchanged between the cells, thus requiring existence of fast, high capacity, high availability communication channels.

The same arguments indicate that approaches assuring consistency of multiple distributed copies of a database require existence of information

exchange capabilities which will not be available at the global level. Consistency, as well as some of the other desirable system properties already discussed, must be considered to be an ideal goal generally attainable only to a limited extent, which is related to the availability of computational and communication resources. Thus, a whole spectrum of possible degrees of consistency attainment must be considered whenever user perceived characteristics of the system are discussed. This quality assessment approach is markedly different from that used for existing data base management systems which, to a large extent, are characterized in terms of strong requirements of mutual consistency.

The database management functions should address the following areas:

- o Creation of new data types and transactions
- o Creation and deletion of database objects (remote as well as local objects)
- o Database partitioning and replication
- o Dynamic relocation of database objects
- o Uniform interfaces to access and update the database distributed across the mininet or the maxinet
- o Probabilistics algorithms to address the consistency issues
- o Error detection and handling techniques to deal with the following problems
 - node failures in a mininet
 - loss of a cell
 - intermittent communications failure in the maxinet
 - partitioning of the maxinet or the mininet
- o Query decomposition.

3.3 AN EXAMPLE SCENARIO FOR DESIGN EVALUATION

In this section we present an example scenario using the organization of a typical tactical command and control system. We will be using this example as a basis for evaluating the Zeus system design. A command and control system consists of four basic elements, sensors, navigation aides, command centers and communication facilities that are used to support its function of providing communication, navigation, data collection and decision support to Air Force personnel so that they can efficiently and effectively deploy defense resources.

The major activities supported by a command and control system are:

- o mission planning
- o mission monitoring
- o surveillance and identification
- o air traffic control and navigation
- o aircraft weapons control
- o logistics
- o operations management
- o coordination with other headquarters

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

These activities require the acquisition processing and distribution of information related to:

- o weather
- o personnel
- o weapons
- o intelligence
- o airspace control
- o enemy positions
- o terrain descriptions
- o other command related messages

3.3.1 Organizational Components of a Tactical C2 System

The major organizational components that use a tactical air command and control system are located in different shelters which are connected by unreliable, low-bandwidth communications facilities. These organizations and their functions are described in this section.

The Tactical Air Control Center (TACC) exercises operational control over all operations and resources associated with the command, plans and monitors missions, communicates with Air Force headquarters, plans tactical air defense operations and provides support to Army operations. All data related to operations, e.g., available resources, weather data, intelligence information, airspace and ground deployment information, geographical information, etc., must be available in this organizations shelter.

The TACC is a high value target of opposing forces and it must be possible to move TACC functions to another shelter if the original shelter and its equipment are destroyed.

The Control Reporting Center (CRC) and Air Traffic Regulation Center (ATRC) use radar to control in-flight aircraft and to perform surveillance within assigned areas of a combat zone. This center provides air situation data to TACC and may operate as an alternate TACC. The center's activity consists of collecting, displaying, evaluating, and distributing air traffic activity information. Such a center may be duplicated for survivability, but at any given time only one center has operational responsibilities. The distance between the TACC and a CRC may range from a few miles to about 200 miles.

Control Report Posts (CRP) work under the Control Reporting Center. They provide radar control and surveillance within assigned areas forward of the CRC to extend coverage and provide a link to Forward Area Control Post (FACP). A Control Reporting Post may communicate with other CRPs.

Forward Area Control Posts (FACP) with radar and communication elements are assigned to a Control Reporting Post for surveillance in forward combat areas. They also provide radar control of missions to and from control points for hand-off to Forward Air Controllers (FAC). An FACP can be a mobile unit.

An Air Support Operations Center (ASOC) operates with the Army Tactical Operations Center (TOC) to direct air support operations to support the Army's requirements. Several Tactical Air Control Parties (TACP) work for an ASOC. Several ASOCs can be deployed within the control of a TACC.

Tactical Air Control Parties (TACP) consist of Air Force personnel with mobile communications. These may also be attached to Army command posts as Air Liaison Officer (ALO). These units gather information on enemy positions and communicate this information to ASOC. They also act as a Forward Air Controller (FAC) in front lines.

A Tactical Unit Operations Center (TUOC) or Wing Operations Center (WOC) receives combat plans from TACC and ALCC; acts as an air operations center for tactical unit headquarters. TUOCs plan for and control the use of aircraft weapons.

An Airlift Control Center (ALCC) may operate as a separate element or as an integral part of TACC. Airlift requests may be for equipment delivery, personnel movement, resupply, medical evacuation, etc.

A Sensor Reporting Post (SRP) acquires, processes, and disseminates ground sensor data. Important information is sent to TACC.

A Tactical Airborne Warning and Control System (TAWACS) is used to extend radar and radio ranges beyond that available with ground-based reporting posts. Since it may be much more difficult for an opponent to destroy an aircraft, these systems enhance the survivability of the reporting units.

Communications Centers: are the switching units that handle the communications for the TACC, CRC and the CRPs. Associated with these are electronic telephone central offices, and teletype communications centers.

Figure 3-1 shows an example distributed command and control system. In this scenario we have four shelters which are completely connected by a long-haul network. Tactical Air Command Center (TACC) represents the hub of the tactical system. Control Reporting Center (CRC) performs data collection and other functions related to communications, navigation and intelligence. Squadron 1 and Squadron 2 are the air squadrons for supporting tactical missions. A number of FACPs communicate with the Control Reporting Center to access and update the databases stored there. Each shelter has a local area network connecting its computing resources. For the purpose of this scenario we have identified six databases along with their consistency requirements and distribution. These databases are described below. For the purpose of our evaluation, instead of concentrating on the actual structure and the contents of these databases we focus on their intent, replication, consistency requirements, and the access patterns by various transactions in the system.

The replicated databases in the system would be updated in two modes: weakly consistent and strongly consistent. In general, most of these databases would consist of a number of records ranging from thousands to millions. For the purpose of updating, only a reasonably small portion of the database would be locked by a transaction. In most situations, updating the database would involve inserting new records or changing certain fields of

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

some existing records. Generally, the information stored in this database would be marked with some time-stamp and validity constraints. Under the weakly consistent updates, each copy would be updated using a separate transaction, whereas in the strongly consistent mode of updating either all or a majority of the copies would be updated within the same transaction. In cases of duplicated copies, availability of only one copy would suffice for the purposes of reading or updating.

INTELLIGENCE: This contains all plausible intelligence information. This database is stored at two locations, the TACC and the CRC. In most cases this database would be updated in the weakly consistent mode. Most of the updates and queries would originate at the TACC and the CRC. The squadrons will be mostly querying this database.

MISSION PLANS: Copies of this database are located at TACC and CRC. These copies would be always updated in the strongly consistent mode. Queries to this database would occur from all locations in the system; updates to it will be made either at the TACC or the CRC (when it is acting as a backup for the TACC).

WEATHER: This database is kept at the TACC and the CRC; all updates are made in the weakly consistent mode. Queries to this database may originate from any location in the system.

SUPPLIES: This contains the available resources in the system along with their current status and the commitment. This will include, for example, fuel, food, spare parts, ammunition etc. The copies of this database are stored at TACC, SQD-1 and SQD-2. This database, in most situations, will be updated in the strongly consistent mode. However, we also envision some transactions that will modify the database in the weakly consistent mode. This database would be updated during mission planning and normal day-to-day operations.

SQDB (Squadron Database): This would include the number and the types of aircraft in the squadron, their status and commitment, and the available personnel for mission operations. Copies of this database are stored at the squadron base and the TACC. This database will always be updated in the strongly consistent mode. Updates to this database would be made during mission planning when a set of aircraft is assigned to a mission.

NAVIGATION: This database contains navigation and air-traffic related information. Updates to this database would be made during mission planning and the normal operations. Most updates would be made in the strongly consistent mode.

MISSION PLANNING TRANSACTION:

The following is a generic mission planning transaction. This type of transaction will be executed either at the TACC or the CRC. A typical structure would be as shown below:


```

Begin Transaction
  Read INTELLIGENCE database (large number of records accessed)
  Read SQDB-1
  Read SQDB-2 (both squadrons are involved in the mission)
  Read SUPPLIES
  Read WEATHER
  Computation
  Update MISSION PLANS
  Update SQDB-1
  Update SQDB-2
  Update SUPPLIES
  update NAVIGATION
  Using MTM functions, send order messages
End Transaction

```

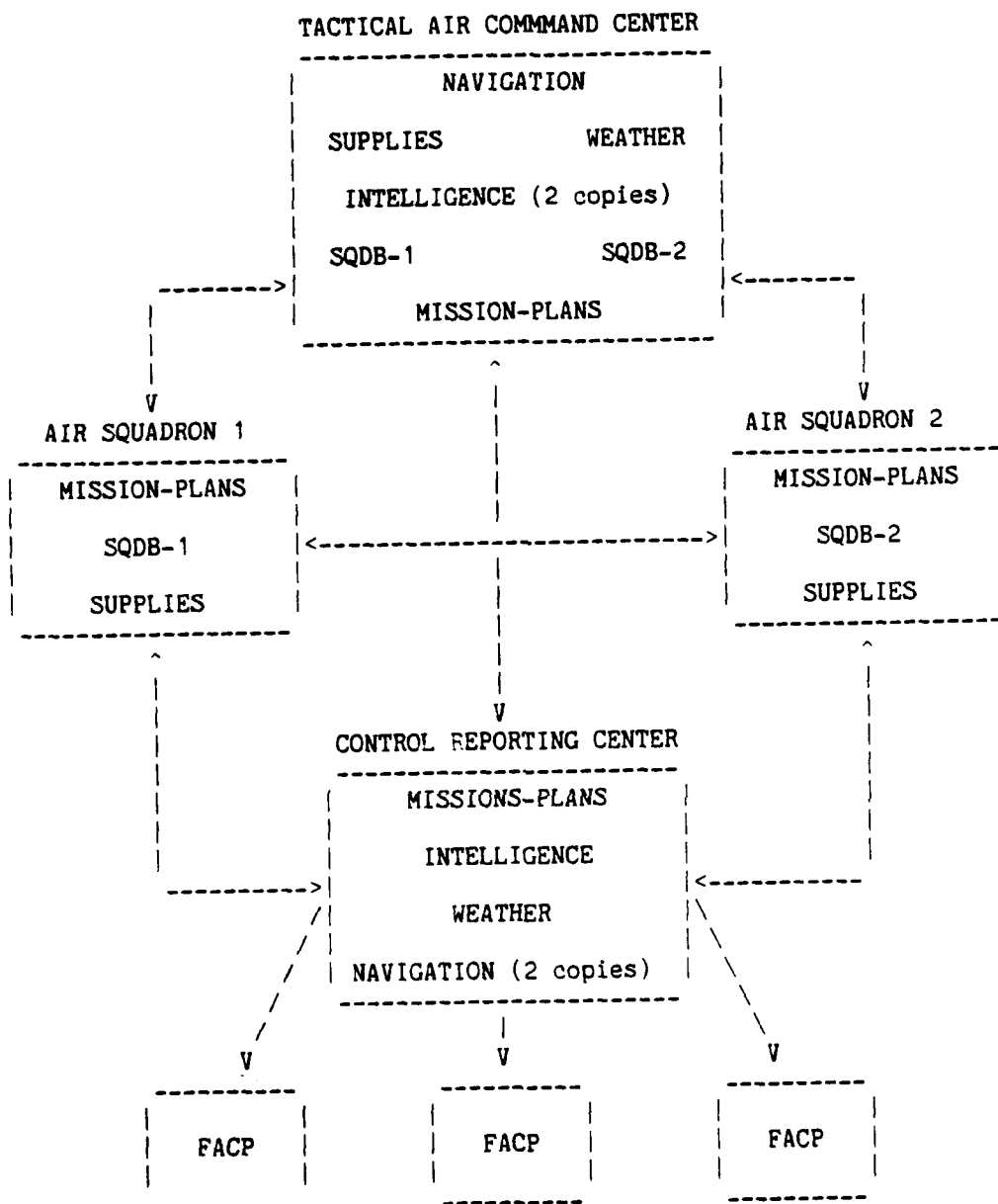
Some variants of the above transaction can be defined to represent planning of large and small missions in the system. For example a smaller mission plan may involve only one squadron. In the body of the mission plan transaction one may use the MTM facilities to send messages containing orders.

The Table 3-1 shows the types of operations performed by the various shelters in this distributed air command. At this point the frequency of these operations and the failure characteristics of the shelters and the sites within a shelter needs to be specified.

3.4 Conclusions

In this chapter we have reviewed the salient requirements for distributed operating systems for command and control applications. The important goals of such systems are to support flexible sharing, updating and retrieving of global information, to reliably provide computing resources for information processing tasks, and to provide a secure environment for the users and the information. Considering the evolutionary nature of such systems, it is essential that the operating systems for distributed C2 system provide powerful, yet convenient, means of introducing new functionalities, new hardware and software into the system. The operating system should provide some basic "building block" functions for expansions and growth. Most importantly, the operating system should also provide support for maintaining consistent system state in the event of failures and loss of resources. Constructing systems as a collection of objects and defining appropriate consistency management techniques to suit the requirements of each object type in the system looks like an attractive approach to this problem.

DISTRIBUTED COMMAND AND CONTROL SYSTEMS



An Example Distributed C2 System
Figure 3-1

Table 3-1

SHELTER	TRANSACTION TYPES
(1)	TACC Mission Planning, Update WEATHER, INTELLIGENCE, SUPPLIES NAVIGATION Query SQDB-1, SQDB-2, NAVIGATION INTELLIGENCE, WEATHER, SUPPLIES Send command messages (local and remote) Send Alert and Emergency messages to all other shelters
(2)	SQUADRON Update SQDB-1 and SUPPLIES due to daily operations Query of WEATHER, INTELLIGENCE NAVIGATION, MISSION PLANS, other SQDBs Send command messages (mostly local) Send command messages to TACC
(3)	CRC Mission Plans and most of the other TACC functions as a backup, Update WEATHER, INTELLIGENCE, NAVIGATION Send Alert and Emergency messages Send command messages to FACPs
(4)	FACP Query NAVIGATION, WEATHER, MISSION PLANS, INTELLIGENCE Update INTELLIGENCE Send Alerts and Emergency messages

CHAPTER 4

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

The architectural features of distributed systems offer a great potential for designing reliable systems. This potential is derived from the fact that in a distributed system there is a physical isolation between system components, which tends to reduce correlation among component failures, and there is redundancy of resources to support continued operations in the presence of component losses. However, this potential has largely remained unexploited because of the lack of a formal discipline to integrate the existing and known recovery techniques into the designs of distributed systems. In this chapter we describe the techniques for reliability and consistency management in distributed systems. We also present an object-oriented design model for distributed systems which facilitates a systematic and well-structured integration of known recovery and consistency management techniques into the designs of distributed systems.

Object-oriented designs offer an attractive approach to constructing reliable systems to confine errors in the system, to define a consistent system state to support rollback and restart, and to limit propagation of rollback activities in concurrent systems. In the object-oriented approach, a system is viewed as a collection of objects that are accessed or updated by users through transactions. A transaction is defined as a sequence of primitive operations on a set of objects. A transaction is viewed as a unit of error recovery and synchronization in the system. The key principles for designing reliable systems are the atomicity of transactions and a sufficient level of redundancy in the system to support continued operations in the event of loss of objects (system components).

Another problem, which is functionally orthogonal to recovery, is concurrency control in distributed systems. The maintenance of recoverable consistent states of objects is essential for system recovery. The techniques to solve these two problems in a design interact closely with each other. An additional problem that arises due to concurrency in distributed systems is the propagation of rollback activities that can lead to a cascade of rollbacks known as the domino effect.

The consistency requirements in a distributed system are characterized by four distinctly different criteria. The first criterion is the internal consistency of the data. This refers to the semantic integrity of the data in the system. The second criterion is related to the consistency among replicated distributed data. This is called mutual consistency. One example

of a mutual consistency requirement is that all copies of a replicated data converge to the same value sometime after the updating of data is stopped. The third criterion for consistency is referred to as the external consistency. This refers to the consistency of the system interactions with the users. For example, if a user invoking a transaction is given a response indicating successful completion, then the updates made by the transaction must be reflected in the database. The external consistency requirements are dependent upon the definition of the user-system interface. Interactive consistency [LAMP82], the fourth criterion, requires that all correctly functioning nodes in the system have an identical view of the system despite the malfunctioning of some nodes. Interactive consistency requirements are defined as the Byzantine Generals' Problem described below.

In a system where a set of processors communicate with one another to make some common decision, the Byzantine Agreement requirement states that:

- (1) All non-faulty processors make the same decision.
- (2) A small number of faulty processors can not cause the non-faulty processors to make different decisions.

It has been shown in [LAMP82] that the algorithm for achieving the Byzantine Agreement in a system functions correctly only if the total number of processors in the system is larger than three times the number of faulty processors in the system.

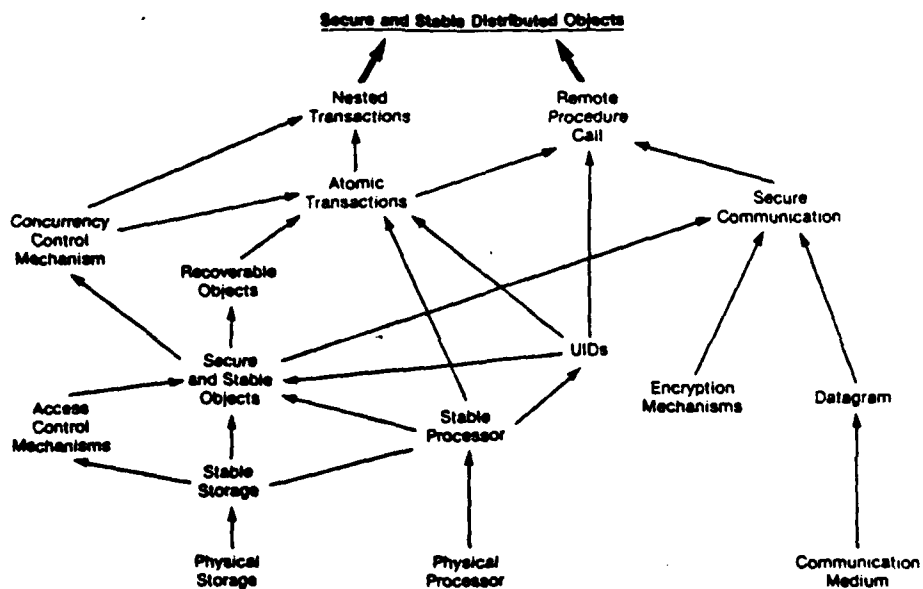
In distributed systems, the operating conditions that may arise due to concurrency and component failures strongly influence the consistency management techniques. This is shown in Figure 4-1. Concurrency of operations requires techniques to maintain mutual, internal, and external consistency requirements. Depending on these consistency requirements, serializability of transactions may be a necessary requirement for the consistency management techniques; therefore, in Figure 4-1, the consistency requirements have been further divided into two classes: those that require serializability as a necessary requirement, and those that do not require serializability. The concurrency control techniques to ensure serializability are based on locking protocols or time-stamp based protocols. Most of the work in the area of consistency management has been in the context of maintaining serial consistency of distributed, replicated or partitioned databases [BERN81] [THOM79]. The term serializability means that the final effect of executing interleaved operations of concurrent transactions on the database is equivalent to some serial execution order of those transactions. Not many researchers have addressed the consistency requirements that permit non-serializable interleaved executions of transactions. Applications with such consistency requirements can be important if continued operations are to be permitted in spite of network partitioning.

The component failures have been divided into two classes: silent failures and malicious failures. Silent failure of a component means that the failed component does not generate or forward any information. In a malicious failure, the failed component may generate wrong messages or distort the

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Consistency Requirements and Consistency Management Techniques			
Operating Conditions	Concurrency of Operations		Component Failures
			Silent Failures Malicious Failures
Consistency Requirements	• Mutual • Internal • External		• Mutual • Internal • External • Interactive
Consistency Management Techniques	Serial Consistency	Non-Serial Consistency	• Logs/Audit Trails — REDO/UNDO logs • Commit Protocols • Checkpoint/ Rollback • Data-Patch and Log Transformation (under network partitioning) • Byzantine Agreement — Network Connectivity — Digital Signature and Authentication
	• Locking — Two-Phase — Tree Locking • Time-stamps Based Schemes • Optimistic Methods	• Semantic Analysis of Transactions	

Figure 4-1



A Design Model for Reliable Distributed Systems

FIGURE 4-2

messages it forwards. Silent failures of components affect the internal, mutual, and external consistency in the system. The techniques for maintaining system consistency under such failures are based on the concept of atomic actions. The implementation of atomic actions in distributed systems requires some commit protocols. Lampson and Sturgis [LAMP76], and Gray [GRAY79] independently introduced the concept of commit protocols to implement atomic actions on distributed objects in the presence of system crashes. The problem of interactive consistency arises in the presence of malicious failures of components. The solutions to such problems are based on the solution to the Byzantine Generals' problem.

Management of redundancy in the system in the form of replication of objects or creation of backup objects is important for supporting continued operations in the event of loss of resources. The major problem in redundancy management is the maintenance of consistency among replicated objects, and the maintenance of sufficient up-to-date state information with the backup modules to support reconfiguration. Several strategies are used to manage such state information. For example, some techniques keep a majority or a survivable set of the replicated units in a consistent state.

This chapter describes the reliability and consistency management techniques in distributed systems. Section 4.1 of this chapter describes an object-oriented design model for structuring highly reliable distributed systems. A system is viewed as a collection of objects that are accessed and modified by atomic transactions. Section 4.2 of this chapter describes techniques for consistency management under concurrent operations. The topic of concurrency control in distributed systems for serial consistency of transactions has been widely addressed over the last few years. It is only recently that the area of non-serial consistency has been explored [GARC83]. Section 4.3 presents recovery techniques that maintain system consistency under system crashes and component failures. In Section 4.4 the recovery and consistency management techniques described in Section 4.2 and 4.3 are integrated into the design model described in Section 4.1. Atomicity of transactions is based on constructing recoverable objects using multiple versions and commit protocols. These concepts are extended to nested transactions. The operations on distributed objects are performed as remote procedure calls. This requires implementation of remote procedure calls in a reliable fashion. The facilities of reliable nested transactions and remote procedure calls are used to synthesize distributed objects that are highly reliable.

4.1 A DESIGN MODEL FOR RELIABLE DISTRIBUTED SYSTEMS

A design model for the construction of reliable distributed systems is shown in Figure 4-2. This model has been inspired by Lampson's lattice model [LAMP81a] for reliable distributed systems. The objective of reliable distributed system designs is to synthesize secure and stable distributed objects that survive crashes of system components and support high availability of functions. Such objects are constructed using unreliable resources such as physical storage (disks), physical processors, and the communication media.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

In this section we describe the design model shown in Figure 4-2. The description is given in a bottom-up fashion. The model described here is one possible approach to designing reliable distributed systems. This model is particularly suitable for object-oriented systems in which interactions among objects are transaction-based. In the description presented here, we identify the properties at each point in the functional abstraction graph shown in Figure 4-2. In Section 5 we describe the application of various recovery mechanisms to achieve these properties at each level of this design model.

The physical storage refers to non-volatile disk storage that has a non-zero probability of information loss. For example, a page on a disk may get corrupted due to a head-crash or some other malfunction. Such failures can be characterized by reliability measures such as the mean-time-to-failure or a reliability function. Another problem with the physical storage is the non-atomicity of write operations on pages. For example, a crash may occur in the disk system during writing a new value on a page. This leaves the page in an undefined state because the old value has been destroyed and the new value has not been written completely.

The stable storage facility, which is constructed from the physical disk storage, provides atomic write operations on pages, and also enhances the availability of data by increasing its mean-time-to-failure.

A physical processor loses its control state data on crashes; a restart operation can only cause a process to execute from the beginning. A stable processor facility, on the other hand, supports saving of process states on the stable storage, and restarting a process from some previously saved process state. The operation of saving process states is called checkpointing. Processes are considered as objects that are supported by a stable processor facility.

In our model the system consists of a collection of objects each of which is of certain well-defined type. The object manager (which is an object of process type) for a given type manages all operations on the objects of that type. In addition to supporting operations associated with the type definition, the object manager for a type also supports creation of objects of that type, or the destruction of some existing objects of that type. A system-wide object called Type-Type Manager facilitates the introduction of new type definitions in the system. This approach is based on the principles followed in the design of Hydra [COHE75]. The Type-Type Manager object in our model corresponds to the TYPE-TYPE object in Hydra.

The next level of abstraction provides secure and stable objects based on stable storage, stable processor and unique identifier (UID) facilities. Stable objects are those that survive system crashes with a high probability and for which the primitive operations (i.e., the operations supported by the type definition) are atomic. Secure objects are protected objects which can only be accessed by authorized users.

Every object in the system is given a globally unique name using the UID facility. This name is never reused in the entire life-time of the system. From this unique identifier, the type of the object can be inferred. The UID

also contains the identification of the node where the object was created. Objects in the system may migrate from one node to another. The UID facility defines the logical name space in the system. Operations on an object are invoked by specifying the UID of the object and the operation name. Because the type of an object can be determined from the unique identifier of that object, operation invocation on an object is directed to the appropriate object manager for that type. The operations on the remote and the local objects are invoked in an identical fashion. It is for this reason that we find the remote procedure call paradigm a convenient abstraction.

The UID generation is based on the stable storage and the stable processor facilities. The UID generation facility is based on a local clock process or a sequence counter that uses the stable storage to survive system crashes and to ensure that the same UID is not regenerated on restart of a node after a crash. The UID for an object indicates the type of the object and the node where it was created. A scheme for generating UID in a reliable fashion is described in [SCHA83].

The abstraction of recoverable objects provides mechanisms to restore the state of an object after having made some changes to it, or to commit a change to the object state. The concept of commitment forbids any restoration to states before commitment. Commitment of a change to an object essentially implies permanence of the changes made to the object since the last commit operation on it.

We use the concept of immutable versions to implement mutable recoverable objects. An immutable object is one that is never changed once it is created, i.e., every change to an object creates a new object. In our model every change to an object creates a new version of that object; this version is uniquely identifiable by using the UID of the object and the version number. These principles have been discussed in detail in [REED78] and [SVOB81].

Atomic transactions are implemented using the facilities described above and some concurrency control mechanisms. A transaction should be atomic in the presence of concurrent operations and system crashes. Atomicity of concurrent transactions requires suitable mechanisms for concurrency control. There are basically three distinct approaches to concurrency control: locking protocols [ESWA76], time-stamp based schemes [BERN81], and optimistic techniques [KUNG81]. Recoverable objects support schemes to achieve atomicity of transactions in the presence of system crashes. Transactions in our model are treated as objects of process type. As in the case of any other object in the system, a transaction is assigned a UID.

Nested transactions provide the facility to construct higher levels of abstractions by composing a set of already defined transactions into one larger transaction. The commitment of computations by each of the nested transactions is dependent on the commitment of the parent transaction. Concurrency control mechanisms are required to synchronize nested transactions of the same or different parent transactions.

The remote procedure call mechanism is based on an atomic transaction facility to ensure the atomicity of operations in the presence of system

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

crashes and other concurrent transactions. The remote procedure call mechanism uses an unreliable datagram facility that supports high probability of successful delivery of messages. In [SHRI82a] and [LSK82a] arguments are given in favor of building a reliable remote procedure call facility using less sophisticated facilities such as a datagram. These are examples of end-to-end arguments [SALT81] that point out the wasteful duplication of functions at different levels. Secure communication is achieved by encryption of messages and storing unencrypted messages in protected buffers.

4.2 CONSISTENCY MANAGEMENT IN DISTRIBUTED SYSTEMS

In this section we present an overview of the concurrency control techniques used for maintaining consistency of data under concurrent update operations. In the past, most of the research in this field has focused on the serializability of concurrent updates, i.e., the schedule of interleaved operations of transactions is equivalent to some serial execution schedule of the transactions. We will refer to this kind of consistency requirement as serial consistency. Not much work has been done in the area of concurrency control techniques that permit non-serializable schedules, but still maintain the internal consistency (semantic integrity) of the data.

Numerous protocols have been suggested for maintaining consistency in distributed database to permit concurrent execution of transactions. Some are applicable to specific network structures, others are more general. Before discussing a few protocols of the many surveyed, we shall define a classification of protocols which will be helpful in studying them.

One criterion of classification is whether a protocol is scheduler based or certifier based. If we have a 'do nothing' scheduler, all the steps are scheduled immediately and the resulting changes are kept in temporary copies. At the end of the execution of each transaction a certifier checks for any inconsistency. If the transaction is certified, the changes are made permanent. A protocol is a set of rules which, when followed by each site in the distributed network, leads to implicit or explicit communication required to maintain the consistency. If the set of rules is checked by a scheduler, the protocol is said to be scheduler based. If after executing a transaction the certifier checks that no rule was violated, the protocol is said to be certifier based. There could be some protocols which use a mixed policy in which part of the work is done by the scheduler and part by the certifier. In some cases, the scheduler may decide that a transaction needs to be resubmitted.

4.2.1 Serialization Consistency

The concurrency control problem basically consists of four tasks. The first is to assign an order to all the transactions. The second is to identify conflicting transactions and conflicts. The third is to realize the inter-site synchronization required to achieve this order for the conflicting transactions. The fourth is to achieve the required intra-site

synchronization. There are three basic techniques to achieve serial consistency: timestamps, locks, and optimistic.

4.2.1.1 Timestamp Based Protocols

In timestamp based protocols, every transaction is assigned a globally unique timestamp. This timestamp is used for conflict resolution and it determines the serialization order.

The following description of the basic timestamp ordering (TO) protocol is from [BERN81]. In the model presented there, transactions request read or write operations on objects. These operations are executed by the data manager (DM) at the nodes. In this description $R_i[x]$ and $W_i[x]$ represent the read and write operations on object x by transaction T_i , and $TS(i)$ represents the time-stamp of transaction T_i .

A basic T/O scheduler outputs operations in essentially first come, first served order, as long as the T/O scheduling rule holds. When the scheduler receives $R_i[x]$ it does the following:

```
if  $TS(i) <$  largest timestamp of any Write on  $x$  yet 'accepted'
then reject  $R_i[x]$ 
else 'accept'  $R_i[x]$  and output it as soon as all Writes
    on  $x$  with smaller timestamp have been acknowledged by the DM.
```

When the scheduler receives $W_i[x]$ it behaves as follows.

```
if  $TS(i) <$  largest timestamp of any Read or Write
    on  $x$  yet 'accepted'
then reject  $W_i[x]$ 
else 'accept'  $W_i[x]$  and output it as soon as all Reads
    and Writes on  $x$  with smaller timestamp have been acknowledged by the DM.
```

A conservative T/O scheduler avoids rejections by delaying operations instead. An operation is delayed until the scheduler is sure that outputting it will cause no future operations to be rejected. Conservative T/O requires that each scheduler receive Reads and Writes from each TM in timestamp order. To output any operation, the scheduler must have an operation from each TM in its 'input queue'. The scheduler then 'accepts' the operation that has the smallest timestamp. 'Accept' means to remove the operation from the input queue and to output it as soon as all conflicting operations that have smaller timestamp have been acknowledged by the DM.

Basic T/O and conservative T/O are endpoints of a spectrum. Basic T/O delays operations very little, but it tends to reject many operations. Conservative T/O never rejects operations, but it tends to delay them often. One can imagine T/O schedulers between these extremes. To our knowledge, no one has yet proposed such a scheduler.

Thomas' write rule (TWR) is a technique that reduces delay and rejection [THOM79]. TWR can be used only to schedule Writes, and it needs to be

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

combined with basic or conservative T/O to yield a complete scheduler. If we're interested only in ww scheduling, TWR is simple. When the scheduler receives $W_i[z]$ it does the following:

```
if      TS(i) < largest timestamp of any Write on x yet 'accepted'
then    'ignore'  $W_i[x]$ 
else    'accept'  $W_i[x]$  and output it as soon as all
        operations on x with
```

4.2.1.2 Locking Protocols

A database can be partitioned into entities, which can be locked. By locking an entity, a transaction can prevent other transactions from accessing it, until it releases the lock. Various implementations of locks are possible. The larger the entity, the less will be the overhead for locking. But at the same time a larger entity size leads to a lower level of permissible concurrency. Both the implementation of locks and the size of entities are irrelevant to the understanding of the protocols, and will not be discussed further. The level of concurrency can be improved by having various lock modes. These lock modes could be mutually permissible, and they may be compatible and/or convertible to one another. By identifying these properties, execution sequences which would otherwise be forbidden could be permitted; this obviously increases the concurrency. Some of the terms related to these modes are defined below.

A read mode lock held by a transaction T on an entity E allows it to read the value of E and use it for computations, as often as it needs, till it releases the lock. We shall denote this mode by R.

A write mode lock held by a transaction T on an entity E allows the transaction to read and/or write and use the value of the entity for computations, as often as it needs, till it releases the lock. We shall denote this lock mode by W.

A locking mode X is said to be compatible with a mode Y if a transaction may acquire X on an entity while another transaction holds Y on it. For example, an R lock on an entity is compatible with another R lock on the same entity but is incompatible with a W lock on this entity.

Unless the logical data base is specified as having a specific structure (e.g., Directed Graph Structure) all the transactions must follow two phase locking protocol [ESWA76] to ensure serializability. Thus, we have the following two subclasses under the locking protocols.

1. Two-phase locking.
2. Non-two-phase locking

Two-Phase Locking Protocols

A two-phase locking protocol specifies that in each transaction all the locking operations must precede any unlocking operation, and all transactions be well-formed. A well-formed transaction always acquires appropriate locks on the objects before reading or updating them. It has been shown in [ESWA76] that if all transactions follow the two-phase locking protocol, then the schedules of their executions are serializable. In the two-phase locking it is easy to see that deadlocks are possible. To avoid deadlocks we could set an order to all the entities and stipulate that all the transactions request locks only in the said order. Alternatively, when a transaction has been permitted to start executing, it may put intention locks on all the entities it would ever need. These locks may be used to rule out the possibility of a deadlock before permitting any other transaction to put its intention locks. Thus we need to superimpose a mechanism on top of the two-phase locking protocol to ensure deadlock freedom.

Rosenkrantz, et al., [ROSE78] have proposed deadlock prevention schemes for locking protocols based systems. These schemes use timestamps to assign priorities to transactions. There are two basic variations of deadlock prevention in their schemes:

1. Wait-Die: In this scheme, a transaction, say X, requiring a resource that is currently being held by another transaction, say Y, waits for transaction Y to complete if the transaction X is older than transaction Y. Transaction X dies and restarts if it is a younger transaction.
2. Wound-Wait: Here, a requesting transaction wounds a younger transaction holding the required resource. A wounded transaction completes successfully if it has already initiated its termination, else it aborts. If the requesting transaction is older, then it waits resource it needs is released.

The Wait-Die scheme has the disadvantage that a younger transaction may restart and die several times before completing successfully. The restarts will consume some of the system resources. However, this scheme has the advantage over the Wound-Wait scheme that after a transaction has acquired all resources it needs, it can not be pre-empted and restarted. In the Wound-Wait scheme, even when a transaction has locked all the resources it needs, but has not yet initiated its termination, it is possible that it gets wounded and is forced to restart.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Non-Two-Phase Locking

As the name suggests, these protocols do not use the two-phase locking policy. Very few protocols have been suggested which belong to this class. One of these, proposed in [SILB81] is applicable for hierarchically organized database systems. It presumes a tree-structured, hierarchically organized database. Unlike the two-phase locking protocols, it is free from deadlocks. It locks an entity, i.e., a node in the tree, without implying any locks on the entity's descendants in the tree. A transaction following this protocol must satisfy the following conditions:

1. T_i may lock any node to start with.
2. T_i may lock other nodes only if it already holds a lock on its father in the database.
3. After unlocking a node, it may not lock it again.
4. It may access only those nodes on which it holds a lock.

Note that the transaction need not be two-phase. It has been proven that the schedules produced by this protocol are serializable. An intuitive understanding of this fact is easy to see by the following argument: each transaction has a frontier of lowest nodes in the tree on which it holds the locks. The protocol guarantees (conditions 2 and 3) that these frontiers do not overlap. If the frontier of T_i begins above the frontier of T_j, it will remain so, and every item to be locked by both will be locked by T_j first.

Locking in Distributed Systems

The consistency management techniques in distributed systems can be divided into categories depending on whether they require global locking. Some techniques do not require any global synchronization to lock objects at a node. The mutual consistency among multiple copies is ensured by time-stamps as in SDD1. The techniques that require locking of objects at the global level use either 1) a primary copy located at a node for granting locks, or 2) local locking along with some deadlock detection/prevention scheme, or 3) some variation of the circulating (migrating) token scheme. An example of the circulating token approach is Minoura's migrating true-copy scheme.

The circulating token protocols require that there be an order for passing the token. Permission to access the entities, requiring mutual exclusion between sites, is passed from one site to another in a pre-determined order. A token is used as a symbol of the permission. Since there is only one token, inter-site mutual exclusion is ensured.

As already mentioned, there are two ways of achieving centralized locks. In the first case, one site is designated to be primary and transactions running on all sites seek locks from it. It is clear the the primary site protocols would have high communication requirements and that the primary site would tend to be a bottle-neck.

In the second case, one copy of each entity is designated to be primary and locks are sought only on these copies. Here communication overhead tends

to distribute over all the sites, since different entities have their primary copy on different sites.

4.2.1.3 Optimistic Concurrency Control

The optimistic method for concurrency control was proposed by Kung and Robinson [KUNG81]. This is a non-locking method of concurrency control; it relies on transaction backup as a control mechanism, allowing transactions to proceed concurrently until their validation points, "hoping" that the transactions will not conflict.

In this method every transaction goes through three distinct phases: read phase, validation phase, and write phase. During the read phase, the transaction reads the objects and creates local copy of the object for update operations. The validation phase ensures that making the updated local copy of the objects will not violate serial consistency requirements. If this test fails, then the transaction is aborted.

In order to ensure the serializability of the transaction, each transaction is assigned a unique integer number. For this purpose, a global integer counter tnc (transaction number counter) is maintained in the system. The transaction serialization order is the same as their assigned numbers. The validation procedure is based on rules that ensure that one of the following three conditions holds. T_i and T_j are two transactions such that $t(i)$ (the transaction number of T_i) is smaller than $t(j)$.

- (1) T_i completes its write phase before T_j starts its read phase.
- (2) The write set of T_i does not intersect the read set of T_j , and T_i completes its write phase before T_j starts its write phase.
- (3) The write set of T_i does not intersect the read set or the write set of T_j and T_i completes its read phase before T_j completes its read phase.

Condition (1) states that T_i actually completes before T_j starts. Condition (2) states that the writes of T_i do not affect the read phase of T_j , and that T_i finishes writing before T_j starts writing, hence does not overwrite T_j (also, note that T_j cannot affect the read phase of T_i). Finally, condition (3) is similar to condition (2) but does not require that T_i finish writing before T_j starts writing; it simply requires that T_i not affect the read phase or the write phase of T_j (again note that T_j cannot affect the read phase of T_i , by the last part of the condition).

The transactions are assigned their transaction numbers after they complete the read phase. The reason for not assigning transaction numbers before the start of the read phase is to avoid the situations in which a more recent transaction with a short read phase might get blocked due to a long read phase of some earlier transaction. This scheme of assigning transaction numbers does not require the validation of condition (3) above. The following

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

is a serial implementation of the validation rules. The statements enclosed within "<" and ">" are treated as a critical section.

```
start_tn:=tnc;
  read phase
  <finish_tn:=tnc;
  valid:=true;
  for t from start_tn+1 to finish_tn do
    if (write set of transaction with number t
        intersects the readset of the transaction
        to be validated)
      then valid:=false;
  If valid then
    (write phase;
     tnc:=tnc+1;
     tn:=tnc)>
  If valid then cleanup else backup;
```

In [KUNG81] a parallel version of the validation algorithm is presented which validates all the three conditions and permits concurrent write operations (i.e., the write phase is not a part of any critical section).

4.2.1.4 Basic Timestamp Ordering Versus Locking

The following example from [AGRA83] shows that the timestamp ordering algorithms in centralized systems tend to behave very similar to locking but have the disadvantage of inducing larger numbers of restarts.

The timestamp ordering scheme a priori determines the serialization order, whereas, locking protocols dynamically determine the serialization order. This leads to restarts of conflicting transactions in timestamp ordering schemes. The following example illustrates this observation. Consider two transactions T1 and T2 such that ts (T1) (i.e., timestamp of T1) is smaller than ts (T2). Suppose the following sequence of operations is executed in the two transactions.

```
T2 : Read (X)
T2 : Commit
T1 : Write (X)
```

The basic timestamp ordering scheme will abort T1 but locking schemes as well as the optimistic scheme will allow both T1 and T2 to complete successfully. In both the timestamp ordering scheme and the optimistic scheme locks are required to implement critical sections. For example, in basic TO scheme locks are required while reading updating the timestamps associated with the objects.

More importantly, in the timestamp ordering scheme some form of logical locking might be required in order to prevent triggered aborts. Such a situation arises when a transaction is allowed to read objects that have been

updated by a transaction that is still uncommitted. Consider the following example where $ts(T2) > ts(T1)$.

T1 : Write (X)
T2 : Read (X)
T1 : Abort

When T1 aborts, T2 is also aborted and all updates made by T2 are undone. Therefore, to prevent such a situation, when a transaction starts updating an object, access to that object by other transactions is blocked until the transaction either commits or aborts. This is equivalent to holding a write lock on the object. It should be noted here that the optimistic scheme avoids locking and accepts the possibility of transaction aborts.

4.2.1.5 Deadlock Prevention vs. Deadlock Detection

Some deadlock prevention schemes, Wait-Die and Wound-Wait, were discussed in Section 4.2.1.2. Deadlock detection is another approach to coping with deadlocks in the system. In the SCOT project some simulation studies were conducted to compare various schemes for deadlock detection and prevention. The results of this study are presented in [BALT82]. In these studies the evaluation was based on the measurement of transaction response time and the system throughput expressed as transactions completed per unit of time. A high conflict rate was assumed in these studies; a large number of transactions access a relatively small part of a large database. The response time and the throughput were measured as a function of the concurrency level of the system.

The first part of the studies focused on the evaluations in centralized systems. In these simulations two deadlock detection schemes were considered. In the first scheme there was no limit on wait queues for resources. The second scheme limited the number of transactions in a wait queue to some predefined value. The deadlock prevention schemes studied were Wait-Die, Wound-Wait, and improved Wound-Wait [ROSE78]. The results of these simulations can be summarized as follows:

- (1) For low concurrency levels, all schemes perform equally well.
- (2) Deadlock detection schemes with wait queue limitation perform better than most of the other schemes at all levels of concurrency. This is because the wait queue limitation strategy reduces the effective level of concurrency in the system.
- (3) At all levels of concurrency, the deadlock detection scheme without wait queue limitation performs the worst of all techniques. In this case the throughput starts decreasing sharply after achieving an optimum value as the concurrency level is increased.
- (4) In all cases Wound-Wait performs better than Wait-Die; however, this improvement is only marginal.
- (5) The improved Wound-Wait scheme performs worse than the ordinary Wound-Wait scheme.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

It was concluded in these studies that the optimum throughput is reached when the length of the wait queue is limited to one. In such a case a transaction is allowed to wait only for a running transaction, but not for a waiting transaction. Simulations were performed for distributed concurrency control schemes with deadlock prevention schemes (Wait-Die and Wound-Wait) and deadlock detection with wait queue limitation. The observations summarized above were found to be valid in this case also.

Another technique that can be used for deadlock prevention is the use of time-outs. This technique was not used during the evaluations on the SCOT project. Heuristics for selecting the optimum time-out periods need investigation.

4.2.2 Non-Serial Consistency

It is only recently that researchers [GARC83b] [FISH82] [BLAU83] have started investigating consistency management techniques that exploit the semantic knowledge of the database during concurrency. Such a knowledge can lead to certain acceptable schedules that are not serializable. This area of research is relatively little explored.

In [GARC83b] Garcia-Molina investigates how the semantic knowledge of an application can be used in a distributed database to process transactions efficiently and to avoid some of the delays associated with failures. In [GARC83], the main idea is to allow nonserializable schedules which preserve consistency and which are acceptable to the system users. To produce such schedules, the transaction processing mechanism receives semantic information from the users in the form of transaction semantic types, a division of transactions into steps, compatibility sets, and countersteps. Using these notions, in [GARC83], a mechanism is proposed which allows users to exploit their semantic knowledge in an organized fashion.

4.3 RELIABILITY TECHNIQUES IN DISTRIBUTED SYSTEMS

In this section we review various error recovery techniques and their applicability in distributed systems. Our discussion of recovery techniques starts with a brief overview of the concepts and definitions in this area. Detailed discussions of these concepts and definitions can be found in some of the surveys [RAND78] [KOHL81] [VERH78] in this area.

A system is said to have failed when it no longer meets its specifications. The transition into the failed state is characterized by the failure event.

The term error is used to characterize an incorrect system such that any further computation activity using the normal algorithms would result in a failure of the system.

A fault is the mechanical or algorithmic malfunction (i.e., failure) of a system component that may cause an erroneous state.

All reliability techniques are based on one basic principle -- incorporation of redundancy in the system to support recovery from errors and continued operation. This kind of redundancy is called protective redundancy; it is incorporated into the system as additional components, data and algorithms.

Protective redundancy is incorporated in systems in two distinct ways: masking redundancy and dynamic redundancy. As the name implies, masking redundancy tends to hide the effects of erroneous states so that the system behavior is not affected due to component failures. The most common example of masking redundancy (also called static redundancy) is a Triple Modular Redundant (TMR) system.

In the case of dynamic redundancy, additional resources are used within the component to dynamically detect malfunctioning of a component and to compensate for it using another set of external redundant resources which act as stand-by or spares.

Conceptually, there are three fundamental strategies incorporated in every reliable system design. These are error detection, damage assessment, and error recovery. The following parts of this section describe the most common techniques for error detection and recovery.

4.3.1 Error Detection Techniques

The purpose of error detection techniques is to detect the erroneous states of the system that could lead to system failures. The reliability of a design depends on how rigorous the techniques are for detecting erroneous states in the system.

Some general techniques for error detection [ANDE79] are described below.

- (a) Replication Checks: In such schemes, an activity is replicated and the results from replicated activities are checked for consistency. An inconsistency among results indicates a possible error condition. Errors can be masked by majority voting as in Triple Modular Redundant systems.
- (b) Reversal Checks: This involves application of inverse computation to check what the input to the system should have been. The calculated input and the actual input are compared for consistency.
- (c) Coding Checks: This is the most popular error detection technique. Redundant information in the form of checksum or parity is associated with objects to detect erroneous states.
- (d) Acceptance Tests/Consistency Checks: At certain well-defined points in the execution, tests are applied to the objects that define the state at that point. Such tests ensure that the state at that point conforms to certain specifications. Any inconsistencies imply an erroneous state. Consistency checks can also be applied to some mutilated data structures that are reconstructed on recovery.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

- (e) **Interface Tests:** These tests ensure that the interactions among system components meet certain acceptance criteria. Tests are applied to the parameters and the results of interface functions. Such tests limit propagation of errors from one component to another through the interfaces. The confinement of errors is strongly dependent on how rigorous the acceptance tests are. In distributed systems, interfaces provide well-defined and controlled means for the propagation of exception conditions between modules. If the interface function execution encounters error conditions, then an error condition is returned to the caller through the interface.
- (f) **Diagnostic Checks:** In such techniques, explicit tests are conducted on system components for which expected outputs for given test inputs are known. The failures of components to be tested and the components conducting the tests should be independent. As pointed out in [ANDE79], diagnostic tests are rarely used as a primary error detection mechanism, rather used as a supplement to other detection mechanisms.
- (g) **Interval Timer/Time-Out Mechanisms:** In distributed systems, time-out techniques are frequently used to detect possible error conditions. A process invoking a remote operation waits for a certain specified period (called the time-out period) to receive the response. If no response is received within this period, then an exception condition is raised and appropriate forward error recovery is initiated.

4.3.2 Error Recovery Techniques

Depending on the way a consistent system state is regenerated, error recovery techniques are divided into two broad categories: backward error recovery and forward error recovery. In backward error recovery, a prior consistent state in the execution history is restored. The forward error recovery techniques rely upon using the present error state to arrive at some consistent state. The techniques in the latter category are application dependent, whereas the techniques in the former category are more general in application.

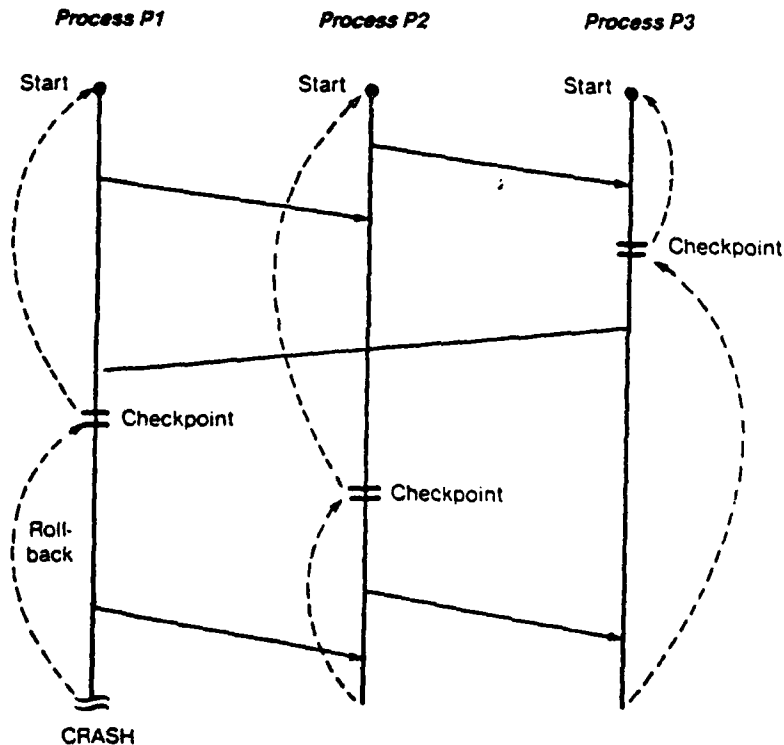
Backward Error Recovery

Backward error recovery requires facilities for establishing recovery points which support reconstruction or restoration of the state at that point. Some of the important techniques for backward recovery are described below.

4.3.2.1 Checkpointing and Rollback

In this technique, the complete state of the process to be checkpointed is saved on a stable storage. In a process-oriented design, a checkpoint also saves on the stable storage the current state of all the objects bound to that process. A checkpoint creates a backup version on a stable storage of the complete execution environment of the process that existed at the time of checkpointing.

The rollback of a process in a system of communicating processes may cause rollback of other processes. This happens when a process that is rolled back to a previous checkpoint has communicated some information to some other processes after establishing that checkpoint. Thus, all messages sent after that checkpoint are revoked, and all activities performed by the recipient processes after receiving such messages are invalid; hence, all recipient processes are also rolled back to their respective checkpoints established before receiving these messages. This can cause a cascade of rollback activities, a phenomenon referred to as the domino effect. An example of the domino effect is shown in Figure 4-3 where a crash of process P1 causes all processes to be rolled back to their starting points. The basic reason for domino effect in distributed systems is the revocation of information exchange from one process to another.



Domino Effect
FIGURE 4-3

The following discussion shows that object-oriented transaction-based systems in which the interactions among processes follow certain commit protocols are free from the domino effect. In such systems, a process executes a transaction to update a set of shared objects. During the

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

transaction execution, these objects are locked and other processes are prevented from accessing the objects. Only when the transaction commits successfully are the updated values made permanent and can be used by other processes. Committed values of objects can never be revoked. The only way to undo the updates of a committed transaction is to run a compensating transaction. During process rollback only the uncommitted transactions are undone. Thus, there is no cascade of rollback activities.

4.3.2.2 Careful Replacement

The careful replacement technique avoids updating objects "in place". Updates are made to a copy of the original, whereas the original copy, also called the "shadow" copy, remains unaffected in case of failures during the updating procedure. Only on commitment is the shadow copy replaced by the updated copy.

An example of this technique is the scheme proposed by Lampson and Sturgis [LAMP81] for making page write operations atomic in order to implement stable storage facility. The Put and Get operations on a physical disk are not atomic in the sense that a crash of the system during the Put operation for a page may leave that page only partially updated. The CarefulPut operation is defined to ensure that a Put operation completes successfully provided no processor or disk crash occurs. A CarefulPut is defined as follows:

```
Procedure CarefulPut (var p:DiskPage, b:MemoryPage),  
begin  
    repeat  
        Put (p,b)  
        Get (p,b)  
    until status (b)=ok,  
end
```

Similarly, the careful get operation is defined as follows. This operation reads a page repeatedly until either it gets a clean page or some prescribed bound is exceeded.

```
Procedure CarefulGet (var p:DiskPage, b:MemoryPage),  
const N:integer;  
begin  
    i:=0  
    Get (p,b)  
    repeat >i:=i+1;  
    until (status (b) = ok) or (i>N)  
end;
```

The Cleanup operation periodically checks the status of the two pages; if one of the pages is corrupted and the other page is in good state, then the cleanup procedure replaces the contents of the corrupted page by the contents of the good page. This operation is periodically applied to each StablePage in the system. If T_c is the period of invoking the cleanup procedure, then for a StablePage to be reliable and highly available the period T_c must be

small enough so that the probability of both DiskPages of a StablePage getting corrupted is infinitesimally small.

```
Procedure Cleanup (var p1, p2:DiskPage);  
var b1, b2: MemoryPage;  
begin  
    CarefulGet (p1, b1);  
    CarefulGet (p2, b2);  
    If (status (b1) = good) and (status (b2) = bad)  
    then CarefulPut (p2, b1);  
    else if (status (b1)=bad) and (status (b2)=good)  
    then CarefulPut (p1, b2)  
    else if (status (b1)=bad) and (status (b2)=bad)  
    then page_error;  
end;
```

A StablePage is constructed from two disk-pages and by using CarefulGet and CarefulPut operations. StablePut and StableGet operations on a StablePage are defined below.

Let p be a StablePage which is implemented using two DiskPages p1 and p2.

```
Procedure StablePut (var p: StablePage, b:MemoryPage);  
begin  
    CarefulPut (p1, b);  
    CarefulPut (p2, b);  
end  
  
Procedure StableGet (p:StablePage, var b:MemoryPage);  
begin  
    CarefulGet (p1, b);  
    If (status (b)=bad) then CarefulGet (p2,b);  
end;
```

The construction of a stable storage facility in this fashion is valid if the following conditions hold.

- (1) Get (p, b) operation never returns good status for a bad page.
- (2) Get (p, b) operation never alters the contents of a page such that it still appears to be good during status checks.
- (3) Both physical pages for a StablePage never get corrupted during the same interval between two consecutive cleanup operations.
- (4) Contents of a good physical page on the stable storage never get changed such that the page still appears to be good.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

- (5) Contents of a bad physical page on the stable storage never get changed such that the page appears to be good.

Another example of careful replacement is the technique of using a shadow copy of an object to facilitate recovery. This technique avoids in-place updating of objects by keeping two copies of an object: the current version and the shadow version. The updates from an uncommitted transaction affect only the current version of the object. On transaction commitment, the current version is made the shadow version thereby making the updates permanent. On transaction abort, the current version is deleted and the shadow version is made the most current version, thereby undoing the transaction. It is required that the operation of replacing the shadow version by the current version must be atomic, and it must be done in one instruction. The technique described below accomplishes this objective. This technique was proposed by Lorie [LOR177].

Suppose that an object is represented by a set of StablePages $\{P_1, \dots, P_n\}$ in the stable storage. There is a page table PT associated with each object, and we will assume that this table occupies one StablePage in the stable storage. When a transaction accesses the object, it first reads the page table in the primary memory. As a notation, we will use capital letters to refer to the page addresses on the stable storage, and small letters to denote the pages in primary memory. For example PT is the page table address on the stable storage and pt is the page table address in primary memory. We use the notation $PT[i]$ to refer to P_i , the i -th StablePage address in the page table.

The shadow copy update algorithm is described below:

```
begin
  StableGet (PT, pt); {read the page_table in the primary memory}
  {To read/write the page PT[i]}
  If PT[i] not in the primary memory
  then StableGet (PT[i], pi)
    {read page  $P_i$  in the primary memory buffer pi from the
     stable storage address  $P_i$ }
  else read/write in buffer pi;
end;
```

Procedure commit; {On transaction commitment, execute the following procedure}

```
begin-commitment
for i:=1 to n do
  begin
    If pi has been updated
    then get 'unused page  $P_i$ ' on the stable storage,
    StablePut ( $P_i$ , pi) {write the contents of the buffer pi
     into a new unused StablePage  $P_i$ }
    pt[i]:=P_i'
  end;
  StablePut (PT,pt);{This will change the page table, on the stable
   storage and hence the updated version will
```


become permanent}
end-commitment

Before the execution of the last StablePut operation, any crash during the execution of this procedure will abort the transaction. Successful completion of the last StablePut operation implies permanence of the updates.

4.3.2.3 Logs/Audit Trail

In this technique, actions performed on an object are recorded in a log or audit trail. The purpose of the logs is to support either undo of the logged action for state rollback, or redo the logged action to ensure permanence of results produced by some committed transaction. Logs/audit-trails are used to either restore an object to a state prior to executing a sequence of operations on it or to ensure the permanence of the effect of executing a sequence of operations on it. The logs that facilitate object state recovery record the undo operation corresponding to every action performed on an object, and the logs that are used to ensure permanence of effect record the redo operation for every operation performed on the object. An undo record for an operation on an object specifies the actions to be executed to nullify the effect of executing that operation on the object. A redo record for an operation basically records the actions performed by the operation.

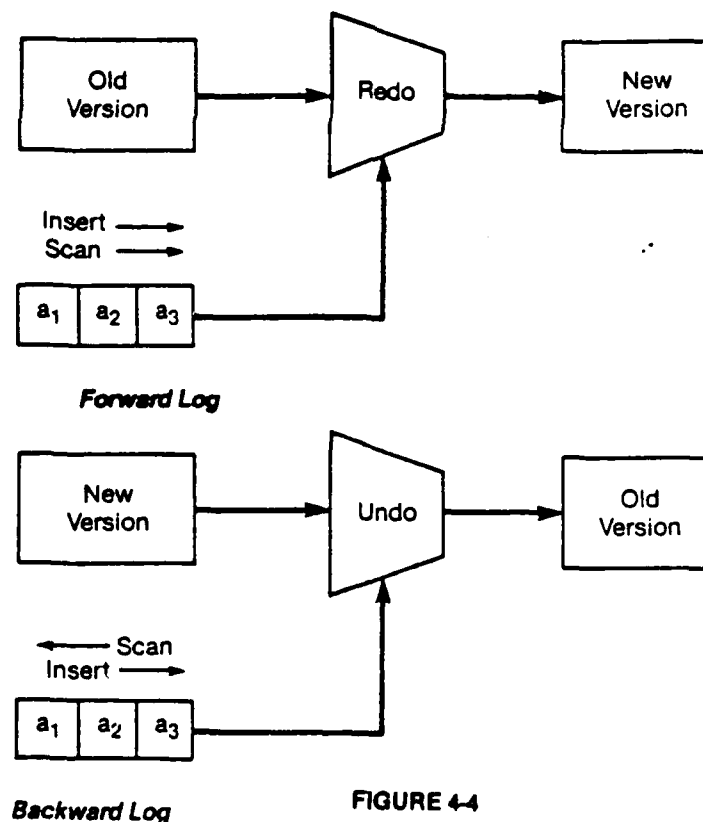


FIGURE 4-4

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Logs that contain the redo actions are called the forward logs, and the logs that record the undo actions are called the backward logs. The backward logs either record the inverse operations or the values of the object before the application of the logged action. During a recovery process, a backward log is used by scanning it backwards for undoing actions in a last-in, first-out fashion. Thus a backward log can be viewed as a push-down stack. During system recovery, a forward log is scanned in the FIFO order as a queue. This use of forward and backward logs for recovery is shown in Figure 4-4.

A forward log is said to be idempotent if any number of (complete or aborted) repeated executions of the log from the beginning leave the updated objects in the same state. Such logs are also referred as intention lists. One way to implement forward logs is to use differential files. This is described below.

In this technique, all updates to an object are recorded on a differential file. The updates from the differential file are periodically merged into the main copy of the object and such updates are then deleted from the differential file. The differential file technique provides a relatively inexpensive means of maintaining multiple versions of a large object. Intention lists and forward logs are forms of differential files containing redo actions that record the new values of the objects and have the property of idempotency. The property of idempotency implies that repeated executions (some of which may be incomplete) of this sequence of actions would always bring the updated object to the same state.

The backward log technique is used when changes are made in-place in the stable storage. The recovery techniques based on backward logs follow the write-ahead-rule: (1) Before performing an operation in-place on an object, record the corresponding UNDO action in the log and force the log on the stable storage, (2) Before committing a transaction (i.e., sending a commit response to the user), either the updated versions of the objects or the corresponding forward logs must be forced on the stable storage. This rule makes sure that if the system crashes or the transaction aborts, the backward log can provide a means for restoring the object which has been updated in-place. Similarly, for a committed transaction, the updates made by it are guaranteed to be made permanent by using the forward logs. This rule is described below for some operation A to be performed by transaction T. Here UNDO(A) means the undo action corresponding to A, and LOG(T) is the log maintained for transaction T.

To perform operation A, transaction T executes the following actions:

- (1) Record UNDO(A) in the LOG(T) in volatile storage;
- (2) Update the object, by executing A, in the volatile storage;
- (3) Write LOG(T) from the volatile storage to the stable storage;
- (4) Write the updated object in-place in the stable from the volatile storage.

It is important to note here that the steps (3) and (4) must execute as one atomic action; otherwise, a crash in between executing (3) and (4) will render the object recovery inconsistent. This is because the log will contain an undo operation for an action that was not executed. This can be achieved by the using the technique of careful replacement described in 4.3.2.2.

To rollback a transaction T, during recovery from a system crash, the following actions are performed:

- (1) Read LOG(T) in volatile storage from the stable storage;
- (2) Read the object in volatile storage from the stable storage;
- (3) Repeat
Select the last UNDO action from LOG(T);
Delete this UNDO action from LOG(T);
Execute this UNDO on the object image in the volatile storage;
until all action in LOG(T) have been processed;
- (4) Write LOG(T) from volatile storage to the stable storage;
- (5) Write the object image from volatile storage to the stable storage;

Here again, operation (4) and (5) must be executed in an atomic fashion. The recovery technique based on the concept of forward logs is described below.

- (1) Read the object in the volatile storage from the stable storage;
- (2) Update the object image in the volatile storage;
- (3) Record the update as a REDO action in LOG(T);
- (4) When all actions have been performed, write LOG(T) in the stable storage from the volatile storage;

During system recovery,

On transaction abort: do nothing;

On transaction restart:

- (5) Read LOG(T) in volatile storage from the stable storage;
- (6) Read the object image in the volatile storage from the stable storage;
- (7) Repeat
Select the first REDO action from LOG(T);
Delete this action from LOG(T);
Update the object image in the volatile storage by performing this REDO action;
Until all actions in LOG(T) have been processed;
- (8) Delete LOG(T) from the stable storage;

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

- (9) Write the object image from volatile storage to the stable storage;
- (10) Continue processing according to steps (2) through (4).

To process a committed transaction:

- (11) Process LOG(T) as in step (7).
- (12) Write the transaction status for T as COMPLETED;
- (13) Write the new object image from the volatile storage to the stable storage;

In the above algorithm steps (8) (9), and steps (12) (13) should be implemented as atomic sequences of operations. A detailed description of a design model based on these principles for transaction processing is presented in Section 6.

4.3.2.4 Commit Protocols and Atomic Actions

Commit protocols are used for implementing atomic actions in distributed system. The commit protocols ensure the all-or-nothing property of atomic actions in the presence of node crashes and communication link failures. The concept of commit protocols was independently introduced by Gray [GRAY79], and Lampson and Sturgis [LAMP76].

A distributed transaction performs operations on distributed global objects. To perform a set of operations on objects at a node, a transaction starts a worker process on its behalf at that node; such a process is also referred to as a cohort. The global transaction is initialized at a site by the creation of a transaction process. A transaction can spawn new transactions by creating new transaction processes at the same or different sites. Such transactions are called nested transactions. The cohort processes and the transaction processes execute the commit protocols to ensure that everyone of them makes the same decision regarding the completion or abortion of the transaction. This maintains database consistency by ensuring the "all-or-nothing" property of the global transaction.

As mentioned in [MOHA83], some of the desirable characteristics for a commit protocol are: 1) guaranteed transaction atomicity, 2) minimal overhead in terms of log writes, 3) optimized performance in no-failure case, 4) ability to "forget" the outcome of commit processing after a while, and 5) exploitation of read-only transactions.

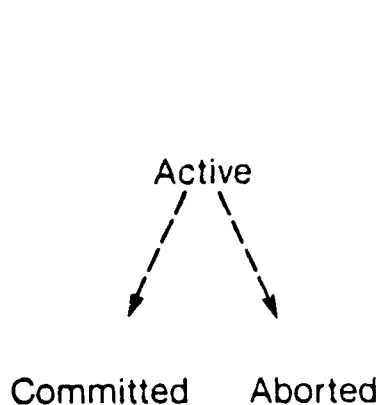
The most common commit protocols are either one-phase or two-phase. In the following parts of this section we describe the basic model for these two classes of protocols and compare their performance tradeoffs. Most of the following discussion on these two protocols is from [BALT81]. In both these classes of protocols one of the processes acts as a coordinator to resolve the completion or abortion of the distributed transaction. Because of a central coordinator, these protocols make the commitment decision with a centralized

control. The selection of the process to perform the coordinator function is an important issue and it is discussed later.

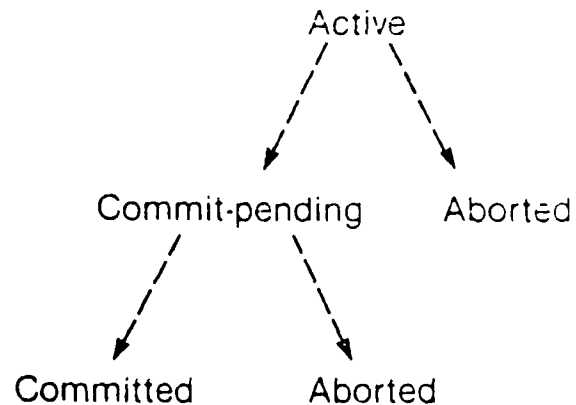
A transaction creates a cohort process at other nodes by sending a message to the host nodes. In the diagrams we will mark such a message by CREATE-PROCESS. All cohort processes are assumed to be well-formed [ESWA76], i.e., they always properly lock the resources before accessing them. The resources acquired by a cohort process or a transaction are released only when the transaction commitment decision has been made. If the transaction commits, the updates performed by the cohort processes are made permanent; otherwise, the objects are released with their states prior to the transaction execution. On completing the requested work, the cohort processes send DONE messages to their respective initiators.

4.3.2.4.1 One-Phase Commit Protocols

In a one-phase commit protocol, the initial transaction process sends CREATE-PROCESS messages to create cohort processes at other nodes. A worker process, on completing its activities, enters the commit-pending state from the active state by saving the updates to the local resources on a stable storage in a recoverable fashion as described in Section 4.2.4. A cohort process that is unable to perform the requested activity enters the abort state from the active state, and it sends an ABORT message to the coordinator. A cohort process entering the commit-pending state sends a DONE message to the coordinator. After the coordinator has received DONE messages from all cohort processes, it enters the committed state and sends a commit message to each of the worker processes. If the coordinator decides to abort the transaction, then an ABORT message is sent to each of the cohort process. The state transitions for the cohort processes and the coordinator are shown in Figures 4-5(a) and 4-5(b) respectively.



States Transition for the Coordinator
FIGURE 4-5 (a)



State Transitions for a Cohort
FIGURE 4-5 (b)

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

The coordinator decides to abort a transaction if it either receives an abort message from any of the cohort processes or it does not receive any message from at least one worker process before the time-out condition occurs. Figure 4-6 shows an example of a one-phase commit protocol.

Effect of System Failures

The possible system failure events are: coordinator crash, a cohort process crash, and communication system failure. The crash of a cohort process before sending an ABORT or DONE message to the coordinator will cause the coordinator to time-out and abort the transaction. Any cohort process crash during the commit-pending state requires the following recovery steps:

- 1) On restart of the node, an inquiry procedure is invoked to find out from the coordinator process the decision, if any, taken during the failure period.
- 2) All locks on resources being held by the cohort process must be maintained after recovery until a decision regarding commit or abort is obtained from the coordinator.

A coordinator failure may leave a cohort process that has sent a DONE message and entered the commit-pending state in the blocked state; in this state the process holds locks on resources until it hears the commit decision from the coordinator. Similarly a communication link failure may leave a commit-pending process in that state with locked resources for a long period of time. For these reasons the period during which any of the cohort processes is in the commit-pending state is called the in-doubt period. No algorithm can avoid this in-doubt period; however, certain techniques, such as the two-phase commit protocol, tend to minimize this in-doubt period where the system is sensitive to coordinator failures causing processes holding resources to be blocked.

4.3.2.4.2 Two-Phase Commit Protocols

As mentioned above, the two-phase commit protocols minimize the in-doubt area. In this protocol also one of the processes is designated to act as the coordinator. As in the one-phase commit protocol, each cohort process sends either DONE or ABORT message to the coordinator. However, unlike the one-phase commit protocol, the cohort process does not enter the commit-pending state immediately after sending the DONE message to the coordinator. It waits to receive either a PREPARE or ABORT message from the coordinator. During this waiting period, a cohort process is free to release all its resources and terminate itself; this would of course abort the whole transaction. Only on receiving the PREPARE message, does a cohort process enter the commit-pending state. The actions of the coordinator and the cohort processes are described below.

In order to ensure proper recovery of transactions, during the execution of the commit protocols proper information related to the commitment is logged

by the coordinator and the cohorts in the transaction log. Such a log is maintained on the stable storage. When a log record is written, the write operation can be done either synchronously or asynchronously. The synchronous writing of a log record, which is also called forcing a log record, means that this log record and all the precluding log records from the primary memory that have not been written on the stable storage yet, are forced on the stable storage and the process invoking this operation waits until the forcing is complete. After a force write operation the forced record will survive crashes.

For asynchronous writing of a log record, the record is written in a buffer in the volatile storage. This record migrates to the stable storage sometime later; this happens due to a subsequent force or due to the page buffer being full. For an asynchronous write, the invoker can proceed once the record has been written into the buffer storage. A record written using asynchronous write may get lost due to crashes. The synchronous write increases the response time of the invoking process.

Commit Coordinator

Phase 1:

- (1) On receiving a DONE message from all the cohort processes, the coordinator sends PREPARE messages to each of the cohorts.
- (2) The coordinator waits to receive a reply from each of the cohorts.

The following two conditions may arise:

- (a) If any of the cohorts sends an ABORT message, then
Write ABORTED in the transaction log in stable storage;
Send the ABORT message to all the cohorts;
Wait for acknowledgement from all the cohorts;
Terminate when all acknowledgements have been received.
- (b) If a READY message is received from each of the cohorts then start of the commitment procedure, or else start the abort procedure of Phase 2.

Phase 2:

- (1) Force write COMMITTED in the transaction log in the stable storage.
- (2) Send a COMMIT/ABORT message to each of the cohorts.
- (3) Wait for a positive acknowledgement from each of the cohorts; retransmit the commit (abort) messages if no acknowledgement is received from a cohort before the timeout.
- (4) When all cohorts have replied with positive acknowledgements, write COMPLETE in the transaction log in the stable storage.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Cohort Processes

Phase 1:

- (1) Wait for a PREPARE message from the commit coordinator.
- (2) In response to the PREPARE message, the cohort process writes the updates to objects on the stable storage in a recoverable fashion as described in Section 4.2.3.
- (3) At this point the cohort process enters the commit-pending state by force writing COMMIT-PENDING/ABORT in the local transaction log.
- (4) If steps (2) and (3) are completed successfully, then the cohort process sends a READY message to the commit coordinator, or else it sends an ABORT message.

Phase 2:

- (1) The cohort waits in the commit-pending state for the commit decision from the coordinator. This is the in-doubt period for this cohort.

The following two cases can arise:

- (a) If the decision is ABORT, then restore the object state by processing the backward logs or discarding any forward log;
Release all the resources;
Force write ABORTED in local transaction log;
Send an acknowledgement to the coordinator;
- (b) If the decision is COMMIT, then make the updates permanent by either processing the forward logs or discarding the backward logs;
Release the resources;
Write COMPLETED in the transaction log;
Send a positive acknowledgement to the commit coordinator.

Effects of System Failures and Recovery:

The recovery protocol described below deals with the possible system failures such as the coordinator crash, cohort crash, or communication link failures. During the recovery phase on restart of a node, the following protocol is executed by the coordinator.

- (1) Any coordinator crash before completing step 1 of Phase 2, will result in the abortion of the transaction. This is due to the fact that on restart the transaction log at the coordinator node will contain status as UNDECIDED; therefore, on restart, the coordinator will send ABORT messages to all the cohorts.
- (2) If the coordinator crashes any time during Phase 2 after completing Step 1 but before completing Step 4, then the transaction log at the coordinator will contain COMMIT but not COMPLETED entry. On restart, the

coordinator will send COMMIT messages to each of the cohorts. It is possible that a list of cohorts that have not sent positive acknowledgements is maintained on the stable storage. If such a list is available then send the COMMIT messages only to the cohorts in this list. Such a list needs to be updated when positive acknowledgements are received.

- (3) A coordinator crash after its completing Step 4 of Phase 2 does not require any recovery processes. This is because the transaction log will contain a COMMIT entry indicating that all cohorts have successfully completed their tasks.

A cohort process will execute the following recovery protocol on crash recovery of the host node.

- (1) On restart from any failure before completing Step 3 of Phase 1, the cohort process aborts the transaction by undoing all the updates to objects and then terminating itself. The coordinator, in this case, will timeout and abort the whole transaction.
- (2) If a failure occurs after Step 3 of Phase 1 but before receiving the decision and successfully recording in the log, the cohort process has to send an inquiry message to the coordinator to get the decision.
- (3) If the status in the local transaction log is COMMITTED or ABORTED but not COMPLETED, then process the forward logs for commitment and backward logs for abortion.

4.3.2.4.3 Coordinator Selection

In the one-phase and two-phase commit protocols, one of the cohorts acts as the coordinator to resolve transaction commitment and to enforce the decision on other cohorts. The commit coordinator can be selected based on the following criteria.

- (a) Initial Transaction as the Coordinator:

The end-user is connected to the node where the transaction is started. Selecting this transaction process as the coordinator ensures that the end-user is never in the "in-doubt" period. If any other remote process is designated as the coordinator, then a failure of the coordinator or the network during the "in-doubt" period will leave the end-user in a state of doubt.

- (b) Cohort with the most critical resources as the coordinator:

Because the coordinator does not have any "in-doubt" period, the resources held by the coordinator are never locked indefinitely. Therefore, it is attractive to make the cohort that is locking the most critical resources as the commit coordinator.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

4.3.2.4.4 Comparison of One-Phase and Two-Phase Commit Protocols

These two kinds of protocols can be compared on the basis of i) the response time, and ii) the duration of the "in-doubt" period. We see that the suitability of these two protocols is highly dependent upon the structure of the transaction. This is illustrated below using a set of examples. The following two examples have been taken from [BALT81].

Example 1:

Consider the example where a transaction consists of only two cohort processes A and B. Transaction A is the initial transaction that starts B; A also acts as the coordinator in both one-phase and two-phase commit protocols. The one-phase and two-phase commit protocols for this case are shown in Figures 4-6(a) and 4-6(b) respectively.

We will use the following notation:

Dmin = Minimum possible delay in the network

Dmax = Maximum delay in the network

tcc = Maximum time required for the coordinator to locally process transaction commitment after receiving all DONE messages in one-phase and READY messages in two-phase protocols.

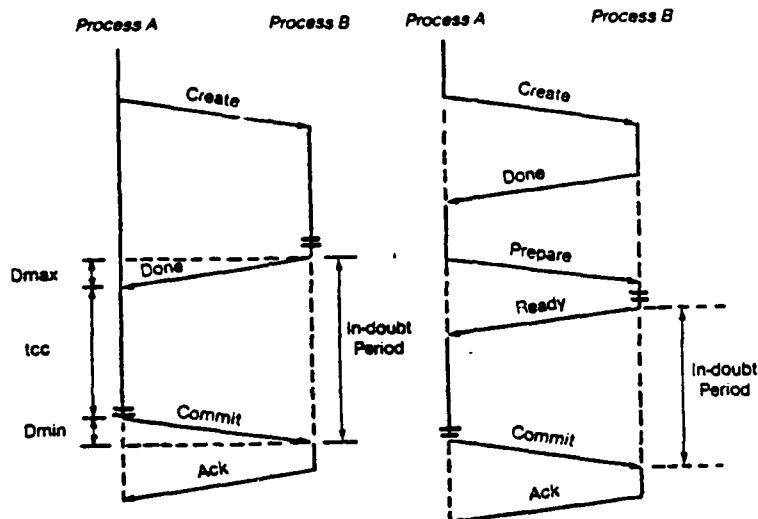
tmin = Shortest time for any cohort to complete execution of its assigned task; this time is measured from the instant of starting the initial transaction.

tmax = Longest time for any cohort to complete execution of its assigned task; this time is measured from the instant of starting the initial transaction.

Pmax = Maximum time required for any cohort to "prepare" itself after receiving a PREPARE message and to send out a READY or ABORT message.

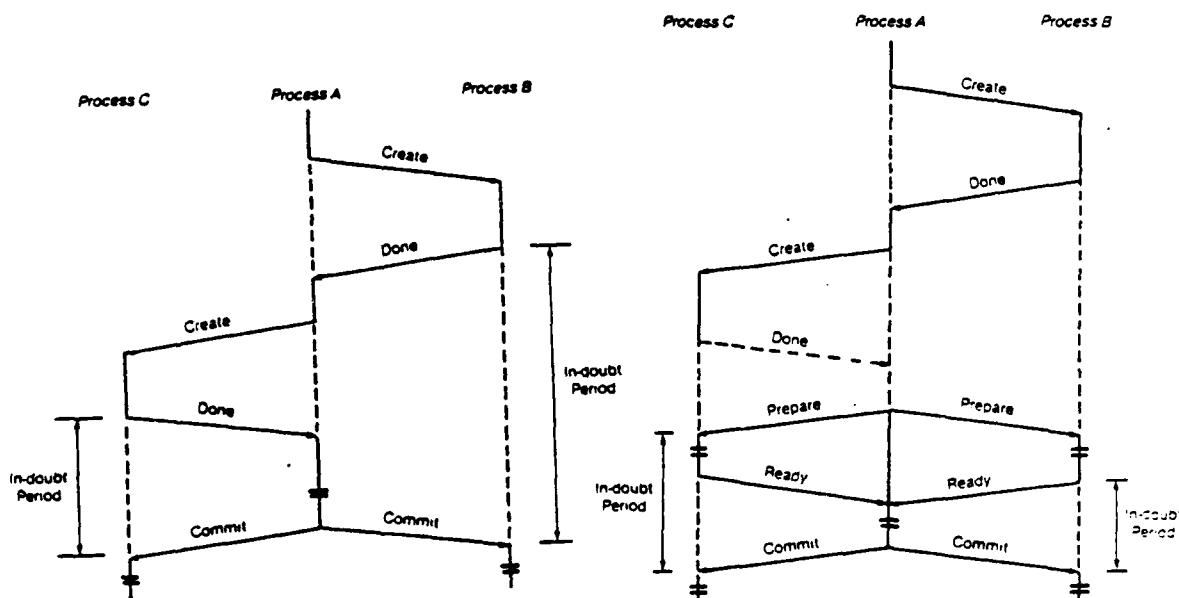
Pmin = Minimum possible time required by any cohort to complete processing the PREPARE message and send out a READY or ABORT message as a response.

In this example we see that the in-doubt period is the same for both cases, and it is equal to $2D_{max} + t_{cc}$. However, the extra phase of the 2-phase commitment introduces additional delay which increases the response time; therefore, two-phase commitment is undesirable in this case.



One-Phase Commitment
FIGURE 4-6 (a)

Two-Phase Commitment
FIGURE 4-6 (b)



One-Phase Commit Protocol in Example 2
FIGURE 4-7 (a)

Two-Phase Commit Protocol in Example 2
FIGURE 4-7 (b)

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Example 2:

Next, we consider an example that consists of three transactions. Transaction A, which is the initial transaction, first creates cohort B, waits for some response, and then based on this response initiates another cohort process C. Figures 4-7(a) and 4-7(b) show the corresponding one-phase and two-phase commitment protocols.

The in-doubt period, assuming no coordinator crash, in the two-phase commit is at-most:

$$(D_{\max}-D_{\min}) + (P_{\max}-P_{\min}) + 2D_{\max} + t_{cc} \dots (E1)$$

In case of one-phase commitment, the in-doubt period for B is equal to:

$$\text{Processing time for C} + P_{\max} + 2D_{\max} + t_{cc}$$

If the availability of resources held by B is important, then the two-phase commit protocol is recommended. Assuming that the terms $(D_{\max}-D_{\min})$ and $(P_{\max}-P_{\min})$ are relatively small as compared to the processing time of the cohort C, we can see that in this case the two-phase commit protocol is definitely more desirable than the one-phase commit protocol.

In general, in a one-phase commit protocol the longest possible duration of the in-doubt period for a cohort (one that completes execution earliest) assuming no coordinator crash is given by:

$$(t_{\max}-t_{\min}) + 2D_{\max} + t_{cc} \dots (E2)$$

For a two-phase commit protocol, this period is equal to (E1).

The two-phase commit protocols offer one distinct advantage over the one-phase commit protocols with respect to releasing the read locks. In a two-phase commit protocol, the read locks held by a cohort can be released during the commit-pending state. This does not violate the serializability of the transaction. In fact, the commit decision by the coordinator corresponds to the end of the growing-phase of two-phase locking.

4.3.2.4.5 Presumed Abort and Presumed Commit Protocols

The two phase commit protocol described in Section 4.2.4.2 requires force write of two log records at each cohort. These records are COMMIT-PENDING ABORT and COMMIT (ABORT). At the coordinator, two log records are written: COMMIT/ABORT and COMPLETE. Only the COMMIT/ABORT log record is forced. One of the important criteria in designing commit protocols is the number of synchronous write (force) operation. This number should be minimized in order to decrease the response time. Secondly, a commit protocol should be able to forget about the outcome of commit processing after certain period. This is

important to prevent the process manager database in the volatile storage from growing to an unmanageable large size.

The Presumed Abort (PA) and Presumed Commit (PC) protocols were proposed in [MOHA83] to address some of these issues. These protocols also address the problem of exploiting read-only transactions during commitment.

Presumed Abort (PA) Protocol

The basic principle in this scheme is that, regarding queries from cohorts or the coordinator, the absence of any information about the transaction status at any node means that the transaction was aborted. This leads to the asynchronous writing of ABORT records both by the coordinator and each of the cohorts; secondly, no acknowledgements need to be sent for an ABORT message from the coordinator. Additionally, the coordinator need not write a COMPLETE record after an ABORT record. In the two-phase commit protocol, the coordinator can "forget" the transaction only after it has made sure that all cohorts are aware of the abort decision. Such a protocol requires two log records to be written; however, only one of these records, viz., the COMMIT record, has to be forced.

A read-only cohort sends a READ-ONLY vote in response to the PREPARE message from the coordinator, releases its locks and "forgets" the transaction. The cohort writes no log records. No COMMIT/ABORT message need to be sent to a read-only cohort by the coordinator.

If the coordinator gets all READ-ONLY votes, then it is called a read-only coordinator. Such a coordinator writes no log records. A transaction is partially read-only, if at least one of the cohorts votes READY/ABORT. For a successful partially read-only transaction, there is at least one READY vote and others are READ-ONLY votes. In such a case, the coordinator writes a COMMIT record in the log and sends the COMMIT message only to those cohorts that sent a READY vote. If any of the cohorts votes ABORT, the coordinator processes abort according to the Presumed Abort protocol rules.

Presumed Commit (PC) Protocol

Because in most of the cases transactions commit, it seems more reasonable to require ACKs from ABORT messages instead of COMMIT messages. This would tend to make commit processing cheaper. In case of no information, the transaction is presumed to have committed; therefore, the name Presumed Commit (PC) is given to this protocol.

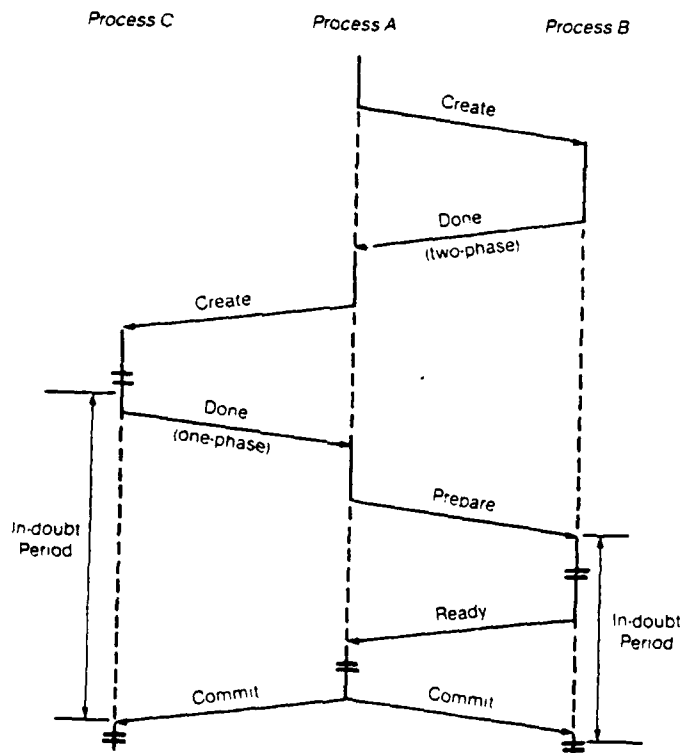
In this scheme, before sending PREPARE messages to any of the cohorts, the coordinator records the names of all the cohorts on the stable storage. This is called the collecting record; therefore, when the coordinator process crashes and the transaction is aborted, the recovery process makes sure to send the ABORT messages to all the cohorts and to get their ACKs. Only after getting all the ACKs can the transaction be forgotten safely.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Here in this scheme all the abort records are to be forced and all ABORT messages are acknowledged by the subordinates. The commit records need not be forced by the subordinates. The coordinator force-writes both commit and abort records. The coordinator writes a COMPLETE record in the log only after an ABORT record. For read only transaction, in PA no log records are written, whereas in PC, a COMMIT record is written at the end of the first phase and then the coordinator "forgets" the transaction.

In summary, for a completely read-only transaction, the coordinator writes two records: COLLECTING (forced) and COMMIT (not forced). The coordinator sends one message (PREPARE) to the cohorts. The cohorts write no log records, but each one of them sends one message (READ-ONLY).

For committing a partially read-only transaction, the coordinator sends two messages (PREPARE and COMMIT) to update cohorts and only one message (PREPARE) to read-only cohorts. It writes two log records: COLLECTING and COMMIT, both of which are forced. A read-only cohort, as in the completely read-only case, i.e., it sends a READ-ONLY vote. An update cohort sends one message (READY) and writes two log records, one of which (COMMIT-PENDING) is forced and the other (COMMIT) is not forced.



Hybrid Commit Protocol in Example 2
FIGURE 4-8

4.3.2.4.6 Hybrid Commit Protocols

It has been pointed out in the preceding discussion that:

- (i) The two-phase commit protocols unnecessarily increase the response time in the case of transactions with simple execution schemes. In such cases the one-phase commit protocols are to be preferred.
- (ii) Two-phase commit protocols are recommended where the availability of the resources is an important issue.

The hybrid protocol scheme proposed in [BALT81] tries to provide a flexible selection of both these techniques within the same transaction structure. The coordinator selects different commitment schemes with different cohorts. An example of hybrid commitment is shown in Figure 4-8. As shown in this figure, process B uses two-phase commit protocol whereas process C uses one-phase commit protocol. At the time of creating a cohort process, the commit protocol to be followed is specified. It can be observed from this example that using the two-phase commit protocol for process C would have only added more delay in the completion of the transaction without providing any advantage.

4.3.2.5 Replication Management in Distributed Systems

A distributed computer system can offer benefits if objects are replicated and their management adjusted to take advantage of the multiple copies. The benefits can include improvements in performance and reliability. The former is possible due to the reduction in communication cost to access an object and the increase in parallelism of operations on an object. The latter is possible because operations can continue despite the loss of system components. For example, if a directory is replicated on every site on a distributed system, the cost of reading it is the cost of accessing a local storage device (e.g., there is no overhead incurred due to communication between two sites). It is possible for users on multiple sites to be simultaneously accessing the directory, further improving a system's performance. Finally, if a site fails, the directory can still be accessed by any operating sites.

Unfortunately, increases in reliability and performance do not come for free and in many cases are not mutually attainable. This tradeoff in system attributes is often determined by a correctness criterion that describes a relationship between the values of the replicas of a distributed object at any point in time. The correctness criteria must ensure that a replication update algorithm satisfies the mutual consistency property: all replicas of an object converge to the same state and become identical if update operations cease.

The most common requirement of consistency has been based on the notion of serializability of transactions -- the effect of the execution of a set of transactions is equivalent to some serial schedule. This is called a strong consistency requirement. It requires that some subset of the set of copies of

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

an object converge to a common state within the time it takes for a single transaction's execution.

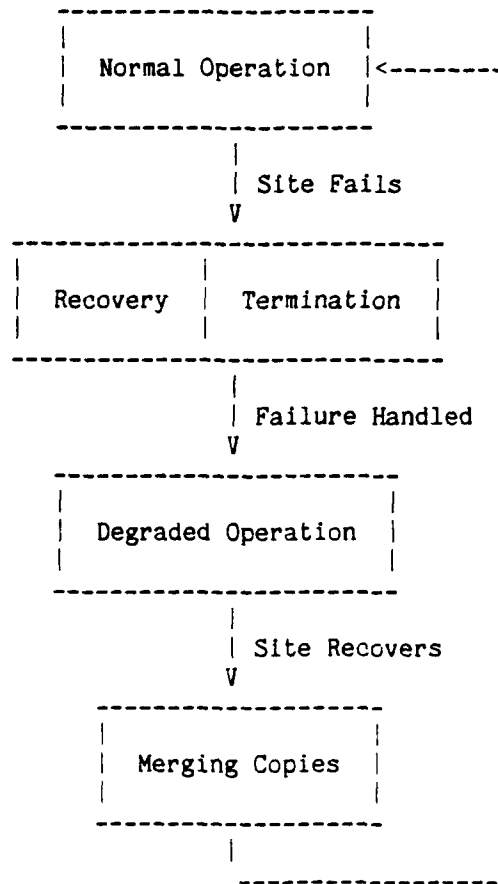


Figure 4-9. Replication Management Phases

A different requirement for consistency may be derived from observing applications such as directories, calendars, or network resource tables. The use of these objects does not require that they have the most up to date information. For example, a network name server may access an object's old site and be directed to its new site, or a message may be routed through a network over a longer than optimal path because its routing table is slightly out of date. But the services may be achieved with using non-identical copies of an object. The consistency requirement for them is that they eventually converge to a common state if changes to the object stop. This convergence may span the time it takes for multiple transactions to execute. This correctness criteria is called weak consistency. It is a property of the application.

A third correctness criteria related to consistency exists. It is called semantic consistency and is a property of a set of transactions of an application. Semantic consistency seeks to find relationships (e.g., commutative, inverses, etc). between the effects of transactions that allows them to be executed according to a non-serializable schedule. To the best of our knowledge, semantic consistency has not been applied to performing updates on replicated objects. However, it has been proposed as a technique for merging replicated objects that existed in different network partitions when the partition is repaired.

In some sense, consistency criteria can be seen as points on a spectrum differentiated by the amount and type of activity that may occur in a system at any point in time. The three consistency criteria discussed are points in this spectrum that are currently known and are not meant to be interpreted as the only possible criteria.

The problem of managing replicated objects can be divided into four parts -- normal operation, detecting a failure and transitioning into a degraded mode of operation, operating in a degraded mode, and merging partitions during recovery (see Figure 4-9). The first and third part of the problem are the same problem but in a different operating environment. They are almost always addressed by a single mechanism and will be discussed as a single problem in this paper. Transitioning into a degraded mode has two subparts -- termination and recovery. Termination is the action taken by operational sites when they determine that a site has failed. Recovery is the action taken by a site to clean up any old transactions when it becomes operational after previously failing. Finally, merging is the act by a set of sites of bringing multiple copies of an object into a consistent state. It is helpful to recognize these distinct parts in order to understand the advantages, disadvantages, and applicability of the algorithms to be discussed.

A number of algorithms have been designed to ensure that the copies of an object meet some consistency correctness criteria. This section discusses how some of these algorithms operate and their effect under normal and degraded operation. Degraded operation exists when either a site goes down, a communication link is lost, a network is partitioned, or a message is lost or duplicated. Some update algorithms are tolerant to some of these failures and have no explicit distinction between normal and degraded operation. Other algorithms cannot tolerate failures and may block an operation until recovery from the failure has been completed, or abort the operation.

An attribute of interest is availability: the probability that an object can be accessed and an operation successfully performed. Those algorithms that ensure weak consistency result in a higher availability of objects. Strong consistency requirements typically restrict the concurrency level to a single update transaction and multiple read only transactions. They further restrict access to the replicated object by only one partition during degraded operation.

There are some general relationships among a replication algorithm's distribution of control, consistency criteria, reliability, and performance. Centralized control supports strong consistency and freedom from deadlock well, but is susceptible to single points of failure. It potentially can

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

create performance problems (bottlenecks) and thereby reduce the availability of an object under both normal and degraded operation. Decentralized control can potentially increase the throughput of a system and the tolerance of a system to single point failures. Weak consistency is not appropriate for centralized control; it is naturally achieved through decentralized control. Weak consistency increases a site's throughput and response time, an object's availability, and a system's resilience to multiple failures.

4.3.2.5.1 Centralized Control

In these schemes some distinguished node is assigned the function of coordinating the concurrent updates and accesses on the multiple copies. These schemes have the obvious disadvantages in terms of reliability and performance characteristics. Most of the centralized schemes tend to be simple from the view point of design and implementation. In this section we describe some representative centralized schemes for managing multiple copies of an object in a distributed system.

The first scheme that we describe here is known as the primary copy algorithm, originally proposed by Alsberg and Day [ALSB76]. In this scheme one of the nodes is made the primary node, responsible for serializing the updates. All transactions are forwarded to this centralized node where they are executed serially. All computation is performed at this node. When the update values have been computed, the primary node broadcasts them as 'perform update' messages to all other nodes in the system. The primary node then waits to receive acknowledgements from all nodes in the system before processing the next transaction.

The major problem with this scheme is that it permits no parallelism among transaction executions; therefore, this is not a very attractive scheme from the viewpoint of performance. This scheme can be made more efficient by eliminating the acknowledgements to the update messages. The acknowledgements indicate that an update has been made by the node sending the ack. Before proceeding with the next transaction, it is not necessary for the primary node to wait until all nodes have performed the updates of the preceding transaction. The only thing that needs to be ensured is that every node performs the updates by different transactions in the same order. This is achieved by assigning a sequence number to each update transaction that is executed at the primary node. Each time a new transaction is processed by the primary node, it is assigned a sequence number that is one larger than the sequence number of the last transaction processed by the primary node. This sequence number is attached to the update messages sent to the other nodes, which process the update messages in the order of their sequence numbers.

In both these schemes the primary node performs two functions, it (a) performs computations for the update operations, which includes accessing databases, computing updates, and finally updating the local database and broadcasting update messages, and (b) enforces concurrency control among the transactions. The centralized scheme that we discuss next moves the function (a) to the other nodes to reduce the load on the primary node. The primary node essentially supports function (b) by granting requests for locks on the

data to be updated by the remote nodes. Thus, in this scheme the primary node acts as the lock manager.

In the centralized locking scheme an update algorithm is processed as follows:

- (1) A node desiring to execute an update transaction A requests locks from the primary node for all objects referenced by A.
- (2) The primary node grants the lock if there is no conflict; otherwise, it queues the request. In case the lock is granted, a 'grant' message is sent to the requester. The primary node maintains a queue for each item and a request waits in only one queue at a time.
- (3) Once a node gets all of the requested locks, it processes the transaction using its local copy of the database. After computing the updates, it sends 'perform update' messages to all nodes in the systems and waits for their acknowledgements. The local copy of the database is updated meanwhile.
- (4) When all acks have been received, a 'lock release' message is sent to the primary node managing the locks. The primary node releases the locks in response to such messages, and grants the locks to the next request in the queues.

This algorithm can be improved by eliminating the waiting for acknowledgements in step (3) by using the sequence numbers as described in the previous scheme. In the modified scheme, the primary node, in addition to granting locks, also assigns a sequence number to the transaction request. This sequence number is also attached to the 'perform update' messages that are broadcast by the node processing the transaction. A node can process a transaction only if it has seen all updates from the other transactions that have sequence numbers smaller than the sequence number of the current transaction.

One major disadvantage of this scheme is that the sequence numbers sometimes cause unnecessary delay. For example, even if two transaction have no common data items in their read or write sets, the transaction with the smaller sequence number will not be processed at the originating node until the updates from the other transaction (in this case, one having a smaller sequence number) have been seen at that node. Thus, a long transaction would unnecessarily delay other transactions even if those transactions do not conflict with the long transaction. One approach to improve this is by maintaining some additional information at the central node. This information consists of keeping track of the last update transaction that referenced each item in the database. The central node maintains a table LAST, where LAST(i) is the sequence number of the last update transaction that locked item i. When a transaction requests locks from the central node, along with the grant message, a list of sequence numbers called WAIT-FOR list is sent to the requester. Before processing the transaction after receiving the grant message, the requester must wait to see the updates from the transaction whose sequence numbers appear in this list. This requires that each node must

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

maintain a list of the sequence numbers of all the 'perform update' messages that it has seen. This list can potentially be very large. Another overhead arises due to the preparation of the WAIT-FOR lists at the central node. This improved scheme requires some extra processing and storage overhead.

Another scheme that is similar to the above scheme uses DO-NOT-WAIT-FOR list instead of a WAIT-FOR list. This list is also called the hole list [GARC81]. When the primary node grants locks to some transaction A, it also attaches a list containing the sequence numbers of all those transactions that are still holding locks on some resources. All those transactions that appear in this list obviously do not conflict with transaction A; otherwise, A would not have been able to acquire its locks. The hole list indicates the sequence numbers of those transactions for which A need not wait to complete. The hole list is only a partial list of such transactions. There may be other transactions which are not in this list but they do not conflict with A either. However, it is hoped that the update messages from such transactions would reach the node requesting to process A without delaying A excessively. It is assumed that such transactions would not cause excessively large delays. Transaction A is processed at its originating node only when updates from all transactions with smaller sequence numbers but not in the hole list have been seen at that node.

4.3.2.5.2 Voting

Voting algorithms have a collection of cooperating object managers using a common set of rules to determine whether or not an update can be made. The control of the algorithm may be centralized or decentralized depending on whether the voting is done at one site or multiple sites. A strong consistency criteria is applied to a subset of the copies and weak consistency may be applied to the complete set of copies. Because only a subset of the copies are required for an update, certain failures can be withstood. In general, if there are n total copies/votes and m copies/votes are needed to perform an operation, $n-m$ sites may suffer failures and the operation can still succeed (assuming that there is a single copy/vote at each site). The failures can include site outage and lost and duplicate messages. If a network partition occurs, there must exist a partition with at least m copies/votes in order for an update to be made. The requirements for a read operation to be made are considerably weaker.

4.3.2.5.2.1 Majority Voting

Thomas proposed an approach for managing replicated data that uses a single timestamping mechanism for enforcing both internal and mutual consistency [THOM79]. All objects are replicated at every site and all of the computation for a request (transaction) is done locally. To determine if an update is to be committed, each site votes based on the currency of its copy of the object compared with the currency of the copy used to compute the new value. If a majority of the sites agree that the update request is based on current variables, a new uniquely identifiable version of the object is created at all sites. Because the request is executed locally it is not known

if the base variables are current or in conflict until an attempt is made to commit the update. This approach to updating is an optimistic approach to concurrency control.

The serial consistency of the multiple copies is guaranteed by the fact that if any two updates are accepted then at least one site voted to accept each. This common site (or sites) ensures that the two updates have a consistent view of the replicated object.

The following describes the majority consensus algorithm in terms of an object oriented system. An object manager and the instances of the type of objects it manages are replicated on every site in a network. An application program executes operations on the object by issuing requests that are serviced by the object manager on its site. The object manager performs the operation by reading a set of objects and possibly computing new values for the objects. The objects read are called the base objects. If there are no updates, there is no interaction with remote object managers. If the operation involves updates, the object manager submits a request to its peer object managers on which they vote to determine if the update is to be committed. The majority consensus algorithm defines the interactions between the peer object managers. The algorithm consists of five rules -- inter-object manager communication, request generation, voting, request resolution, and update application. Each rule is described below.

Inter-Object Manager Communication Rule: The communication rule defines how object managers are topologically organized so that they can exchange voting information. The two extreme points are a logical star and a virtual ring or daisy chain. There are obviously many control points in between. A star has the advantage of using broadcast to minimize time at the expense of extra messages (results being returned) and all sites having to execute the voting algorithm. A star represents centralized control and would be very similar to a one or two phase commit protocol. The daisy chain has the advantage of minimizing the number of messages transmitted and the number of sites that have to execute the voting algorithm at the expense of increased time to reach a majority consensus. A daisy chain represents decentralized control and can be more resilient to coordinator failures.

The majority consensus algorithm, as specified by Thomas, uses a daisy chain topology. An object manager receives a request, executes the algorithm, and forwards the request to the next site in the chain. To make the algorithm resilient to site failures and lost messages, a timeout and retransmit procedure is assumed. After a site has forwarded a request it sets a timer based on how close to a consensus the vote is (the more votes needed, the larger the timeout value). If the timer expires prior to receiving the voting result, the site attempts to determine if the site it forwarded the message to is available. If it is the timer is reset; otherwise, the request is forwarded to a different site. Note that this can result in a diffusing computation.

```
communication_rule is
  next_site := successor(this_site);
  loop
```

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

```

timeout := f(number_for_majority - number_of_sites_voted);
send(request) to next_site;
select
    receive(result);
    exit;
or
    delay timeout;
    timeout := f(1);
    send(are_you_alive) to next_site;
    select
        receive(next_site_is_alive);
    or
        delay timeout;
        next_site := successor(next_site);
    end select;
end select;
end loop;
--the function successor returns the name of a site
--that has not voted yet;
--the function f returns a timeout value based on the
--the number of sites to be visited

```

Request Generation Rule: When an object manager has completed the computation for an update, it submits a request for voting. The request must be uniquely identifiable by all of the object managers. A globally unique time-stamp (say TS) is generated to identify the request. There are two parts to a timestamp - a counter and an id. The counter is a monotonically increasing integer. The id is the name of the site or manager that generated the timestamp. The high order bits of the timestamp are used for the counter and the low order bits for the id. A timestamp uniquely orders requests throughout a system. The smaller the timestamp, the older the request. Every time a request is generated at a site the value of the local counter is increased.

Timestamps are used to determine the priority of operations. A slow or less active site may continually generate what appear to be older requests. This is prevented by adjusting the local counter when an update is committed from a remote site. The local counter is set to the maximum value of the local counter and the counter in the timestamp request.

In addition, a request also identifies the base variables that are to be updated. In order to determine if a request used a current object, a timestamp is associated with each copy of an object which identifies the last request that changed the base object. An update request therefore contains the request timestamp and the base objects with their respective timestamps.

```

type timestamp is
    record
        counter: integer;
        object_manager: id; --a subtype with base type integer
    end record;
type object is
    record

```

```

    k: key; --an identifier that names an object instance
    v: value;
    ts: timestamp --for the last committed update
    pending: boolean;
    pending_ts: timestamp; --for a pending update
  end record;
  type request is
    record
      update_id: timestamp;
      base_objects: {object};
      votes: {vote};
    end record;
  request_generation_rule is
    counter := 1 + max(counter, max({base_object.ts.counter}));

```

Voting Rule: A site receiving a request uses a set of voting rules to implement concurrency control. The timestamps of the base objects of the request are compared with those of the local copies of the objects. If the request is based on current values and there is no pending update that conflicts with this request, the manager votes for accepting the request. If any base object is old, a reject vote is cast. If a base object is more current than the local copy, the request is deferred.

When the base objects are current and there is a conflicting request a decision must be made about the current request that prevents deadlock. To do this, timestamps are used as request priorities. The greater the timestamp, the higher the priority of the request. The policy assumed is to always vote on an older request to force it to completion and to defer voting a newer request. The vote cast for an older request that conflicts with a pending request is a pass. This serves as a signal to other managers that there may be a deadlock.

Once a manager has voted on a request, it cannot change its vote. When a majority of the managers vote for accepting a request, all conflicting requests must be rejected. Any two non-conflicting requests that have some object in common are guaranteed to be serializable because there exists at least one manager that voted for accepting both requests.

Voting "okay" for an object is equivalent to setting a lock on an object and the beginning of an in-doubt period for a two phase commit protocol. However, at this stage it is more resilient than a two phase commit protocol because there is not a single coordinator and therefore no single point of failure. This part of the algorithm is resilient to failures and is non-blocking in the event of failures (provided there is a majority of active, reachable sites) because of the communication discipline and moving locus of control.

The following assumes that each site names its local copy of the object "lo" and the base object in the request "bo."

```

voting_rule is
  if lo.ts = bo.ts and lo.pending = false then
    vote := okay;

```

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

```
    lo.pending := true;
    lo.pending_ts := bo.ts;
  elsif lo.ts > bo.ts then
    vote := reject;
  elsif lo.ts = bo.ts and lo.pending = true and
    bo.ts < lo.pending_ts then
    vote := pass;
  else vote := defer;
  end if;
```

Request Resolution Rule: After voting on a request an object manager checks to see if it has resolved the fate of a request. If a manager resolves a request it becomes the coordinator for completing the request and is responsible for notifying all other object managers of the result. This means that the resolving manager must remember the request and its result until all object managers have successfully received the result.

Resolution involves tabulating the votes and determining if a consensus has been reached. A consensus is reached when there is a majority of the votes cast for one action, or it is impossible to reach a majority vote. When a result is reached any deferred conflicting request can be addressed. If the request is accepted, conflicting requests are rejected (i.e., they must be restarted using the new current values). If the request is rejected, any deferred conflicting requests can be voted on.

The resolution rules are described below using a function named majority. The function tabulates the previously cast votes and the vote of the object manager that is executing the resolution rule and returns a value of yes, impossible or no which means a majority consensus has been reached, a majority consensus is not possible, no result has been reached yet.

```
resolution_rule is
  if vote = okay and majority({vote}, okay) then
    send(accept, request) to {object_manager};
  elsif vote = reject and
    majority({vote}, reject) = impossible then
    send(reject, request) to {object_manager};
  elsif vote = pass and
    majority({vote}, pass) = impossible then
    send(reject, request) to {object_manager};
  else forward_request;
  end if;
```

A weaker reject rule is to reject a request as soon as it has a single reject vote. However, if the computation does diffuse because of the timeout and retransmit strategy of the communication rule, the request may be resolved at multiple sites and additional overhead due to extra message transmission and processing will be incurred.

Update Application Rule: When notification of the acceptance of a request is received the value of the local copy of the object is updated. The

notification is received from the object manager on the site that resolved the request. If a site is currently unavailable, the resolving object manager will periodically attempt to notify that site's object manager of the update until it succeeds. It is possible for a site to be unavailable for an interval of time during which multiple requests are accepted. There is no guarantee that when the site becomes available again that the object manager will receive the updates in the order that they were accepted. Since the updates have been accepted and are serialized, it is not necessary for a site to apply every update to its copy of the object. Only those updates that are more current than the local copy must be applied. Less current updates may be ignored.

```

update_application_rule is
  if bo.ts > lo.ts then
    lo.value := bo.value;
    lo.ts := bo.ts;
  end if;

```

The availability of updating the data using the majority consensus is given by:

$$\sum_{k=\lceil \frac{N}{2} \rceil}^N p^k (1-p)^{N-k} \binom{N}{k}$$

4.3.2.5.2.2 Weighted Voting

Gifford [GIF79] designed an algorithm that generalizes voting by letting each participant in a distributed decision have a varying number of votes. This allows multiple control policies to be implemented. For example, if one site has all of the votes, we have a primary site, or centralized, control policy. Whereas, if each site has an equal number of votes, a decentralized control policy exists.

A participant is an object manager that has a copy of an object. Each object is assigned some number of votes. (It is also possible to assign the votes to the object manager and assume that every instance of an object controlled by an object manager has the same number of votes.) In order to perform an operation, a transaction must collect a quorum -- the number of votes required for that operation. Each object manager has a collector and an agent that it uses to collect a quorum. A collector, at the site where the operation is initiated, sends requests to agents at remote sites to forward their votes to it. The collector in turn tabulates the votes and determines if an operation can be executed.

The operations assumed by Gifford are read and write. The total number of votes allocated to the copies is constrained by the relation $r+w > v$, where r is the number of votes needed to read an object, w is the number of votes needed to write an object, and v is the total number of votes allocated. This relation ensures that there is an intersection between read quorums and write quorums and is the basis for achieving mutual consistency. It guarantees that there is at least one current copy in a read quorum. Version numbers are associated with each copy in order to determine what is a current copy.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Gifford designed the algorithm to manage multiple copies of files. The algorithm is designed to run in an environment that has an underlying file system and transaction system. The file system implements objects of type file and the operations create, delete, open, close, read, and write. The transaction system implements transactions and the operations begin transaction, commit, and abort. It is assumed that the underlying system provides for the consistent management of individual copies of a file. The problem of internal consistency of individual objects has been separated and made a more primitive operation from the management of copies of an object. This is a departure from Thomas's design and allows for a simpler, more flexible replication mechanism.

A copy of an object has two kinds of information. The first is a prefix to the object that describes the voting rules and the configuration of the copies of an object. The voting rule information includes the size of a read quorum (r), the size of a write quorum (w) and the number of copies of the object. The configuration information includes a list of the names of the copies of the object, the version number of a copy, and the number of votes per copy. This information is stored with each copy of the object.

The second kind of information describes a copy of the object. It includes the value of this copy of the object and the version number of this copy. A version number is a monotonically increasing integer that describes the currency of a copy. A subset of the copies of the objects may have differing version numbers and, therefore, differing values.

```
type configuration is
  record
    name: id;
    votes: integer;
    version: integer;
  end record;
type copy (no_copies: integer) is
  record
    total_votes: integer;
    read_quorum: integer;
    write_quorum: integer;
    copies: array(1 .. no_copies) of configuration;
    version: integer; --of this copy
    value: v; --of this copy
  end record;
```

In order to perform a read operation, a transaction must collect a read quorum. A read quorum is collected by attempting to gain access to remote copies of the object and acquire their votes. This is achieved by an agent acquiring a lock from the object manager. The control block of the remote copy is read and its version number is returned to the requesting site. The number of votes allocated to that copy is added to the number of votes collected from other copies. When a sufficient number of copies have been locked so that r votes have been obtained, one of the copies with the most current version number is read. Once a quorum has been collected for a

transaction, all subsequent reads of the object by the transaction do not encounter delays due to collecting a quorum.

```
--the following is assumed:
--there is a copy of the object manager on each site
--that consists of a set of objects of type copy and
--the code below.
--there is an object c of type copy that has been
--declared and initialized on a site.
--there is a begin_transaction and end_transaction that
--encapsulate a sequence of read_replicated_data and
--write_replicated_data calls.
--a read quorum is only collected the first time that
--read_replicated_data is called.
```

```
agent is
  receive(request);
  case request is
    when form_quorum =>
      file.read(name.version, name.votes);
      --lock is implicitly set when object is read
      send(name.version, name.votes) to requester;
    when read_data =>
      file.read(name.value);
      send(name.value) to requester;
    when write_data =>
      file.write(name.version, name.value);
  end case;
```

```
collect_read_quorum is
  read_votes := 0;
  current_version := unknown;
  for i in c.no_copies'range loop
    c.copies(i).version := 0;
  end loop;
  --the following statements for efficiency should be
  --done in parallel by i spawned tasks
  for i in c.no_copies'range loop
    send(form_quorum, c.copies(i).name) to agent(i);
    receive(version, votes);
    read_votes := read_votes + votes;
    c.copies(i).version := version;
    if version > current_version then
      current_version := version;
    end if;
  end loop;
  exit when read_votes >= c.read_quorum;
end loop;
```

```
read_replicated_data is
  if first_read then
    collect_read_quorum;
    first_read := false;
  end if;
```

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

```

i := agent_with_current_copy;
send(read_data, c.copies(i).name) to agent(i);
receive(object_value);
return object_value;
end if;

```

A write operation is performed by a transaction first collecting a read quorum, next collecting a write quorum, and finally updating copies of the object. A read quorum is collected first to ensure that multiple transactions can not be concurrently writing disjoint copies of the same logical object. (This can occur since $w < r$ and $w < v/2$ is a permissible allocation of votes. For example, we could have $v=10$, $r=9$, and $w=2$. This would allow four concurrent write operations.) A transaction collects a write quorum by using the same protocol used to collect a read quorum with two minor changes. It is assumed that any read locks that the transaction holds can be converted to write locks. Only the votes of those copies that are current are used in obtaining a quorum. When a quorum is collected, an update may be done. An update involves incrementing the current version number for the object, and writing the new version number and data into the copies represented by the quorum.

```

collect_write_quorum is
loop
write_votes := 0;
for i in c.no_copies'range loop
if c.copies(i).version = current_version then
write_votes := write_votes + c.copies(i).votes;
end loop;
exit when write_votes >= c.write_quorum;
for i in c.no_copies'range loop
if c.copies(i).version < current_version and
c.copies(i).version <> unknown then
background_copy(current_version) to agent(i);
end if;
end loop;
wait done;
end loop;
--the function background-copy makes a known active copy
--that has an obsolete value current, updates the local
--copy of the configuration, and signals when done.
--note that when a copy comes online, its initialization
--code must check for any outstanding and incomplete
--attempts at forming a quorum.

write_replicated_object is
if first_read then
collect_read_quorum;
first_read := false;
end if;
collect_write_quorum;
current_version := current_version + 1;
--the following statements for efficiency should

```

```

--be done in parallel by i spawned processes
for i in c.no_copies'range loop
    if c.copies(i).version = current_version - 1 then
        send(write_request, current_version, new_value)
        to agent(i);
    end if;
end loop;

```

Note that the size of a read quorum and a write quorum may vary from the total number of votes to the minimum number of votes that any site has. A quorum represents the minimum number of sites (in terms of votes) that must be operating in order to permit a read or write to occur. When an update is done to an object, there can exist copies of the object that have obsolete values because they were not included in the quorum. Similarly, when a write quorum is collected it is possible to have to collect votes from more sites than anticipated if some sites have obsolete copies of an object.

Several optimizations have been proposed for the algorithm. Two changes can improve the access time of a read operation. The first is to maintain a table that describes the time that it takes for a local site to access every site in the system. When a read quorum is obtained, the site that can be accessed fastest and has a current copy is the site from which the object is read. A second read optimization is to introduce so called "weak" representatives. A weak representative has no votes and does not participate in forming quorums. It contains a copy of the object on a high speed device and can improve the performance of accessing the object.

Two improvements for write operations are worth discussing. The first is to have servers that periodically look for obsolete copies of an object. When an obsolete copy is found it is made current. This can improve the probability of forming a write quorum and therefore the performance of a write operation. A second is to observe that in order to perform a write operation a read quorum and then a write quorum must be formed. Since $r + w > v$, then $r > v/2$ and/or $w > v/2$. A minimum of the ceiling[$v/2$] votes must be obtained in order to do a write operation. Therefore, a write quorum should always be greater than or equal to the ceiling[$v/2$] votes. This also solves the problem of multiple write operations and eliminates the need to initially collect a read quorum when a write operation is to be executed.

The quorum approach can increase object availability under both normal and degraded performance. This is possible because not all of the sites on which a copy exists have to be up for an operation to succeed, nor do all of the active copies have to be available for an operation to succeed. It seems to share some of the same attributes as Thomas's majority consensus voting algorithm.

The weighted voting algorithm can tolerate a limited number of site failures, link failures, and lost messages up to the point where an agent commits an object manager to joining in a quorum. However, because the act of collecting a quorum uses centralized control (i.e., one site initiates and acts as the coordinator for collecting a quorum) an operation can only succeed if it is executed within a partition that contains sufficient sites to form a

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

quorum. If a site or link fails after a quorum has been formed and breaks the quorum prior to an update being committed, all of the resources of the participating object managers are blocked or the transaction is aborted. Because decentralized control is used in the majority consensus voting algorithm, it can tolerate failures of sites after they have voted (equivalent to joining a quorum). However, the added resilience is an attribute of decentralized control (specifically the timeout and retransmit strategy). It would appear that weighted voting could be made more resilient by using decentralized control.

Another difference between the algorithms is the number of sites that are notified of an update by the coordinator after an update has been decreed. Thomas requires the coordinator to update all copies; Gifford updates only those copies in the quorum. Since Thomas requires for serializability that each update intersect the previous update, only a majority of the object managers need to be notified of the update. The role of the update coordinator in the majority consensus algorithm can be weakened to notifying the majority of object managers that have voted okay for the update and then forget the result. However, if a timeout and retransmit strategy is used, there is a possibility for there to be a diffusing computation and therefore some object managers may think that a resolved request has not been resolved. Achieving a clean termination becomes a more difficult problem.

4.3.2.5.2.3 Weighted Voting and Directories

The weighted voting algorithm allows replicated objects to be read and updated. It enforces the serializability of transactions updating the copies of an object. But by so doing, it only allows one transaction at a time to be updating any part of the object. But what if the object is not logically a single object but a collection of objects, for example, an object of type directory that has multiple, potentially independent, entries in it? Applying the algorithm to directories may result in two problems. First, the algorithm may adversely restrict the concurrency level because a single version number is associated with each copy of the directory. Second, if a version number is associated with every entry in a directory, anomalies may arise in forming quorums due to an object being inserted at a subset of the sites or deleted from some site.

Daniel and Spector [DANI83] proposed extensions to the weighted voting algorithm that addressed the above problems. They assumed that a directory is accessed by giving a key that names the directory. There are several requirements on the keys: the total set of possible keys is statically defined; there is an ordering relation on the set; and there is a known least element, LOW, and greatest element, HIGH, in the set. The ordering relation dynamically partitions the keys into a set of disjoint ranges. A range consists of a directory entry (and its key), or a gap - the set of possible entries between two existing entries. To determine the currency of any object, a version number as defined by Gifford is associated with it. A range is the smallest object in a directory that can be accessed by a transaction and becomes the object granularity that requires concurrency control.

It is assumed that a type manager exists for object of type directory. It implements five operations - lookup, predecessor, successor, insert, and coalesce.

4.3.2.5.3 Operations Under Weak Consistency Requirements

The requirements of serial consistency and high availability pose somewhat conflicting goals. Under network partition conditions almost all of the serial consistency management techniques would permit at most one partition to perform operations. It is possible to support continued operations in different partitions when the consistency requirements are weakened. There are many applications where one can take advantage of the transaction semantics and the weak consistency requirements. Some examples of such applications are distributed dictionary, mail system, appointment calendars, routing tables in a communication, and some operating system data such as resource utilization information in a distributed system. Most of these systems have commutative transactions, and while performing a transaction one normally does not need to know the most up-to-date state of the data. Two examples of such applications are described below.

Example 1: Announcement Calendars

Announcement calendars are generally used for informing people about some important and interesting events happening in the community. Such calendars generally describe event along with its date, time, and the place. Such information records are inserted in the calendar for announcement. It is possible that an event may get cancelled; in such a case the event record is deleted from the calendar. The system supports the following functions: INSERT, DELETE, and LIST. The INSERT(x) function is used to insert an event record x in the announcement calendar. The DELETE(x) function is used for cancelling an announcement. This function may also be used to delete the announcements that are outdated. The LIST function is used to get the listing of all the announcements that are present in the calendar. If one needs to change an announcement, for example change the date or the place of the event, then we will assume that the old announcement is deleted and a new announcement is inserted. Once an event record has been inserted, none of its fields can be updated.

Now let us consider a centralized implementation of this system. Such an implementation would maintain the calendar as a database shared among many users. A user would be allowed to invoke INSERT and LIST functions to insert an announcement or to list the current announcements respectively. The LIST function is the read operation on this database. A user would be allowed to DELETE an announcement if it had appropriate access rights to do so. The implementation of such a system should support concurrency among the users, i.e., several users might be invoking these operations concurrently. We will assume that the INSERT and DELETE operations (which are the write operations on this database) are implemented as atomic actions. This will mean that effect of an insert or delete operation will be visible only after it has been successfully completed. In this application we can now permit reads and

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

writes to proceed concurrently. A read operation will not see the effects of an incomplete write. We can also allow write-write concurrency because the write operations do not share any data records. Thus, while a user is listing the current announcements it is possible that another user might be deleting some of the announcements. The serialization of these transactions does not solve this problem, i.e., a serialization that permits the LIST operation to execute before the DELETE operation would still keep the preceding reader unaware of the change. In fact, this particular characteristic of the calendar database is exploited to allow read-write and write-write concurrency.

Next we consider a distributed implementation of this calendar system. In this implementation let us assume that each user has a workstation, and these workstations are connected by a communication network. A copy of the calendar database is maintained at each of the workstations. A user performs the INSERT, DELETE and LIST operations on its database. The results of these update operations are periodically communicated to the other workstations which then update their local databases accordingly. The details of such a protocol by Fischer and Michael is given later in this section. Thus, while a user finds an announcement in the calendar by invoking the LIST operation, it is possible that another user might have deleted that record at some remote site. This uncertainty problem is the same as in the centralized implementation; however, in a distributed implementation the time required for the changes to be known at all copies would be significantly more than that in a centralized system.

Example 2: Directory Systems

In this example a directory consisting of names, telephone numbers and addresses of some persons are maintained in a database. The operations on this directory are the same as described in the previous example, namely, INSERT, DELETE and LIST. Names along with their associated records are inserted or deleted from the database. If any of the fields in an existing record are to be changed, then the current record is deleted and new record with the updated field values is inserted. The centralized and distributed implementations of such a system are identical to those described for the system in Example 1.

4.3.2.5.3.1 Fischer and Michael's Scheme

In [FISH82] one such system is described that implements a dictionary system. The algorithm presented there implements a "best effort" approximation to serial consistency. As mentioned earlier, the correctness of this algorithm depends on some special properties, such as commutativity, of the dictionary operations.

Basically, in this technique each node maintains its own copy of the database which represents that node's view of the world. Periodically communication between nodes takes place to send one node's view to another. In the distributed dictionary system there are three operations: INSERT, DELETE, and LIST. INSERT(x) adds an element x to the set of identifiers maintained by the dictionary system. DELETE(x) removes the element x from the

set if x is present in the set, otherwise it does nothing. LIST operation enumerates the current members of the set. Additionally, SEND(m) and RECEIVE(m) primitives support internode communication to exchange their current views

It is assumed that in this dictionary system, for each element x there is at most one INSERT(x) operation; DELETE(x) is legal at a node only if x is currently in the database maintained by that node. When an INSERT(x) operation is performed at a node, x is said to be in the view of that node until that node knows about a subsequent DELETE(x) operation.

The most simplistic approach to merging views of two nodes i and j would be to form a union, $V_i \cup V_j$, of their views. This simple scheme does not work because of the following reason. For an element x that is present in only one view, it is ambiguous whether x should be deleted or retained in the merged set. The algorithm presented in [FISH82] solves this problem by maintaining some additional information with each element in a view. The additional information records (T_x, C_x) , where C is the identity of the node where the element was inserted and T is the time of the insertion. T is measured with a clock at the creator node; it is assumed that this clock generates monotonically increasing numbers. Each node i also maintains a posting-time table PT_i such that the entry $PT_i[j]$ in this table indicates the latest time (with respect to j 's clock) node i got j 's view. This means that node i is aware of all insertions and deletions that have taken place at node j until time $PT_i[j]$.

The following predicate is defined as function of view V , posting-time table P , and element x .

$$\text{del}(V, P, x) \text{ iff } [x \in V \text{ and } T_x \leq P[C_x]]$$

With this definition it means that $\text{del}(V_i, PT_i, x)$ holds iff x has been deleted. The algorithm is described below:

```
INSERT(x)   $PT_i[i] := \text{clock}_i$ 
            $C_x := i; T_x := PT_i[i]$ 
            $V_i := V_i \cup \{x\}$ 
           {All these operations are performed as an atomic action}
```

```
DELETE(x)   $V_i := V_i - \{x\}$ 
```

```
LIST       return  $V_i$ 
```

```
SEND       send message  $\langle V_i, PT_i \rangle$ 
```

```
RECEIVE    For a message  $\langle V, P \rangle$ 
            $V_i := \{x \mid (V_i \cup V) \mid$ 
               not  $\text{del}(V_i, PT_i, x)$  and not  $\text{del}(V, P, x)\}$ ;
            $PT_i[k] := \max(PT_i[k], P[k])$  for all  $k$ ;
```

The initial conditions at all nodes are :

```
clocki=0, PTi[k]=0 for all  $k$  and  $i$ 
Vi=null for all  $i$ 
```

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

In [FISH82] a formal proof of correctness of this algorithm is presented.

The major advantages of this algorithm are:

- (1) It functions even in the presence of lost or duplicated messages.
- (2) It does not require synchronized clocks or transaction logs.
- (3) No commit protocols are required.
- (4) It supports continued operations even under network partitioning.

This algorithm has the following disadvantages:

- (1) This scheme requires transmission of the whole view. Therefore, it is unsuitable for applications in which the view at each site tends to be large.
- (2) It is not possible to update an item in the database. Updating can only be done indirectly by deleting the current item and inserting a new item. The system treats these two items totally unrelated.

The scheme proposed by Allchin addresses these problems. This scheme is described below:

4.3.2.5.3.2 Allchin's Suite of Algorithms

The two main differences between this scheme and the one described above are that (i) this scheme does not require sending the views from one node to another, and (ii) it permits updating an entry once it has been inserted in the database. Introduction of the update operation raises a new problem: if the same item gets concurrently updated by two nodes, then how may mutual consistency be ensured. The solution to this problem in Allchin's scheme is similar to the data-patch concepts of forward error recovery, i.e., depending on some predefined criteria one of the updates is aborted and the effect of the other will prevail in the system.

In this report we present a simplified version of Allchin's scheme. As mentioned above, this scheme does not require sending the view from one node to the other nodes. Instead, a list of updates, called the synchronization set, is sent to other nodes. The scheme uses four different design options. The first set of options is related to the propagation characteristics of the changes to the database by a node. The propagation option means that when a node makes certain change to the database, then every node in the system has the responsibility of propagating that change to the other nodes in the system. On the other hand, the no-propagation option means that only the node making the change has the responsibility of ensuring that every node gets to know the change.

The other design options are related to the update operations on the items. If the database is independent then any node can update an item, provided the item is present in its view. The database is said to be dependent if only one node in the system can update an item. Given these design options, we will describe the scheme for the general case of propagation-independent options.

Every node maintains the following objects locally:

- V_i Local view of the database
- SS_i Synchronization Set which consists of a list of changes that may not have been seen by the other nodes.
- T_i An array of time-stamps such that $T_i(j)$ indicates the time (w.r.t node i 's clock) up to which node j has seen node i 's changes to the database.

An item in the view V_i has the following structure:

```

item      = record
            itemname: uniqueness;
            val: value;
        end;
```

The unique name for an item is of uniqueness type as defined below:

```

uniquename = RECORD
            id:string;
            creator:node_id;
            creationtime:timestamp;
        END;
```

Each synchronization set contains items of type changeitem:

```

changeitem = RECORD
            citem:item;
            op:(INSERT, DELETE, UPDATE, AbsentSeen, AbsentNotSeen);
            cn:node_id;
            KnownBy: Set of node_id;
        END;
```

The field KnownBy records the names of the nodes that have seen this change and have recorded it in their local view. At any point in time when this set for an object of type changeitem becomes equal to the set of all nodes in the system, the item is deleted from the synchronization set.

The global variables at each node are defined as follows:

```

V : SET OF item;
SS : SET OF changeitem;
T : ARRAY [node_id] of timestamp;
```

Periodically a node sends messages to other nodes containing $\langle T, SS \rangle$ pair. A set of rules are defined which dictate the actions to be taken by a node when it receives such a message.

First we describe the actions performed by a node on its local database (i.e., V , SS , and T) when the INSERT, DELETE and UPDATE operations are invoked locally by some user. The actions performed are as follows:

INSERT(xname:itemname; val:value): returns (ok, AlreadyExists)

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

```
IF xname is in the set V THEN INSERT:=AlreadyExists
ELSE
  BEGIN
    time:=clock;
    x:= <xname, val>;
    T(my_node_id):=time;
    Add x to the set V;
    Add <x,INSERT,my_node_id,time,{my_node_id}> to the set SS;
    INSERT:=ok;
  END

UPDATE(xname:itemname; val:value): returns (ok, Nonexistent);

IF (xname is not in the set V) THEN UPDATE:=Nonexistent
ELSE
  BEGIN
    time:=clock;
    x:= <xname, val>;
    T(my_node_id):=time;
    Replace x in the set V;
    Add <x, UPDATE, my_node_id, time, {my_node_id}> to the set SS;
    UPDATE:=ok;
  END

DELETE (xname:itemname; val:value): returns (ok, Nonexistent);

IF (xname is not in the set V) THEN DELETE:=Nonexistent
ELSE
  BEGIN
    time:=clock;
    x:= <xname, val>;
    T(my_node_id):=time;
    Delete x from the set V;
    Add <x, DELETE, my_node_id, time, {my_node_id}> to the set SS;
    UPDATE:=ok;
  END
```

x old y new	Insert	Update	Delete	AbsentSeen	AbsentNotSeen
Insert	SS: $M(x,y)$ V: nc	SS: nc V: nc	SS: nc V: nc	SS: nc V: nc	SS: $A(y)$ V: $A(y)$
Update	SS: $R(x,y)$ V: $R(y)$	SS: * ₁ V: * ₂	SS: nc V: nc	SS: nc V: nc	SS: $A(y)$ V: $A(y)$ if absent $R(y)$ if present
Delete	SS: $R(x,y)$ V: $D(x)$	SS: $R(x,y)$ V: $D(x)$	SS: $M(x,y)$ V: nc	SS: nc V: nc	SS: $A(y)$ V: $D(y)$ if present
Absent Seen	SS: $D(x)$ V: nc	SS: $D(x)$ V: nc	SS: $D(x)$ V: nc	SS: - V: -	SS: - V: -
Absent Not Seen	SS: nc V: nc	SS: nc V: nc	SS: nc V: nc	SS: - V: -	SS: - V: -
nc: no change					
* ₁ if $(x.cn = y.cn)$ and $(x.ct = y.ct)$ then $M(x,y)$ else if $(x.ct < y.ct)$ or $(x.ct = y.ct)$ and $x.cn < y.cn$ then $R(x,y)$ * ₂ $R(y)$ if x replaced in * ₁					
SS: $A(j)$ = Add j to oldSS, union local node number into knownby of j . if knownby = allnodes, then $D(j)$. $R(j,k)$ = Replace j on oldSS with k , union local node number into knownby of j . if knownby = allnodes, then $D(j)$. $D(j)$ = Delete j on oldSS. $M(j,k)$ = Merge knownby sets into j . if knownby = allnodes, then $D(j)$. V: $A(j)$ = Add j .item to V. $R(j)$ = Replace the item with the same name in V with j .item. $D(j)$ = Delete item with the name j .item from V.					
Resolution Table					

Figure 4-10

The other two procedures are related to receiving (transmitting) views from (to) other nodes. When a message containing the synchronization set and the array T is received by a node, the remote SS and the local SS are compared. The conditions that may arise described below:

1. A changeitem for an object is present in the remote SS, but not in the local SS.
2. A changeitem for an object is present in local SS, but not in the remote SS.
3. A changeitem for an object is present both in remote SS and local SS.

Case 1 arises because of two conditions. The first is that the change has already been seen by the local node and removed from the SS or the change has never been seen by the local node. These two conditions are characterized as AbsentSeen and AbsentNotSeen respectively.

AbsentSeen T[node where the change occurred] > timestamp of change

AbsentNotSeen T[node where the change occurred] < timestamp of change

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

The same conditions can be defined for the remote node in Case 2.

For Case 3, if the both items refer to the same change, then their KnownBy sets are merged. If at any time the KnownBy sets for an entry in an SS set becomes equal to the set of all nodes, then that entry is deleted from the SS. This is because that change has been seen by all nodes in the system.

If the two entries for the same item refer to different changes, then the change with the larger timestamp is retained and the other entry is deleted. For example there is an entry corresponding to an update change in each of the SS, then the more recent update is retained. In case both the updates have the same timestamp, then the selection is made on the basis of the IDs of the nodes where the changes originated. The resolution table (from [ALLC83]) for the various possible cases is shown in Figure 4-10. In this table, X refers to an entry in the local SS and Y refers to any entry in the remote SS. In this table, X=Delete and Y=AbsentSeen means that in the local SS there is an entry for some object with delete operation, and there is no entry for that item in the remote SS. However, on the basis of the timestamp array T one can say that the remote node has already seen that change sometime in the past. The absence of the entry also indicates that all nodes in the system must have seen that entry, because then only that entry would have got deleted from the synchronization sets. For this case, delete the entry for X in the local SS, and no change is to be made to V.

4.3.2.5.3.3 Some Performance Measures for Weak Consistency Management Schemes

There are several measures one might consider to evaluate the effectiveness of a consistency management scheme that allows weak consistency among replicated copies. Some of these are defined below:

Probability of inconsistency: This is the probability that an entry present in one view has been either deleted or modified at some remote node.

Convergence Period: This is the time interval starting with the instant a change is introduced in the system to the instant when its KnownBy set becomes equal to the set of all nodes in the system.

These two measures are functions of the following design parameters and operational characteristics:

- (1) Number of nodes in the system
- (2) Frequency of broadcasting the synchronization sets
- (3) Failure characteristics (MTTF, MTTR) of the communication links
- (4) Message transmission delays
- (5) Frequency of updating an object
- (6) Probability of deleting an object

4.3.2.5.4 Detecting and Entering Degraded Mode

This part of replication management is largely ignored in most designs. It is an important issue for those designs that have a correctness criterion of strong consistency or can set locks at multiple sites during a transaction. If a transaction involves a failed or inaccessible site, a decision must be made whether to block the transaction and tie up its resources until the failure is recovered, or to immediately terminate the transaction and thereby free any allocated resources. The former approach is called blocking and the latter non-blocking. This problem must be addressed for transactions involving non-replicated objects, but is exacerbated for replicated objects because of the difficulties that may arise when partitions are being merged.

If a site involved in a transaction fails, any operational sites that are also involved in the transaction will eventually detect the failure (e.g., by a timeout or some other mechanism). The operational sites at that point may execute a termination protocol and complete the transaction if 1) they can arrive at a consistent state and 2) a failed site can transition to a consistent state when it recovers.

Recovery for a replicated object must address 1) transitioning a failed site into an operational state and 2) merging replicas of objects in different partitions into a replicated object with a single state.

If a site was participating in a transaction when it failed, the transaction must be terminated so that the objects are left in a consistent state. The mechanism by which this is done is called a recovery protocol and has a purpose similar to a termination protocol that is executed by operational sites following a failure.

A failure that results in network partitions has two issues that must be addressed when merging the copies of an object during recovery. First is the detection of inconsistencies between the copies of an object. Second is the reconciliation of any inconsistencies between copies of an object.

4.3.2.6 Network Partitioning and Continued Operations

Under the conditions of network partitioning, allowing sites to update a replicated database some copies of which are in an inaccessible partition, may result in inconsistency among the copies. This inconsistency among the copies is required to be resolved when the partition is repaired. In [BLAU83] two schemes, called Data Patch and Log Transformation, have been proposed for integrating the inconsistent copies of the database. The technique called Data Patch [GARC83a] relies on the data values and the semantic knowledge of the database. The technique of Log Transformation uses the logs of the transactions executed during network partitioning for the integration purpose. In this section, we describe these two techniques.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

4.3.2.6.1 Data Patch

This scheme is an example of the forward error recovery technique. In this approach the data values before the partition and the data values at different sites after the partition are examined during the partition repair time. Depending on various different criteria and consistency requirements, the final merged value of the data is determined. The criteria and techniques for determining the repaired values are determined at the time of the database design; the database administrator uses tools based on these policies to integrate different copies of the database during the partition repair.

Some of the examples of the data-patch rules are described below.

- (a) Computed Attributes: Certain attributes can be recomputed, based on a pre-defined formula, during the partition repair.
- (b) Data Insertion Rules: When a new item is inserted into the database during the partition at one site but not at another, the data insertion rules specify the possible actions that can be taken during the repair. Some of the possible rules are:
 - (i) KEEP - add the data item to other copies of the database,
 - (ii) REMOVE - delete it from all copies where it was inserted,
 - (iii) NOTIFY - notify the database administrator that an unanticipated situation has occurred. The database administrator will make the decision to either retain or delete this item.
- (c) Data Integration Rules: When two replicated copies of the same data item have been modified independently during a partition, the data integration rules specify how to determine the final value of the data item after the integration. These rules are specified for each attribute of the data type. Some of the possible rules of integration are described below.
 - (i) Select the latest value of the data,
 - (ii) Select the value from a given site,
 - (iii) Use some constant or default (e.g., NULL) values when it is not possible to determine any consistent value,
 - (iv) Notify the database administrator if no action can be taken.

In addition to these rules, some other rules are also specified which apply certain integrity constraint checks on the database and invoke compensating and corrective actions. Some of these actions might involve notification to certain users.

The usefulness of the data-patch technique strongly depends on thorough understanding of the application environment. This technique fails to deal with network partitioning during integration. Data-patch allows ad hoc updates, but such updates are required to be restricted to keep the integration rules appropriate. As new transaction types are added, the integration rules must be updated appropriately. The compensating actions that require only the database values at the merge time are efficient to

execute as compared to those actions which require examination of the execution logs to determine which transactions generated these data values.

4.3.2.6.2 Log Transformation

As the name suggests, this technique relies on the logs of transaction executions at different sites for merging the partitioned copies of a database. This technique does not make use of the data values at the merge time. The transaction logs are merged according to some pre-defined rules which depend on the semantic properties of the transactions. This technique assumes that all transactions are pre-defined.

The database administrator specifies the semantic properties and relationships between the pre-defined transaction types. For example, transaction T1 and T2 commute, or transaction T2 overwrites all data written by T1.

During the partition merge time, the execution logs from each partition are exchanged and new merge logs are built. In constructing merge logs some conflicting transactions are undone and re-run. The merge logs are generated independently at each site; therefore, they may be different at different sites. However, the log transformation technique assumes that there is a system-wide policy to re-order conflicting transactions. One possible criterion for determining the order of execution for transactions in the merged logs is their execution time.

Generally, a merged log can be divided into two parts: the roll-back log and the re-run log. Suppose that for each transaction T_i , an inverse transaction T_i' is defined such that the log $[T_i, T_i']$ is equivalent to an empty log, a log containing no transactions. It is possible that some transactions do not have any inverse transaction. We will discuss this case later.

Consider the following example that illustrates these concepts. The notation $[T_1...T_n]/x$ means that the log $[T_1...T_n]$ is executed at site x . Suppose that during a partition the logs $[T_1, T_3]/x$ and $[T_2, T_4]/y$ have been executed. During partition repair, merge logs are constructed at both x and y such that their executions at these two sites will have the final effect as if at each site the transactions executed in the order $[T_1, T_2, T_3, T_4]$. The most trivial way to achieve this is to construct the merge logs

$[T_4', T_2', T_1, T_2, T_3, T_4]/y$ and $[T_3', T_2, T_3, T_4]/x$.

The log-transformation technique provides rules for generating merge logs efficiently, i.e., it generates merge logs which are short.

A log transformation is a mapping from one log to another equivalent log. The transformation rules depend on the properties of the transactions; these properties are specified during the design of the database. The following properties are found useful in performing log transformations:

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

- (1) Transaction Commutativity: This means that the two logs $[T_i, T_j]$ and $[T_j, T_i]$ are equivalent.
- (2) Transaction Overwrite: If transaction T_i overwrites T_j , then $[T_i, T_j]$ and $[T_j]$ are equivalent.
- (3) Quasi-Commutativity: Two transactions are commutative if some extra transactions are executed.

In [BLAU83] a graph-theoretic formulation of the log transformation technique is presented. During the construction of the merge logs at a site, the transactions are grouped into two classes: local and non-local. An arc exists from transaction T_i to transaction T_j (shown as $T_i \dashrightarrow T_j$) if, and only if, T_i preceded T_j , and T_i and T_j do not commute.

The local node graph transformation rule presented in [BLAU83] exploits the commutativity of transactions to perform log transformations. The rule is as follows: For each local transaction T_j such that there is no non-local transaction T_i with a path from T_i to T_j , delete the transaction T_j from the local merge log. In addition to this rule, the log transformation process moves some of the local transaction, depending on the commutativity constraints, to the head of the re-run log. This allows them to be merged with their inverses, hence generating a smaller log.

Consider the following example with the two logs in a partitioned system $[T_1, T_3]/x$ and $[T_2, T_4]/y$. Suppose $T_2 \dashrightarrow T_3$. At site Y , the rollback log is $[T_4, T_2]/y$ and the re-run log is $[T_1, T_2, T_3, T_4]/y$. At site y , transaction T_2 is local and it is not preceded by any non-local transaction such that a path exists from that transaction to T_2 . Therefore, T_2 and T_2 are deleted. Because T_4 commutes with T_1 and T_3 , it is moved to the head of the re-run log. The new merge log is $[T_4, T_4, T_1, T_3]$, which is equivalent to $[T_1, T_3]$.

The case when a transaction does not have an inverse is discussed in [BLAU83]. An example of such a transaction is a transaction which assigns a new value to a data item. To restore the old value, one needs a restoring transaction. A restoring transaction is independent of the semantics of the original transaction and only depends on the state of the database that existed before executing the original transaction. On the other hand, an inverse transaction depends purely on the semantics of the original transaction and is independent of the state of the database.

In some cases compensating transactions are needed to be inserted in the merge logs. For example, suppose a person has \$200 in his account, and he withdraws \$100 at site A during network partition. Sometime later during the partition condition he withdraws \$150 from site B (site A and B are in different partitions). After partition repair, the system will discover that this person's account is overdrawn; a compensating transaction will charge the person a \$10 fine and send him a notification for the charge.

The applicability and usefulness of the log-transformation technique is dependent on the application. As in case of data-patch, the transactions in

the system are of pre-defined types. As new transaction types are added to the system, necessary information is required to support correct operations of this technique.

4.3.2.7 Self-Identifying Objects

In this technique suitable descriptors are attached to the objects to facilitate reconstruction of directories by salvation programs. Salvation programs are used only in cases of extreme failures where not enough information is left in a consistent state to support automatic rollback and restart. Such programs need an intervention by the operator.

4.3.2.8 Forward Error Recovery

Generally, every reliable system design incorporates both forward and backward error recovery techniques. The most common technique for forward error recovery is exception handling [GOOD75]. Exception conditions are the anticipated error conditions in the system. An exception handler is a program block that is invoked when a specified exception condition arises during run-time. The purpose of an exception handler is to bring the system to a consistent state. Generally the exception handlers are application specific. Forward error recovery requires a complete understanding of the application for which the system is being designed. We have already presented two techniques, log transformation and data-patch, that are based on the concepts of forward error recovery. In this chapter we do not consider the forward error recovery techniques in any more detail.

4.4 INTEGRATION OF RECOVERY MECHANISMS IN THE DESIGN MODEL

In this section we describe the reliability techniques that are suitable at each level of abstraction in the design model shown in Figure 4-2. The discussion is divided into four major parts: object management, transaction management, remote procedure calls, and the management of distributed objects. We focus on the problems related to recovery rather than protection and security issues.

4.4.1 Stable Storage

Lampson presented the techniques for constructing stable storage from unreliable disc storage facility [LAMP81a]. The primary goal of his scheme is to make the operation of writing disc pages atomic.

Lampson's scheme is based on the technique of careful replacement. The atomic operation for writing pages on the non-volatile storage is called StablePut. The StablePut operation first writes the page on an unused disc page rather than writing it over the original; thus, any failure during

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

execution of the StablePut operation leaves the original page intact. Periodically the two pages are compared, and the old page is replaced by the new one. The pages are also checked for any corruption of data by applying suitable parity or checksum tests. The corrupted page is replaced by the data of the other page if that page is still unspoiled. This replication of pages also increases the availability and mean-time-to-failure for the pages, provided the pages are stored on different storage units such that their failure is independent.

4.4.2 Object Management

An object manager supports primitive operations on the objects of its type, as well as other functions such as the construction of recoverable objects, concurrency control, and access control. Objects for which recovery and synchronization are provided by the object manager are called atomic objects [LISK82b].

Generation of UIDs is an important part of reliable object management. A crash resistant scheme for generating UIDs in the system is described in [SCHA83]. In this scheme, every node in a subset of nodes, which forms a survivable set, must possess a stable storage facility. A global sequence counter is replicated over this subset and sometimes global synchronization among these nodes is required. On restart, nodes not having a stable storage obtain the sequence number from one of the members of this survivable set of nodes.

Recoverable Objects - Conceptually, constructing recoverable objects in our design model is based on the multiple version techniques [REED78], [SVOB81]. Every change to an object creates a new version of that object; such versions are finalized upon committing the enclosing transaction. On transaction aborts, the tentative versions created by that transaction are discarded. The versions are forced onto the stable storage to make them recoverable under node crashes.

Reliability techniques most suitable for constructing recoverable objects include multiple versions, differential files, intention lists, audit trails/logs, and self-identifying objects. Generally, a combination of several of these techniques is used in constructing recoverable objects at a node.

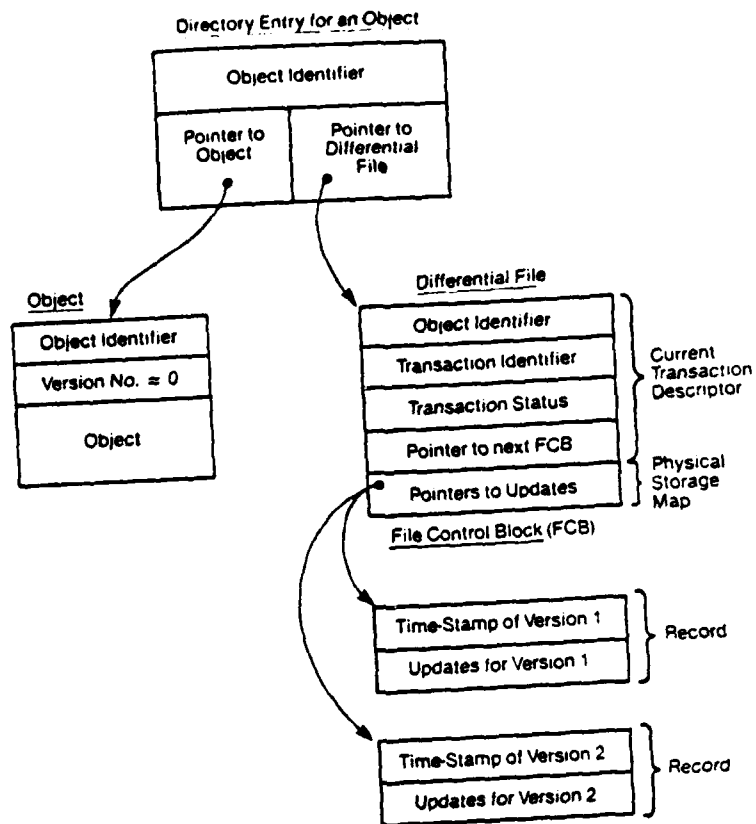


FIGURE 4-11 Differential File Organization for Maintaining Multiple Versions of an Object

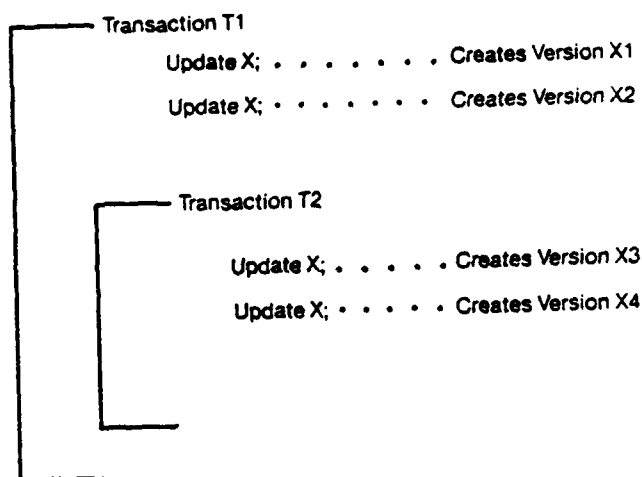


Figure 4-12

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

Maintaining multiple versions as a forward log is less expensive than as copies of the original object. A forward log in which the sequence of changes is idempotent can be used as an intention list to ensure the permanence of results on the commitment of a transaction. Backward logs are used for restoring objects by undoing the actions recorded in the log. Whenever a new uncommitted state of an object is to be forced in-place on the stable storage from the volatile memory, it is essential that (in order to keep the object recoverable) the backward log be forced on the stable storage before forcing the uncommitted object in-place on the stable storage.

(a)	Object X Committed None	Version 1 Uncommitted T1	Version 2 Uncommitted T1		
(b)	Object X Committed None	Version 1 Uncommitted T1	Version 2 Uncommitted T1	Version 3 Uncommitted T2	Version 4 Uncommitted T2
(c)	Object X Committed None	Version 1 Uncommitted T1	Version 2 Uncommitted T1	Version 3 Uncommitted T1	Version 4 Uncommitted T1
(d)	Object X Committed None	Version 1 Committed None	Version 2 Committed None	Version 3 Committed None	Version 4 Committed None

FIGURE 4-13

Self-identifying objects and consistency checks play an important role during restart after a crash in reconstructing objects, object headers and directories during the restart after a crash. For example, with multiple versions, differential files and logs, additional information such as the object UID, state of the versions (committed, uncommitted, commit pending, etc.), pointers to other versions, logs and differential files is incorporated for crash recovery. After reconstructing the data structures on crash recovery, the consistency checks are important in checking the validity and correctness of the reconstructed data structures.

At this point we describe a scheme and its associated data structures for maintaining multiple versions in the system to construct recoverable objects. Logically, every version in this scheme contains a descriptor which contains the UID of the object, version number, UID of the transaction currently holding this version, a time-stamp indicating its creation time, and a status of the version. The status field can be in any one of the following states: uncommitted, commit-pending, committed, and aborted.

The time-stamp field of the versions is useful for discarding the versions created by a transaction since its last checkpoint. The commit-pending state is used during execution of the two-phase commit protocol [LAMP76], [GRAY79] with the current user transaction. The commit protocol is initiated by the user transaction by sending a prepare-to-commit message to the object managers of all the objects it has updated. On receiving such a message, the object managers change the status field of the current versions to commit-pending, and return a positive acknowledgement. A version in the commit-pending state cannot be unilaterally discarded by its object manager.

In the scheme proposed here, we use forward logs to maintain multiple versions. The versions of an object are maintained in a log as records of changes to the existing committed copy of that object. Applying these changes to the object has the property of idempotency; therefore, the log also serves as an intention list. For every transaction, one such log file is created as shown in Figure 4-11. The file control block (FCB) plays an important role in this scheme. The FCB for a forward log file has two parts: Current Transaction Descriptor and Physical Storage Map. Current Transaction Descriptor contains the identifier and the status of the transaction that has recorded new uncommitted versions of the object in the log file. Physical Storage Map points to the records on the stable storage containing the updates for the new versions. By rewriting this FCB using the atomic StablePut operation the entire FCB can be changed in one atomic action. This use of FCB for atomic updates is similar to the scheme described in [LORI77] [PAXT79]. To record an action on the file, the changes are written on new pages, the FCB is modified and re-written using the StablePut operation. At this point, the change has been successfully recorded.

4.4.3 Process and Transaction Management

In this section, we discuss the principles of reliable transaction processing and the reliability techniques that we propose to evaluate for reliable process and transaction management in distributed systems. As noted in Section 3, processes are considered as objects. Transactions are atomic processes; transactions are objects of process type with some additional properties; therefore, transaction type is a sub-type of process type.

A process object once created can be in one of five states: Inactive, Running, Suspended, Completed, or Aborted. The operations for process objects include: Create, Destroy, Start, Restart, Status, Suspend, and Resume.

From an object-oriented viewpoint, new versions of a process object are created during execution of the process, i.e., a new version of the process object is created whenever its program counter changes. This view is

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

consistent with the one for multiple versions of data objects. There is, however, a difference between these two types of objects in handling rollback recovery: with data objects, it is usually possible to save all versions of an object before these versions are committed so that rollback to a previous version is relatively simple; with process objects, it is too expensive and impractical to save the process states of all execution steps. Checkpointing can be viewed as the selective saving of versions of process objects and is used to establish recovery points for process objects.

The following operations for process objects are used to support checkpointing and rollback:

- o Establish_Recovery_Point - saves the current process state of the process object in stable storage.
- o Discard_Recovery_Point - discards the checkpoints of a process object.
- o Rollback - continues the execution of a process from a checkpoint.

Note that with the above scheme for checkpointing, only the state of the process object is saved; the states of objects modified by that process are not saved in the checkpoint. This approach may create problems for error recovery since not all state changes of the process are recorded. It is necessary, therefore, to follow some discipline in using checkpoints and atomic transactions.

First, we require that a non-transaction process (e.g., a user process) must invoke a transaction in order to modify an object or a set of objects. The changes to an object are recorded as new versions of the object. New versions of the object are committed to become permanent at the end of a successful completion of the commit protocol among the invoked transaction, the invoking process, and the object managers of the modified objects. Uncommitted versions are discarded by explicit abort commands from the transaction process or by timeout on inactivity.

If a transaction is nonidempotent, i.e., multiple executions of the transaction produce different results, a problem may arise in error recovery since rollback of the process may cause a committed transaction to be re-executed. One solution to this problem is to always force the invoking process to perform a checkpoint before the transaction completes committing the modified objects. Checkpointing is part of the commit protocol; if the protocol determines to abort, the checkpoint is discarded. With this mandatory checkpoint, rollback recovery of a process can avoid undesirable repetition of transaction execution; however, this may cause too frequent checkpointing of the invoking process. The second solution, therefore, is to make checkpoint of the invoking process an option that is to be specified at the time of invoking a transaction. This checkpoint apparently is not required for idempotent transactions to guarantee correct execution; a process, however, invoking an idempotent transaction may elect to force a checkpoint during the transaction commit protocol for efficiency reasons. For example, if the transaction requires extensive computation compared to checkpointing the invoking process, and if the possibility of a failure is significant, it may be desirable to have a checkpoint as described above. That decision is left to the process that invokes the transaction.

The following example illustrates the flexibility provided by the second solution. Consider the following scenario in which a process receives some item from a buffer, then processes the item. GetItem is the transaction that is invoked by the process to receive an item from the buffer. Commitment of this transaction leaves the buffer in a new state in which the removed item is no longer present and the previous state can never be restored. If the process invoking this transaction checkpoints itself on the commitment of the transaction, the received item is a part of the checkpointed state of the process. Any subsequent rollback will restart processing of this saved item, and there will not be any need to re-invoke the GetItem transaction. On the other hand, if the process does not checkpoint on committing the GetItem transaction, on a subsequent rollback the old item would be lost; the process would invoke the GetItem transaction once again; and processing would be performed on a new item. In certain applications such as process control systems, the second scenario may be a valid mode of error recovery.

A question on checkpointing still exists: Because the checkpoint of a process does not include the current states of the objects that are modified by the process, how does it guarantee correct rollback recovery? This question can be answered by considering the ways in which objects are affected by a transaction: first, a transaction may directly modify an object by invoking an operation on the object; second, it may invoke another transaction that modifies the object.

We first consider the object modified by the transactions by invoking an operation on the object. One requirement for correctly implementing rollback is that the object manager for transactions must maintain for each transaction a list of UIDs of the objects that are affected by the transaction so that in the event of failure, the transaction will be rolled back to its latest checkpoint. This requires that all changes to objects made by the transaction after the checkpoint be discarded. The list of UIDs of the objects that are affected by the transaction provides a means for notifying these objects to discard the unwanted versions. This list is also used at the end of the transaction to conduct the commit protocol.

The discussion in the preceding paragraph implies that for implementing rollback, timestamps must be recorded with each checkpoint and each version of objects. This is necessary because checkpoints do not record all versions of a process object; and thus, there is not a one-to-one mapping between process checkpoints and versions of objects affected by the process. These techniques allow us to rollback a process correctly with modifications to objects in the first way.

The correctness of rolling back a process with the second way of modification to objects is guaranteed by the principles followed in committing a nested transaction. The updates made by a nested transaction are made permanent only if its parent transaction is committed. Any rollback within a transaction may cause abortion of some committed nested transactions. In case a transaction is aborted, the changes made by the transaction are discarded (transactions are atomic). The objects are brought back to the state before the transaction was started. Failure of the invoking process poses no problems to these objects.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

For idempotent transactions, the case is even simpler. Because transactions are atomic, changes to objects are either not done or made permanent from the invoking process point of view; and because the transactions are also idempotent, the rollback recovery is always correct, no matter where the invoking process is rolled back to.

Nested Transactions - As described in the previous section, at the end of a successful transaction the objects that were changed by the transaction are committed to become permanent; however, for a nested transaction, (a transaction invoked by another transaction) that completes successfully, commitment of the changes to objects will be dependent on the success of the parent transaction. If a nested transaction is aborted, the changes to objects made by the transaction will be discarded regardless of the success of its parent transaction; however, the failure of a nested transaction may not always cause its parent transaction to abort. In this section, we will describe a technique for implementing nested transactions. This technique requires only minor modifications to the technique for implementing single level transactions.

The technique that we use here requires recoverable objects as described in Section 6.2, i.e., each update to an object creates a new version of the object. Each version carries the information to indicate whether it is a committed, uncommitted, or commit-pending version. In order to support nested transactions, additional information is needed for each version to indicate on which transaction the version is dependent. This information is attached to a version when it is created by a transaction.

At the end of a transaction, the transaction either commits or aborts the changes to the object. If it aborts, only the uncommitted versions that are dependent on this transaction are discarded. If it commits, all versions of the object that are dependent on this transaction are changed to become dependent on the parent transaction of the current transaction. In this case, if the current transaction is at the top level, i.e., if it is invoked by a non-transaction process, these versions are committed to be permanent.

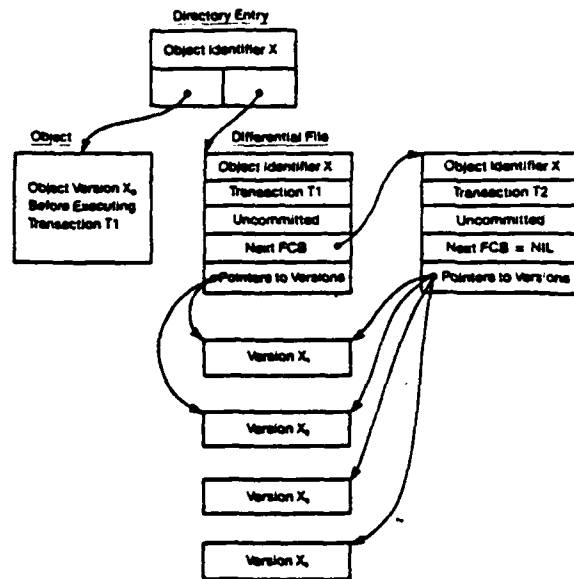


FIGURE 4-14 Multiple Versions of an Object for Nested Transactions

Figure 4-12 shows an example of nested transactions. Transaction T1 updates the object X to create Versions 1 and 2. Both versions are uncommitted and contain the information that they are dependent on T1. A logical view of this result is shown in Figure 4-13(a). T1 then invokes transaction T2, which creates Versions 3 and 4 of X (Figure 4-13(b)). When T2 is completed successfully, all versions that are dependent on T2 are changed to be dependent on T1, the parent transaction of T2 (Figure 4-13(c)). Since T1 is a top-level transaction, i.e., it is invoked by a non-transaction process; when it is completed, all versions that are dependent on T1 are committed to become permanent (Figure 4-13(d)). Version 4 of X is now the current committed copy of object X; other versions can be discarded at this point.

In order to implement the above scheme for nested transactions, we can use differential files to maintain multiple versions as described in Section 6.2 and Figure 4-11. In the example in Figure 4-12 for a nested transaction, a new FCB and descriptor block is created as shown in Figure 4-14. When a nested transaction completes, the transaction UID field in the descriptor of the version that is being committed is replaced by the UID of its parent transaction, and the status field is changed to the uncommitted state. The status field of a version changes to committed only when the transaction committing it is the outermost level transaction. In Figure 4-14, transaction T2 is nested within transaction T1, and T1 created versions X1 and X2 for object X. Transaction T2 appends new versions X3 and X4 to the differential file, and these changes are visible only in the FCB that is being used for transaction T2. On the commitment of T2, the old FCB is replaced by the new one, and the user transaction field contains T1. When T1 commits, the status field in the descriptor is changed to committed, and the updates from the

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

differential file are applied to the object. If any crash occurs during this updating, the procedure can be restarted from the beginning.

Some Principles of Process and Transaction Management:

First we present the commitment protocol for nested transactions, then we present some principles for establishing recovery points and rolling back within transactions and processes.

In the proposed execution model a process invokes a transaction to perform certain operations on the shared global objects. Such a transaction is called a top-level transaction. A transaction may either directly modify an object or invoke one or more other transactions (called nested transactions) to perform operations on global objects. When a nested transaction completes its execution, it executes a commit protocol with the objects modified by it directly. At this point all of its own nested transactions, if any, are in the commit_pending or aborted state. During the execution of the commit protocol, a PREPARE message is sent to the object managers of the updated object. If all object managers respond with a ready message, then the decision is to commit the transaction. However, this being a nested transaction the transaction enters the commit-pending state, all the versions, created by this transaction, of the updated object are marked dependent on the commitment of the parent transaction. The parent transaction is sent a completion signal. A nested transaction commits when its top-level transaction commits.

Figure 4-15 shows this commit protocol. It can be seen that the nested transaction executes two-phase commit protocol with the object managers of the updated objects and one phase commitment with its parent transaction.

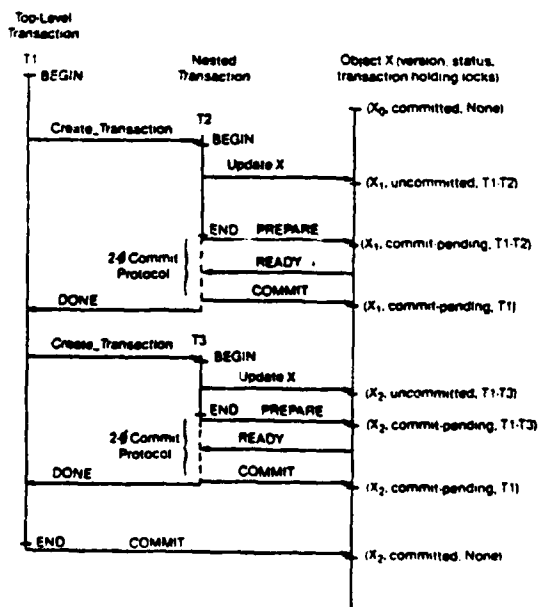


FIGURE 4-15

A transaction is either parallel or sequential with respect to its parent transaction. When a process (or transaction) declares a part of its sequential execution stream to be treated as a transaction, then such a transaction is sequential with respect to its parent. If a process or transaction creates a new execution stream which executes concurrently with it, then the new transaction is parallel with respect to its parent. The primitive for creating a sequential transaction is to enclose the part of the process code by `Begin Transaction` and `End Transaction` commands. For creating a parallel transaction, the `Create Transaction` command is executed. The protocol shown in Figure 4-15 is valid for both parallel and sequential nested transactions. For a top-level sequential transaction, during the execution of the `End Transaction` command, a decision to commit implies final commitment of all updates. This commitment is not dependent on any other information or decision. After the execution of this command, the execution of the parent process continues, and the transaction no longer exists.

A top-level parallel transaction enters the `commit_pending` state if the decision, during the execution of the `commit protocol`, is to commit the transaction. Only when the parent process executes the `commit` command is the top-level parallel transaction committed.

Recovery Points and Rollback within Processes

A process can establish recovery points by calling the `ERP` command. This establishes a new recovery point; all recovery points for a process are serially numbered. A process can rollback by invoking the `Rollback` command. The following principles are followed during the rollback of a process to a recovery point:

- (1) No committed transactions that were created after the establishment of the recovery point are undone.
- (2) All incomplete transactions, i.e., those which have not committed or aborted yet, that were created after the establishment of the recovery point are aborted.
- (3) All recovery points established after the establishment of the recovery point to which the process is being rolled back are discarded.

Recovery Points and Rollback within Transactions

The commands for establishing recovery points and rolling back can be executed within transactions. The following principles are adopted in the designs of these two commands for transactions.

- (1) On the invocation of the `ERP` command by a transaction, a recovery point for the transaction is established only when all of its nested parallel transactions have either committed or aborted. This principle is important for defining a well-defined state on rollback and recovery.
- (2) On rollback of a transaction, all of its nested transactions (even those that are `commit-pending`) that were created after the establishment of this recovery point are aborted. The reason for supporting abortion of

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

commit_pending transactions in this case is that no information created by these transactions has moved out of the sphere-of-control of the transaction being rolled back.

- (3) When a nested transaction commits or aborts, all its recovery points are discarded.
- (4) When a transaction rolls back, all objects modified since the establishment of the recovery point to which the transaction is being rolled back are restored back to their versions at the time the recovery point was established.
- (5) No explicit commit commands are executed to commit a nested transaction.

Sometimes it is desirable to establish a recovery point for the parent process (or transaction) when one of its child transaction commits. For a sequential transaction, the End_Transaction command can be executed with the establishment of a recovery point for the parent process or transaction. Similarly, when a parent process or transaction commits one of its child transactions, it can specify the recovery point option with the commit command. It must be noted here that the recovery point for the parent process (a transaction) is established when all of its other concurrent transactions have terminated their executions.

4.4.2 Remote Procedure Calls

The problems related to reliable remote procedure calls have been discussed in [LAMP81b] and [SHRI82a]. One problem associated with the implementation of remote procedure calls is their execution semantics. In case of a retransmitted request message, should repeated executions be permitted? To address this problem, Nelson [NELS81] has classified the semantics of remote procedure calls as follows:

- o "At most once" - In this semantic, at most one execution of the procedure takes place. It is possible that no invocation occurs. In this case the call returns with some error condition.
- o "At least once" - This semantic means a successful return from the call guaranteeing at least one execution of the procedure.

In most of the applications "at most once" is preferred. One problem in the implementation of "at most once" is detecting duplicate requests at the server end. If the client process crashes after sending the call request and retransmits the request after the restart, the server should be able to detect the duplicate request. For this purpose the UID facility is used to assign a unique name to the request.

In the following discussions we present three design schemes for implementing the remote procedure calls in a reliable fashion with "at most once" execution semantics.

If a requester crashes after the server has started the procedure execution, the procedure invocation is termed an "orphan". After a restart from the crash, the requester process will retransmit the remote procedure call request. At this point we have two options in the design. The first option is to retransmit the request with the same UID as was used for the initial call request. If the original request was lost, this retransmitted request will invoke the remote procedure. If the server received the original request and started procedure execution that was later rendered "orphan" due to the requester crash, the server would detect the duplicate request, continue the "orphan" execution which is no longer an orphan, and return the results of the "orphan" to the restarted requester. This scheme requires that every requester process must have access to a stable storage facility to store the request along with its UID so that on a restart the retransmitted request has the same UID. Because of this limitation we reject this scheme and propose the second scheme in which every remote procedure call is an atomic action which commits only after executing a commit protocol with the requester; thus, the results produced by the "orphans" are discarded because the commitment protocol fails. This scheme eliminates the need for a stable storage at every node at the expense of decreased performance due to commitment protocols.

Because commitment of each remote operation can decrease the performance significantly, it might be more desirable to commit all remote operations performed within a transaction only once, i.e., at the time of the transaction commitment. In the third scheme that we present here, a remote procedure call within a transaction creates only an uncommitted version of the object. If the requester crashes after dispatching the remote procedure call, the server would eventually time-out and discard the uncommitted version. When the requester rolls back and restarts its execution from some recovery point, it makes sure that all remote procedure calls that were made after this recovery point, during the previous execution, are aborted by sending explicit abort messages to their object managers. To facilitate this, every remote procedure call and recovery point is time-stamped. During rollback, the abort messages to the object managers indicates the time-stamp of the recovery point. This forces the object managers to discard all currently executing procedure calls for that requester with time-stamp larger than the time-stamp of the recovery point. It also forces the object manager to discard any versions created by that transaction after establishing the recovery point. Instead of time-stamps, one can use the UID facility. This scheme does not require a stable storage at every site, and it does not require the execution of a commit protocol after each remote procedure call.

The reliability of the datagram facility can be enhanced by introducing appropriate reliability techniques into the network layer and the link layer supporting this facility. At the network level, the network topology is an important design issue. A network topology with higher connectivity would generally exhibit better reliability characteristics. At the link level, appropriate retransmission protocols are used to deal with transient errors.

RELIABILITY AND CONSISTENCY MANAGEMENT TECHNIQUES

4.4.5 Distributed Objects and Transactions

The reliability techniques at this level deal with maintaining redundancy in the system. The redundancy in the system is maintained in the form of object replication, primary/backup copies, or survivable sets of objects. The techniques suitable for managing redundancy at this level are based on the principles of voting [THOM79] [GIFF79] together with some commit protocol [LAMP76] [GRAY79]. At the distributed application level, the reliability techniques deal with the synthesis of reliable objects by redundancy management and the construction of recoverable transactions. Atomic transactions play a key role at this level. At the application level, these fundamental mechanisms are integrated into some higher level techniques, such as a recovery block, for system structuring. Forward error recovery based on exception handling is an important part of reliability techniques at this level.

The problems associated with the management of redundancy in the system have been discussed earlier. The nested transaction facility provides a convenient and powerful abstraction to perform atomic operations on a set of distributed objects. Replication management techniques based on quorums or majority consensus are used within nested transaction structures.

The concept of recovery blocks can be used conveniently at this level to define a primary transaction along with a set of backup transactions and some acceptance test. This can be done easily in our model because transactions are atomic. Integrating the backward recovery techniques, such as a recovery block, with forward error using exception handling can create very effective recovery mechanisms in a design. Such an integration of these two concepts has been described in [MELL77]. For forward error recovery, exception conditions can be associated with primitive operations on objects. Exception handlers can be introduced within a transaction; this does not affect the atomicity of a transaction. If a transaction is a part of a recovery block, an acceptance test is applied on its completion, but before its commitment. The transaction is committed only if this test is passed or else the transaction is aborted and an alternate transaction is tried.

4.5 CONCLUSIONS

In this chapter, we have presented a comprehensive survey of the recovery techniques for a distributed system. We have also presented an object-oriented design model that supports structuring of distributed systems for high reliability and error recovery. In this model, we have identified the error recovery problems at the different levels of functional abstraction and have shown how various error recovery techniques are integrated into this design model. For example, techniques based on multiple version concept are used for constructing recoverable objects, checkpointing. Commitment techniques are used for constructing atomic transactions, and the techniques based on replication and primary-backup modes of operation are used for constructing reliable distributed objects.

CHAPTER 5

ZEUS DISTRIBUTED OPERATING SYSTEM

In the previous chapter we presented several recovery mechanisms and a design model for constructing reliable distributed systems. This design model provides a framework for integrating these recovery mechanisms into a system design in a structural fashion. Ideally, a distributed operating system should make the low level recovery mechanisms, such as logs and commit protocols, transparent to application programmers by providing some high-level functions for constructing reliable software. This chapter describes a distributed operating system called Zeus which has been designed with this goal. This design illustrates how various recovery mechanisms are integrated according to the design model presented in the previous chapter.

This chapter presents the principles followed in designing Zeus, an object-oriented distributed operating system for integrating recovery mechanisms into the designs of distributed command and control systems. The main contribution of this work is an operating system design that provides an integrated set of functions to application programmers for reliable management of objects in distributed systems. These functions transparently provide complex recovery mechanisms, commit protocols, concurrency control mechanisms [KOHL81] [BERN81], and remote object accessing to application programmers. For now the primary goal of the Zeus design is to define reliable object management functions for distributed command and control systems and to evaluate the performance and the correctness of the recovery mechanisms for these functions; therefore, no implementation of this design exists at this stage. The user visible functions support definition of object types, creation of objects, and updating of distributed objects using atomic transactions. The second volume of this guidebook presents the detailed designs of Zeus using Concurrent System Description Language and Ada. The results of performance and reliability evaluations of these designs using the techniques of Chapters 7 and 8 are presented in that volume. The formal methods to prove the correctness of the recovery mechanisms using Gypsy methodology, events and state transition based models, and Path Pascal functional simulation models are described in Chapters 9. To achieve these goals we have refined the Zeus design to a significantly detailed level. To date we have explored this design only from the viewpoint of these goals.

A distributed operating system for highly reliable applications must not only include suitable recovery mechanisms that are transparent to the application developers but it should also provide transparency of the distributed nature of the system. The second feature is important to make development of distributed software no more difficult than the development of

conventional software systems. The Zeus design has made a significant contribution in this direction. Some other systems, such as LOCUS [POPE83], have integrated these two concepts in their designs; however, in most of these systems, object management is limited only to the file storage level. To date, Argus [LISK82] is the only other system besides Zeus which provides a set of general mechanisms for reliable management of distributed objects of any type. Zeus not only provides such general mechanisms, but also addresses several other issues not included in the Argus design such as object naming, object relocation, authentication and object protection. We have made an effort to address these issues in the Zeus design making it novel as compared to any other distributed operating systems. Another novel feature is the integration of the conventional database management functions into the operating system object management functions. This is an important advance in the operating system designs because most of the todays popular operating systems do not provide efficient mechanisms for database applications [STON81]. Even with respect to its recovery model, the Zeus design differs significantly from other known designs.

Much of the recent research in reliable system design is actually exploration into system structuring techniques which are more significant for distributed system designs than the conventional centralized systems. Such systems are intrinsically more complex than centralized systems. A structured approach reduces design complexity by factoring the designs into layers that create different levels of functional abstraction; the design of each layer can then be carried out somewhat independently of the design of the other layers. The layers in the system can be viewed as creating horizontal partitions in the system design.

Another structuring concept, which is dual as well as orthogonal to layered approach, is object-orientation which creates vertical partitions in the system. The interactions between these partitions occur through some well-defined interfaces; thus, each partition in the system represents an independent domain where the internal structure of a domain can not be directly accessed by other domains. A vertical partition essentially embodies the concept of objects in the system. The whole system is viewed as a collection of objects. All state transformations in one partition by other partitions are performed through the interfaces defined by the partition. The advantage of such an approach is that the design of the internal structure of any given partition is independent of the designs of other partitions. These are the fundamental principles of data abstraction. From the viewpoint of reliable system design, such an approach is very attractive because it supports confinement of errors within an object boundary. This also implies that the recovery mechanisms for a given partition can be designed to suit its reliability requirements.

There are two distinct approaches to designing reliable systems. The traditional approach takes a process-oriented view of the system where objects are bound to the address space of a process at the time of process creation and execution. The process is responsible for maintaining the integrity of these objects in the presence of faults and system crashes, and for recovering its locus of execution in the presence of faults. This approach uses checkpointing and rollback as primary recovery mechanisms for constructing resilient processes. Most previous operating systems have used system-wide

ZEUS DISTRIBUTED OPERATING SYSTEM

checkpointing, saving the state of all processes in the system, irrespective of need. The research in this area has addressed the problems of separately checkpointing interacting concurrent processes [KIM79] [RUSS80]. The major problem here is to avoid a domino effect in which the rollback of one process may lead to a cascade of rollbacks.

A second, more recent, approach, takes an object-oriented view of systems [LISK82]. In this view, objects are of distinct types; each type provides a defined set of externally visible operations. Each object is permanently bound to the address space of its object manager. Processes act upon these objects by invoking the visible operations implemented by an object manager. The object manager is responsible for enforcing necessary concurrency control rules and recovering objects from faults and system crashes. The primary recovery mechanisms include forward/backward logs, careful replacement, and object replication [KOHL81]. Processes are no longer responsible for recovering the objects they access during their execution; however, they are still responsible for recovering their execution locus. This requires establishing recovery points, and rolling back a process to some recovery point. A major advantage of the object-oriented approach is the clean separation between the recovery functions for processes and objects. Another advantage is that, for each object type, the recovery mechanisms and their design parameters can be selected to match the type's integrity requirements.

The concept of object-oriented design has been used in some recent distributed system designs such as Cronus [SCHA83], SWALLOW [SVOB81], Argus [LISK82], and in the approach presented in [SHRI81]. Argus provides object-oriented linguistic mechanisms for constructing reliable distributed systems, and SWALLOW provides reliable object management. These systems do not support some of the other operating system functions such as access control, naming, sharing, and resource management. Some of the functions supported by Zeus, such as naming, authentication, and interprocess communication, exist in some other recent operating systems such as Pilot [REDE80] and Grapevine [BIRR82], developed for network-based applications. Neither of these two systems can be regarded as a general purpose distributed operating system. Grapevine is intended only to support a distributed mail system.

The design of the Cronus operating system has significantly influenced the design of Zeus, largely because both these systems are intended for highly reliable applications such as command and control systems. Zeus provides users with reliable object management, which is not present in the current design of the Cronus system. Like Cronus, Zeus has the character of a general purpose operating system mainly because the nature of the command and control applications includes a wide range of processing characteristics. This is in sharp contrast to the requirements for banking or airline reservation systems where the application environment is well-defined. Zeus provides capabilities for defining and creating objects and transactions required by the application systems. It also provides mechanisms that support management of such objects in a reliable fashion. Zeus can be used for constructing any high reliability application system.

This chapter presents the basic object-oriented building block mechanisms provided by the Zeus distributed operating system. The concept of object managers is the basis for system structuring.. An object manager provides the encapsulation for a given type of objects; all objects of that type are accessed or updated via that object manager. The object-oriented recovery model underlying the Zeus design is described in Chapter 4. In this model the construction of reliable distributed objects is based on an atomic transaction facility and a remote procedure call mechanism. This approach is summarized in Figure 5-1.

DISTRIBUTED OBJECT MANAGMENT FUNCTIONS
(Partitioned and Replicated Objects)

RELIABLE REMOTE PROCEDURE CALL MECHANISM

ATOMIC TRANSACTION FACILITY

LOCAL OBJECT MANAGEMENT FUNCTIONS
(Concurrency Control, Recovery, Access Control,
Object Storage Management)

KERNEL FUNCTIONS
(Host Resource Management, Communication, Scheduling,
Remote Call Handling, Interrupt Handling)

HARDWARE

A Model for Reliable Distributed systems
Figure 5-1

The lowest layer in this figure represents the kernel functions that execute at every host node of the distributed system. Above the kernel layer are the local object management functions such as storage management, access control, synchronization, and object recovery. This layer represents the functions that are associated with every object manager in the system; the functions at this level deal only with the centralized object management. The next layer provides facility of atomic transactions; thus, a sequence of

ZEUS DISTRIBUTED OPERATING SYSTEM

operations can be performed on a set of objects in an atomic fashion. The remote procedure call mechanism facilitates operations on objects that are not local. We have adopted the remote procedure call mechanism because it provides a uniform way of accessing remote as well as local objects; thus, location of the object is transparent to the users during access or update operations. It is important to make the semantics of remote and local procedure calls identical in the presence of host crashes and communication link failures. In our design we have adopted the "at most once" execution semantics for remote procedure calls; thus, in the presence of duplicate messages or on server node crash-restart, effectively only one execution of the remote procedure will occur. The combination of the remote procedure call mechanism with the atomic transaction facility is used for managing objects that are either partitioned or replicated. Based on these mechanisms one can suitably create type definitions for replicated or partitioned objects such that one can access or update those objects in the same manner as updating centralized objects.

The object management model used in the Zeus design is based on the concepts developed in the Hydra [COHE75] design. In an object-oriented approach, the system is comprised of a set of objects, and each object is of a well-defined type. A Type Manager object for a given type manages all objects of that type. All operations on permanent and shared objects in the system are executed via their type managers. There are some obvious differences between the protection models used in the Hydra and Zeus designs. The protection mechanism in the Zeus design is based on access control lists while the Hydra model is capability based. Although both these models are equivalent in terms of their functionality, they differ with respect to their operational environment. The prime reason for using the access control list model in our design is to be able to change the access rights dynamically. Although it is not very efficient to change access rights dynamically in a capability based system, dynamic changing of access rights is important in a command control system where some of the nodes might be taken over by hostile forces.

5.1 PRINCIPLES OF DISTRIBUTED OBJECT-ORIENTED DESIGN IN ZEUS

5.1.1 Structure of Object-Oriented Systems

An object-oriented system consists of a collection of Type Managers and the objects created by them. As described above, the Type Managers create vertical partitions in the system. For a given type in the system, a Type Manager would exist at all those nodes which may be required to store objects of that type. A Type Manager at a node manages all objects of that type at that node. The multiple instances of Type Managers for a type function cooperatively to provide the abstraction of a single Type Manager for that type in the system. Each Type Manager defines an address space in which all the objects of that type reside. A Type Manager is logically viewed as a single process that performs all the state transformations on the objects in its address space in response to execution requests by some other objects of the same or different type.

At a physical node, several different Type Managers may reside, each managing objects of its type at that node. The abstract machine to support such an object-oriented system can be constructed from almost any hardware/software system architecture. The system architecture, which includes the hardware, software, and the firmware architecture, of the processors to support such a system must have: (i) a mechanism for switching the processor between Type Managers, (ii) a mechanism for partitioning secondary memory resources among Type Managers, and (iii) a mechanism for exchanging messages between Type Managers.

It can be seen from the preceding model of Type Managers that there is no concept of a system-wide state or uniform control and/or recovery mechanisms. Resource management functions and recovery mechanisms are partitioned along with the set of Type Managers. The traditional functions of system-wide software units such as operating systems and database systems are incorporated into a collection of Type Managers which implement the basic elements of the model of distributed computations. This is a radically new view of operating systems.

Object Type Managers are the primary building blocks for the permanent elements of the system. The Type-Type Manager is an object in the system that manages "types" in the system. It is the means by which new types are introduced into the system. The concept of the Type-Type Manager is essentially the same as that of the TYPE-TYPE object in the Hydra design [COHE75]

The objects in the system are accessed in a uniform fashion regardless of their locations. All operations on permanent objects are performed within a transaction. A transaction is basically an atomic action that is defined as a sequence of operations on local or remote objects. A transaction ensures atomicity of distributed operations. It is possible to introduce concurrency within a transaction by creating one or more nested parallel transactions.

5.1.2 Object Naming

The most basic requirement at the lowest level of the system architecture is to identify and refer to objects unambiguously. This requires that each object must be associated with some system-wide unique identifier (UID). In the design model adopted for Zeus, a unique identifier is associated with every object in the system; from this identifier the "type" of the object can be inferred. One of the fields in the UID contains the ID of the host where the object was originally created; this field acts as a hint for the most probable location of the object. Based on the "type" field in the UID, the access to an object is directed to the appropriate Type Manager at the node given by the "host" field of the UID. In the Zeus design we have adopted the concept of "extended" UID; an extended UID contains one additional field which records the ID of the host where the object was present during the most recent access; thus, any subsequent access is directed to the new host address as an object relocates.

ZEUS DISTRIBUTED OPERATING SYSTEM

5.1.3 Functions of the Type Managers

The functional characteristics implemented by the Type Managers are the original basis for defining abstract data types. Extending abstract data type concepts to include a formal basis for the integration of recovery, synchronization, and access control mechanisms generates a number of additional functions for the Type Managers:

1. Each Type Manager is directly responsible for the mapping of the occurrences of the objects they define to physical storage.
2. Each Type Manager implements access control policies for the occurrences of its type.
3. Each Type Manager supports concurrent execution of its procedures and/or functions.
4. Each Type Manager ensures the consistency of the objects it stores under concurrent and distributed use.
5. Each Type Manager implements the necessary levels of redundancy to ensure the level of fault tolerance given in its specification.

This obviously integrates many functions that have been conventionally associated with database systems into the object management functions of this operating system.

5.1.4 Structure of Type Managers

Externally viewed, a Type Manager is a collection of functions and procedures which can be invoked on the objects of its type by specifying the identifier of the object along with the operation name. This causes an invocation request message to be sent to the Type Manager regardless of its physical location in the system. Internally, these operations are executed by the Type Manager using one or more server processes; such server processes may be dynamically created or destroyed by the Type Manager. The operations on remote and local objects are invoked by the clients in the same fashion as procedure call. Such invocations on remote objects are performed by implementing remote procedure calls [NELS81] [SHRI82] with "at most once execution" semantics. A Type Manager consists of:

- Data structures for the objects of that type;
- Procedures/functions defining the type;
- Concurrency protocols;
- Recovery mechanisms;
- A database to manage the objects in its domain;
- A controller process that schedules/executes the requests.

A Type Manager is responsible for the permanent storage of the object instances of its type. Each Type Manager interfaces directly with some set of permanent storage devices. The Type Manager generates the mapping from the UID for an object of its type to the physical storage on some permanent storage devices. It also realizes object instantiation in the executable

volatile storage from the permanent storage. There is no system-wide file system. The object management system takes the place of a file system.

A Type Manager consists of a controller process whose purpose is to schedule server processes to serve client requests. The server process is given the same UID as that of the client process; thus, a client process is conceptually viewed as migrating into the address space of the Type Manager. This view of the migrating client process is useful from the viewpoint of enforcing access rights associated with the client process. On the completion of the requested service, the server process is deallocated. The controller process accepts the incoming or outgoing invocation request messages, performs security checks, and interfaces with the kernel procedures. Effectively, the controller process plays the part of a local operating system for the Type Manager; the scheduling policies can thus be tailored to the specific requirements of the Type Managers. The controller process manages the server processes performing the operations and provides them with a set of procedures that perform resource management, communication, protection and other services that are normally provided by an operating system.

A Type Manager's controller has several responsibilities related to protecting its objects from unauthorized access. Upon receiving an invocation request, the controller must obtain and store the requesting process' identification. This information is made available to the operation via a callable procedure so that the Type Manager's controller may check the access list of the object. In addition, the controller appends the identification of a process which is making an outgoing invocation request to some other Type Manager.

When an incoming invocation request is received, the controller attempts to locate the object whose UID is given in the request. First, the controller looks for the object in its own local pool of objects. If found, the program which will perform the operation on the object is parameterized with the object's local address and then is scheduled as the server process. If the object is not found locally, the controller determines if a "forwarding address" has been left for that object. This might occur if the object has been relocated to some other host. If the object is not found locally, the controller sends a reply message indicating that the object was not found and includes the forwarding address if any.

In response to an update request, the Type Manager creates a new version of the object. This version is committed only when the transaction that created it commits; the uncommitted versions are discarded if the transaction aborts.

Each Type Manager maintains a database which records the necessary information pertaining to the objects in its address space. This database records the identifiers of the objects of that type currently present at that node, their physical addresses, and the commitment status of their most current versions. A Type Manager is also responsible for aborting a new uncommitted version by timing out if it detects no activity of the transaction that created this version. Every time a new version of an object is created by a transaction by invoking an update operation, the Type Manager ensures

ZEUS DISTRIBUTED OPERATING SYSTEM

that this new version is written onto the stable storage before sending an acknowledgement for the operation. A scheme for maintaining such multiple versions using differential files is described in Chapter 4.

Type Managers are responsible for ensuring that each of their defined operations is atomic. The operation must either complete successfully or else abort, leaving the object completely unmodified. This is not difficult to achieve if only local objects are being modified in the operation. However, if the operation involves invoking operations on other Type Managers, then the controller uses the transaction facility to ensure the atomicity of the update. If the Type Manager is structured so that operations may be executed concurrently, the controller ensures that objects are not being modified by two operations simultaneously or read by one operation while being modified by another. Each type, in general, has its own set of constraints on the allowed order of execution of its operations on a given object. These constraints are supplied when the Type Manager is created.

5.1.5 Distributed Types

The reason for introducing the concept of distributed types in the system is to make transparent the distributed nature of an object that is logically viewed as a single object. The components of an object may be distributed by replication or partitioning. The transparency of the replicated or partitioned nature of an object is a convenient abstraction which makes updating and accessing of distributed and centralized objects identical.

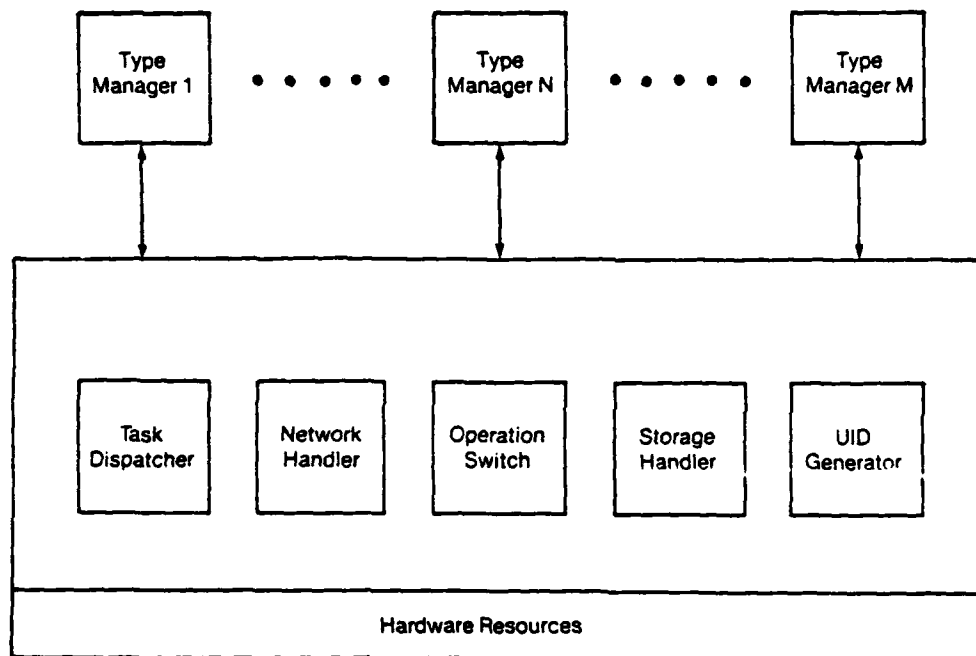
A distributed type is an abstract data type whose concrete representation is distributed. For example, an abstract type called reliable-file might be implemented using physically distributed replicated copies of a file, or a global database might be implemented as a set of partitioned distributed components. The consistency and coordination among the distributed components of the concrete representation is specified in the type definition and enforced by the distributed Type Manager. Unlike the centralized objects, an occurrence of a distributed type does not have a unique host location, i.e., an object of a distributed type may "reside" at more than one host for reliability and performance reasons. An occurrence of a distributed type is given a UID, the Type Manager then maps the operations directed to this UID into a set of operations, which are executed as a transaction, on the components that comprise the distributed object's concrete representation. This mapping can be done at any of the hosts where the distributed object is conceptually "residing". The operations defined for a distributed type are implemented as transactions.

5.2 STRUCTURE OF THE ZEUS SYSTEM

Zeus is essentially a collection of Type Managers (TMs); typically, many different Type Managers coexist on a host node. The core of the operating system consists of a set of Type Managers that support capabilities for defining new types and object instances in the system, authentication of

users, naming environment for each user, and reliable process and transaction management functions. These system-defined Type Managers reside at every node in the system.

The lowest level of operating system at each node is called the kernel; the kernel virtualizes the resources at the host so that each Type Manager can be viewed as having its own virtual processor. The kernel supports interprocess communication, primary storage management, processor scheduling, interfaces to secondary storage devices, and UID generation. As shown in Figure 5-2, all Type Managers at a node execute over the abstract machine interface provided by the kernel. The kernel multiplexes the processor between the Type Managers; it also handles all interrupts due to storage devices and the communication devices.



Structure of a Zeus Host

Figure 5-2

5.2.1 Structure of the Zeus Kernel

The Zeus kernel provides low level services to the type managers of the system. These services include three important functions 1) interprocess communication, 2) storage management and 3) unique identifier (UID) generation. The UID generation in turn depends on the failure detection and recovery of hosts in the Zeus system. The kernel consists of a task dispatcher and a number of interrupt handlers. The task dispatcher schedules

ZEUS DISTRIBUTED OPERATING SYSTEM

the different Type Managers at its host node and handles their requests for resources. It also handles the restart of the system and initiation of the Type Managers. The resources managed by the kernel include volatile and non-volatile storage, the processor and the communication handler. The kernel interface consists primarily of three parts: invocation requests to other Type Managers, requests for unique numbers, and requests for resources.

Interprocess communication is achieved by the mechanism of remote procedure call (RPC) which consists of four messages interchanged between caller and callee. These are call, call acknowledge, response and response acknowledge. For each call that is made from or to a type manager the status of the call parameters and status must be stored. To do this each type manager has a call handler to perform this function. The synchronous nature of the RPC is achieved by the type managers who will first issue a call and then on getting the response will inform the caller of it.

The storage functions of the kernel are performed at the object level; thus, calls to the kernel can retrieve, store and delete objects. Further stable storage operations can be executed by the kernel, where stable storage is implemented using the Lampson [LAMP81] scheme. Storage management in the kernel is minimal. Storage is available in fixed sized blocks and the Type Managers request one or more of these blocks at any time. A Type Manager is solely responsible for the data it writes to the blocks of storage. The kernel keeps track of the ownership of blocks of storage. The routing of invocation requests to Type Managers is the major function of the kernel. Each call is an operation invoked against an object that is held by some Type Manager. Operation Switch, which is a component of the kernel, supports this function.

UID generation is a function used by the RPC and by the type managers so that calls and objects can be uniquely identified. This function must continue despite failure and recovery of hosts. To achieve this the hosts participate in a distributed computation to keep track of active hosts and to let new or recovered hosts join in the UID generation function.

5.2.1.1 The Operation Switch

The function of the Operation Switch is to forward an invocation request to the appropriate Type Manager at the local or a remote node. These calls may be from a Type Manager or from the network driver. Each call contains the following information:

1. The extended UID of the object against which the call is invoked.
2. The extended UID of the process invoking the operation.
3. The extended UID of the principal on whose behalf the operation is being invoked.
4. The operation and a set of parameters.

The Operation Switch uses the host hint field of the target object's extended UID to determine whether the object is on the host or not. If it is, it uses the type unique number of the object to direct the call to the proper

Type Manager. If the object is on another host, the Operation Switch instructs the Network Handler to send the call to the other host.

5.2.1.2 Unique Identifier Generation

The "type" and "instance" fields of an extended UID are unique numbers. Each of these unique numbers consists of three fields, the host identifier of the host at which they were generated, the incarnation number and the sequence number within an incarnation number.

In a system where no failures can occur, each host will generate a monotonically increasing sequence of unique identifiers. If we permit failures, but stipulate that every host in the system has stable storage, then each host will store the next incarnation number, and as soon as it restarts on crash recovery, it will retrieve this number and write to stable storage the next incarnation number; thus, even though some part of a range of sequence numbers may not be generated, the hosts will generate a monotonically increasing sequence of unique numbers.

Reliable unique identifier (UID) generation is one of the important and critical problems in a Zeus-like system. There have been many general algorithms proposed for implementation of UID generation systems. These include sequencers in the shared memory model, and several algorithms for mutual exclusion in arbitrary communicating networks that allow process failure. This section presents a reliable UID generation scheme that uses a broadcast communications medium to implement an efficient reconfiguration under process failure conditions using timeouts.

In this section we construct an efficient failure tolerant sequencer by replicating copies of the sequence value. We implement the sequencer in a distributed system connected by a broadcast medium, such as an Ethernet. This configuration is important since it is in wide use for local area network design. We drastically reduce the overhead in the algorithm by using a timeout mechanism based on dynamic priorities, where the priority calculations do not require any message overhead. The following description is presented in terms of a general scheme for generating sequence numbers reliably.

The system consists of N processes, where Q processes contain a copy of the sequence value, and are termed servers. Only one server should be incrementing and delivering sequence values at any time, and is termed the sequencer. All servers not functioning as the sequencer simply record each successive value of the sequencer until at some time they may function as the sequencer. The problem we address is how to efficiently determine when a server is to function as a sequencer, and how to efficiently reorder the remaining servers.

A distributed system is defined by the behavior of the communication structure and the processes resident in the system. The communication structure of the system discussed in this section should meet the following requirements:

The communication medium connects all active processors and is continuously available.

ZEUS DISTRIBUTED OPERATING SYSTEM

The set of servers active from the time a message was broadcast on the medium will either all receive the message, or no server will receive the message (this is sometimes known as reliable broadcast, and is required only of the set of servers).

All messages sent by a process are received in the order they were sent.

Additions to the ethernet protocol have increased the probability of reliable broadcast. Several distributed operating systems are being built using algorithms based on the assumption that reliable broadcast will hold for a significant time span. For these algorithms, checking and correction procedures may occasionally need to be invoked. In the last section we discuss how configuration decisions can increase the probability that a reliable broadcast will hold, and additions to the algorithm which take care of a certain number of spurious messages or inconsistent values of the sequence number.

Any process resident in the distributed system must have continued access to the communications medium. A process can be in one of three states:

failed: the process is not running, and the contents of its local memory may have been lost,

entering: a process is either entering the system for the first time, or had failed and is now reentering the system, or

active: a process has recreated a state consistent with the rest of the network and is executing.

All active processes have a unique process number. The server processes have several requirements necessary for the algorithm. All processes in the system can function as server processes, but we introduce the distinction of the server for efficiency and clarity of purpose. For the algorithm to proceed, at least one server process must be active at any time. Initially server P_1 is assigned priority 0 and is intended to function as the first sequencer. Every active server possesses:

A copy of the most recent (largest) value of the sequencer.

A unique priority number calculated from the server's own process number and the process number of the most recent sequencer.

A timer based on a local clock and a time constant T , such that T measured on the local clock is larger than the maximum time an active server can take to receive and immediately respond to a request.

Notice that only the active servers need have a current copy of the sequence value, and this value is not assumed to be known by any other process in the system. This is an important restriction for both the reentering server and the algorithms which we implement using the sequencer in section four. The time constant T may be determined by some constant times the round trip delay propagation of the broadcast communications medium and the time it

takes the slowest server to receive a message, process a request, and send the corresponding message. The active server with priority number 0 functions as the sequencer for the distributed system. Several sequencers can be executing concurrently, as long as each sequencer variable is named uniquely in the distributed system. For simplicity, we describe only how one sequencer is implemented on the network.

We now introduce the basic algorithm to implement sequencers on a system that allows process failure. We begin by explaining how one server answers a sequence request message. We then explain how the other servers update their value of the sequence number, and finally how the dynamic reconfiguration is done using the same message sent in answer to the sequence request.

At system initiation time process numbers $1..Q$ are assigned to the servers and $Q+1..N$ are assigned to the remaining processes. For simplicity we assume server numbers are identical to process numbers, and if they are made distinct each reference to a process number of a server below should be replaced by a reference to the server number. Each server P_i has $\text{priority}(P_i)=i-1$ initially. Each server has the initial value of the sequencer set to the same predetermined value (for example, if we are assigning process numbers to new processes joining the network, it should be set to N). When a request sequence(S) for the next sequence number is sent over the medium, the following algorithm begins for each active server in the system.

When a server P_i detects a sequence(S) request from P_j , its timer is set to $(T)*\text{priority}(P_i)$. This time interval is sufficient for an active server with a lower valued priority number to respond to the request. If the timer expires before the answer to the request is detected on the communications medium, P_i will function as the sequencer for the request. P_i increments S and then replies to P_j with message (SEQ,i) , where SEQ is the current value of S and i is the process number of P_i . We note that the response to the request for the next sequence number should be of high priority, so when P_i 's timer expires server P_i will block the communications medium until it sends the response; therefore, the time constant (T) is a constant and does not reflect the utilization of the communications medium.

If server P_k detects an answer to the request on the communications medium before its timer expires, P_k reads the value of the response. P_k sets its local value of S to SEQ to update its copy of S . If P_k is just reentering, it may not have received the request that prompted the response, but will update its value of S . This is correct behavior, since all messages are received in the order they are sent on the network, and P_k will not receive the request anytime in the future. Hence, a server need not receive the original request in order to continue in the algorithm, in case it was failed when the request was broadcast.

If the previous server P_i with priority 0 had failed, a reconfiguration algorithm must take place. A new sequencer is selected by the timeout mechanism described above. This process P_j must set its priority to 0 in order to ensure fast response to the next sequence request. Any other process P_j should also reorder and obtain a smaller priority number to

ZEUS DISTRIBUTED OPERATING SYSTEM

decrease the amount of delay calculated by $(T) * \text{priority}(P_j)$. A simple way this can be done without any message overhead is to have a server P_j that sends or receives a response (SEQ,i) calculate its priority number by $(Q - i + j) \bmod Q$, where j is the process number of P_j , i the process number of the sequencer, and Q is the maximum number of servers in the system. This calculation also has the desirable property that, if all processes with lower priority number fail, eventually a reliable server with a larger process number will have its priority set to 0. It is a very inexpensive reconfiguration algorithm, since it does not incur any message overhead for the reassignment of priorities.

We now present a simple example to illustrate the use of the time out mechanism to reconfigure the sequencer in the event of process failure. Consider a case where processes $P_1..P_4$ are servers and process P_5 periodically requests a sequence number. Initially priorities for $P_1..P_4$ are $(Q-1 + j) \bmod Q$, or $j-1$. The first event occurs when P_5 requests a sequence number. P_1 responds by sending $(x,1)$. Each server P_j updates its value of S to x , and any priority calculations return a value of $j-1$, since P_1 remains the sequencer. Next, P_1 and P_4 fail, which is followed by a sequence request from P_5 . P_2 waits for $1*(T)$, then sends $(x+1,2)$. P_1 recovers in time to receive this message. Hence, P_3 and P_1 update their value of S and recompute their priorities to be 1 and 3, respectively. In this manner it is easy to see that stable processes with high process number will eventually obtain a priority of 0. This also demonstrates how easily P_1 returns to an active state, in this case with no additional overhead to the distributed system.

As illustrated above, this same algorithm can be used to allow a process to switch from an entering state to an active state. A process P that is entering the system for the first time sends a sequence number request to obtain a process number j , and places it in memory. A server that has failed and is reentering needs to reestablish the current value of S and calculate its priority before it can become active in the computation. It can obtain the present value of S and calculate its priority by reading the value of a response being sent across the communications medium. The reentering server can also prompt this response by sending a sequence(S) request. However, it would be best for this server to use the primitive read(S), where the current value of S is sent without incrementing the value of S . Hence, if several processes are reentering the network, only one process needs to send a read(S) request, any other processes can simply update their value of S and recalculate their priority number from the values contained in the message if they are server processes.

5.2.1.3 Network Handler

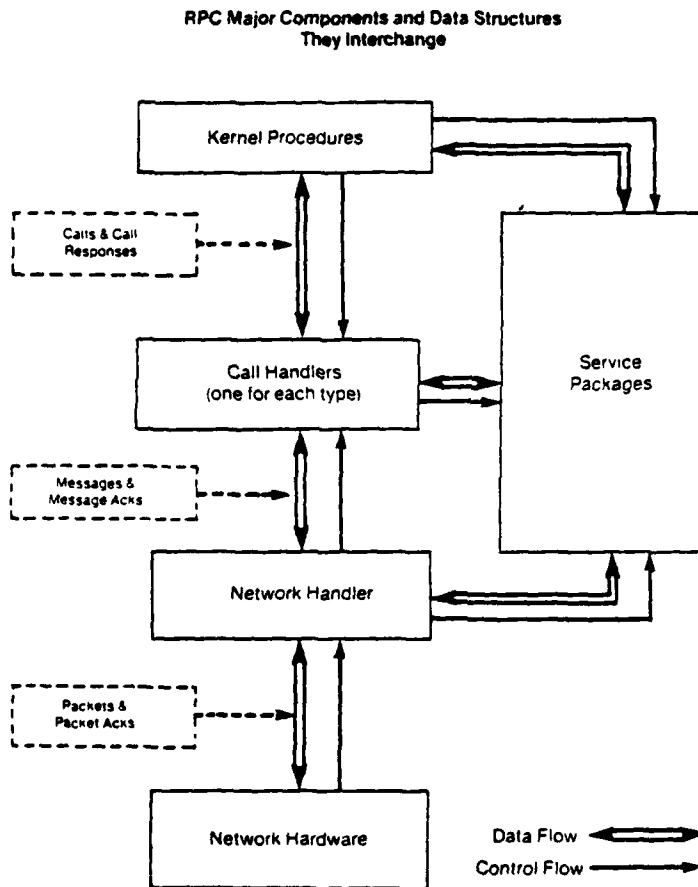
This component provides a simple datagram level of transport mechanism between different kernels. It interfaces with the Operation Switch. The invocation requests for remote nodes are handed over by the Operation Switch to the Network Handler, which has the responsibility for delivering it to the Operation Switch at the destination host. Similarly the response messages are returned from the server to the invoker by the network handler via the Operation Switch.

5.2.1.4 Kernel Initiator

The kernel initiator has two functions. The first function is to restart a host when it recovers from a failure. The second is to initiate a task. Both tasks require a certain amount of housekeeping. Host recovery implies the setting up of tables for the dispatcher of the kernel, using the log for the Type-Type Manager to create, delete, or modify the Type Managers on the host, and obtaining a new incarnation number. After the above actions are successfully completed, the initiator can hand control to the task dispatcher.

5.2.1.5 Kernel Design

The overall structure of the three kernel functions is given in Figures 5-3 through 5-5. The structure of each function is described next.



Major Components and Data Structures of RPC
Figure 5-3

ZEUS DISTRIBUTED OPERATING SYSTEM

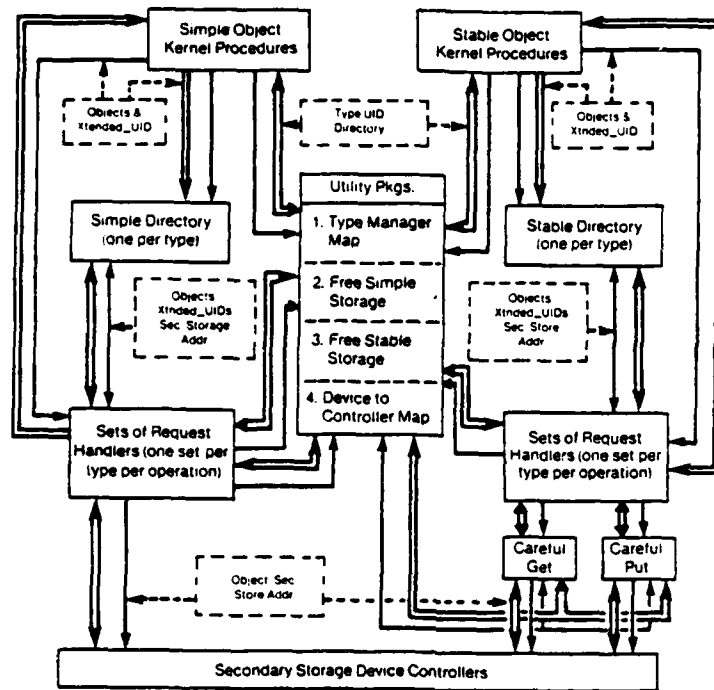


Figure 5-4 Object Storage and Retrieval:
Data and Control Flow

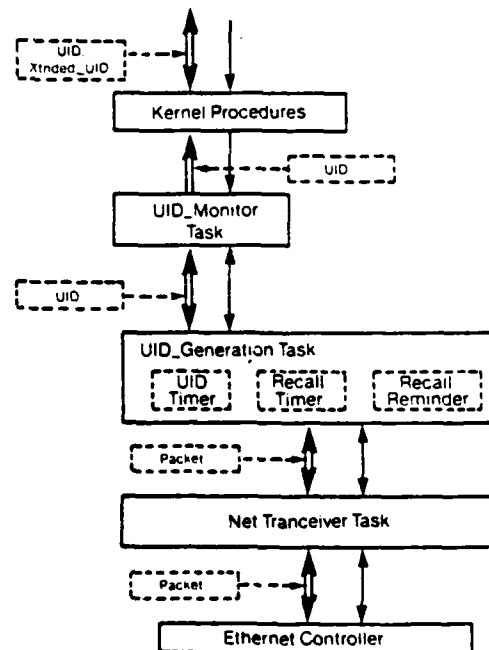


Figure 5-5 UID Generation and Site Recovery Architecture

Each remote procedure consists of four messages. Each message must be packetized at its source and re-assembled at its sink. There are four levels in the RPC mechanism. The first is a set of kernel procedures that form the kernel interface for the RPC mechanism. These procedures are invoked to send calls and obtain responses. In addition they differentiate between local and remote objects thus performing the function of the operation switch.

The next level consists of one call handler per object type and is the repository of the state of all outgoing and incoming calls. Each call handler is invoked by the kernel procedures from the level above it and the network handler processes from the level below.

The network handler consists of three processes, the send driver, the receive driver and the network traneiver. Together the send and receive drivers implement the link level protocol. This includes the processing of acknowledgements (positive and negative) at the packet level. The send and receive drivers interact with the network traneiver to send and receive data packets to and from the network. The network traneiver interacts with the UID generation task and the net controller in addition to the send and receive drivers.

Object storage and retrieval has a very simple structure. Each object type has a simple object directory and a stable object directory. Associated with each directory are sets of tasks, one set for each operation to be performed. These tasks are called request handlers. The kernel interface procedures first ask the directory to perform some operation. The directory initiates the task and returns to the kernel procedure a pointer to the request handler assigned to the task. The kernel procedure then asks the request handler for the result. The directories contain pointers to all objects of a given type. The directories always reside in main memory and directory storage and retrieval has not been considered in the design. Storage of the directories would involve setting up a directory structure. Request handlers are allocated to each directory based on the designer's estimate of how often operations on a particular object type will be invoked and how long those operations will take. Secondary device space is allocated on an object basis and simple and stable storage have free storage managers. The two kinds of storage are managed separately since they will be kept on separate devices.

The third major function of the kernel is to generate unique identifiers for objects. A unique identifier consists of a host field, an instance field and a sequence field. The instance field is generated collectively by all the hosts to the system. For each value of the instance field the UID monitor process will generate a range of UIDs whose cardinality equals that of the sequence field. The generation of a new instance number is accomplished by the UID generation process of the kernel.

Additionally, the UID generation process participates in a roll call computation to keep track of active hosts, and a host restart computation to permit a new or restarting host to join the set of active hosts.

All the UID generation processes on the active hosts in a cluster participate in the generation of a new instance field. The algorithm for UID

ZEUS DISTRIBUTED OPERATING SYSTEM

generation is based on the principle that all the active hosts in a cluster have a dynamically maintained linear order at all times. When all the active hosts in the cluster have received a new instance number request from one of their members they each set a timer in the process UID timer and wait. If a host's timer expires before that host receives a response to the request. This message contains the new instance field value and the responding host's position in the linear order. When a host receives a response to a request (either its own or that of a host with a lower position in the linear order) it accepts the new instance value and subtracts the responding hosts position in the linear order from its own. Thus each active host receives the new instance number. Further, if some host in the linear order has failed, the other hosts ranked lower than it will have their position moved higher in the order. This will over time result in reliable hosts reaching the highest positions in the linear order.

To periodically check for the active hosts the active hosts execute a roll call protocol. To do this they set the RECALL REMINDER process's timer is set to be equal to some time delay plus a delay proportional to the host's position in the linear order. The first host whose RECALL REMINDER timer expires starts the roll call computation by sending out a start roll-call computation message. To start the roll-call computation, each host sets two timers, RANK TIMER and FULL TIMER in the process RECALL timer. The RANK TIMER's time is set proportional to the current rank while the FULL TIMER is set proportional to the maximum number of hosts in the cluster. When the RANK TIMER expires a signal is sent to the UID GENERATION task which then broadcasts its own "I AM ALIVE" message on the network. When a host receives such a message from some other host, it enters its host ID in the new rank list that is being prepared by the roll-call procedure. The roll-call procedure terminates when the FULL TIMER interrupt occurs.

Hosts that fail rejoin the system by initiating a distributed computation among the active hosts UID generation process's. This computation is identical to the host roll call computation described earlier. This computation pre-empts the host roll call or UID generation computations. Thus the host restart computation is a poll of active hosts triggered by a new active host.

The messages sent by UID generation process take precedence over the remote procedure call messages. The UID generation monitor process when the latter exhausts the number's in the previous instance field's range. The UID monitor is invoked by the get_UID kernel procedure which in turn is invoked by the kernel users.

5.2.1.5.1 The Kernel Interface

The kernel interface consists of three parts; remote procedure calls, object storage management and unique identifier generation. Each part has a set of procedures that can be invoked from the type managers or by user proceses.

5.2.1.5.1.1 The Remote Procedure Call

The Remote Procedure Call functions provide the call invoker with the facilities to initiate a call, receive the response to a call and to inquire about a call's status. Similarly, the recipient of a call has the facility to receive a call, make a response to a call, and inquire about a response's status. Functions also exist to cancel a call or retain a call. Each call is identified uniquely in the system by a unique identifier. A request for a call or a response will return to the caller any call or response that is waiting for the caller; it is then the caller's responsibility to deliver the call or response to the correct process. The kernel interface procedures are specified below:

1. procedure `make_call` (type of caller, source of call, destination of call, call contents, call options, call unique identifier, call status). The call contents include the operation to be invoked and the parameters of that operation.
2. procedure `get_rasp` (type of caller, call unique identifier, call's response). The call unique identifier and response and output by the kernel; thus, the invoker cannot make a request for a particular response.
3. procedure `c_status` (type of caller, call identifier, call status). The call status is returned and will tell the invoker of the current status of the call, i.e., whether it has been delivered, whether the response to the call has been received or whether the call has failed.
4. procedure `make_resp` (type of caller, source of call, destination of call, call identifier, response options, response status). The `make_resp` procedure has very similar parameters to the `make_call` procedure.
5. procedure `get_call` (type of caller, call identifier, response contents).
6. procedure `v_status` (type of caller, call identifier, response status).
7. procedure `kill_call` (type of caller, call identifier, call status).
8. procedure `keep_call` (type of caller, call identifier, call status). The `kill_call` and `keep_call` procedures are used to update the local tables for a call.

5.1.4.2 Object Storage Management

The Object Storage Management functions permit type managers to store, retrieve and delete objects. Objects can be stored on simple or stable devices; thus the interface has two sets of calls.

The procedure definitions for the storage management functions are:

1. procedure `get_obj` (type of object to be retrieved, identifier of object to be retrieved, object contents, operation status). This is for simple object retrieval.
2. procedure `put_obj` (type of object to be stored, identifier of object to be stored, object contents, operation status)
3. procedure `del_obj` (type of object to be deleted, identifier of object

ZEUS DISTRIBUTED OPERATING SYSTEM

- to be deleted, operation status)
- 4. procedure `stable_get`
This has identical parameters to `get_obj`.
- 5. procedure `stable_put`
This has identical parameters to `put_obj`.
- 6. procedure `stable_del`
This has identical parameters to `del_obj`.

5.1.4.3 Unique Identifier Generation

This part of the interface permits an invoker to obtain a new unique identifier, construct an extended unique identifier, obtain the host hint of an object and change the host hint of the object.

The functions for UID generation are:

- 1. Function `get_UID` returns UID.
- 2. Function `build_xt` (host hint, object type UID, object instance UID, object version UID) returns extended UID
- 3. Function `give_host_hint` (object extended UID) returns object host hint
- 4. Function `change_hint` (object extended UID, new host hint) returns modified object extended UID.

5.2.2 System-Defined Type Managers

As mentioned previously, Zeus is a set of Type Managers whose members may potentially change dynamically as Type Managers are created, deleted, and modified. There is, however, a subset of Type Managers called the System Type Manager which perform the essential services provided by the kernel of a conventional operating system. In this section, the Type Managers for these system types are defined. The following are the System Type Managers which exist at each node in the system:

- (1) Type-Type Manager
- (2) Process/Transaction Manager
- (3) Principal and Authentication Manager
- (4) Symbolic Name Manager
- (5) Program Type Manager
- (6) Message Type Manager

The functions provided by these Type Managers along with their structures are described below. Each of these Type Managers is considered as an object of distributed type; an instance of each of these Type Managers resides at every node. The distributed Type Managers for a given type function cooperatively to provide the abstraction of a single system-wide Type Manager.

5.2.2.1 Type-Type Manager

The definitions of new Type Managers is introduced in the system by using the mechanisms supported by a system-wide object called the Type-Type Manager;

thus, the Type-Type Manager implements functions to create, alter, delete and replicate Type Managers. The definition of the Type-Type object given here is an adaptation and extension of the Type-Type concepts originating in the HYDRA [WULF81] operating system. The facilities provided by the Type-Type Manager include an explicit command on where to locate copies of a Type Manager.

Type managers are active objects. At any point in time, one or more copies of the Type Manager for a given type may be active. By active we mean that either within the Type Manager calls against its object instances are in progress, or that some of the functions it implements have invoked calls on some other Type Manager and are waiting for a return. This complicates the Type-Type manager because it must ensure that all copies of a Type Manager are in a quiescent state and will stay in that state before an operation can be invoked against that Type Manager. However, we believe that operations to modify existing Type Managers will be quite infrequent; therefore, schemes based on global synchronization can be used for consistency management.

5.2.2.2 Process/Transaction Manager

Processes and transactions are active objects in the system through which a user carries out operations in the system. Transactions are atomic processes, i.e., they have an "all or nothing" property. The transaction facility with its atomic property provides a powerful mechanism for reliable operations. A transaction either commits or aborts on termination, and if it aborts then no trace of its execution is left. On the commitment of a transaction, all updates made by it are permanent.

We require that a process must invoke a transaction in order to modify permanent shared objects in the system. The changes to an object are recorded as new versions of the object. New versions of the object are committed to becoming permanent at the end of a successful completion of the commit protocol among the invoked transaction, the invoking process, and the Type Managers of the modified objects. Uncommitted versions are discarded on explicit abort commands issued by the transaction process or on timeout due to inactivity.

Processes and transactions can establish recovery points by checkpointing. Such points are used for the purpose of rollback and restart of a process or transaction. Checkpointing is the selective saving of versions of process or transaction objects. Note that with the above scheme for checkpointing, only the state of the process (or transaction) object is saved; the states of objects modified by that process are not saved in the checkpoint. This approach may create problems for error recovery since not all state changes of the process are recorded with the checkpoint. However, one must remember that all updates made within a transaction to permanent objects via their Type Managers are saved on the stable storage as uncommitted versions. It is, therefore, necessary to exercise some discipline in using checkpoints and atomic transactions. In the following parts we discuss how checkpointing can be used correctly to support recovery.

The first problem that we want to address is how one guarantees correct rollback recovery. One requirement for correctly implementing rollback is that the object manager for transactions must maintain for each transaction a

ZEUS DISTRIBUTED OPERATING SYSTEM

list of UIDs of the objects that are affected by the transaction. The reason for this is that in the event of a rollback, it requires that all changes to objects made by the transaction after the checkpoint be discarded. The list of UIDs of the objects that are affected by the transaction provides a means for notifying these objects to discard the aborted versions. This list is also used at the end of the transaction to conduct the commit protocol. The discussion in the preceding paragraph implies that for implementing rollback, timestamps must be recorded with each checkpoint and each version of objects. This requirement stems from the fact that there is not a one-to-one mapping between process or transaction checkpoints and versions of objects affected by them.

The second problem is the interaction between process checkpointing and commitment of a transaction invoked by the process after that checkpoint. Suppose a process crashes after committing a transaction. In such a case the process restarts from its last checkpoint, but the transactions that have been committed since the establishment of this checkpoint are not undone; thus, some committed transactions might be executed more than once due to the restart. If a transaction is nonidempotent, i.e., multiple executions of the transaction produce different results, a problem may arise in error recovery since rollback of the process may cause a committed transaction to be executed again. One solution to this problem is to always force the invoking process to checkpoint concurrently with the committing transaction. Checkpointing is part of the commit protocol; if the protocol determines to abort, the checkpoint is discarded. With this mandatory checkpoint, rollback recovery of a process can avoid undesirable repetition of transaction execution. However, this may cause too frequent checkpointing of the invoking process; therefore, the second solution is to make checkpointing of the invoking process an option that is to be specified at the time of invoking a transaction. Apparently this checkpoint is not required for idempotent transactions to guarantee correct execution. However, a process invoking an idempotent transaction may elect to force a checkpoint during the transaction commit protocol for efficiency reasons. For example, if a transaction requires extensive computation compared to checkpointing the invoking process, and if the possibility of a failure is significant, it may be desirable to have a checkpoint as described above. The decision of when to checkpoint is left to the process that invokes the transaction.

The Process/Transaction Manager also supports nesting of transactions; such nested transactions can execute concurrently with the parent transactions. The nested transaction facility provides the users mechanisms to introduce concurrency within a transaction. The commitment of a nested transaction is dependent on the commitment of the parent transaction.

5.2.2.3 Principal and Authentication Manager

The object protection system in Zeus depends on the ability of the individual Type Managers to identify any process which requests an operation be performed. In addition, the Type Managers need to be able to determine the ultimate initiator of the action which resulted in such an invocation request. We call these initiators of actions principals. Principals are permanent objects in Zeus and they are the only objects which carry the authority to

perform computations involving other objects. When a new process is created, it is "owned" by a single principal and it retains this principal association throughout its lifetime.

The two fundamental problems of the protection system, authentication and authorization, both involve principal objects and the association of processes to principals. The problem of authorization, that is, determining on whose behalf a given process is currently working is a fairly simple matter since each process is always working for a single principal only. When a process invokes an operation on a Type Manager, the information regarding its UID and principal association is transported onto the virtual machine of the target Type Manager. In this way, the principal which owns a particular process is always known by any Type Manager on which it makes invocation requests. In addition, since process identifiers are transported to and from Type Manager machines by system code, a process is unable to forge its own principal association to gain access to objects its real principal is not authorized to access.

During login, a user is first asked to identify himself by giving his unique principal symbolic name. The login process (also called the Authentication Manager) tries to find a principal object containing the same symbolic name. The principal object contains all the pertinent information about that user. The user's password is stored with the principal object, allowing the Authentication Manager to perform necessary authentication checks. Two other pieces of information regarding the user are maintained within the principal data object. One is the unique identifier (UID) of the user's symbolic name context, which is described in the next section. The other is the UID of the command interpreter or shell program of the logged-in principal.

Since the authentication manager must find a principal object given only its symbolic name, it follows that this name must be unique. In order to make it convenient for unique names to be assigned to principals, Zeus has the concept of a working group (WG). Working groups are used to form a strict hierarchy of principal names. This hierarchy of names is similar to that used in the Multics system. They contain members which may be either principals or other working groups. The root working group has a null name and is called the null working group. The unique name of a principal or working group is formed by concatenating the name of the principal or WG with the names of all of its containing working groups. This hierarchical structure also forms the basis for other symbolic names in the system.

5.2.2.4 Symbolic Name Manager

To provide user convenience, an object can be given a symbolic name that is used when referencing that object. A user in the system should be able to use symbolic names within its context independent of other users. For example, the same symbolic name can be used by different users to refer to different objects. Similarly, different symbolic names can be used by different users to refer to the same object. The Symbolic Name Manager maintains the mapping between a symbolic name for an object and that object's UID. The mapping function is many-to-one in that several symbolic names may

ZEUS DISTRIBUTED OPERATING SYSTEM

be mapped to one object UID. The symbolic names within a context must be unique.

5.2.2.4.1 Symbolic Name Contexts

The objects which are managed by the Symbolic Name Manager are symbolic name contexts, where a context object contains the above mentioned mapping. A context may be viewed as a private directory of relative symbolic names. Each principal is given a context when the principal is created. It is initialized with the symbolic name to object UID mappings of certain system objects which a principal must know in order to function properly. The Symbolic Name Manager maintains a data base that contains the context objects and the current state of the context operations. In the event of a failure the data base provides the recovery of the state of the Symbolic Name Manager and the recovery of the context objects. This type manager is solely responsible for the creation, deletion, modification, and management of instances of the symbolic name context type. The structure of these objects are not known outside the symbolic name manager (SNM), storage is not allocated for them outside the SNM, and their values may not be assigned or checked for equality by any process other than the SNM. In short, all access to symbolic name contexts is completely controlled by the SNM.

Symbolic name contexts are the only supplied means within Zeus for a user to define and use symbolic (non-numeric) names for other system objects. Fundamentally, a context is a single-valued functional mapping from user-supplied symbolic names to system defined unique identifiers (UIDs). As such, the context plays a very important role for the user/principal since the system itself deals only with the bit string UID which is decidedly non-mnemonic but, nonetheless, efficient as a system name.

Each principal which has permission to log-on to Zeus is associated with at least one context object which contains his own private names for objects with which he may interact. This arrangement allows a relative symbolic name space for principals in the sense that different users will, in general, have different symbolic names for identical system objects (that is, objects with the same UID). Such a relative scheme is very efficient in a distributed environment such as Zeus since names must remain unique only within a single context. This eliminates the need for a central (and, therefore, vulnerable) naming authority or a complicated hierarchical scheme as in other distributed systems.

5.2.2.4.2 Context Sharing

The usefulness of context sharing is exemplified by considering the case in which one process wishes to establish one-to-one communication with another process. In order to do this, each process must know the process id of the other process. This "exchange" of process ids is possible through context sharing.

The primary shortcoming of this relative naming scheme is that it makes it difficult for two or more principals to become aware of a common shared object by passing its symbolic name. The principals are likely to have

different names for the shared object and thus will be unable to find "common ground" on which to agree on a single name. Contexts, like other objects, may be shared among principals having the proper access and are part of the mechanism by which principals may share objects. If principal A wishes to share object X with principal B, A must give B access rights to X and must also give B the UID of X. When A shares its context with B, B is able to obtain the UID of X through an agreed upon symbolic name for X. It is important to note that sharing a context in no way enhances or alters the access rights to any of the objects whose UIDs are in the shared context. Access to an object is still coordinated by its associated Type Manager.

Context objects may also be used to implement the absolute naming scheme for principal objects. This is done by providing a single additional context object which contains the name => UID mappings for all the principals currently authorized to use the system. The symbolic name of this context (contexts may have symbolic names just like any other object) should be well known throughout the system. This can be accomplished by initializing each of the principal's contexts with the symbolic name of the unique user name context.

5.2.2.4.3 Reliability Issues

Due to the essential nature of a principal's symbolic name context, it will be desirable to provide the capability to define highly reliable contexts which are likely to survive or perhaps continue to be available in the presence of certain types of errors. The standard mechanism in Zeus for providing such reliable objects is object replication. Context objects are the first significant example of object replication which has been encountered in the detailed design.

Despite the requirement to provide distributed, replicated context objects in Zeus, the consistency requirements of the operations which will be defined for contexts are relatively simple. Straightforward read/write consistency control will be used in the SNM since the five operations defined therein appear semantically identical to four write operations and one read operation. In general, of course, some object types may have more complicated semantics for their operations such that two or more update operations turn out to be compatible since they modify mutually exclusive parts of an object instance. This, however, is not the case with symbolic name contexts.

5.2.2.4.4 Users' View of the SNM

For user processes in the Zeus system, the SNM (and in fact all other type managers as well) appears as a single autonomous server process which defines an abstract data object type and a set of operations which access and/or modify instances of that type. The actual structure of the defined data type, as previously mentioned, is not known by the user process. In addition, the physical host boundaries (indeed most physical characteristics of the system) are hidden from the user process so that the physical network of hosts appears as a logical network of connected user and type manager processes.

ZEUS DISTRIBUTED OPERATING SYSTEM

The user-visible features of a type manager are completely defined by the set of operations which it provides. The symbolic name type manager provides five such operations on context objects; CREATE context, DELETE context, ADD name, REMOVE name, and LOOKUP name. The operations have the obvious semantics. CREATE and DELETE operate on whole contexts while ADD, REMOVE, and LOOKUP modify and access the individual name/UID pairs in an existing context object. The LOOKUP operation is the only read-only operation while the others all cause at least part of the context object to be modified in some way. The reason that ADD and REMOVE are considered to be modifying the entire context object rather than only a single entry in it is that the most likely implementation of context is as a hash table. This means that any modification of the pointer structure within a hash container to insert or remove a name/UID pair would cause the pointers to become temporarily inconsistent thus requiring that the entire table be made unavailable.

The following are operations that affect the whole of some context object.

```
create ()
    --> (context-id, return-code)
    Creates a context object with the denoted access
    list. The possible return-codes are (1) true and
    (2) false.
```

```
delete (context-id : in out context)
    --> (return-code)
    Deletes a context object. The possible return-
    codes are (1) deleted and (2) non-existent
    context.
```

The three operations, ADD, REMOVE, and LOOKUP are provided to maintain the symbolic names to object UID mapping. Of these three, only the LOOKUP operation is read-only while the other two cause a name to UID mapping within the context to be modified.

```
add (name : in symbolic-name;
     object-id : in UID;
     context-id : in context)
    --> (return-code)
    Create a symbolic-name / object UID entry in the
    given context. The possible return codes are (1)
    successful and (2) non-existent context.
```

```
lookup (name : in symbolic-name;
        context-id : in context;
        name-id : out UID)
    --> (return-code)
    Find the symbolic-name in the given context and
    return the object UID associated with that name.
    The possible return codes are (1) successful, (2)
    non-existent context and (3) name is not found.
```

```
remove (name : in symbolic-name;
        context-id : in context)
```

--> (return-code)
Delete the symbolic-name / object UID entry from
the specified context. The possible return-codes
are (1) successful and (2) non-existent context.

5.2.2.5 The Program Type Manager

The Program Type Manager is the repository of both program text and object code. Program text is defined to be a text object that compiles correctly; thus, the creation of a program object requires the user to supply the Program Manager with a correct program or a separately compilable unit of a program. The Program Type Manager, in addition to its function as a repository, acts as a builder of programs; thus, a user can call upon the program Type Manager to build a new program from some specified components. This linking function of the Program Type Manager is useful to the system to build new user types. A program object is defined to be a collection of versions of a single program. The criteria for retaining program versions in the system are defined by the users.

5.2.2.6 Message Type Manager

Communication among application processes is facilitated by message objects. Message objects are created by the sending process and are accessible only by the receiving processes. Messages in Zeus are typed objects. They can be send or received either synchronously or asynchronously. The Message Type Manager (MTM) provides messages as a means of inter-process communication in either a synchronous or an asynchronous fashion. A message that is transferred from one process to another is viewed as an object upon which operations are performed to effect this inter- process communication. The operations are send_msg, receive_msg and msg_status. The send operation creates a message object and initiates the transfer of the object from the sender to the intended receiver. The receive operation completes the transfer when the object's message content is returned to the receiver. The sender or receiver can determine the status of a message object by performing a msg_status operation.

5.2.2.6.1 Reliability of Message Objects

At the time a message is created, the sender can specify the reliability class for that message. The reliability class of a message reflects its availability to the receiver in the face of one or more host failures in the network. At the low end of reliability there are volatile message objects that disappear upon host failure (if the object resides on the failed host). At the high end of reliability stable message objects have a replication factor of n where n is the number of hosts in the network. The four reliability classes are volatile, non- volatile, resilient and stable.

5.2.2.6.2 Scope of Inter-process Communication

Inter-process message communication may occur between processes that are local to a host, or remote. In either case the send and receive operations are performed on the MTM local to the host of the calling process. Any remote communication that might be performed to effect the respective operation is

ZEUS DISTRIBUTED OPERATING SYSTEM

carried out between MTMs and is unseen by the caller. Figures 5-6 and 5-7 depict the flow of information in local and remote inter-process communication respectively.

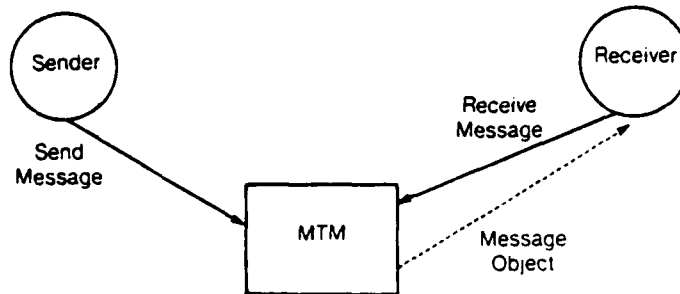


Figure 5-6 Local Message Communication

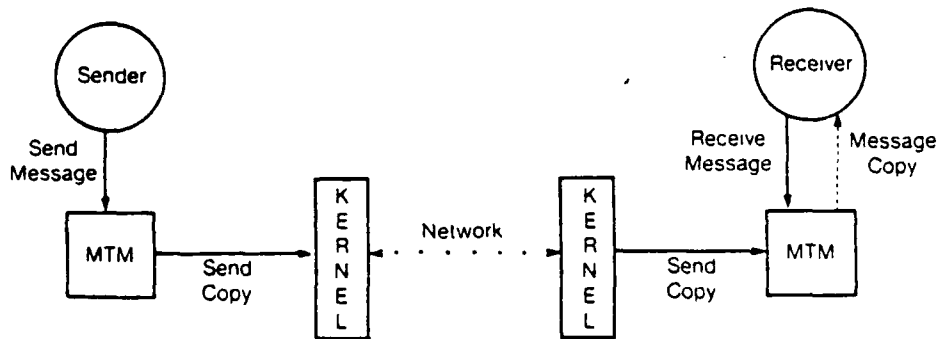


Figure 5-7 Remote Message Communication

5.2.2.6.3 Message Operations from the User's Viewpoint

The messages are sent by invoking a `send_msg` operation with the specification of the message content, the list of receivers, the reliability class of the message, and whether or not the message operation is to be performed synchronously or asynchronously. If the send is a synchronous send, the sender is delayed the shorter of the sender timeout value or the time it takes to route all of the copies of the message to any remote hosts. The routing is a function of the reliability class and the receiver list. When a send is asynchronous, the sender is blocked only for the time it takes to create the message and notify the receivers of its existence. As a result of the send operation, the message unique identifier is returned. If the message

status is determined at a later point by the sender, it is this unique identifier that is passed as a part of the `msg_status` call.

In order to receive a message, the receiver invokes a `receive_msg` operation and specifies which processes to receive a message from, and whether or not the operation is asynchronous or synchronous. This simply means no wait if there is no message available in the asynchronous case and a wait in the synchronous case until a message is available or until a timeout occurs. An additional parameter allows the user to further qualify which message is received. The qualification may indicate that either the most recent message be received, or the oldest, or the first since a host failure.

The `msg_status` operation returns to the caller the current status of the message relative to its receipt by each of the intended receivers of the message. Possible statuses are `received`, `not_received`, `unavailable` and `message non-existent`.

5.3 DESIGN OF RELIABLE TRANSACTION MANAGEMENT

This section describes the design rationale, architecture and functions of the process and transaction management system in Zeus. This section focusses on these aspects of the Zeus Process Manager design: maintenance of databases for reliable process (transaction) management, commit protocols for process managers and object managers, protocols for site crash recovery and process rollback, and the commitment of nested transactions.

5.3.1 Functions for Reliable Application Systems

Reliable applications are built in Zeus by manipulating objects using transactions. The Zeus kernel offers only unreliable remote procedure calls [NELS81] [LAMP81b] which are made reliable by invoking them within a transaction. The transaction facility also provides a powerful mechanism for managing replicated or partitioned objects reliably. This section presents the computational model for managing processes and transactions, and the application visible operations. These operations are summarized in Table 5-1.

Processes are active objects that perform state changes on behalf of system users by modifying shared permanent objects. They have a (system defined) type, `PROCESS`, and are managed by an object manager called the Process Manager. Processes are created and deleted using the `CREATE_PROCESS` and `DELETE_PROCESS` functions, respectively.

Transactions are `PROCESS` objects with the additional property of atomicity. Atomicity, or the "all or nothing" property, means that either all or none of a transaction's updates become permanent. The `TRANSACTION` type is derived from the `PROCESS` type; thus, all operations defined on processes are

ZEUS DISTRIBUTED OPERATING SYSTEM

Table 5-1: Application Visible Operations

OPERATION	REMARKS
INVOKE	input parameters: UID(1) of object to which operation is applied, operation name and operation parameters
CREATE_PROCESS	if successful, returns UID of the new process, otherwise, returns error signal
DELETE_PROCESS	input parameter: UID of process to be deleted
BEGIN_TRANSACTION	if successful, returns UID of the new sequential transaction; otherwise, returns error signal
CREATE_TRANSACTION	if successful, returns UID of the new concurrent transaction; otherwise, returns error signal
END_TRANSACTION	initiates commit protocol between Process Manager and Object Managers
WAIT	input parameters: transaction UID(s) on which parent, waits, optional timeout value
COMMIT	invoked by processes only; input parameters: transaction UID
ABORT	cancels all of a transaction's updates
ESTABLISH_RECOVERY_POINT	returns recovery point number
DISCARD_RECOVERY_POINT	input parameter: recovery point number
ROLLBACK	input parameter: recovery point number; without parameter, process rolls back to most recent recovery point

applicable to transactions. Additional operations are defined for transaction objects. Shared objects can be updated reliably only by transactions.

The Zeus design uses the transaction concept for reliability and to avoid the domino effect during process rollback by enforcing disciplined interactions among processes. First, all information-flow among processes which affects global state takes place via shared objects. Second, all shared global objects must be accessed within a transaction. A transaction defines a "sphere of control" (SOC) [DAVI73]; all objects modified by a transaction are

-
- (1) A UID is a globally Unique Identifier; every process, transaction and object in the system has a UID.

said to belong to its sphere of control. Third, no other process/transaction is allowed to access objects belonging to a transaction's sphere of control during that transaction's execution. If the transaction completes successfully, the updated objects are "committed"; otherwise, they are restored to their state before the transaction began execution.

A process can create sequential and concurrent transactions. The parent process of a sequential transaction is suspended until that transaction terminates. However, the parent process of a concurrent transaction process executes concurrently with its child. When a concurrent transaction process terminates, an appropriate condition is signalled to the parent process.

The Zeus design allows a transaction to invoke other (sequential and concurrent) transactions, called nested transactions [MOSS81] [RIES82]. A top-level transaction is one whose parent is a non-transaction type process. Zeus supports nested transactions (i) to introduce concurrency into an atomic action, and (ii) to allow a transaction to invoke procedures which may contain transactions. Nested transactions also provide a means for constructing recovery blocks [HORN74], and updating replicated objects using majority consensus [THOM79] or weighted voting [GIFF79].

To create a sequential transaction, a parent process or transaction invokes the BEGIN_TRANSACTION function. The parent process is then suspended until its child terminates. The sequential transaction created by BEGIN_TRANSACTION inherits its parent's address space and runtime environment. The transaction terminates by executing either END_TRANSACTION or ABORT. Invoking END_TRANSACTION causes the Process Manager to execute commit protocols with the object managers of the objects accessed by the transaction. The code between a BEGIN_TRANSACTION and a corresponding END_TRANSACTION is executed as an atomic action, that is, as a transaction.

A process or transaction creates a concurrent transaction by invoking CREATE_TRANSACTION. When a concurrent transaction completes, a condition is signalled to its parent. At this point the child transaction is still not committed; it is either in the aborted or the commit-pending state. The commit-pending state indicates that the transaction was successful and is waiting for its parent to issue a commit command. If the parent is a non-transaction process, then it explicitly issues the COMMIT command. Nested transactions are implicitly committed when the top-level transaction commits.

A parent process or transaction can wait for a completion signal from a concurrent child transaction by invoking the WAIT function. WAIT can include a time-out option will cause the invoker to be suspended until either the transaction completes or an interval of time passes. A process may wait on any of several transactions, or until each of a set of transactions has completed.

Processes and transactions perform operations on shared global objects using the INVOKE function. Remote and local shared global objects are accessed identically.

A top-level transaction may make a commit decision based on the status of its nested transactions (e.g., completed or aborted). It is undesirable to

ZEUS DISTRIBUTED OPERATING SYSTEM

require a top-level transaction to revalidate the state of the objects accessed by a completed nested transaction if the top-level transaction decides to commit. Revalidation of object states can be avoided if a nested transaction follows an appropriate commit protocol. A nested transaction can follow either a one-phase or two-phase commit protocol [BALT81] with its parent. Using a one-phase commit protocol means that, when a nested transaction completes, all the objects it modified are in the commit-pending state. The commit-pending versions cannot be aborted unilaterally by an object manager. In contrast, following a two-phase commit protocol leaves modified objects in the uncommitted state. Such uncommitted versions can be aborted unilaterally by their object managers, thereby aborting that nested transaction.

Zeus uses the one-phase commit option for nested transactions. This allows the use, within a transaction, of a conditional statement that depends on the successful completion of one of the transaction's nested children. Such conditionals may be used because the one-phase commit option prevents a unilateral abort by an object manager from invalidating conditional decisions made by the parent transaction. It also eliminates the need for a parent transaction to revalidate the status of completed nested transactions.

Object managers and process managers follow a two-phase locking protocol [ESWA76] so that all concurrently executing transactions are serializable. All concurrently executing nested transaction with the same parent are also serializable according to these rules.

A process or transaction may establish a recovery point by invoking `ESTABLISH_RECOVERY_POINT` (ERP). When ERP is invoked, the Process Manager saves the current state of the process on stable storage and returns a recovery point number to the calling process. A process can explicitly roll back to some previous recovery point by invoking the `ROLLBACK` function. If no parameters are given, the calling process rolls back to its last recovery point. If a recovery point number is supplied, the process rolls back to that recovery point.

It is possible to establish a recovery point for a parent process when a sequential transaction commits, by using the ERP option with the `END_TRANSACTION` command. If the parent process subsequently crashes, it would be started either from this recovery point or from a subsequent recovery point, avoiding re-execution of a transaction that has already been committed.

The Process Manager establishes the initial state of every process as the recovery point numbered 0. All subsequent calls to ERP return sequentially increasing integer numbers. When a process completes, all of its recovery points are discarded. A process can also discard any of its recovery points by invoking `DISCARD_RECOVERY_POINT` (DRP), with the number of the recovery points to be discarded.

Table 5-2: Rules for Rollback and Checkpointing

- (1) When a non-transaction process rolls back, no committed transaction is undone.
- (2) Within a transaction, a rollback causes abortion of any nested transactions that were created after the recovery point was established, including successfully completed nested transactions. Transactions outside the sphere of control (SOC) of the transaction that is being rolled back are not aborted, because no information has yet passed out of that sphere of control.
- (3) When a process invokes ERP, its recovery point is established only when all of its concurrent child transactions have completed.
- (4) A transaction can refer only to its own recovery points.
- (5) When a transaction completes, all of its recovery points are discarded.

Table 5-2 shows the rules that determine the effect of rollback and checkpoint operations. Rule 1 prevents a domino effect during rollback. Rule 3 avoids the domino effect within the invoker's SOC. Figure 5-8 shows how a domino effect can occur without Rule 3. The figure shows a concurrent transaction, T1, being created before establishing recovery point 1. When this recovery point is established, transaction T1 is still executing. After establishing this recovery point, the parent process, T, creates another transaction, T2. For a while T1 and T2 execute concurrently; T2 completes and commits before T1 completes. After T2 commits, some of the objects it modified are accessed by T1. Later, T1 commits. If the parent transaction, T, rolls back to recovery point 1, transaction T2 is undone (aborted). Because some information flowed from T2 to T1, abortion of T2 causes the abortion of T1. This can result in a cascade of aborts.

5.3.2 Design of Reliable Transaction Management

Application processes at a Zeus host are managed by the Process Manager, a system-defined object manager. Application visible functions are implemented by the Process Manager and the object managers of the objects accessed by the application processes. This section describes the system architecture and the interactions among the Process Manager and the object managers. Figure 5-9 shows the relationship among application processes, object managers and the Process Manager (PM). Section 5.3.2.1 presents the system architecture and the structures of the Process Manager and a generic object manager. Section 5.3.2.2 explains transaction management.

ZEUS DISTRIBUTED OPERATING SYSTEM

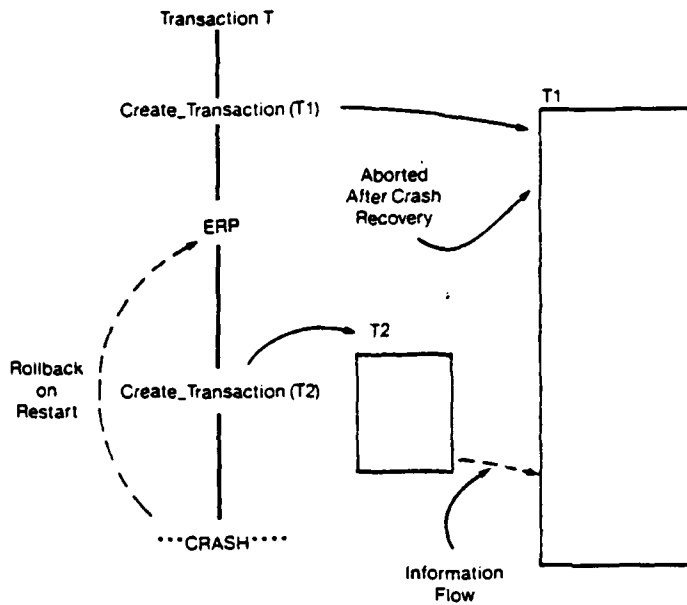


Figure 5-8

Domino Effect with Nested Transactions and Checkpoint

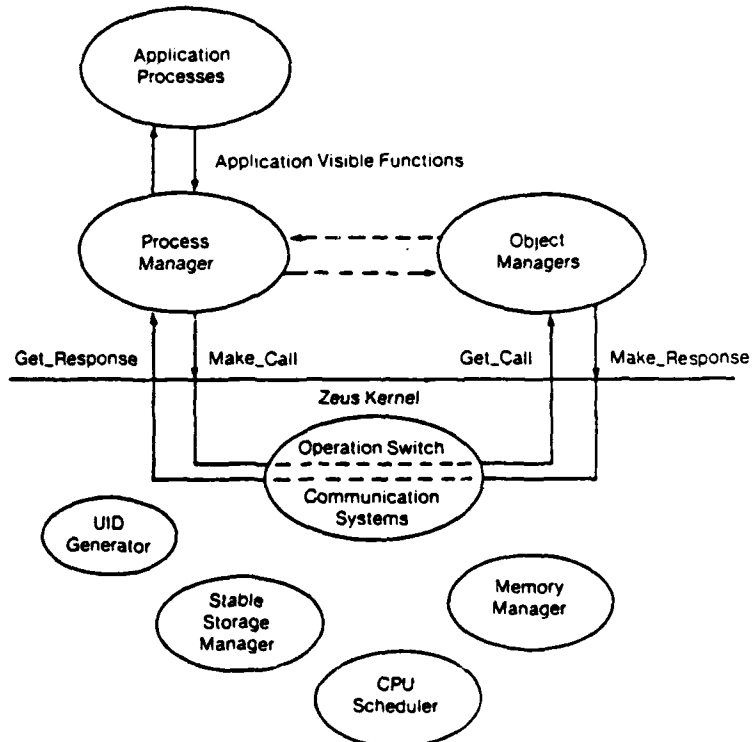


Figure 5-9

Overview of the System Architecture

5.3.2.1 Overview of the Architecture

The lowest level of the operating system at each host is called the kernel. It supports interprocessor communication, management of primary and secondary storage, CPU scheduling and unique identifier generation. Secondary storage management provides a stable storage as defined by Lampson [LAMP82a]. An object manager communicates with another object manager by calling the kernel's procedure invocation functions. A client process makes a call by invoking the kernel function MAKE_CALL, with the unique name of the object it wants to access. The system directs the call to the appropriate object manager which receives it by invoking the kernel function GET_CALL. After performing the requested operation, the object manager responds to the client by invoking MAKE_RESPONSE, and the client receives the response by invoking GET_RESPONSE.

As shown in Figure 5-9, application processes access objects with the INVOKE function. A Process Manager executes the INVOKE function by calling the kernel functions MAKE_CALL and GET_RESPONSE. Section 5.3.2.1.2 explains the actions performed by the Process Manager in response to the INVOKE call.

5.3.2.1.1 Process Manager (PM) Structure

A Process Manager provides the application visible functions which create, manipulate and destroy processes and transactions. In order to provide these functions, it maintains a database (PMDB) for processes and transactions at its node consisting of the following tables:

Active_Process_List: This list contains a record for each active process/transaction. A record holds: a unique process/transaction name, the last recovery point number, and information about priority and access rights. The commitment status of transactions is also kept here.

Parent_Child_Info_List: This database consists of two lists: the Descendent Transaction List (DTL) and the Accessed Object List (AOL). For each active process/transaction, the DTL contains the UID of its parent and a list of its children processes and transactions. The Accessed Object List stores the UIDs of objects a transaction accesses. When a transaction calls INVOKE, PM inserts the UID of the object to be accessed in AOL if that UID is not already present. If an object is being accessed for the first time by a transaction, the request message to the object manager is marked as an initial-access message. On the invocation of END_TRANSACTION by a process, PM uses AOL to determine which object managers are to be involved in a commit protocol.

Current_Operation_Table (COT): This table plays a crucial role in site recovery from crashes (Section 4.1). When a process invokes some critical operation such as ROLLBACK or END_TRANSACTION, the occurrence of the operation is stored in COT and forced onto stable storage. In general, the crash-recovery procedure attempts to start every crashed process from its last recovery point. However, if a process appears in COT, the recovery procedure restarts the process not from its last recovery point, but from the operation invoked by the process at the time of crash.

ZEUS DISTRIBUTED OPERATING SYSTEM

These databases are kept in both primary and secondary storage; they are represented in secondary storage by a differential file [SEVE76] and a main copy. Updates to the primary memory databases are recorded in a buffer called PMDB_Log_Buffer which resides in primary memory. When the PM explicitly invokes a synchronous write (FORCE) operation on the database or when the buffer gets full, its contents are appended to the differential file in secondary storage and deleted from primary memory. Periodically and during the site restart from a crash, the contents of the differential file and the main copy are merged.

5.3.2.1.2 Object Manager Structure

An object manager performs operations on objects in response to operation invocation messages from client processes and transactions. The object manager is responsible for enforcing concurrency control rules on requests to preserve object consistency. It enforces a two-phase locking protocol [ESWA76] and executes commit protocols with the PM to ensure the transaction atomicity.

When an object manager receives an initial access message, it attempts to lock the object in the mode required by the operation. Only read mode locks may be shared by transactions. If the object can not be locked, the request is queued. It is possible to follow a deadlock prevention scheme such as wound-wait or wait-die [ROSE78] at this point. The rules for granting locks to clients are defined in Table 5-3. In Section 5.3.2.2.2, these rules are modified for nested transactions.

Table 5-3: Lock Compatibility

Lock Requested	Lock Held			
	none	read-r	read-o	write-o
read	grant	-	grant	queue
write	grant	grant*	queue	queue

read-r: lock held in read mode by the requestor

read-o: lock held in read mode by some other transaction

write-o: lock held in write mode by some other transaction

grant*: grant only if no other transaction is holding a lock

For each object, the object manager maintains an object header, which records the object's UID, current state, version number, timestamp, UIDs of the transaction(s) currently holding a lock on the object, the lock mode of

current access, and the complete context of every transaction that is currently accessing the object. The context comprises the UIDs of all transactions on the path starting with the top-level transaction and ending with transaction currently accessing the object. This list is called the Transaction Context List. The object may be in uncommitted, commit-pending, committed, or aborted states. These states have the usual meaning defined in a commit protocol.

When a lock is granted to a client, the object manager schedules a server process for the client request. For update operations, the server creates a new uncommitted version of the object and forces the version onto stable storage before sending a response to the client. This version is committed if and only if the client commits. Update locks on the object are released at the time of commitment. If the client is a non-transaction process, updated versions are committed immediately, i.e., at the time they are forced onto stable storage. In this case sending response and writing the new version onto stable storage can be done asynchronously.

A timestamp is attached to each new version; it records the timestamp of the invocation message that was attached by the client's PM. The timestamp is used during transaction rollback to determine which versions of the object were created by the transaction after establishing the recovery point. The object manager discards these versions, thus ensuring object consistency. The rollback protocol is defined in Section 5.3.2.2.1.

The object manager maintains an inactivity timer for uncommitted and commit-pending versions of objects. If a time-out condition occurs, the object manager may abort an uncommitted version. However, an object in the commit-pending state, may not be aborted unilaterally by its object manager. It is either committed or aborted depending on the commit/abort decision received from the client transaction's PM.

5.3.2.2 Transaction Management Design

Section 5.3.2.2.1 describes the management of top-level transactions that do not contain nested transactions; Section 5.3.2.2.2 introduces the extensions required for managing nested transactions. Both sections describe the actions a PM executes when a process invokes the transaction management functions described in Table 5-1.

5.3.2.2.1 Top-Level Transactions

BEGIN TRANSACTION and CREATE TRANSACTION

In response to the BEGIN_TRANSACTION request, the PM creates a new Process Control Block (PCB) and assigns a new UID to this block. The Process Manager's database is modified by: (1) inserting a record for the new transaction in both the Active_Process_List and the Parent_Child_Info_List, (2) marking the parent process's status as suspended, (3) saving all local objects in the parent's address space as part of a checkpoint, and (4) recording the checkpoint identifier in the parent's database. These changes are written as one record in the PMDB_Log_Buffer. Writing these changes as one record ensures that all or none of them are forced onto secondary storage;

ZEUS DISTRIBUTED OPERATING SYSTEM

therefore, all or none of them are visible after a site crash and recovery. At this point, the new transaction is given to the CPU scheduler for execution.

On the invocation of `CREATE_TRANSACTION`, if the request is for transaction creation at a remote node, the PM at the invoker's node uses `INVOKE` to send a request to the PM at the remote node for creating a transaction. If the operation is successful, the remote PM returns the UID of the newly created transaction. The local PM updates the invoker's Descendent Transaction List, and the invoker is resumed. If the requested transaction is to be created locally, the PM follows the steps as described for `BEGIN_TRANSACTION` except that in the last step both the parent and the child are made active by queueing them in the CPU scheduler's queue. In this case the child transaction does not inherit parent's address space and the run-time environment.

INVOKE

A process invokes operations on remote or local objects identically with the `INVOKE` function. If the invoker is a transaction and the object to be accessed is not in the invoker's AOL, the PM inserts the UID of the object in this list, and marks the invocation as an initial request. The PM invokes the kernel function `MAKE_CALL` to send it, attaches a timestamp to the request, and sets a timer for timing out the response. Responses are received by the PM using the kernel function `GET_RESPONSE`.

END_TRANSACTION

To commit a top-level transaction that does not contain any nested transaction, the PM acts as a coordinator for a two-phase commit protocol with the object managers in the transaction's AOL. Before starting the commit protocol execution, PM records the UID of the invoker in the Current Operation Table indicating that the process is executing `END_TRANSACTION`. This information is cleared when the commit/abort decision is made and written into the database. It then sends `PREPARE` messages to all these object managers and waits to receive responses. If all responses are `READY`, then it commits the transaction; otherwise, it aborts the transaction. The protocol and related actions are described in Table 5-4. The protocol for committing a top-level concurrent transaction is only slightly different than that for a top-level sequential transaction. For a top-level concurrent transaction, when `READY` messages are received from all object managers, the transaction enters the `COMMIT_PENDING` state instead of the `COMMITTED` state. The transaction enters the `COMMITTED` state only after receiving a commit command from its parent.

We follow the Presumed-Abort protocol [MOHA83], which means that a PM sends an `ABORT` response to an object manager from which it receives a status query about a transaction for which it has no information. Because of this protocol, the PM can safely delete all information about an aborted transaction from its database without waiting for acknowledgements of the abort from the object managers.

Table 5-4: Commit Protocol for Top-Level Transactions (No Nested Trans.)

Process/Transaction Manager(PTM)		Object Manager(OM)
1. Send PREPARE messages to all accessed objects	----->	Receive PREPARE;
		[] Uncommitted version aborted --> Forget transaction; Send ABORT
2. Receive responses;	<-----	
[] Any response ABORT --> Send ABORT to all OMs that voted already; Clear PTM database		[] Uncommitted version not aborted --> Force write status=COMM-PEND; Send READY
[] All responses READY --> [] Concurrent transaction --> Force write status=COMM-PEND; [] Sequential transaction --> Force write status=COMMITTED Send COMMIT to all objects;		In COMM-PEND state of the objects:
	----->	[] Receive ABORT --> Discard COMM-PEND version; Restore previous committed version of the object; Forget transaction;
3. [] Sequential transaction --> resume the parent (restore local variables if ABORT)		[] Receive COMMIT --> Force write status=COMMITTED; Send ACK;
[] Concurrent transaction --> signal DONE to parent;	<-----	Forget transaction;
4. In the COMM-PEND state of the transaction: Receive COMMIT --> Force write status=COMMITTED; Send COMMIT messages;	<-----	[] Timeout --> Send Status Query
	----->	
5. In COMMITTED state of the transaction:		
[] Receive ACK --> If all ACK received, Clear PTM database;		
[] Receive Status Query --> Send COMMIT	----->	
[] Timeout --> Send COMMIT to all accessed objects;		
6. In case of ABORT: Receive Status Query --> Send ABORT	----->	

ZEUS DISTRIBUTED OPERATING SYSTEM

Checkpointing and Rollback

In Zeus, process/transaction checkpointing is confined only to the PM, whereas rollback involves coordination with object managers. This makes checkpointing cheaper than rollback, which is desirable because a system executes checkpointing more frequently than rollback.

When a process or transaction invokes the ERP (ESTABLISH_RECOVERY_POINT) function, the PM stores its recovery point data on secondary storage and gives this recovery point an integer number identifier. The PM database is modified to include this recovery point number as the last recovery point for the process, and the PMDB_Log_Buffer is forced onto secondary storage. A timestamp based on PM's local clock is recorded with each recovery point which indicates the time it was established.

When a process (or transaction) invokes ROLLBACK, the PM records the process's UID in its Current Operation Table before starting the rollback operation, indicating that this process is currently being rolled back. This information is forced onto stable storage. Thus, if a crash occurs during the rollback, the recovery procedure attempts to restart this process not from its last recovery point, but from some well-defined point in the rollback procedure.

To rollback a transaction, the PM first restores the transaction state from the checkpoint on the secondary storage. This checkpoint is not necessarily the last recovery point established by the process. The process is then suspended while the PM sends a rollback message containing the timestamp of the recovery point to every updated object and waits for a positive acknowledgement. The receiving object managers discard all those object versions which were created by the transaction to be rolled back after establishing the recovery point. The rollback procedure is complete only when a positive acknowledgement is received from every object manager. If a timeout occurs, the PM re-sends the ROLLBACK message to all those object managers that have not yet responded.

Upon receiving a rollback message, an object manager discards all those versions of the object that were created by the transaction and which have timestamp greater than the timestamp attached to the rollback message. The modified object header is forced onto stable storage, and then sends an acknowledgement to the PM. If no versions are to be discarded, still an acknowledgement is sent to the PM.

5.3.2.2.2 Nested Transactions

This section describes the features in the Zeus design that support nested transactions. Modifications are required to the locking protocol, commit protocol, and the PM database contents. The commit protocols must ensure that a nested transaction is committed only when its top level transaction commits. The locking rules must ensure that all of a parent transaction's descendents are serializable within the parent's sphere of control.

Based on the parent child relationship, a collection of nested transactions that cooperate to perform a computation forms a transaction tree. A transaction has one of three distinct roles within a transaction tree -- top level (root node), intermediate, and leaf. A top level transaction coordinates the commitment of updates to objects made by its nested transactions. An intermediate transaction provides a program structuring mechanism. For simplicity, we require that all objects are accessed by leaf transactions. This rule can be relaxed if the appropriate checks are made.

Initiation of Nested Transactions

A sequential nested transaction is created by invoking BEGIN_TRANSACTION. The actions performed by the PM are basically the same as described earlier. A concurrent, nested transaction is created when a transaction invokes CREATE_TRANSACTION. The action executed by the PM are the same as those for creating a top-level concurrent transaction. In both these cases the Parent_Child_Info List is updated for both the parent and child transaction in their respective PM's databases.

Accessing Objects by Nested Transactions

Nested transactions cause a minor change in how a PM handles an INVOKE operation and a significant change in how an object manager grants locks for objects. When an object manager is accessed by a transaction, the PM adds a Transaction Context List to the request. It lists the UIDs of the transactions on the transaction tree path from the top level transaction to the leaf transaction. The PM maintains an accessed object list for a leaf transaction in the same way that it does for a top level transaction.

The object manager locking rules are complicated by the fact that an object may be accessed by multiple leaf transactions within one transaction tree. The locking rules ensure the serializability of nested transactions. Since all transactions in a transaction tree must be eligible at some point to be granted locks to objects accessed by other transactions within that tree, the locks granted to a leaf transaction must be released when it executes END_TRANSACTION.

The question arises which transactions an object manager may consider as candidates for access to an object. Since a transaction tree is a unit of consistency on which commit or abort may be executed, none of the objects updated by a transaction tree can be accessed by a transaction that is not a member of the tree until the top level transaction has terminated. Similarly, none of the objects read by a transaction in a transaction tree can be updated by a transaction that is not a member of the tree until the top level transaction has terminated.

ZEUS DISTRIBUTED OPERATING SYSTEM

Table 5-5a. Lock Releasing Rules

Lock Released By Child	Lock Previously Held By Parent		
	read	write	none
read	read	write	read
write	write	write	write

Table 5-5b. Lock Granting Rules

Lock Requested	Current Lock Mode							
	none	read-a	write-a	read*	read-r	read-na	write-na	read*-na
read	grant read	grant read	grant read*	grant1 read*	-	grant read	queue request	queue request
write	grant write	grant write	write	grant2 write	grant3 write	queue request	queue request	queue request

Comments:

- read-a: read lock held by ancestor transaction
- write-a: write lock held by ancestor transaction
- read*: write lock held by ancestor has been converted to a read lock for a descendant
- read-r: read lock held by requester
- read-na: read lock held by non-ancestor transaction
- read*-na: write lock held by non-ancestor has been converted to a read lock for descendant
- write-na: write lock held by non-ancestor transaction
- grant1: granted if if no transaction on the path from the current lock holder to the the successor of the least common upper bound of the holders and requester has inherited a write lock
- grant2: grant if grant1 and grant3 met
- grant3: granted if read lock is held by no other transaction

A lock released by a nested transaction is inherited by its parent transaction; the mode of the inherited lock is defined in Table 5-5a. The parent is determined by the Transaction Context List which is attached to an object's header when a lock is granted. An object manager may grant any lock held by a parent transaction to any of its descendants; thus, any subsequently created descendants of the parent are eligible for the lock. The rules for granting locks are defined in Table 5-5b. In order to implement the rules, an object manager maintains a list of lock modes currently held by the transactions on a Transaction Context List. Note that a parent transaction in such a list will have a lock mode of nil until it inherits a lock from one of its descendants.

The granting and regranting of locks within a transaction family enforces the notion of a sphere of control at a finer granularity than that of a top level transaction. However, it also allows for a deadlock between transactions in a transaction tree. Deadlock can be avoided through a technique such as time-outs, and wound-wait or wait-die schemes [ROSE78].

Abortion of Nested Transactions

A nested transaction may be aborted either due to a ROLLBACK or ABORT command. The goal of an abort is to ensure that the objects and transactions are returned to a previously consistent state. This requires the PM and object managers to undo the effects of the nested transaction. PM follows the protocol for a top-level transaction with one change to abort a nested transaction. It includes the list of all nested transactions aborted in the ABORT message. This aids an object manager in determining which versions of an object may be deleted if there is a dependency between the versions and one is in the COMMIT_PENDING state.

To abort a nested transaction, an object manager follows the protocol for aborting a top-level transaction with one change in lock releasing. If the nested transaction had completed, the parent inherits the lock mode it held. The object will eventually be released when the top level transaction commits because the object was added to its Accessed Objects List when the nested transaction completed. If the nested transaction had not completed, the parent will gain the lock mode it had before the nested transaction accessed the object. Note that if no other nested transaction of the parent had accessed the object, the parent's lock mode will be nil and the object will be released completely from the parent.

Commitment of Nested Transactions

A commit protocol is executed when an END_TRANSACTION command is invoked. There are three kinds of commit protocols for nested transactions:

- (1) A leaf transaction commits an object.
- (2) An intermediate transaction commits a child transaction.
- (3) A top level transaction commits a transaction tree.

The combined effect of the above is a two phase commit protocol.

Two data structures are important for committing nested transactions -- a Descendant Transaction List (DTL) and an Accessed Objects List (AOL). (For a leaf transaction the DTL is obviously empty.) When a nested transaction completes successfully, its DTL and AOL are appended to its parent's lists. Ultimately, the list of all successful nested transactions in a transaction tree and the objects involved in those transactions are accumulated in the top level transaction's DTL and AOL, respectively.

A leaf transaction executes the same first phase of the two phase commit protocol as is followed by a top level transaction without nested transactions. In the second phase, the PM of the leaf transaction sends COMPLETED messages (instead of COMMIT messages) to the object managers and signals completion to the parent transaction. A COMPLETED message causes an

ZEUS DISTRIBUTED OPERATING SYSTEM

object manager to transfer the locks held by the leaf transaction to its parent. The completion signal identifies the transaction and contains its AOL. Note that only those objects that have a version in the commit pending state are in the AOL. Steps 1-4 in Table 5-6 describe the above commit protocol.

Based on the responses (ABORT and DONE) from its children transactions, an intermediate transaction makes a decision whether or not to commit. If it decides to commit, a list of its successful children, and the DTLs and AOLs of the successful children are appended to its DTL and AOL. The PM database is updated to indicate these changes and the transaction status as commit pending. This information is forced onto stable storage before signalling completion to its parent. Finally, a COMPLETED message is sent to the object managers of the objects on the AOL and a DONE message is sent to its parent. The messages have the same effect as for a leaf transaction. Steps 1-2 in Table 5-7 describe the commit protocol.

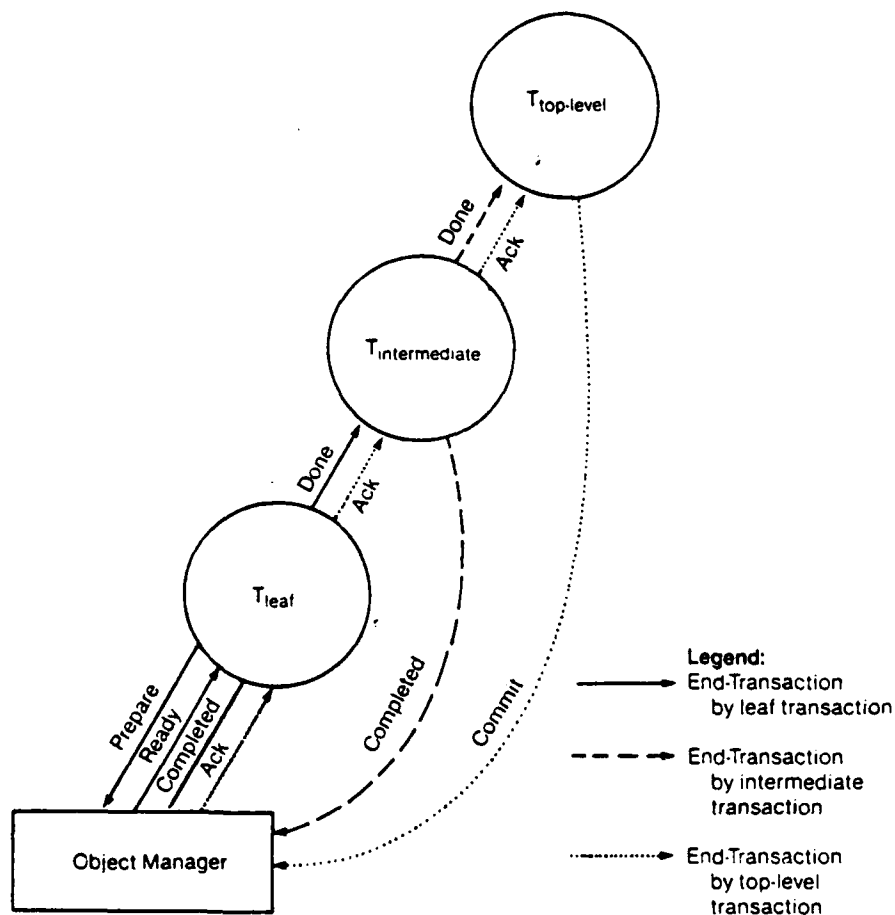


Figure 5-10 Information Flow for Nested Transaction Commit Protocol

Eventually, a top level transaction decides to commit or abort a transaction tree. If it decides to commit, the second phase of the two phase commit protocol that a top-level, non-nested transaction normally follows with an object manager (see Section 5.3.2.2.1) is executed with two modifications.

First, the commit message contains the DTL to handle the case where two or more nested transactions may have updated a common object and the most recent version of the object is invalid because the update was done by a nested transaction that aborted. Second, the acknowledgment will come from a nested transaction instead of from the object manager. The object managers also follow the second phase of the normal commit protocol with the modification that an acknowledgment is directed to a leaf transaction. The acknowledgement is forwarded up the transaction tree from leaf to intermediate to the top level transaction. Table 5-8, step 5 in Table 5-6, and step 3 in Table 5-7 describe END_TRANSACTION for a top-level transaction. Figure 5-10 shows the information flow between the transactions and object managers.

It is possible for a site to crash during the commit protocol, a message to be lost, or a timeout to occur. These events may cause a transaction or object manager to obtain the status of a transaction by querying the Process Manager with the function TRANSACTION_STATUS. This function is invoked with the UID for the transaction as its parameter; the function returns one of the following status: uncommitted, commit-pending, committed, non-existent, and aborted. The non-existent status is interpreted as either aborted or committed depending on the context as explained below.

Status Queries by Object Managers: An object manager may query the status of a transaction if that transaction is currently holding a lock on a commit-pending version of an object. If the transaction status is non-existent, then the transaction is presumed to have aborted. If an object manager, on receiving a COMMIT message from a transaction, does not find any information for that transaction in its database, then it presumes that the versions created by that transaction were successfully committed and sends an acknowledgement. Status Queries by Process Managers: A transaction may query the status of its descendant transactions. If the transaction status is non-existent and the inquiring transaction is in the commit pending state, the transaction is presumed to have committed. If the transaction status is non-existent and the inquiring transaction is not in the commit pending state, the transaction is presumed to have aborted. If a PM, on receiving a COMMIT message for a transaction from its parent transaction, finds no information for the transaction in its database, then it presumes that the transaction was committed successfully and sends an acknowledgement.

ZEUS DISTRIBUTED OPERATING SYSTEM

Table 5-6. Leaf Transaction END TRANSACTION

Process/Transaction Manager	Object Manager
1. send PREPARE messages to all accessed objects.	
2.	receive PREPARE; [] uncommitted version aborted --> send ABORT to PM; give locks that were inherited from parent back to parent; forget transaction; [] uncommitted version okay --> force write COMMIT_PENDING; send READY;
3. receive responses	
[] any response ABORT --> send ABORT to all OMs that sent READY messages; send ABORT to parent; clear PMDB;	
[] all responses READY --> force write COMMIT_PENDING; send COMPLETED to all OMs; send DONE and AOL to parent;	
4.	receive command; [] command = ABORT --> forget transaction; [] command = COMPLETED --> transfer locks to transaction's parent;
5. receive ack (from OM); clear PMDB; send ACK to parent;	

Table 5-7. END TRANSACTION of Intermediate Transaction

Process/Transaction Manager	Object Manager
1. force write COMMIT_PENDING and successful nested transactions; send DONE message with DTL and AOL to parent; send COMPLETED message to OM;	
2.	receive COMPLETED; transfer locks to transaction's parent;
3. [] all ACKS received (from successful children) --> clear PMDB; send ACK to parent;	

Table 5-8. Top Level Transaction END TRANSACTION

Process/Transaction Manager	Object Manager
1. force write COMMIT_PENDING; send COMMIT message with DTL to OM;	
2.	receive command; [] command = ABORT --> send ACK to leaf transaction; forget transaction; [] command = COMMIT --> force write COMMIT; send ACK to leaf transaction; forget transaction;
3. [] all ACKS received (from OM via successful children) --> clear PMDB;	

Checkpointing and Rollback of Nested Transactions

Protocols for checkpointing and rollback of leaf transactions are the same as those described for top-level transactions in Section 5.3.2.2.1. When a top-level or intermediate transaction invokes ERP, a checkpoint is established only when all its descendents have terminated. To rollback a non-leaf transaction, abort messages are sent to all descendent transactions that were created after establishing the recovery point to which the transaction is being rolled back. The rollback operation completes only when all such descendents have been aborted. At this point the rolled back transaction resumes.

5.4 CONSTRUCTING RELIABLE SYSTEMS

This section describes how reliable distributed systems are constructed using the primitives defined in the preceding sections. First, we present the steps required to restart a site after a crash. Next, we explain how the INVOKE operation is combined with the nested transaction facility to build reliable remote procedure calls. Finally, we describe techniques to build a reliable object type from its non-reliable counterpart by object replication and the structure of object managers.

5.4.1 Site Restart From Crashes

During crash recovery, the restart procedure first recreates the PM database from secondary storage. The PM database on secondary consists of two parts, the main copy and the differential file. The recovery procedure merges the changes in the differential file into the main copy. It replaces the old

ZEUS DISTRIBUTED OPERATING SYSTEM

database with the new merged database and installs an empty differential file as an atomic action.

Once the database is recreated, the recovery procedure attempts to restart all processes and transactions whose status is marked as ACTIVE in this database. Processes are restarted from their last recovery point as recorded in the PM database, except for the processes in the Current Operation Table (COT). These are restarted with the invocation of the PM function that they were executing at the time of crash.

5.4.2 Reliable Remote Procedure Calls

As mentioned earlier, operations on remote and local objects are invoked in an identical fashion using the INVOKE function. The problems that can arise in the implementation of remote procedure calls in distributed systems are discussed in [LAMP82b]. In our design, we want to ensure that the remote procedure calls have "at most once" execution semantics in spite of duplicate messages, and server or client site crashes. The following failure conditions which can arise during the invocation of a remote operation can potentially violate the "at most once" operation semantics:

These conditions are:

- (1) Arrival of duplicate messages at the server due to network retransmissions,
- (2) Crash of the client leading to a restart and re-invocation of the remote request,
- (3) Crash and subsequent restart of the server,
- (4) Loss of the response message .

We now present an RPC design that addresses these problems. An operation is invoked reliably within a transaction; reliability is achieved by the techniques described in Sections 4.2.1 through 4.2.4.

5.4.2.1 Duplicate Invocation Messages

Duplication of invocation messages may occur due to retransmissions within the communication network's transport layer. If the server has already acted on a request, then it should be able to detect and ignore a duplicate request. The simplest means to allow detection is to assign a globally unique timestamp or sequence number to every request in the system. A scheme based on this approach is described in [SHRI82]. For every object, its object manager maintains the timestamp of the last request that it has processed, and ignores newly received requests for that object with timestamps smaller than or equal to the timestamp of that last request. However, this simple scheme performs poorly due to the the global synchronization required to generate timestamps. This performance overhead can be lowered significantly by generating the timestamps by roughly synchronized clocks. But there is a

somewhat larger risk that an operation is rejected due to the skew in the clocks, or due to delays caused by the transaction executions. This rejection of operation requests could lead to the abortion of an unacceptably large number of transactions.

In the Zeus design, operation requests are accepted by an object manager only if they arrive in increasing timestamp order; any out-of-order requests are considered duplicates and rejected. So far the scheme is similar to [SHR182], except that the timestamps may be generated using unsynchronized local clocks. However, when a transaction completes, a server may accept a request with a lower timestamp from some other transaction, thus avoiding the unnecessary transaction aborts of the last scheme. The overhead of avoiding unnecessary aborts is that the server must maintain the timestamp of the last request from its last completed client transaction for each object, in order that a retransmitted initial access message for that transaction not look like a new request message to the server.

The question arises about how long the server should maintain information about a completed client transaction. The server maintains a list of those client transactions that completed over the last T units of time. Any request from a transaction appearing in this list is rejected. If clocks are perfectly synchronized, an initial access is accepted if and only if the transaction UID is not in the list mentioned above and

$$(\text{clock} - T_s) \leq T$$

where T_s is the timestamp of the initial access message. If T_{skew} is the maximum skew between any pair of clocks in the system, then the last condition is

$$(\text{clock} - T_s) \leq T - T_{\text{skew}}.$$

5.4.2.2 Client Transaction Site Crash

The crash and restart of a client transaction due to crash recovery of its site can cause retransmission of some uncompleted invocation messages. The rollback protocol executed during a crash recovery (Section 4.1) ensures that the "at most once" operation semantics of remote calls is preserved. The rollback procedure sends a rollback message to all objects accessed since the last recovery point, and waits until all object managers have acknowledged before restarting the transaction. The server process, on receiving such a message, aborts any on-going RPC and aborts object versions created by the INVOKE calls since the last recovery point. If the client never restarts, or restarts after a significantly long period, the server eventually times out and aborts the RPC. This will undo any updates made to the object by the aborted transaction.

5.4.2.3 Server Object Manager Site Crash

Suppose an object manager crashes while it is servicing a call. If it restarts from some recovery point before accepting the call, then the client will never hear from the server and will eventually time-out, thus aborting

ZEUS DISTRIBUTED OPERATING SYSTEM

the transaction. If the object manager restarts from some recovery point after accepting the call, and if the restart occurs shortly after the crash, chances are the client will never sense the server crash. In this case the server restarts after the client has timed-out, the server will complete the RPC and wait for commitment. This new version will ultimately be aborted because the client transaction has aborted on the timeout condition.

If a client transaction in the above situation decides to commit instead of abort, it is possible for an INVOKE operation that was timed out to be committed. To avoid this uncertainty, either the client transaction must be aborted or each INVOKE must be called within a nested transaction by enclosing it by a BEGIN TRANSACTION - END TRANSACTION pair. This facilitates explicitly aborting a timed out INVOKE call, and then committing the top-level (client) transaction.

The case of lost response messages is handled similar to a server crash.

5.4.3 Replication Management

Reliable objects can be constructed by replicating the object and distributing the copies of an object. A replicated object logically appears as one single object; however, its concrete representation consists of multiple copies. A replicated object belongs to a new type which is constructed from the copies of some base type. Each copy in the concrete representation of a replicated type is managed by the type manager for the base type. The operations defined on a replicated type object are executed by its type manager which invokes operations on the copies. The transaction facility in Zeus provides a means for updating distributed objects atomically. A replicated object is given a UID like any other object; however, unlike non-replicated objects, a replicated object is not located at any one location. A replicated object is logically located at all those type managers that have information about its concrete structure; therefore, it can be accessed via those type managers by invoking the type defined operations.

The type manager for a replicated type maintains for each instance of its type the UIDs of the copies in its concrete representation. This information may also include votes allocated to each copy if the weighted voting scheme is used for replication management. This structural information may be present at more than one object manager thereby making it possible to access the object via more than one object manager, and hence increasing its availability.

In the following paragraphs we describe management of multiple copies of an object by its type manager. Here we describe maintaining strong consistency among multiple copies using the majority voting approach similar in concept to the scheme described by Thomas [THOM79]. Before presenting this description we review the basic functions performed by a generic object manager when it receives an invocation request. An object manager, on receiving an invocation request, performs the requested operation by creating a worker process. If the operation modifies the object, then a new

uncommitted version of the object is created. This version is forced on the stable storage before sending a response message to the client process. A lock is maintained on the object and no other process is allowed to access the objects locked in update mode. On receiving the PREPARE message, the object status is changed to commit-pending and a READY response is returned to the client. An uncommitted version can be unilaterally aborted by its object manager. In the commit-pending state when the COMMIT message is received, the object status is changed to committed and the lock is released. In case of an ABORT message, the commit-pending version is deleted and the old committed version is restored.

An object manager for a replicated type executes transactions structured as shown in Figure 5-11. It consists of one nested transaction within which the replicated copies are updated. These updates are not committed until the outermost transaction is committed.

```

1  BEGIN_TRANSACTION (T1)
2      BEGIN_TRANSACTION (T2)
3          Invoke operations on the replicated copies;
4          Wait to receive responses from a majority of copies;
5          If majority response OKAY then
6              Establish_Recovery_Point;
7              Send "OKAY" to the client;
8              Wait to receive PREPARE/ABORT message from client;
9              If ABORT then ABORT else begin {COMMIT}
10         END_TRANSACTION with ERP
11
12         If T2.status=ABORTED then begin
13             Send "ABORT" to the client;
14             ABORT; end
15         else begin
16             Send READY;
17             Wait for COMMIT/ABORT message;
18             If message is ABORT then ABORT else {COMMIT}
19         END_TRANSACTION

```

Figure 5-11. Operation Execution by Type Manager of a Replicated Object

The replicated copies are updated within the nested transaction T2. If the majority of copies respond with OKAY response, a response message is sent to the client. Transaction T2 waits to receive ABORT/PREPARE message from the client. In response to an ABORT, both T2 and T1 are aborted. On receiving PREPARE, T2 is committed by executing END TRANSACTION. If the execution of END TRANSACTION is successful, then T2 will be in the commit-pending state. T1 is aborted if T2 fails to commit. If T2 commits, then at least a majority of copies would be in the commit pending state. A READY message is sent to the client if T2 committed. At this point T1 waits to receive the COMMIT/ABORT message. T1 is committed if the COMMIT message is received, else T1 is aborted, which results in the abortion of T2.

ZEUS DISTRIBUTED OPERATING SYSTEM

5.5 CONCLUSIONS

In this chapter we have presented an overview of the Zeus distributed operating system which is suitable for highly reliable applications. Zeus is an object-oriented system which is novel in the sense that it integrates many of the conventional database management functions into the operating system. Recovery and synchronization are transparent to the application programmers. The software for this system looks no different than the conventional software because of the remote procedure call mechanism which makes accessing of remote and local objects identical. This distributed operating system is basically a collection of system and application defined object managers (also referred to as Type Managers because they manage the objects of a specific type). In this chapter we have described some system define Type Managers that provide the essential facilities to the application developers for creating their own Type Managers. This chapter has presented certain principles that have not been tested yet in a real implementation; there still remains a significant amount of research to be done to demonstrate these ideas in a real system.

In this chapter we have also presented the application visible functions for constructing reliable distributed software systems and the rationale and details of a design for implementing these functions. This design separates the recovery functions performed by object managers and process managers, describes the databases maintained by a process manager and an object manager, and identifies the events which cause updates to these databases to be forced onto stable storage. This chapter also presented a detailed design for the commitment and concurrency control of nested transactions. Finally, it describes how Zeus facilities can be used for constructing reliable systems.

CHAPTER 6

COMMUNICATION NETWORK DESIGN METHODS

6.1 INTRODUCTION

This chapter describes some methods for designing communication networks for command and control systems. The primary requirements for such networks are survivability and minimal communication delays. This chapter presents some graph theoretic methods for designing the communication networks to meet these requirements. The contents of this chapter are derived from the various papers on this topic.

A distributed system is viewed as a graph in which the processors represent the nodes (vertices) and the communication links represent the edges. We will consider all the links as undirected. The following characteristics of such a graph are of interest to the designer:

Degree: The degree of a node in the graph is the number of edges incident upon it. In a communication network this corresponds to the number of communication links connected to the processor.

A graph is said to be of degree d if all the nodes have degree less than or equal to d .

A graph is regular if all nodes have the same degree.

Diameter: The diameter of a graph is the maximum of the shortest paths between all pair of nodes.

The diameter of a graph represents the maximum number of branches that must be used to transmit a message between any pair of nodes in the graph. In other words this is proportional to the maximum delay for message transmission between any pair of nodes.

Node Connectivity: This is the smallest number of nodes in the graph which when deleted divide the graph into two disconnected components.

Edge Connectivity: This is the smallest number of edges in the graph which when deleted divide the graph into two disconnected components.

The survivability of a communication network is dependent on the degree of the network graph. The node connectivity of any graph is less than or equal to its edge connectivity, which is less than or equal to the minimum degree of the graph.

The design of communication networks should attempt to minimize the diameter of the network and increase its connectivity for a given maximum degree for its nodes. The maximum degree of the nodes is dependent upon the physical limitations dictating the fan-in and fan-out of the processors.

In the past several researchers have investigated the problem of designing graphs which maximize the number of nodes in the graph for a given degree d and diameter k for the graph. This is called the (d, k) graph problem. Obviously, for this problem the best achievable connectivity is d .

For the (d, k) graph problem, the upper bound on the number of nodes is given by the following expression:

$$N_M(d, k) = 1 + d + d(d-1) + \dots + d(d-1)^{k-1}$$

$$= \frac{d(d-1)^{k-2}}{d-2} \quad (d > 2)$$

This is called the Moore bound because the graphs which achieve this are known as the Moore graphs. Unfortunately, the Moore graphs exist for only a few combinations of d and k . For $d > 2$ only $(3, 2)$, $(7, 2)$, and possibly $(57, 2)$ are realizable.

In the following section we briefly describe some techniques for constructing (d, k) graphs. These techniques either maximize the number of nodes in the graph given d and k , or minimize k given d and the number of nodes. One of the properties of interest for (d, k) graphs is their expandability; the network should be expandable in an incremental fashion with small increments of nodes. The best known results from these construction techniques are presented in a tabular form in [MEMM82]. Parts of this table are given in Table 6-1.

6.2 MULTITREE STRUCTURED (MTS) GRAPH

Arden and Lee [ARDE82] present a method of constructing graphs which tend to minimize k for some given n and d . This construction procedure is given only for $d=3$.

A multitree structured (MTS) construction is defined as follows:

1. These are m trees of $(t-1)$ levels
2. The m roots are connected to form a cycle
3. Each leaf is connected to $(d-1)$ other leaves
4. All leaves are connected in at least one complete cycle.

$\frac{A}{d}$	1	2	3	4	5	6	7	8	9	10
2	3	5	7	9	11	13	15	17	19	21
3	4	10	20	30 S (46)	56 AL (94)	72 AL (190)	120 AL (332)	124 AL (766)	138 C2, S1 (1534)	216 S2 (3970)
4	5	15	35 A (53)	48 T (161)	80 C1 (485)	114 S2 (1457)	212 C2 (4373)	448 C1 (131 121)	644 C2 (39 365)	1024 C1, I (119 097)
5	6	24	42 S (106)	126 A (426)	130 C2 (1706)	232 S2 (6826)	530 C2 (27306)	850 F (109 226)	2 130 C2 (436 906)	3 512 S2 (1 747 526)
6	7	31 (35)	62 S (187)	108 C1 (937)	462 A (4687)	729 I (23437)	2187 I (117187)	6561 I (585 937)	19 683 I (2 929 687)	59 049 I (14 543 137)
7	8	50	80 (302)	150 (1814)	378 A (10 886)	1716 A (65318)	2114 C2 (391910)	3626 F (3 351 462)	12 698 C2 (14 108 774)	22 836 S2 (84 652 646)
8	9	57 (63)	114 S (457)	256 C1, I (3201)	1280 C1 (22 409)	4096 I (156 865)	16 384 I (1 098 057)	65 536 I (7 686 401)	262 144 I (53 804 809)	1 048 576 I (375 633 665)
9	10	74 (80)	150 (658)	240 (5266)	738 C2 (42 130)	1904 S2 (337 042)	5 922 C2 (2 696 338)	24 310 A (21 570 706)	47 394 C2 (172 565 648)	94 416 S2 (1 380 525 202)
10	11	91 (99)	200 (911)	625 I (8201)	3125 C1, I (73 811)	15625 I (664 301)	78 125 I (5 978 711)	390 625 I (53 808 401)	1 953 125 I (484 275 611)	9 765 625 I (4 358 480 501)

Table 6-1

The construction procedure is given only for $d=3$; however, there does not seem to exist a formula to obtain the diameter. Some of the graphs constructed this fashion are the best known solutions. Figure 6-1 shows the MTS graph for m and t both equal to 3. The graphs for $d=3$, and k in the range (5..8), the graphs constructed by this method are the best known solutions [MEMM82].

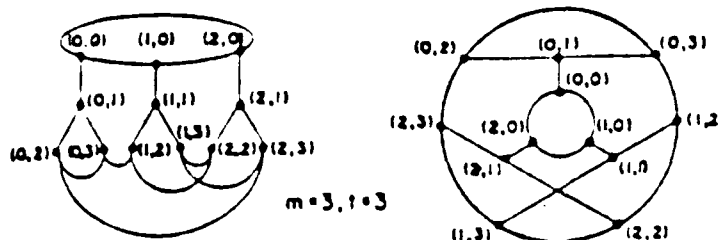


Figure 6-1: Example of Arden and Lee construction

6.3 IMASE AND ITOH CONSTRUCTION

Imase and Itoh [IMAS81] present a method for constructing minimum diameter directed graphs for some given number of nodes n and the maximum out-degree d . They extend this method for undirected graphs by removing the direction from the edges; however, the undirected graphs obtained this way are not regular. The maximum degree for the undirected graphs can be up to $2d$.

This construction is defined as follows:

1. The nodes are labeled as $0, 1, 2, \dots, n-1$.
2. Add an arc from node i to node j if the following equation is satisfied

$$j = i \cdot d + A \bmod(n)$$
 where $A=0, 1, \dots, d-1$

In [IMAS81] it is proved that the diameter k of a graph generated in this fashion is ceiling $(\log n)$. Also, the following inequality holds

$$n \leq (d/2)^k$$

and, for d even, $n=(d/2)^k$.

These graphs have the following properties:

COMMUNICATION NETWORK DESIGN METHODS

1. For arbitrary n and d , the diameter is at most one larger than the lower bound.
2. As d increases, the diameter becomes almost equal to the lower bound.

Figure 6-2(a) shows a graph constructed in this fashion for $n=7$ and $d=2$. Figure 6-2(b) shows the corresponding undirected graph.

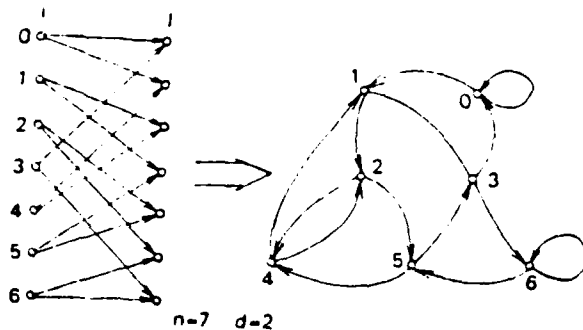


Figure 6-2(a): Image and Itoh Directed Graph Construction

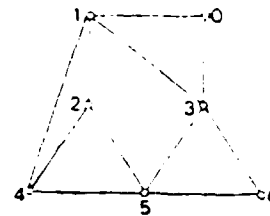


Figure 6-2(b): Undirected Graph

6.4 MEMMI AND RAILLARD CONSTRUCTION

Memmi and Raillard [MEMM82] have defined two construction methods for improving the known (d,k) graphs. These methods generate regular graphs for some given degree d . The graphs constructed using these methods have connectivity equal to d [AMAR82].

6.4.1 Construction C1

In the first construction method, there are $m(d/2)^{m-1}$ nodes distributed around a cylinder in m rows of $(d/2)^{m-1}$ nodes.

Each node x is identified by the pair (i,j) with $0 \leq i < m$ and $0 < j \leq (d/2)^{m-1}$; x is the node j of row i . Each node (i,j) is connected to $d/2$ nodes of the row $i-1 \pmod m$, and $d/2$ of the rows $i+1 \pmod m$ with the following rule, (i,j) is connected to node (k,l) if

$$k = i + 1 \pmod m$$

$$l = (A + d/2 (j-1)) \pmod{(d/2)^{m-1}}$$

with $A = 1, 2, \dots, d/2$.

The diameter k of $C1$ graph defined by d and m is given by

$$k = m \text{ for } m = 3$$

$$k = m + \lfloor m-1/2 \rfloor - 2 \text{ if } m > 3.$$

6.4.2 Construction C2

In this method, first construct d number of identical trees of m levels. Each such tree has $d(d-1)^{m-2}$ leaf nodes because each node has degree d . The i^{th} leaf node of any tree is connected to the i^{th} leaf node in each of the other trees. Thus every leaf node is connected to $(d-1)$ leaf nodes in the other trees. For such graphs the total number of nodes is given by

$$n_{C2}(d,k) = (d/d-2)(d(d-1)^{\lfloor k/2 \rfloor - 2})$$

In this construction, k must be even. Figure 6-3 shows an example of this method constructing a graph for $d=3$, $k=5$ and $n=30$.

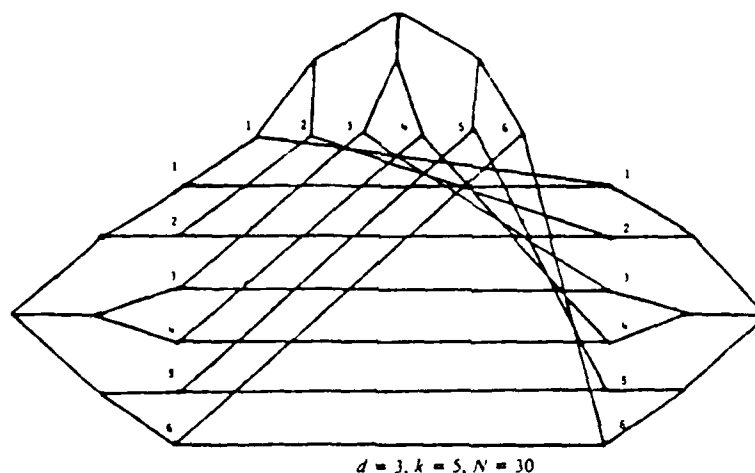


Figure 6-3: Memmi and Raillard's C2 Construction

A comparison of $C1$ and $C2$ constructions with some other techniques is presented in [MEMM82]. Some of these comparisons are summarized below. In the following comparison n_i , n_{C1} and n_{C2} refer to the order of the graphs obtained using Itoh and Imase, $C1$, and $C2$ constructions respectively. The asymptotic behavior is described below:

Asymptotically, we have the following behavior.
When $d \rightarrow \infty$

COMMUNICATION NETWORK DESIGN METHODS

$$n_{c1}(d,k) = \frac{2k+5}{3} (d/2)^{2(k+1)/3}$$

$$n_{c2}(d,k) = d^{(k+1)/2}$$

$$n^I(d,k) = (d/2)^k.$$

When $k \rightarrow \infty$

$$n_{c1}(d,k) = \frac{d^2}{d-2} (d-1)^{k/2}$$

$$n_{c2}(d,k) = \frac{2k}{3} (d/2)^{2k/3}$$

$$n^I(d,k) = (d/2)^k.$$

One notices that Itoh and Imase's construction reaches the order of the Moore bound.

6.4.3 Graph Construction

The second construction method, described before, is based on the principle of generating a new graph from a set of graphs, by performing certain operations. The first operation, called $T(G,h)$, constructs a graph of degree $(d+1)$ and diameter $2h+k$ from a set of graphs each of which has degree d and diameter k . The second operation called $B(G_1, G_2)$ generates a new graph using some graphs of type G_1 and G_2 . The new graph has degree equal to $(\max(d_1, d_2) + 1)$ and diameter equal to $k_1 + k_2 + 2$.

These constructions improve several values of (d,k) graphs. These constructions are described below.

6.4.3.1 $T(G,h)$ Construction

Let G be a graph of order N , degree d and diameter k . The nodes of G are from 1 to N . N trees numbered from 1 to N are then constructed. They have h levels and are of degree $d + 1$ and so they have $L = (d+1)d^{h-1}$ leaves numbered from 1 to L .

$T(G,h)$ is then constructed from L number of graphs of type G numbered from 1 to L and N trees. The leaf j of the tree i is identified to the node i of the graph G numbered j .

Let $NT(G,h)$, $DT(G,h)$, $KT(G,h)$ be respectively the order, the degree and the diameter of $T(G,h)$; we have

$$NT(G,h) = \frac{N}{d-1} (d+1)d^{h-2}$$

$$DT(G,h) = d + 1$$

$$KT(K,h) = 2h + k.$$

When G is a circuit of 4 nodes ($d=2$), for $h=1$ we get the graph $T(G,1)$ as seen in Figure 6-4.

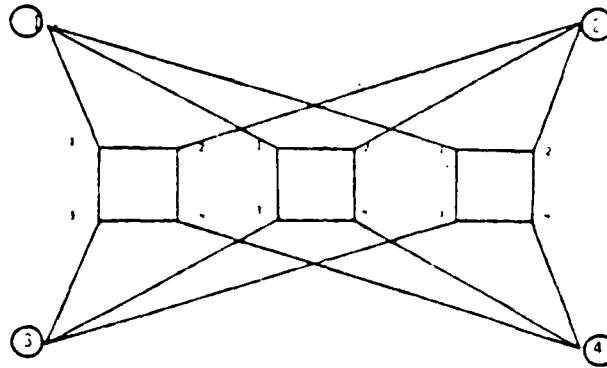


Figure 6-4: Example of $T(G,h)$ Construction

6.4.3.2 $B(G_1, G_2)$ Construction

Let G_1 and G_2 be two graphs of order N_1, N_2 ; degree d_1, d_2 , and diameter k_1, k_2 respectively. Let us assume that $d_1 \geq d_2$.

$B(G_1, G_2)$ is then constructed from N_2 number of graphs of type G_1 numbered from 1 to N_2 and $(d_1 - d_2 + 1)$ number graphs of type G_2 numbered from 1 to $d_1 - d_2 + 1$ N_1 .

The node i of the j th graph G_1 is connected to the node j of the graphs G_2 numbered $kN_1 + i$ with $k = 0, 1, \dots, d_1 - d_2$. The node i of the j th graph G_2 is connected to the node j of the i -th graph G_1 .

Let $NB(G_1, G_2)$, $DB(G_1, G_2)$, $KB(G_1, G_2)$ be respectively the order, the degree, and the diameter of $B(G_1, G_2)$, we clearly have

$$NB(G_1, G_2) = N_1 N_2 (d_1 - d_2 + 2)$$

$$DB(G_1, G_2) = d_1 + 1$$

COMMUNICATION NETWORK DESIGN METHODS

$$KB(G_1, G_2) = k_1 + k_2 + 2.$$

These two methods provide a powerful way of constructing some of the best known solutions to (d,k) graph problem. A complete survey of the best known solutions can be found in [MEMM82].

6.5 SUMMARY OF KNOWN SOLUTIONS

The best known solutions to the (d,k) graph problem are summarized in Table 6-1. In this Table, the values in brackets are the Moore bounds. The results are marked as A due to Akers [AKER65]; A-L to Arden and Lee [ARDE82]; F to Friedman [FRIE66]; I to Imase and Itoh [IMAS81]; and C1 and C2 to constructions by Memmi and Raillard [MEMM82].

CHAPTER 7

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

7.1 INTRODUCTION

The first concern of a system designer is generally the correct functionality of the system he is designing. It is, of course, imperative that a system correctly performs the tasks for which it is intended. Until very recently, designers did not concern themselves with the costs, in terms of resources and time, of providing this functionality until after some or all of the system was operational. In fact, it was found that too much concern for performance too early in the development was often enormously expensive and detrimental to the overall clarity and maintainability of the system. Even so, Knuth probably over-generalized when he stated in [KNUT74]:

"...Premature optimization is the root of all evil."

Modern performance analysis texts such as [KOB78] and [CHAN81] do, however, advocate the early evaluation of performance starting from the initial phases of the design and continuing throughout the useful lifetime of the system. The problem, apparently, was not so much premature concern with performance as it was unguided and uninformed optimization efforts. Too much time was being spent attempting to find optimal designs for non-critical portions of the system. The remainder of this chapter is a discussion, with examples, of the tools and techniques available to the designers of distributed, reliable systems for the purpose of providing guidance and assistance in estimating the performance of such designs.

Early performance predictions and the resulting design iterations are especially important in the design of highly reliable systems. This is because, unlike functional correctness, the reliability of certain functions or modules might be negotiable. If the cost of a reliable function is too high, the designer might be willing to accept a lower degree of reliability for that function which is not so extravagant with system resources. Such tradeoff decisions can only be made if the designer has at his disposal early estimates of performance and reliability.

Reliability analysis is discussed in Chapter 8 of this guidebook while the present chapter is concerned with the methodologies used to predict the performance of a pre-operational system during the design phases. Section 7.2 is a brief survey of the field of performance engineering with references to more extensive surveys in the literature. Section 7.3 is an introduction to some of the notation and tools which were used in the performance evaluation of the Zeus system described in Chapter 5. Excerpts and abstracts from those performance models are used extensively as examples in the later sections of this chapter.

Section 7.4 discusses some of the specific issues involved in analyzing the structure of the system itself. The first part of the section is concerned with issues specific to distributed systems and the second part discusses reliability considerations.

Sections 7.5 and 7.6 take up, respectively, the problems of modeling the external environment in which the system will operate, and the workload the system will be expected to handle. Finally, Section 7.7 summarizes the results of the previous sections and draws some conclusions about performance analysis of distributed, reliable systems.

7.2 PERFORMANCE ANALYSIS METHODOLOGIES

The following brief survey of performance analysis methodologies borrows heavily from a number of more thorough and extensive surveys including [KOB78], [CHAN81], [FERR78], and [LUCA71]. Further details about any of the topics discussed here may be found within these references.

The performance analysis part of the design evaluation phase is concerned with providing quantitative estimates for certain resource, utilization performance measures. Exactly which measures are interesting to the system designer and at what level of detail these estimates are to be made are questions for which it is important to have answers before proceeding with the analysis effort.

7.2.1 Performance Measures

There are several generic performance measures which are typically used to describe and quantify the performance of computer systems. These fall into two distinct categories: user-oriented measures and system-oriented measures.

The user-oriented measure most often used with respect to interactive systems is response time (turnaround time for batch systems). Response time is the elapsed time between the arrival of a request and the completion of that request by the system. Of course the exact moments of "arrival" and "completion" of a request must be carefully defined for any given application.

The two system-oriented measures most commonly encountered are throughput and utilization. Throughput is defined as the average number of requests processed by the system per unit of time. This is typically not a very useful measure of system performance since, as long as the system is performing well enough so that it can complete requests without creating an ever-increasing backlog, the throughput of the system is equivalent to the average arrival rate of the requests. Utilization is defined to be the fraction of time that a particular resource is busy - that is, working on some request.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

7.2.2 Models and Hierarchical Structuring

For operational systems, the most straightforward approach to performance evaluation is to directly measure the performance using some combination of hardware and software monitors. This, of course, is impossible during the design phases of a system since there is nothing yet to measure. In such cases when direct measurement is impractical or impossible, a model of the system must be devised which captures the salient factors that determine system performance. The model is then evaluated and the performance measures thus obtained are used as estimates for the performance measures of the actual system.

The complexity of such models and the degree to which they represent or abstract from the actual system determine to a large extent the amount of effort and expense required to evaluate them. Generally, the more detailed the model, the more expensive it is to evaluate. Luckily, during the design phases of a system, there is normally not a requirement for extremely accurate estimates of system performance. We are typically more interested in rejecting those designs which have a very negative impact on performance and in providing guidance as to which parts of the design should be considered for optimization; Therefore, performance models constructed during the design stage are normally simpler and more abstract relative to the actual system.

Even so, modeling a system which has many interconnected parts, even at a very abstract level, often produces overall models which are large, complex, and for which evaluation is intractable. The solution to this problem is the same as the classical solution to the general problem of software complexity: hierarchical structuring. Models which are decomposed into several smaller sections and structured vertically or hierarchically as in Figure 7-1 prove to be both more manageable and easier to evaluate [KOB78] [BROW75]. Such hierarchical structuring allows the analyst to summarize the performance results obtained from evaluating one level of the model (say the micro level in Figure 7-1) in a form which is easily usable in the next higher level (the intermediate level in Figure 7-1)

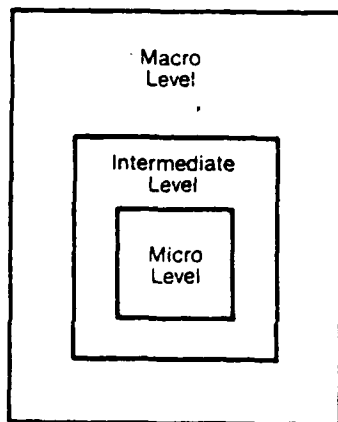


Figure 7-1 Hierarchical Model Structure.

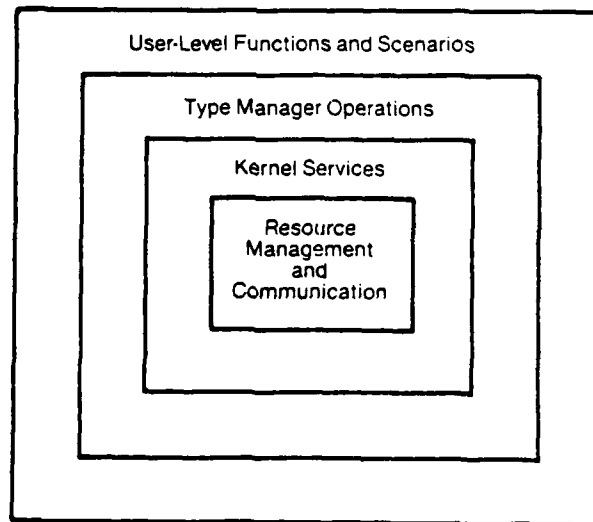


Figure 7-2 Object-Oriented Modeling Hierarchy.

The decomposition of a model into a hierarchy of sub-models should take into account the inherent structures of the machine configuration as well as the system being modeled. A common rule of thumb criterion [KOB78] is that the time constant at a given level of the model should be significantly smaller than the average inter-event times at the next higher level. In other words, a large number of state change events should occur at the lower level between events at the next higher level. In Figure 7-1, for example, the micro level models might have typical inter-event times on the order of micro-seconds or nano-seconds, the intermediate level on the order of milli-seconds, and the macro level on the order of seconds.

For object-oriented, high-integrity systems, there are a number of convenient levels of detail for which performance measures may be obtained. This natural hierarchy of modeling layers is illustrated in Figure 7-2.

The first and final consideration of the system designer must be the response time of the highest-level user operations and functions which constitute the typical system workload. For semi-specialized systems such as distributed command and control systems, it may be relatively easy to choose a representative set of such user-level functions or scenarios. In these cases, it is likely and desirable that all the performance, reliability, and correctness constraints be specified in terms of these user scenarios. Estimating the response time and resource utilization of these scenarios is then the obvious starting point for the performance analysis effort for such distributed systems. The average inter-event times at this highest level of modeling will typically be on the order of seconds, tens of seconds, or longer.

Object-oriented systems with their highly modular structure conveniently accommodate a finer granularity of performance analysis. The parts of an object-oriented system that will need to be studied at the next level of detail will in most cases be the individual type managers and their associated operations. These type manager operations will essentially constitute the high-level instruction set from which the user transactions are constructed. The inter-event times at this level will normally be on the order of hundreds of milli-seconds or seconds.

The various services provided by the system kernel to the type managers is the logical next level of detail in any performance study of an object-oriented system. These services will include notably the remote procedure call mechanisms and protocols provided by the kernel. In addition to this, the kernel will likely provide interface routines to the various system resources such as mass storage. The performance of these system services, of course, eventually affects the performance of the higher level type manager operations and user scenarios. Time constants at this level are likely to be measured in micro-seconds or milli-seconds.

Below the level of the kernel procedures and functions, the analysis of the underlying resource management and communication routines becomes identical with similar analyses for conventional systems. Although the same tools and techniques are generally applicable even at this level of detail, we will not be concerned further here with these issues. In many cases, previous performance results for such conventional subsystems may be applied in

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

modelling high integrity systems in order to simplify the models without sacrificing accuracy. A typical example of this is the use of simple statistical models of common local area network protocols using the results of previously published detailed models of such protocols. Events at this lowest level of the model can be expected to occur very frequently, on the order of micro-seconds or nano-seconds.

7.2.3 Parts of a Performance Model: System, Environment, Workload

There are really three distinct factors which impact the development of performance models for computer systems. The most obvious of these is, of course, the structure of the system which is to be modeled. As previously mentioned, the structure of the model might not operationally reflect the structure of the actual system but might rather abstract from it the main features which affect performance. It is believed, however, that the object-oriented system structure discussed widely in this guidebook will simplify the task of producing performance models of the system. This is because of some of the same reasons that make this approach highly suitable for the formulation of highly reliable distributed systems: inherent modularity and hierarchical structuring.

In addition to modeling the structure of the system, the environment in which the system must operate must also be considered. The environment includes such things as the native hardware and software in which the system is to be embedded and, of particular interest in the modeling of reliable systems, the fault characteristics of that hardware/software configuration.

Finally, the workload which the system will be expected to accommodate must also be modeled in some way. Choosing an appropriate workload and a representation for it is less of a problem for existing operational systems although it is still very much an art and still very difficult to do. For systems which are not yet operational, the problem becomes one of choosing or inventing a hypothetical workload which will hopefully reflect the characteristics of the future workload of the actual system.

7.2.4 Techniques for Evaluating Performance Models

In order to obtain useful predictions of performance measures from the models, they must be evaluated in some way. Performance model evaluation is the process by which values are derived for the chosen performance indices given a "correct" and properly parameterized model and an appropriate workload.

The question of whether a model is "correct", that is, whether the performance of the model accurately reflects the performance of the actual system, is a very difficult problem and deserves some consideration here. This problem is normally referred to as model validation. When modeling existing systems, the most common validation technique is called calibration. Calibration is a validation method whereby the model is evaluated for a limited number of input conditions and the results are compared to the measured performance results for the actual system. If the model is found to

be sufficiently accurate within this limited range of input conditions, then the "domain of validity," the set of all input conditions for which the model is accurate, is assumed to include the other input conditions which are of interest. Naturally, design phase modeling is not susceptible to this sort of validation. Instead, one is forced to trust in the modeler's belief that the conceptual bases of the model are sound and that a correct procedure was followed in its construction. In addition, the results of such models should be subjected to plausibility constraints to ensure that they are not totally out of line with common sense and/or previous performance measures of similar systems.

In any case, once a validated, calibrated, or plausible model has been constructed, it must be evaluated to obtain values for the performance indices of interest. Models may be evaluated analytically, by simulation techniques, or by some hybrid combination of the two.

7.2.4.1 Analytic Methods

In [KOBA78], an analytic evaluation method is defined as, "a solution technique that allows us to write a functional relation between system parameters and a chosen performance criterion in terms of equations that are analytically solvable." The term "analytically solvable" here is usually taken to include numerical solution methods other than simulation as well as closed-form solutions.

Although such a definition of analytic solvability includes deterministic techniques like automata theory and Petri nets, the term is most often used to refer to the mathematical discipline called queueing theory. Mathematical queueing theory provides a framework in which networks of resources (CPU, memory, I/O devices, etc.) are being prevailed upon by jobs to perform some services. Contention for a resource causes jobs to be queued for later service.

Usually, many assumptions and simplifications must be made in order to make a model analytically tractable. For example, an important constraint in queueing theory is that a job cannot hold more than one resource at any given time. Yet, often the results thus obtained are later found to agree surprisingly well with measured results. In general, it is worth the effort to investigate the applicability of analytical techniques for the evaluation of performance models since, if they are feasible, they typically cost considerably less than other (simulation) techniques. An excellent introduction and discussion of analytic evaluation techniques may be found in [FERR78].

7.2.4.2 Simulation Methods

When evaluating an hierarchically structured model of a large system, it is likely that at least some of the submodels will be susceptible to analytical methods. It is also very likely, however, that the analytical solution of some of the submodels will remain mathematically intractable even

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

with simplifying assumptions and constraints. In these cases, the only alternative evaluation method for non-existing systems is simulation.

Simulation is a numerical technique for evaluating queueing network models by mimicking the dynamic behavior of the system being modeled. The principle advantage of this technique is its great generality. Most of the constraints which are necessary for analytical methods have little consequence with respect to simulation. The three main problems with simulation are the expense involved with building the simulator, the expense of running the simulation, and the necessity for statistical analysis of the resulting output data.

The first of these problems with simulation has been relieved somewhat by availability of general purpose simulation languages such as SIMULA [DAHL66], SIMSCRIPT [KIVI69], GPSS [IBM71], SIMPL/1 [IBM72], and PAWS [IRA84]. These languages provide many of the services which are required for writing efficient simulators including event scheduling, random number generation and so on.

The problem of the expense of running simulations derives from the fact that the length of a simulation run is proportional to the number of events which must be simulated rather than to the duration of simulated time. In the example of Figure 7-1, it would probably be desirable to run a simulation of such a system long enough to see perhaps hundreds of events at the macro-level in order to ensure that the simulation reaches a steady state. Since events at this level occur approximately every second, we will wish to run the simulation for something on the order of say 1000 seconds. But if the intermediate and micro levels are also entirely included in the model, the total number of micro events which must be simulated might be on the order of several billion. Such a simulation run will likely be very expensive. This problem is most effectively controlled by hierarchical structuring of the model as explained in Section 7.2.2. This allows low-level models to be evaluated separately and the results summarized at the next higher level in the form of a scaling factor or statistical distribution.

7.2.4.3 Hybrid Methods

A combination of both analytical and simulation methods may be used in evaluating a model of a large system. Again, the hierarchical nature of the model may be taken advantage of to allow lower-level sub-models to be evaluated using either analysis or simulation whichever is more appropriate and least expensive. The results thus obtained may then be summarized in modeling the higher layers.

7.2.5 Performance Measures for Integrity Mechanisms

For the particular purpose of studying the performance of reliability and integrity mechanisms, there are a number of obvious questions which the performance analyst/system designer will be called upon to answer. What are the additional costs incurred simply because the integrity mechanisms are in

place? How costly will certain kinds of failures be in terms of response time and resource utilization? On the average, how long does it take to recover from a certain class of faults?

The design tradeoff decisions concerning reliability and integrity mechanisms and performance are generally more complex than those for conventional systems where high reliability is of less importance. Such tradeoffs are conventionally between different kinds of performance, such as resource utilization and response time. The only other analytical property of such systems is their correctness - the degree to which they satisfy their operational specifications. For obvious reasons, the correctness of a program is rarely purposely compromised in favor of better performance. It may, however, be perfectly valid to design an object so that it is slightly less reliable but responds quicker (or vice versa).

Because of the complex tradeoff decisions which are likely to be involved in configuring a system such as the one with which we are currently concerned, it will be necessary for the designers (and possibly also the system administrators) to have at their disposal reasonably accurate estimates of the costs of the various reliability and integrity mechanisms which are provided by the system. In order to provide such estimates, the analyst/designer must examine at least three different cases:

- o The performance of the system in the absence of the relevant reliability/integrity mechanisms. Obtaining these results will involve deleting or "disconnecting" the mechanisms from the models temporarily. This provides a basis for comparison and a bound on the performance of any implementation which includes that particular reliability feature.
- o The performance of the system with the relevant mechanisms in place but in the absence of the failures which the mechanisms are there to protect against. This class of performance figures, when compared to those obtained as above, will provide a useful estimate of the best case cost of providing protection from faults. In addition, this gives a bound on the performance of recovery strategies which will be invoked when a failure occurs.
- o The performance of the system with integrity mechanisms in place and when failures of the defined class actually occur. Together with the results obtained in the first two cases above, these figures will provide an estimate of the time and resource requirements of the recovery mechanisms of the system. Of course, obtaining performance estimates for the system in the presence of failures requires that the possibility for such failures be built into the analytical or simulation models of the system being used by the analyst. This topic is discussed in Section 7.5.

7.2.6 Example Metrics for Some Generic Integrity Mechanisms

The following is a sampling of some of the performance measures which are likely to be interesting in a distributed, object-oriented reliable system. Three generic classes of reliability and integrity mechanisms are used to illustrate the issues involved; transactions, concurrency control, and object

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

replication. A more detailed discussion of these mechanisms may be found in Chapter 4.

7.2.6.1 Transaction Mechanisms

Of course, user-level scenarios will probably be defined as or in terms of atomic transactions. The response times and throughput of these will be of primary concern. In this section, however, we will be dealing only with the low level performance characteristics of the mechanisms used to attain atomicity and reliability for groups of associated individual type manager operations. The following is a list of some of these low level characteristics:

- o Mean Rollback Time - The mean time required for a type manager to rollback an object to a previous state (the state of the object at the time of the last checkpoint).
- o Mean Size of "Window of Vulnerability" - The mean time during which an object is vulnerable to a failure of the coordinator of the transaction.
- o Mean User In-Doubt Period - The mean time from when a user decides to commit a transaction until the user can be told that the results are committed.
- o Mean Coordinator In-Doubt Period - The mean time (from when a coordinator issues a commit message until it receives acknowledgements from object managers, e.g., second phase) during which a coordinator must retain the state of a transaction.

7.2.6.2 Concurrency Control

The two fundamental concurrency control strategies, locking and timestamps, have two different sets of performance characteristics. Some of the important metrics associated with locking mechanisms are given here as examples.

- o Probability of Object Contention - With a given mix of concurrent user scenarios or transactions, the probability that two incompatible requests for the same object will occur at the same time.
- o Probability of Deadlock - With a given mix of concurrent user scenarios or transactions, the probability that the "wait-for" graph connecting two or more transactions contains a cycle. Such deadlocks may occur when using locking mechanisms without suitable prevention or detection/correction algorithms.
- o Cost of Deadlock Detection/Prevention - The overhead associated with eliminating the possibility of deadlocks in a locking system.

7.2.6.3 Object Replication

There are both costs and benefits associated with maintaining multiple redundant copies of some objects. The costs are in the form of the additional storage requirements for the redundant copies and the time and resources required to ensure that the multiple copies remain consistent. The performance benefit stems from the fact that, in some cases, local copies of an object may be used to provide read-only access thus eliminating the communication costs of accessing a remote copy instead.

- o Redundant Storage Overhead - The additional storage and other resources required to maintain all but one of the identical copies of an object.
- o Multiple Update Overhead - The additional time and resources required to update additional copies of a replicated object.
- o Read-Only Access Improvement - The average improvement in read-only type operations due to the distribution and replication of an object.

7.3 MODELING TECHNIQUES IN THE ZEUS EXAMPLES

Throughout the remaining sections of this chapter we will be discussing in a general sense some of the problems which are peculiar to modeling distributed, reliable systems. In order to make the discussion a little more concrete, a number of examples have been included in the text. These examples are excerpts from models constructed to study the performance characteristics of Zeus, the reliable command and control system design introduced in Chapter 5.

In order to aid in the understanding of these modeling examples, we will digress somewhat in this section to discuss in more detail the tools and representations which were used for the Zeus models. Of these, execution graphs and information processing graphs (IPGs) provided a standard pictorial representation for the designs while PAWS, the Performance Analyst's Workbench System provided a general purpose simulation language for the evaluation of the models. Our choice of these particular tools was partly due to in-house familiarity with them (PAWS is a registered trademark of Information Research Associates), and partly due to their suitability for the tasks of representing and evaluating performance models.

7.3.1 Introduction to Zeus Modeling Techniques

A model of an executable system is formed as a coupling of the system design, the system workload, and the system resource environment. In this section, various techniques are discussed which, when given a design and specification data, render a performance model so that predictions of the performance properties of a system can be made during the early design phases of a system life-cycle. For usability, such techniques should be well documented and relatively straightforward to learn. By necessity, the techniques must capture the most important parts of the system design, workload, and resource environment in order for the model that is developed to

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

be a "good" performance model or a model that exhibits the system's execution behavior with a high degree of accuracy.

The creation of a performance model is a multi-step process involving first a determination of the most relevant execution pathways in the system design (i.e., there are many possible execution pathways in any system, and only a subset of those is used with great frequency). This shifts the focus upon those modules and the system activity those modules represent that is most relevant to the performance of the system. Once the performance determining pathways are defined, they must be coupled in a meaningful fashion with a target resource (hardware) configuration and a specification of the resource usages along the performance pathways.

The above concepts of performance engineering have been developed by Smith and Browne [SMIT79] and [SMIT80]. These authors have developed a graphical means to concisely express the performance pathways in a system. Called execution graphs, the workload specifications are combined with a system design to render directed graphs where each node represents the execution of a module. Evaluatable abstract execution paths are realized by the specification of a hardware resource configuration and the addition of resource usage specifications to the nodes.

Once the performance pathways are determined, execution at the hardware device level may be modeled by mapping the execution graphs to a queueing network model representation of the hardware configuration. This mapping results in a model with units of work flowing between resources and devices according to patterns derived from the paths in the execution graphs.

Execution graphs may be conveniently translated into Information Processing Graphs (IPGs), which are pictorial constructs for modeling information processing systems. As given in an introduction to this tool [IRA83], IPGs are a useful modeling methodology for several reasons: pictures often provide the best method for describing and understanding information flow; it is easier to communicate ideas quickly using a picture; and information processing systems are often designed around a structure of information flow. From these IPGs, it is a straightforward translation to a queueing network model.

In a distributed operating system, information flows through resources on hosts and between hosts in the network. The basic graphical components are nodes, edges, and labels. In an IPG, each node represents a resource (such as CPU, memory, disk units, etc.) while edges connect nodes and represent some form of information flow from one resource to another along an edge. Edges are given labels denoting the form of information flow.

The IPGs are directly mappable to the Performance Analyst's Workbench System (PAWS) [IRA83], which is a simulation language that is used to evaluate performance models. In a model, the information flows which are of interest are given what are termed category and transaction names for which statistics are gathered during the simulation. Additionally, for each resource in the model a set of summary statistics is generated.

In the following sub-sections, a more complete description is given of execution graphs, IPGs, and PAWS which will prove necessary in understanding some of the figures and examples in this chapter. Combining the three into a methodology for creating a performance model and gathering statistics is a straightforward and feasible endeavor. An illustration of this process is given in Figure 7-3.

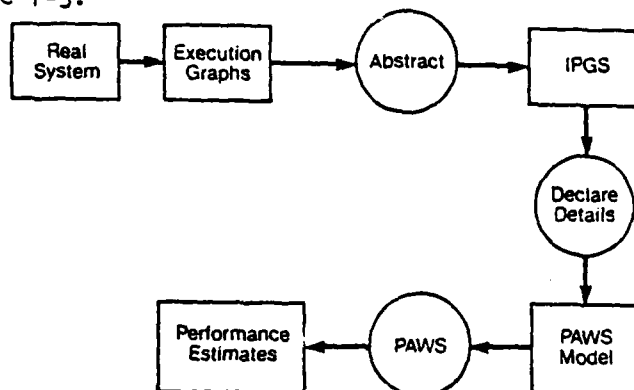


Figure 7-3 The Use of Execution Graphs, IPGS, and PAWS.

7.3.2 Execution Graphs

Much of the information that follows is based on [SMIT80b]. The methodology of Execution Graphs has proven to be a useful technique to apply in following the evolution of software and in analyzing subsequent functional enhancements. The basic methodology captures the code of the design in a graphical form and assigns to each node an estimate of the resource usage associated with that piece of code.

The graphical components of execution graphs are shown in Figure 7-4. Hierarchical structuring of modules is supported as well as conditional execution (e.g., IF-statements), looping, recursion, and nesting. Resource usage times are assigned to each node in the graph and summed for a module execution. These resource estimations become service times in the model for the code which they represent.

Assigning costs is a subjective practice to some degree as it is difficult to know the exact measure of resource that will be used by a module. However, a key point is that the resource usage assignments must be uniformly and consistently applied. This allows the comparison of designs to be meaningful. Additionally, when modifications are made to a design and to its execution graphs, the change in resource usages will be apparent in the execution graphs and show whether or not the change is an improvement in the design with regard to the resource usages.

The execution graphs serve an important function which is to define measures of resource usages by system processes. When a design changes, the execution graphs and the resource usages that change should be reflected in the performance model.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

Symbol	Name	Description
	Basic node	Represents a functional component whose execution characteristics are defined at this level
	Collapsed node	Represents a function whose execution characteristics are included in a graph at the next level of detail.
	Repetition node	Defines the beginning of a loop that will be repeated N times. The last node in the loop has a dummy arc back to the repetition node.
	Dummy node	No processing is associated with the node.
	Arc	Shows a transfer of control or a protection domain switch.
	Bi-directional arc	Shows that control will return to the origin node when processing is completed at the destination node.
	Double bi-directional arc	Same as above except that control returns to the driver, X.
	Dummy arc	No processing time is associated with the arc. They may be bi-directional.
	Self loop	Shows that processing may be completed at this node when there are additional nodes below it in the graph.

Figure 7-4 Execution Graph Notation

7.3.3 Information Processing Graphs

Used to pictorially describe the information flow, Information Processing Graphs (IPGs) are a powerful mechanism for modelers to communicate models among themselves. For this reason, numerous examples in this performance analysis chapter are expressed as IPGs.

An IPG diagrams the flow of information from resource to resource in an information processing system (such as computing, communication and office systems) and can be thought of in terms of work stations or nodes at which information is processed. In a computing system, the nodes most often represent central processors, disk units, device controllers, etc. Edges between nodes have labels denoting the form of information flow along the edge. Information flows in discrete units called transactions.

A transaction in general represents the data on which the nodes of the information processing graph operate and in particular may represent a job, program, task, scheduler, message, process, person, or any such entity useful to the modeler. A transaction gets processed in some manner at each node and, upon completion of processing, moves along the directed edge to some other

node for additional processing. Several transactions may be simultaneously active (being operated upon) at the various nodes of the IPG. Each transaction has some set of local variables associated with it. Additionally, the whole network has a set of global variables which all transactions may access.

We associate a category and a phase with each transaction. The category of a transaction is a name and is permanent; a transaction has one unique category throughout its lifetime. The phase of a transaction may be changed as the transaction progresses through the IPG. The processing of a transaction at a node and the routing behavior of a transaction from node to node in general depend on that transaction's category and phase. The routing from a node to one of a set of nodes is often done probabilistically. The general form for denoting a label on an edge is <category name, phase number, routing probability>. In the examples in this chapter where the category is understood from context, an abbreviated labeling method is sometimes used; <phase, probability>.

There are five classes of nodes used in the IPG notation: 1) resource management nodes, 2) routing nodes, 3) arithmetic nodes, 4) interrupt nodes, and 5) user nodes. Each class of nodes is discussed briefly below.

(1) Resource management nodes represent system resources (processors, memory, communication links, disks, copying machines, people, etc.). A transaction normally requests the use of certain resources and may have to queue (wait) for resource if the request cannot be fulfilled immediately. Resources are either active or passive.

Conceptually, an active resource is something that acts or works, such as a processor or disk unit, and is represented by a SERVICE node in an IPG. A transaction arriving at a SERVICE node requests the use of the resource for a specified amount of time (usually drawn from a specified service time distribution). If the resource is being used, the transaction must queue (wait) until it is scheduled for service according to the queueing discipline specified for that SERVICE node. After receiving service, the transaction exits the node along some edge to another node. Figure 7-5 shows a portion of an IPG representing a SERVICE node named CPU. The circle represents the processor and the open box or square represents the queue for waiting transactions.

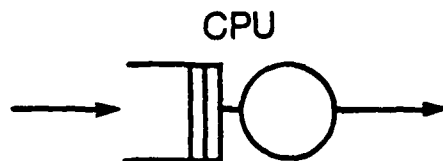


Figure 7-5 SERVICE Node CPU

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

A passive resource doesn't itself do any work but is something that must be possessed by a transaction to do work. Memories, buffers, and control points are examples of passive resources. Passive resources usually occur in groups; for example, memory may be regarded as a group of pages. A passive resource is represented in an IPG by two nodes: one at which the resource is acquired and one at which the resource is released.

The two types of passive resources are: TOKENS and MEMORIES. TOKENS are acquired at ALLOCATE nodes and released at RELEASE nodes. MEMORIES are acquired at GETMEM nodes and released at RELMEM nodes. TOKENS may be used to model input and output buffers, channels, pages, domains or control points, communication links, and other passive resources. MEMORIES are used to model contiguously addressed passive resources such as main memories, extended core storage, and disk space. Associated with each memory is a memory management scheme according to which blocks of memory are allocated to transactions.

TOKENS are created at CREATE nodes and destroyed at DESTROY nodes. Figure 7-6 shows a portion of an IPG in which transactions a) acquire BUFFER tokens at the ALLOCATE node named GET, b) create BUFFER tokens for GET at the CREATE node named MAKE, c) destroy BUFFER tokens for GET at the DESTROY node named KILL, and d) at the RELEASE node named PUT, release BUFFER tokens for the ALLOCATE node named GETMORE.

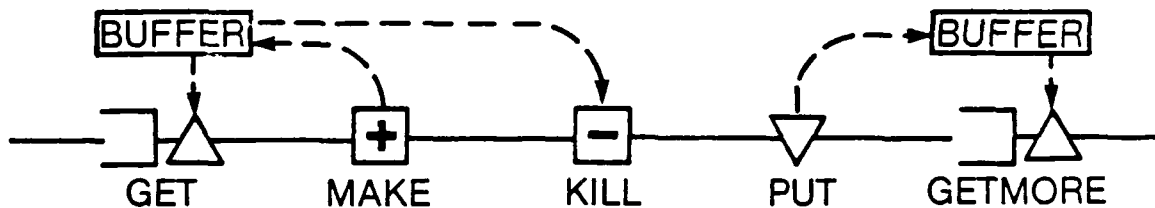


Figure 7-6 Allocate, Create, Destroy, Release Example

The rate at which an active resource processes information or the ability of a transaction to acquire a passive resource may depend on events occurring in other parts of the system. In such cases, a transaction at a SET node may request a modification to the service rate or power of a SERVICE, ALLOCATE, or GETMEM node.

(2) Routing nodes may be used to create and destroy transactions and to alter transaction flow through the system. There are six types of routing nodes: SOURCE, SINK, FORK, JOIN, SPLIT, and BRANCH nodes. At a SOURCE node, transactions are created (arrive) periodically according to a user-specified

interarrival time distribution. At a SINK node transactions disappear from the system forever. A transaction may spawn a number of children transactions at a FORK node, and the children may coalesce at a JOIN node to recreate the parent. FORK and JOIN nodes are useful for modeling the synchronization of concurrent processes. A transaction may create a number of SIBLING transactions at a SPLIT node - much like a FORK node; however, the siblings from a SPLIT node may, for instance, be used to model the operation of message communication. BRANCH nodes may be used to facilitate the specification of branching (routing) probabilities and to collect statistics.

Figure 7-7 illustrates the use of the routing nodes. Each transaction enters the system at the source node START and proceeds to the fork node TFORK, where the transaction creates two children and waits for the children to JOIN.

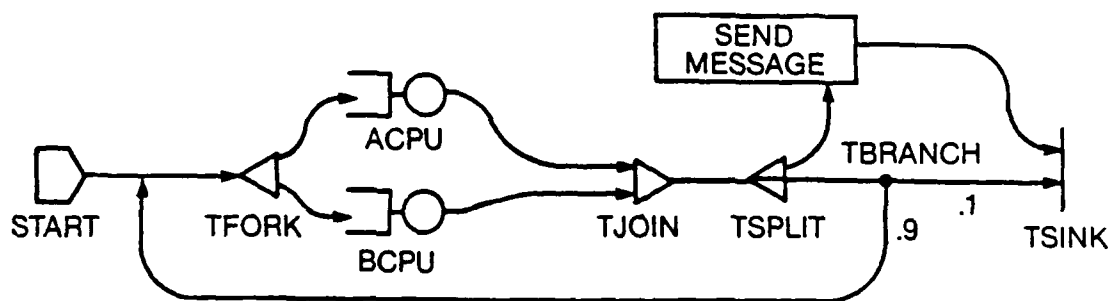


Figure 7-7 Routing Node Example.

Each child requests and uses a processor (ACPU or BCPU) before proceeding to the join node TJOIN. As soon as both children of a transaction reach TJOIN the children disappear and the parent (still waiting at TFORK) replaces its children at TJOIN and proceeds from TJOIN to the split node TSPLIT, where a sibling proceeds to send a message before leaving the system at the sink node TSINK. The original sibling (the transaction that started at START) travels from TSPLIT to the branch node TBRANCH, from which it goes back to TFORK with probability 0.9 or to TSINK with the probability 0.1.

(3) Arithmetic (COMPUTE and CHANGE) nodes are used to carry out computational steps and to modify simulation variables. A COMPUTE node is used for assignment of values to and conditional operation on the local variables of a transaction. A CHANGE node probabilistically alters the phase of a transaction.

(4) An INTERRUPT node is used by one transaction to interrupt the processing of another transaction. The interrupted transaction immediately departs from the service node where it was interrupted with a new phase assigned to it by its interrupter.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

(5) A transaction arriving at a USER node invokes a user-written FORTRAN subroutine that has access to the global variables and the transaction's local variables.

7.3.4 The PAWS Language and Model

Once a model is expressed as a system of IPGs, it is almost directly translatable into PAWS. An IPG is a graphical description of the system to be modelled. An IPG is incomplete in the sense that the characteristics of each node (queueing disciplines, service-time distributions, etc.) are not fully specified. A PAWS program is complete: it completely describes the information processing graph structure as well as the individual nodes of the graph. A PAWS model is also called a PAWS program to reflect the fact that it is in fact a program written in a general purpose simulation language. The PAWS simulator reads and evaluates a model written in the PAWS language to obtain performance statistics for that model.

The PAWS language is declarative rather than procedural: the user simply declares the characteristics of the system being modeled as opposed to coding detailed simulation algorithms. An example CPU node declaration in the PAWS language is:

```
CPU
  TYPE SERVICE
  QUANTITY 1
  QD RRFQ 10.0
  REQUEST <BATCH, ALL> HYPER(10.0, 14.1);
```

Here, the name of the node is CPU, the node type is SERVICE, there is 1 server, the queueing discipline is round-robin fixed quantum with a quantum size of 10 time units, and all BATCH transactions regardless of phase request service times drawn from a hyper-exponential distribution with mean 10.0 and standard deviation 14.1.

It is in the PAWS model that the resource usage figures of the execution graphs are incorporated. The nodes of the IPGs are directly mapped to the PAWS nodes. The edges between nodes are declared in the model, as well as the categories and transactions previously mentioned. For a more detailed description of the PAWS simulation language see [IRA84].

7.3.5 Automating Model Generation

Some of the procedures described in generating a performance model can be automated to some extent. Given the waterfall software life-cycle discussed later in Chapter 10, automation is desirable since the translation from a system design to a performance model is performed iteratively as the evaluation of the performance statistics results in changes in the system design. The possibilities of machine aided design translation have been explored to some degree, and the following summary of the range of possible approaches to the translation of system designs into performance models is the outcome of that exploration.

- (1) Automatic translation. The generation of a database containing a static description of the design with resources based on caller/callee relations in procedure calls, and on the points of synchronization in the design. This aids in the generation of executions graphs.
- (2) Semi-automatic translation. The development of the IPGs. An IPG-like graph of resources and edges can be produced from the database.
- (3) Manual translation. The selection of the appropriate pathways from the database that correspond to a transaction depicting a particular flow of information in the model. The defining of categories, transactions, and phases.

The incorporation of resource and performance parameters into designs, and the generation of a database from which execution graphs and IPG structures can be generated are new concepts to be considered in the evaluation of a design. The following procedure adapted from [FER83] is suggested to evaluate a system design that is coded in some procedural design language.

- (1) Include in the functional design the resource usages and selected performance specifications.
- (2) Process this extended design to obtain a parameterized intermediate form of a performance model, such as a set of execution graphs.
- (3) Bind this intermediate model to a specific execution resource environment and workload, thus producing a specific performance model.
- (4) Evaluate the performance model by simulation or other technique obtaining performance measure estimates for the system.

In Figure 7-8, these steps are given in terms of the modeling techniques previously discussed, with the addition of an automatic translation step from the design to a database from which execution graphs can be generated. The inclusion of resource usages in a design and the automatic translation step are discussed in [PALM83].

The coupling of execution graphs, information processing graphs, and PAWS models form an effective methodology for modeling system designs. The application of those tools and techniques, together with the workload specification and the resource environment, yield an evaluable performance model. An evaluation of the simulation results should reveal areas in the design which, when appropriately modified, will result in a measurable performance improvement.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

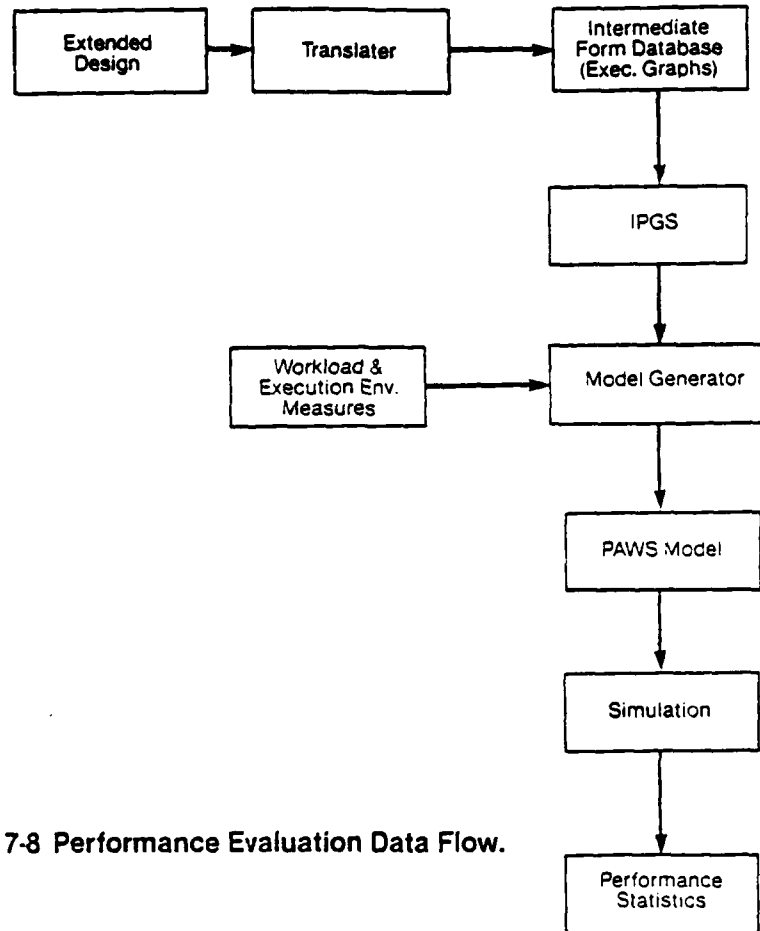


Figure 7-8 Performance Evaluation Data Flow.

7.4 MODELING THE SYSTEM STRUCTURE

In this section we will discuss some of the special problems likely to be encountered in modeling distributed, reliable systems.

7.4.1 Issues in Modeling Distributed Systems

Communication and coordination overhead are two additional considerations in modeling distributed systems that are not present to such a large extent in centralized systems.

Communication overhead is the expense incurred in sending and receiving messages over the communication medium. Communication costs are generally quite high and, therefore, tend to have a relatively large negative impact on the performance of distributed systems.

Coordination overhead is the expense incurred by a system in ensuring that the independent processes and transactions of which it is comprised cooperate with one another to carry out common goals. This includes such things as locking, mutual exclusion, and concurrency control in general.

7.4.1.1 Low-Level Communications

It is expected that the modeling of low-level communication mechanisms such as local-area networks and internet message traffic will be fairly rare. Designing a system totally from scratch is not done very often for obvious reasons and it is likely that most design efforts will utilize some predefined communication system as a basis. In many cases, sufficiently accurate performance results will already be available for those subsystems obviating the need to model them.

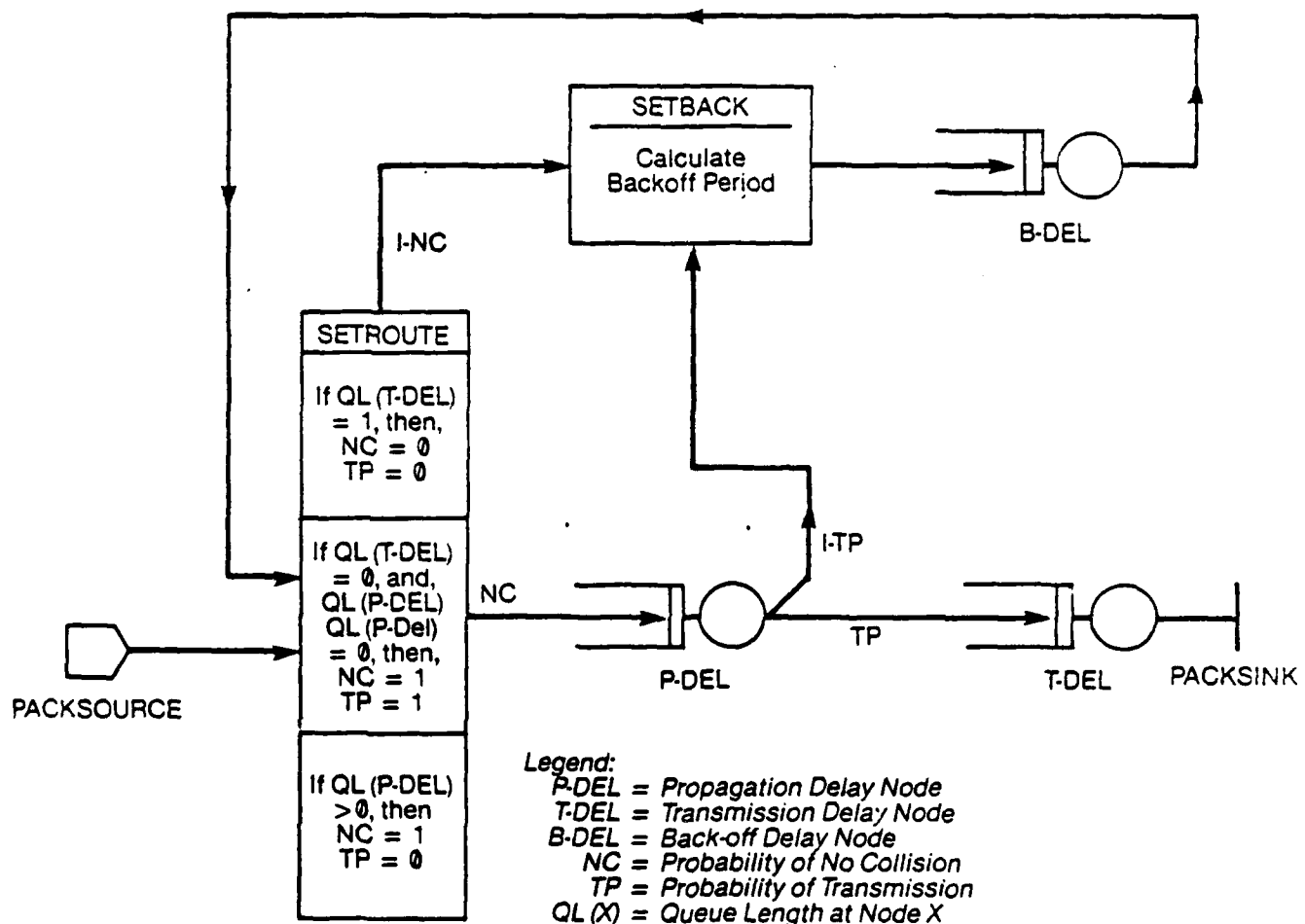


Figure 7-9 CSMA/CD Protocol Model.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

Even so, it may be instructive to examine a simple model of one such low-level communication sub-system whose performance characteristics have already been extensively studied. The Carrier Sense Multiple Access/Collision Detection (CSMA/CD) class of protocols for local area networks was popularized by the ALOHA [ABRA73] and Ethernet [METC76] commercial networks.

The model whose IPG is shown in Figure 7-9 is a simplified model (but at a rather detailed level) of a generic CSMA/CD protocol. The basic notion behind such a network protocol is that a broadcast (a message delivery) has two distinct phases: propagation and transmission. During the former, packet collisions can occur where two or more network users (referred to here as hosts) attempt to use the medium at the same time. During the latter phase, the carrier sense mechanism prevents such collisions. If either a collision occurs or a host finds that the medium is busy when it wants to send a message, then that host (all hosts in the case collisions) will simply "backoff" by delaying for some random time before trying again.

The model in Figure 7-9 models the carrier detect and collision mechanisms by interrogating the lengths of the queues representing the network medium. New "packet" transactions are randomly generated at the node named PACKSOURCE and then proceed to the compute node named SETROUTE. This node looks at the lengths of the queues P-DEL and T-DEL using the function QL(q). If the length of the T-DEL queue is exactly one, this indicates that there is a packet currently being transmitted and, mimicking the carrier sense mechanism, the model sets the branching probabilities NC and TP so that the new packet proceeds to the backoff period calculation node SETBACK. If, on the other hand, no packet is currently being transmitted and there are no packets waiting in the P-DEL queue, then the branching probabilities are adjusted so that the new packet may proceed instead to the P-DEL queue to await transmission.

The third possibility at SETROUTE is that no message is currently being transmitted (no carrier is sensed) but there is already one or more packets waiting to be transmitted in P-DEL. This situation would not, of course, be detected in the actual system and the model reflects this by allowing the new packet to proceed to the P-DEL queue but not on to be transmitted. At some point, the contention for the medium will be detected and the packet will proceed on to node SETBACK and thence to the the backoff delay node P-DEL.

The important performance measure for this model is the average lifetime (in simulated time units) of a transaction from its creation in PACKSOURCE to its deletion in PACKSINK. This corresponds to the mean response time of the network to send-packet requests.

7.4.1.2 Remote Procedure Call

The next higher level of communication protocols in distributed reliable systems is likely to be some sort of mechanism which provides a common interface for remote services and for local procedure calls. The overhead involved with such remote procedure call (RPC) mechanisms might be classified

either as communication or coordination overhead. In one sense, RPC is just a higher level communication protocol and, in another sense, it is a coordination mechanism which enforces strict procedure-call semantics on interactions between independent processes.

As an example of a simple protocol that implements synchronous procedure-call semantics in a message-based system, consider the following:

- On the caller's (client's) side,
 - 1) The caller requests service by sending a REQ message containing the required parameters for the call.
 - 2) On sending the REQ message, the caller resets a timer and then waits for the response message.
 - 3) If the timer expires before the response message is received, the caller retries by sending another REQ containing exactly the same parameters.
 - 4) On receiving the response message RESP from the callee, the caller records the return parameters contained in that message and then sends an acknowledgement message, ACK.
- On the callee's (server's) side,
 - 1) On receiving a request for service message, REQ, from the caller, the callee records the input parameters of the call and, at some future time, performs the requested service.
 - 2) Upon completion of the requested service (or when an error condition is encountered), the callee sends a RESP message containing the completion status of the call and return parameters, if any.
 - 3) When the RESP message is sent, the callee resets a timer and then waits for an acknowledgement message, ACK.
 - 4) If the timer expires, the response is retried by sending another, identical (except perhaps for a sequence number) RESP message to the caller.
 - 5) Upon receiving the ACK message, the callee discards the recorded return parameters and response and forgets about the call.

This is a very simple reliable RPC protocol which has some rather obvious and serious drawbacks. First, the number of retries that the protocol will initiate if one of the processes fails permanently is infinite. This is fairly easily rectified. A more serious flaw in the protocol is that the response message is being used for a dual purpose: to convey the return parameters and to acknowledge receipt of the original request. This means that choosing an appropriate timeout value for the REQ timeout might be difficult unless there is some well known upper bound on the time required to process a request.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

Nevertheless, the above distributed algorithm will serve to illustrate some of the issues of modeling RPC. Figure 7-10 shows an IPG which models the protocol. The graph is meant to illustrate the operation on a single host or process in the system; a host which may act as either a caller or as a server. An entire network could be modelled by forming a array of the given model, one for each host in the configuration. The edges labelled IN and OUT are assumed to be connected to a model of the low-level communications system or to a delay node that summarizes its performance characteristics.

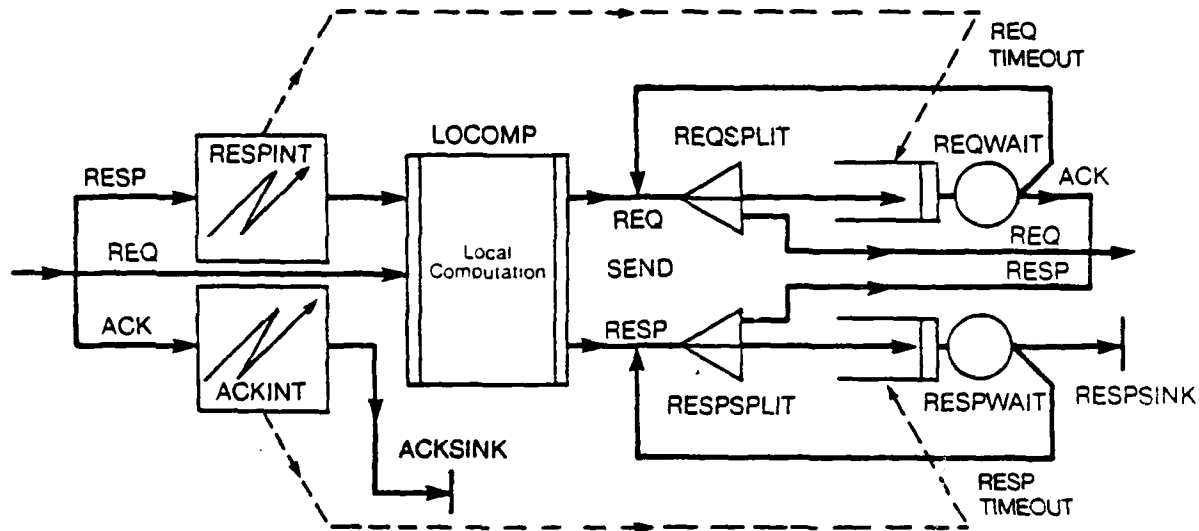


Figure 7-10 Reliable Remote Procedure Call Model.

A process (local computation) that wishes to make a remote call creates and sends a transaction whose phase is REQ to the node REQSPLIT. This transaction would contain, in its local variables, some indication as to the destination host/process, the source host, the type of call, and the size of the input parameters for the call. The split node REQSPLIT spawns a sibling transaction (which inherits all the local variable values of its parent) and sends the newly spawned sibling out over the network as the request message.

The original REQ transaction is sent to a delay node, REQWAIT, where it waits for either an interrupt from the node RESPINT or the expiration of a timer. If the timer expires first, the REQ transaction is sent back to REQSPLIT where it spawns another sibling transaction and retries the request by sending it over the network. This process is repeated indefinitely until a response comes from the server.

When a transaction whose phase is RESP arrives from the network submodel, it is routed through the interrupt node RESPINT where it interrupts the waiting REQ transaction and changes its phase to ACK. The ACK transaction immediately leaves the REQWAIT queue and is sent to the server to acknowledge

the response. The RESP transaction itself, which contains in its local variable the result of the remote operation, then proceeds back to the local computation.

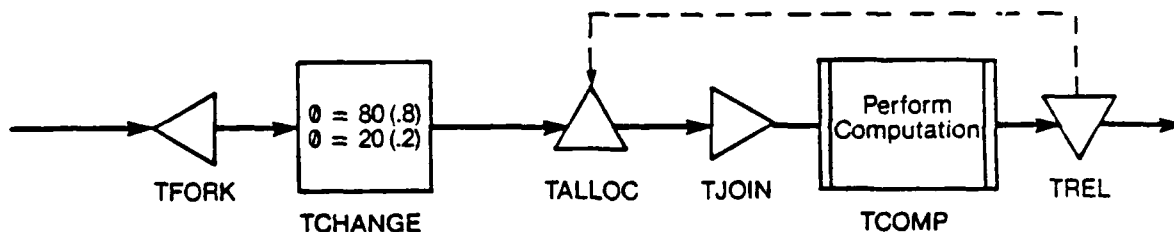
On the other side of this interaction, the process that receives and processes a remote procedure call behaves as follows. When a transaction with phase REQ arrives from the network submodel, it is immediately routed to the local computation which either acts on it or saves it until some later time. When the operation is completed by the local computation, it creates and sends a transaction whose phase is RESP and whose local variables contain the result of the operation. This RESP transaction first goes through the split node RESPSPLIT which spawns a sibling RESP transaction and sends it to the caller over the network. The delay node RESPWAIT waits for acknowledgement or else times out in a way which is exactly similar to the operation of the REQWAIT node described above.

When a transaction arrives with a phase of ACK, it is routed through the interrupt node ACKINT where it interrupts the waiting RESP transactions and allows it to flow into the sink node RESPSINK and be destroyed. The ACK transaction itself is then destroyed in the sink node ACKSINK.

7.4.1.3 Concurrency Control

There are several ways to protect against the situation where two or more concurrent processes update the same data objects at the same time. Of these, the technique of locking is probably the most straightforward and conceptually simplest. Contention for these locks by concurrently executing processes is an important aspect of the system for performance analysis purposes.

Locking strategies may be modelled fairly easily using representations such as IPGs and PAWS. This is done using the passive resources or tokens provided by these modeling tools. A data object is considered to be locked by a transaction if that transaction possesses a token of the correct type.



Talloc Operation

—Initialized with
80% Blue Tokens, and,
20% Red Tokens.

—Transaction with
Phase = 80, Request One Red Token,
Phase = 20, Request One Blue Token.

Figure 7-11 Simple Concurrency Control Model.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

If each data object is being represented separately in the model, then a different token type for each object is required to provide locking. Generally, however, it is desirable to model the system at some higher level of detail so that the data objects are not represented individually but rather collectively. The challenge then is how to model contention for the objects if they are not identified individually in the model.

One simple solution to this problem takes advantage of a phenomena observed in database access pattern. That is that, of a given class of data objects, 20% of the objects will be accessed 80% of the time. Figure 7-11 illustrates a model which utilizes the 80-20 rule to represent contention for objects in a concurrent system.

The general idea of the IPG is to probabilistically change the phase of the transaction that is requesting a lock on an object (or objects). In the change node TCHANGE, the phase of the incoming transaction is changed to 80 (an arbitrarily chosen number) with probability 0.8 and to 20 with probability 0.2. In other words, 80% of the transactions leaving TCHANGE will have their phase=80 while the remainder of the transactions will have phase=20.

The allocate node TALLOC contains a number of tokens which are of two different types, blue and red (again, these are arbitrarily chosen symbols). The number of such tokens in TALLOC is initialized so that 80% of the total number of tokens (of both types) are blue and 20% are red. It is then a simple matter to specify that the transactions entering TALLOC with phase=80 will request one (or more) of the RED tokens, while transactions with phase=20 will request BLUE tokens. If all the tokens of the requested type are currently being held by other transactions, the requesting transaction must wait in the queue until tokens become available.

The tokens thus obtained are released by the transactions after the required computation has been performed on them in TCOMP. This is accomplished in the release node TREL.

The fork and join nodes TFORK and TJOIN are used to save and restore the original phase of the entering transaction before and after the object allocation. Notice that only one child transaction is created at TFORK. This child is destroyed at TJOIN after turning over the tokens obtained to its parent, the original transaction, which then proceeds to the computation subgraph. This form of "context switching" within the models is a very common and useful trick.

7.4.2 Issues in Modeling Reliable Systems

The reliability mechanisms discussed in Chapter 4 can be classified loosely into one of two possible categories: mechanisms related to transactions and recoverable processes and those related to the replication of objects. Analyzing these mechanisms from the performance viewpoint is a rather more complex task than performance analysis for other aspects of the system. For one thing, it is likely that a more complex set of measures will be of interest in modeling reliability mechanisms. Designers will probably be

interested in evaluating models of some of the subsystems both with and without the reliability mechanism models in place. This is to obtain some estimation of the overhead required to provide reliability to the subsystem.

Another reason why modeling reliability mechanisms is potentially a more complex task is that there is a close interaction between these models and the models of the environment of the system. In particular, the ways in which faults are introduced into the models will impact their complexity. Although the effects of faults on the long-term performance of the system will probably be negligible (assuming the faults are very rare), it is often the case that some of the measures of interest in the performance analysis of integrity mechanisms will concern the additional time and resources used by the system during recovery from fault situations.

In order to exemplify some of the techniques used in modeling reliability mechanisms, we will discuss two models which have been simplified and abstracted from the models used to evaluate the Zeus system performance.

7.4.2.1 Transactions and Commit Protocols

The first of these models, illustrated in IPG form in Figure 7-12, is of a variety of transaction commit protocol call two-phase commit. Two-phase commit protocols are used in order to minimize the period during which a data object is vulnerable to the failure of the transaction coordinator. For a more detailed discussion of this and other commit protocols, see Chapter 4.

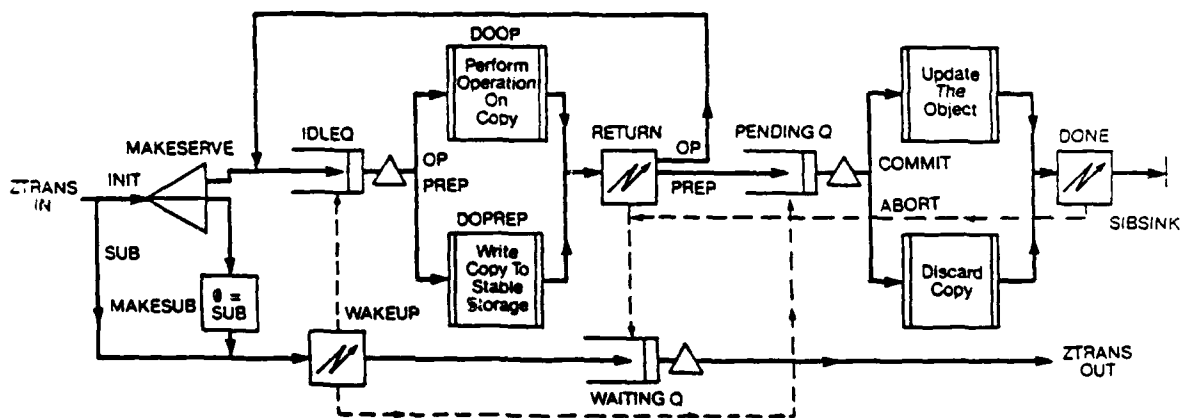


Figure 7-12 Two-Phase Commit Protocol.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

The crux of the protocol is that a data object which has been modified by a transaction may still choose, unilaterally, to abort the entire transaction at any time during the transaction until it receives (and acknowledges) a special operation request called PREPARE. After that point, the object must neither commit nor abort the operations performed on behalf of the transaction unless and until it receives further instructions from the transaction coordinator.

There is an unfortunate collision of terminology in this particular discussion since PAWS transactions are not at all the same thing as the transactions which serve to group a number of macro operations into single atomic actions. We will try to minimize the confusion in the ensuing discussion by referring to the latter as ZTRANS and continuing to use the word "transaction" to mean PAWS transactions

The model of Figure 7-12 represents a single server which is equipped to participate in a two-phase commit protocol. In the model, the ZTRANS enters the IPG from the point labelled ZTRANS IN. On entering a server for the first time, a ZTRANS will have the phase INIT meaning that this is the initial request for this ZTRANS. The ZTRANS is then routed through the split node MAKESERVER where a sibling transaction is spawned. This sibling will perform all of the operations requested by the ZTRANS of the local server. The original transaction, the ZTRANS, then enters a change node, MAKESUB, where its phase is permanently changed to SUB for subsequent request.

The sibling transaction, after having been created, moves to the node IDLEQ where it waits to be interrupted to perform a task for its master, the ZTRANS. The ZTRANS transaction on the initial request as well as on subsequent ones, must pass through the interrupt node WAKEUP in order to awake the sibling transaction and inform it of the operation to be performed.

On being awakened by a ZTRANS, the sibling leaves the IDLEQ and, for normal operations, proceeds to the submodel labeled DOOP to actually do the indicated operation. When the operation is completed on the local server, the sibling passes through the interrupt node RETURN where it informs the ZTRANS (which is waiting in the WAITINGQ) that it has finished. The sibling then returns to the IDLEQ to await further requests by the ZTRANS.

When the ZTRANS is ready to terminate, it will send a special operation request called PREP or prepare to the local sibling. This operation causes the sibling to enter the submodel DOPREP where, among other things, the copy of the object which had been kept in volatile (or non-volatile) storage is moved to stable storage. After informing the ZTRANS that the prepare operation is complete, the sibling, rather than returning to the IDLEQ, instead enters the second phase of the two-phase commit by proceeding to the node PENDINGQ. At this point, the sibling transaction will respond only to requests by the ZTRANS to either commit or abort the copy of the object.

If the decision of the coordinator of the ZTRANS is to abort, then the copy of the object which was stored in stable storage is discarded and the sibling is destroyed in the sink node SIBSINK. The destruction of the sibling means that the ZTRANS has been forgotten by the local server, any further

requests will cause a new sibling to be created and will be treated as a different ZTRANS.

If, on the other hand, the coordinator makes the decision to commit the ZTRANS, then the sibling transaction uses the resources necessary to move the data in the previously stored copy into the object itself, thus permanently recording the ZTRANS' updates. The sibling (and all memory of the ZTRANS) is destroyed in this case as well.

The average time that the sibling remains in the PENDINGQ is a particularly interesting measure with this model since this corresponds to the period during which the local server is totally dependent on the coordinator of the ZTRANS.

7.4.2.2 Object Replication

The second major category of reliability mechanisms is the mechanisms that support and maintain consistency of multiple replicated copies of data objects. A very simple model that might help to estimate the maximum overhead that would be involved in updating multiple copies of objects is shown in Figure 7-13.

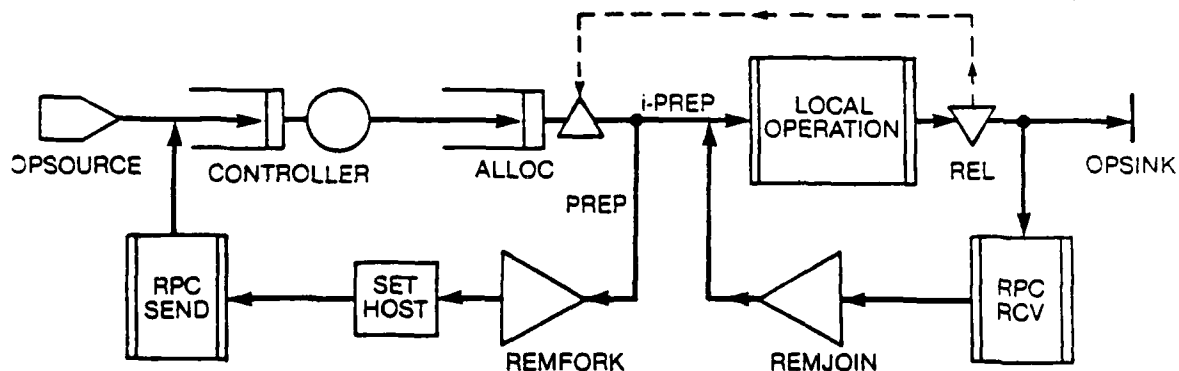


Figure 7-13 Object Replication.

The IPG shown represents, in this case, one member of an array of servers each of which maintains one copy of the data object in question. The actual number of copies is determined by the probability PREP, the probability of replication. If there is only a single copy of the object, then a request

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

generated at OPSOURCE continues through the CONTROLLER node, is allocated a token (representing locking the object) in the node ALLOC, and then moves directly on to the local operation submodel. When the operation is complete, the transaction passes through the node REL where it releases the token (the lock) and then is destroyed.

If, on the other hand, the object is replicated, then it will go to the fork node REMFORK after being granted the lock but before actually executing the local operation. REMFORK spawns a single child, representing a remote request, which then moves to SETHOST where the number of the host (that is the index in the array of submodels) upon which the remote request is being made is obtained (probabilistically). The remote request transaction passes through the RPCSEND submodel (or approximation thereof) and then reenters the model at the CONTROLLER node, exactly as the original request had done.

Eventually, a request will pass through to the local operation node, perform the operation, and then release the object lock. At that point, if the transaction is a remote request, it will proceed to the RPC RCV submodel rather than entering the OPSINK node and being destroyed. After the overhead of the RPC receive has been accounted for, the remote transaction enters the join node REMJOIN. Here, it revives its parent which is then allowed to proceed to its own local operation node (on its own host) and to subsequently complete.

The general principle that is at work in this model is recursion. The calls made to remote hosts to perform the specified operation on a replicated copy are essentially recursive calls to the model. The type of recursion used here could be classified as "head recursion" since the local operation is performed only after the recursive call returns. This method of modeling naturally recursive processes is of general interest and comes up often in modeling distributed systems.

7.5 MODELING THE ENVIRONMENT

The "environment" of a computer system is the pre-existing part (hardware and/or software) of the system. This is the part over which the system designer has little or no control but which affects the performance of the system nevertheless. Often the environment is not modelled in detail but it nonetheless influences the structure and parameters of the system model and, indeed, the system itself.

One aspect of the environment that requires particular attention in the case of performance analyses of distributed, reliable systems is the possibility of fault occurrence within the system. A "fault" is defined as any situation where a low-level hardware or software component (i.e., a part of the environment) ceases to operate according to its specifications. A fault in the environment may result in a "failure" in one or more parts of the system being modeled.

There are many different classes of faults that may be of concern to the system modeler. In Chapter 2, the term "fault model" is defined as the set of

all fault classes that are of interest to the system designer. A discussion of a number of generic fault classes may also be found in that chapter. These generic classes of faults are resource faults, host/site faults, communication link faults, and network partition faults.

Many of the problems in incorporating faults into a performance model are very specific to the system actually being modeled. Nevertheless, a couple of examples from the Zeus modeling effort (which modeled faults at a rather abstract level) may serve to exemplify some of the modeling techniques that are at the designer's disposal. Of the four classes of faults mentioned in the previous paragraph, the first two, resource and host faults, are limited enough in scope so that manageable IPGs can be abstracted from the Zeus models. Therefore, we will present and discuss models which serve to introduce these two types of faults into a system periodically.

7.5.1 Modeling Single Resource Faults

A very common way to model local resources such as secondary storage devices is with the "central server" model illustrated in Figure 7-14 (for the case of a disk resource). A request for service enters the sub-model at REQ IN and then uses some CPU and some DISK resources. The request then loops and repeats this cycle with some probability P or completes and exits the sub-model at REQ OUT with probability 1-P.

Synchronous calls to local resources will occasionally fail due to a fault in the resource. The standard central server model can be modified to include faults by adding a mechanism to introduce the faults and "shut off" the resource and some means of reporting the result of the operation, the return status, to the caller. Figure 7-15 is the IPG of a faulty resource model which returns the status of the operation in the phase of the request transaction. Alternatively, this could be done using a local variable of the transaction. The rest of the model is essentially like the central server model (the two nodes CPU and DISK have been replaced with the single node RESOURCE to simplify the presentation) with requests for service entering at REQ IN and then cycling through the RESOURCE and RETSTAT nodes and eventually leaving through REQ OUT.

Faults in the model take the form of separate transactions that are generated at the source node FSOURCE. The interarrival time for the fault transactions are, as for other transactions, drawn from some statistical distribution whose mean and perhaps standard deviation are parameters of the model. The fault transaction first sets the value of a global variable, FAULT, to true. This has the effect of shutting off any subsequent calls to the faulty resource by forcing them to bypass the RESOURCE node from FBRANCH and then setting their phase to ERROR and immediately flushing them out of the sub-model. Requests that are actually receiving service from RESOURCE at the

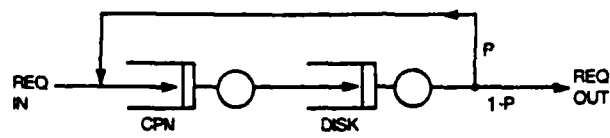


Figure 7-14 Central Server Model

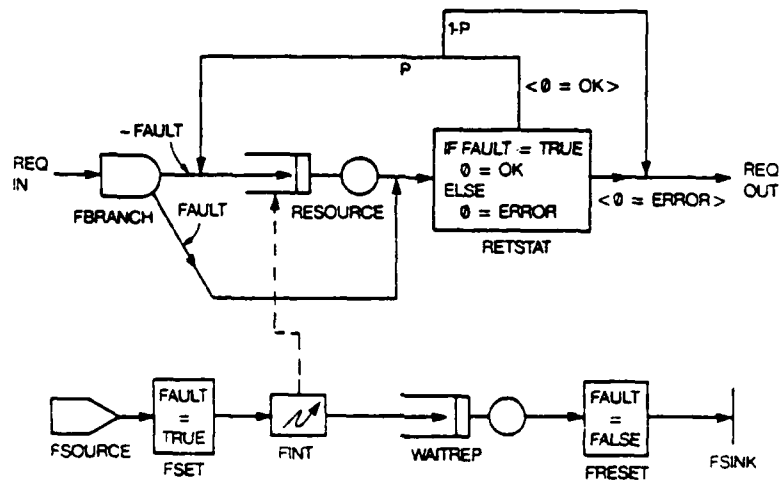


Figure 7-15 Modeling a Faulty Resource

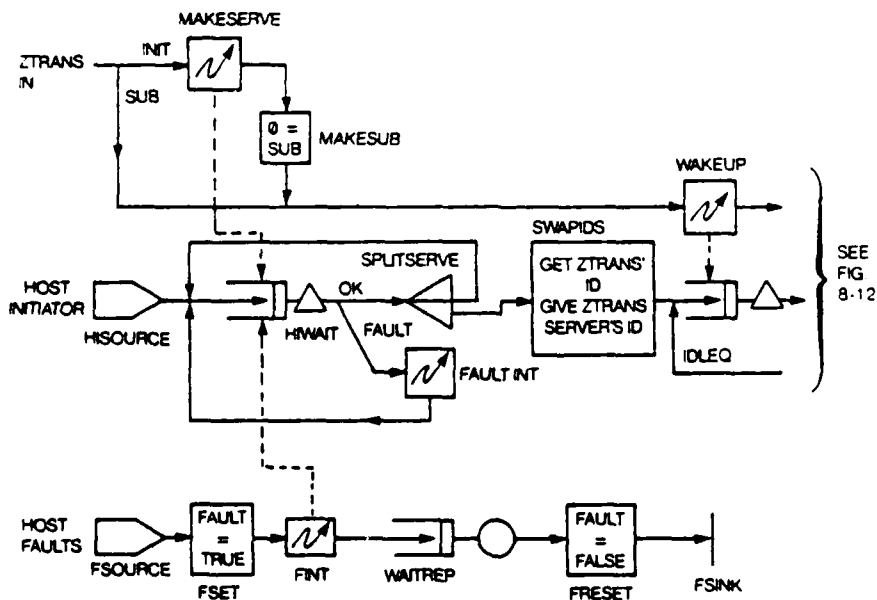


Figure 7-16 Host/Site Faults

time the fault occurs are forced out of the queue when the fault transaction enters the interrupt node FINT. The change node RETSTAT changes the phase of any request transaction passing through the sub-model to ERROR indicating that the resource is not currently working.

After purging all the currently active transactions and arranging that any subsequent requests will be immediately failed, the fault transaction enters the delay queue WAITREP where it waits for some random amount of simulated time after which it resets the value of FAULT to FALSE thus turning the resource back on and allowing subsequent requests to be serviced normally. Finally, after the repair time has expired and the fault transaction has reset the FAULT variable, it is destroyed at the sink node FSINK.

Notice that the conventional fault measures of Mean-Time-To-Failure (MTTF) and Mean-Time-To-Repair (MTTR) have a very natural interpretation in this model. The MTTF is simply the mean interarrival time of the fault transactions which is a parameter of the FSOURCE node. The MTTR is the mean delay time parameter of the WAITREP node. The idea of modeling faults in the system as separate transactions which are generated periodically and then change the behaviour of parts of the model is appealing and fairly pervasive in the Zeus performance models. The technique is used again in a slightly more general way to model faults which affect sets of resources rather than the single devices modeled here.

7.5.2 Modeling Host/Site Faults

In the faulty resource model of Figure 7-15, it is assumed that the fault transaction somehow knows the ID of all the transactions which are receiving service at the time of the fault. This is requirement of PAWS in order for the fault transaction to be able to interrupt the active transaction at FINT. Other modeling languages could conceivably allow one transaction to interrupt all the transactions currently residing at a given node and thereby relieve this requirement. However, a more complex model might include many such queues where transaction may be waiting thus making the problem of finding them and notifying them of the fault even more difficult.

In Zeus, the model of a single host contains as sub-models IPCs for each of the type managers residing on that host. Each type manager in turn may be servicing many concurrent requests at any given time. This means that transactions of various classes and in various phases of their computation will be using resources at many different points in the overall host model at any time. The problem of interrupting or otherwise notifying each of these transactions in the event of a host fault becomes a major modeling concern at this point.

In order to solve this problem it is necessary to ensure that all of the transactions making their way through a host model at any given time are related to one another in some way. In PAWS, the simplest solution is to make sure that all the transaction are siblings of each other. This allows any one of the siblings to interrupt all of the others and to change their phase (to an error phase for example) regardless of where they currently are in the host model and without the need to know their transaction IDs.

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

In order to show how this might be done using IPGs, Figure 7-16 borrows part of the generic type manager model that was presented in Figure 7-12 to illustrate the two-phase commit protocol. Recall that in that IPG a Zeus transaction (ZTRANS) creates a server transaction in the type manager model on its first request of the type manager. Subsequent requests by the ZTRANS of that same type manager then would simply interrupt the existing server transaction changing its phase to reflect the type of operation that was required.

Figure 7-16 shows the path of the ZTRANS transaction on the top of the figure. The ZTRANS enters at ZTRANS IN and, on its first request, proceeds to the interrupt node MAKESEVERE. In the previous model of the type manager, the server transaction was created directly by the ZTRANS in a split node (the split node MAKESEVERE in Figure 7-12). Here we are instead arranging that an intermediary transaction, the host initiator, actually create the servers for all of the ZTRANS on all of the type manager sub-models associated with the same host.

A single host initiator transaction is created at the source node HISOURCE and then remains in existence through the rest of the modeling run. When a ZTRANS enters a type manager sub-model for the first time, the MAKESEVERE interrupt releases the host initiator from the queue HIWAIT and instructs it to create a server transaction with the appropriate characteristics. The host initiator (HI) creates the server in the split node SPLITSEVERE. Consequently, the desired effect that all the server transactions be siblings of one another is achieved because, in fact, they are all siblings of the single HI transaction. After a new server is created, it passes through the compute node SWAPIDS where it arranges that its transaction ID be stored in the local variables of the ZTRANS which had it created and, in addition, stores the ID of the ZTRANS in its own local variables. This enables the WAKEUP and RETURN interrupts used in Figure 7-12 to operate exactly as they originally did when the ZTRANS itself created the server. In fact, after swapping transaction IDs with the ZTRANS, the server proceeds to the wait queue IDLEQ where it awaits further instructions and operation requests from the ZTRANS via the WAKEUP interrupt exactly as in the original model. The host initiator, after having created the server, returns to the HIWAIT queue and awaits other server creation requests.

Host faults are generated in the same way as for single resource faults (Figure 7-15). Fault transactions are generated at FSOURCE and then set the value of the global variable FAULT to true indicating that the host is currently in a faulty state. Rather than directly interrupting the currently active servers in order to notify them of the failure, however, the fault transaction only interrupts the local host initiator (it is assumed that the ID of the HI is well known) changing its phase to FAULT. The HI in turn enters the interrupt node FAULTINT where it interrupts all of its living siblings, which is to say, all of the server transactions that are currently doing work on the local host.

The particular action taken by a server transaction after being notified that the local host has become faulty is not shown in this example. Paths must be provided from each of the queues in the host model so that such failed

servers behave in the desired manner. In some cases, the server may simply enter a sink and be destroyed in the event that its host fails. In other cases, it may be desirable to send the server to a wait queue where it will await notification that the host fault has been repaired and then execute a recovery action. If this is done, another interrupt node could be added in the path of the fault transaction so that all the HI siblings (the ones that didn't simply die when the fault occurred) could be notified of the repair.

7.6 MODELING THE WORKLOAD

The performance of a computer system naturally depends on the nature and frequency of the service demands that are made upon it for a given application. These demands are normally referred to collectively as the workload of the system.

As indicated above, a fully specified workload consists of two parts: the nature of the jobs that the system will be expected to perform, and the frequency with which they will arrive. Typically, the arrival of jobs for processing is modeled using a random variable drawn from some probability distribution (e.g., Poisson). The value of this variable is used as the interarrival time between successive jobs. Where the system being modeled is operational, the arrival rate of jobs may be obtained from a trace of a real job stream. Of course, in the case of performance evaluation during the design phases (our primary concern here), this is impossible and the technique of drawing the interarrival times from a standard distribution is both computationally convenient (for analytic and simulation evaluation) and usually fairly accurate.

In addition to the job arrival model, we must also represent the work demands made by the individual jobs themselves. The demands of a job on a computer system must be specified in terms of the number and amount of resources that the job uses or will use. The term "resource" here is used to refer to any system-provided service facility and will in general include at least the following:

- o The central processor,
- o Any peripheral or secondary processors used,
- o Memory space,
- o Any I/O devices used directly by the job, and,
- o System-provided software components.

In the case of the last item in the list, software resources, the resource utilization of these components may be further decomposed so that the performance of the system-provided modules may be modeled and studied at the lower level. This is an important form of the hierarchical model structuring which was advocated in Section 7.2 and it is of particular utility in the case of object-oriented systems (see the following examples).

There are a number of conventional methods for characterizing the workload of a computer system. Of these, there are at least two, instruction mixes and synthetic jobs, which are applicable to design phase workload

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

estimation. We will describe and discuss these two methods briefly and then relate a simple concrete example from the Zeus design effort.

7.6.1 Instruction Mixes

An instruction mix is a distribution of relative frequencies of instruction types (for a given instruction set) observed or expected in a typical application environment of the system in question. For pre-operational systems, instruction mixes may be chosen which cover a range of possible situations or they may be chosen to reflect the observed mixes of existing similar systems. In either case, the instruction mix specifies only the relative frequencies of the various instructions and nothing else (e.g., the order of execution of the individual instructions is not specified).

The level at which the "given instruction set" is defined has a direct bearing on the quality of this technique as a workload characterization. Conventionally, the instruction set used has been the machine language of the system, which is the very lowest level choice. Instruction mixes at this level have a number of drawbacks which are discussed in [KOB78] among others. Machine language instruction mixes do not, for example, include information about utilization of resources other than the CPU such as I/O devices. By defining the instruction set at a higher level, this problem can be alleviated somewhat. For example, in a strictly object-oriented system such as Zeus, the instruction set might naturally be defined as the set of operations (functions, procedures, methods, etc.) provided by the system type managers. Assuming that one knows the resource usage and performance characteristics of these individual "instructions" (perhaps as a function of the parameters of the operations), then any instruction mix would capture information about total resource utilization, not just the CPU.

A more serious and fundamental problem with instruction mixes (at all levels) is that they are only first-order statistics; they completely lack information about serial dependency and instruction overlap. There are, of course, many cases where the sequence pattern of the instruction and data references, not just the relative frequencies, strongly affects the performance of the overall system.

Notice that the job arrival model mentioned in the beginning of this section is essentially an instruction mix where the "instruction set" is the set of user-level jobs defined for the system. The arrival model is simply a frequency distribution for the jobs.

As an example of workload characterization by instruction mixes, consider a command and control environment such as the one described in Chapter 3. This is the environment in which the experimental system Zeus was designed to operate. We will describe a job arrival model for a C2 system. As was pointed out in the previous paragraph, this arrival model constitutes a high level instruction mix where the instructions are taken to be jobs. The equivalence of the arrival model and an instruction mix is especially accurate in a C2 system since the set of possible user-level jobs (the instruction set) is a relatively fixed set.

In evaluating the performance of the Zeus design, we were interested in examining the characteristics of the system with a wide variety of different job types. However, since no application software was designed with the system, we were forced to define a number of generic scenarios in terms of several performance-affecting attributes. We chose four such job attributes and two possible values for each attribute:

- o Duration (short or long) - This is the total number of type manager operations that the scenario will execute. The difference between short and long scenarios is approximately an order of magnitude.
- o Number of Objects Accessed (few or many) - The total number of objects which are either read or written or both by the scenario. Again, the difference in magnitude between "few" and "many" is about a factor of ten.
- o R/W Ratio (R/O or update) - This indicates whether or not the scenario does any update operations on ANY of the objects that it accesses. R/O jobs do not do any update operations whereas "update" jobs do at least one.
- o Object Distribution (single- or multi-site) - If all the objects accessed by a job reside on a single host (not necessarily the same one that the scenario is running on), then the value of this attribute is "single-site", otherwise, it is "multi-site."

Of the sixteen possible generic jobs classes that may be obtained by substituting values for these four attributes, we chose eight based on information about existing applications on other C2 systems. The following table summarizes the attributes of these eight jobs and defines an instruction mix distribution for them:

Job Number	Job Mix (%)	Duration	# Objects Accessed	R/W Ratio	Object Distrib.
1	15	short	few	R/O	multi-site
2	15	short	few	R/O	single-site
3	15	short	few	update	single-site
4	15	short	few	update	multi-site
5	20	short	many	update	multi-site
6	5	long	few	update	multi-site
7	10	long	few	update	single-site
8	5	long	many	update	multi-site

The percentage figures in the job mix column indicate the percentage of the total number of jobs resident in the system at any given time (after a steady-state has been reached). By way of justification for the job mix given here, notice the following things:

- o 80% of the concurrently executing jobs are short - that is, they perform relatively few operations,
- o 75% of the jobs have a relatively small working set (access only a few objects),

PERFORMANCE ANALYSIS TECHNIQUES AND TOOLS

- o Many of the jobs (30%) are read-only.

Although this method of selecting an example workload may seem somewhat ad hoc, it is expected that it will provide valuable performance data that is sufficiently accurate to guide the design process. More importantly for our present purposes, it provides a concrete example of the use of instruction mixes in real design situations.

7.6.2 Synthetic Jobs

Another useful way to characterize the expected workload for pre-operational systems is with synthetic jobs. These are, as the name suggests, artificial jobs for which the resource usage is similar to the expected characteristics of some future real jobs or to existing jobs being run on other systems.

Synthetic jobs are easier to obtain than the real applications because they summarize the resource utilization of the jobs that they are characterizing. Local CPU usage, for example, is usually represented by a simple idle loop in a synthetic job and remote requests are abstracted so that the parameters of the calls are simplified or left out entirely.

As an example of a synthetic user-level scenario from the Zeus performance analysis, consider the following job:

```
Begin Scenario
  Read Intelligence
  Read Navigation
  Read Weather
  Read Mission-Plan
  Computation
  Update Mission-Plan
End Scenario
```

It is assumed that "Intelligence", "Navigation", "Weather", and "Mission-Plan" are objects (or groups of objects) in the command and control system. The semantics of the scenario is meant to resemble a so-called "Mission Control" job which is a typical (although much simplified) job being run on other C2 systems. The scenario reads the appropriate data, does some local computation to determine how the plan of some in-progress mission should be changed, and then updates that mission plan.

Notice how this synthetic job represents the basic performance-affecting features of the scenario in a very stylized and simplified way. The meaning and functionality of the local computation is not specified and neither are the parameters of the remote calls.

7.7 EVALUATION OF THE EXAMPLE SYSTEM

Appendix J presents in detail the evaluation of an example distributed command and control system. This describes the evaluation of some commit protocols using PAWS simulations.

7.8 SUMMARY AND CONCLUSIONS

In this chapter we have discussed the tools and techniques that are available to aid in the performance analysis of distributed, reliable systems. We began by very briefly surveying the field of performance analysis of computer systems, especially emphasizing the issues that were relevant to performance analysis during the design phases. Modeling and model evaluation techniques are the important topics in this regard.

In addition to the general sketch of performance modeling, we also gave a somewhat more detailed account of several of the representational and simulation tools. These were the tools and techniques which were used in the evaluation of an example system: Zeus.

The remainder of the chapter discussed the specific issues involved with modeling the design of a particular class of computer systems: those that are distributed over several sites and that are designed for the purpose of providing highly reliable service.

It should be apparent from the material in this chapter that performance evaluation during the early stages of design and continuing throughout the lifetime of the system can be an invaluable strategy for producing viable, efficient software products.

CHAPTER 8

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

Similar to performance characteristics, the specification and evaluation of reliability characteristics of a design are an important and integral part of the design process for reliable systems. A design process typically consists of several phases starting with the requirements specifications up to the final design meeting those requirements. These phases may involve several iterations of designing and validation until the design meets the desired requirements. For reliable systems, the requirements statements must include the specifications of the desired reliability characteristics of the target system. Typically a design process consists of decomposing the design into a set of sub-problems. In such cases the requirements statements, which include the reliability specifications, are appropriately extended and augmented for each of the sub-systems. The validation task consists of verifying that the target system constructed from those sub-systems, with the given reliability characteristics, has the desired reliability. The goal of this chapter is to describe a method to perform reliability analysis of interconnected systems.

The traditional approach to specifying reliability characteristics is to use certain numerical measures such as availability, mission time, and mean-time-to-failure (MTTF). Chapter 2 described some discrete measures for reliability specifications. These measures imply that a system has certain failure characteristics under a given set of system faults. The measures capture the level of consistency maintained by the system under this set of faults.

There are essentially two approaches to reliability analysis of system designs -- simulation and analysis. One approach is to simulate the system design along with its failure environment and the recovery mechanisms. This approach is inherently expensive because it requires building simulation models specific to the design to be analyzed. This approach provides relatively more accurate results as compared to the second approach because it captures the structure and functioning of the system to a greater detail. The second approach is based on combinatorial analysis of the system based on the reliability characteristics of its components and their interconnections. The reliability characteristics of the components are specified in terms of availability and MTTF. This approach is, in general, faster and less expensive.

The combinatorial analysis method that we describe in this chapter provides quick first-order evaluations of the system reliability characteristics given the system configuration and the reliability characteristics of its components. These methods can be used to construct a general purpose evaluation tool. One such tool called NetRAT (Network Reliability Analysis Tool) is described in this chapter. The evaluations using this method are somewhat less accurate as compared to using simulation models because they do not capture some of the dynamic operating conditions such as execution delays, system load, and resource contention. In this chapter we also present some simple analytical methods for reliability evaluations in the context of some examples.

The following section describes methods for reliability requirements specifications. Section 8.2 describes the network model for reliability calculations. Using two examples, we show how this model can be used for distributed systems. Section 8.3 describes the basic computational approach used in NetRAT for evaluating the availability, reliability, MTTF, and mission time. The user interfaces are described in Section 8.4. The user interface of NetRAT and features of the NetRAT system are generally oriented towards solving reliability analysis problems in distributed systems. The user can interactively change the distribution of resources in the network, create multiple copies of the resources, or move the resources from one node to another. Other user commands permit changing the network topology and modifying the reliability data of network components. Section 8.5 presents some examples that demonstrate the application of NetRAT and some analytical methods to a set of reliability techniques.

The simulation-based techniques for reliability analysis are not described in this chapter but in Section 9.3, where techniques for building functional simulators using Path Pascal are described. Although the descriptions there are oriented towards the validation of functional correctness, the same simulation models can be extended and used for the evaluation of reliability measures of a design. Similarly, the performance evaluation models using PAWS, which were described in Chapter 8, can be extended for the evaluation of reliability measures such as availability and MTTF.

8.1 SPECIFICATIONS OF RELIABILITY MEASURES

Traditionally the reliability characteristics of a system are expressed in terms of certain probabilistic measures such as the availability, reliability, mean-time-to-failure (MTTF), and mean-time-to-repair (MTTR) for repairable systems, mission time, etc.

For a large system, such as a distributed command and control system, rather than specifying the availability and MTTF of the entire system one would be more realistic in individually specifying the reliability characteristics of its services and virtualized resources as seen by the system users. The approach that we follow consists of specifying the reliability characteristics of the functions executed on the system objects. These characteristics for a function will be different for its invocations from different nodes. For example a system service might be available 100% of

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

the time when accessed from one node, whereas the same service might be available for only 90% of the time when accessed from some other node.

The availability $A(t)$ of a system is a function of time indicating the probability that the system is functioning correctly at any given time t . For non-repairable systems, $A(t)$ tends to zero as time t tends to infinity, the limiting value of $A(t)$, denoted by A , if it exists, expresses the expected fraction of time the system meets its specifications. In this guidebook, the term "availability" refers to this limiting value of $A(t)$.

The availability $A(t)$ of a service (function) only indicates the probability of that function being available at any given time t . It does not indicate the probability of that service remaining available continuously over an interval of time. This probability is expressed by the reliability function $R(t)$ (which is a function of time) for that service. The reliability function $R(t)$ for a service (function) expresses the conditional probability that the service remains continuously available (without failing) during the interval $[0, t]$, given that it is available at time 0.

In distributed systems, we are interested in computing the availability of the functions (services), which expresses the probability of that function (service) being available at any random instant of time. A function execution at a node requires access to some resources which are distributed in the network. A successful execution of the function requires that the resources be accessible from the node where the function is being executed. Therefore, the availability of a function is dependent on the availability of (1) the communication paths to the required resources, (2) the nodes holding the resources, and (3) the nodes executing the function.

The mean-time-to-failure (MTTF) for a service in a distributed system is the expected time interval during which that service remains available before a failure occurs. A service fails if it is unable to access any of the required resources or if the node executing the service fails. MTTF is an important measure of reliability in distributed systems because of the possibility of large delays encountered in communication.

In Chapter 2, some of the problems with using the numerical reliability measures for requirements specifications were discussed. There it was pointed out that one of the problems is dealing with small numbers in specifying these measures. Another problem is related to the fact that the reliability measures of the system components are significantly altered in the combat conditions. Under such circumstances the reliability analysis techniques should focus on determining whether the system performs correctly and in a timely fashion if a certain set of resources are unavailable. This leads to specifying a discrete set of reliability levels corresponding to the consistency levels maintained by the system under various fault conditions within the system.

In Chapter 2, four discrete reliability classes for objects were defined. These classes help us in defining reliability requirements of a system in

terms of kinds of failures to be tolerated. These classes are summarized below. In later parts of this chapter we show how one can use NetRAT to validate these characteristics in a design.

Volatile objects are unstable with respect to a set of faults F and are subject to failure if any of the faults in F should occur. Furthermore, subsequent repair of the offending fault can not repair the object. A volatile object becomes permanently inconsistent whenever a fault in the class F occurs.

Non-Volatile objects are stable with respect to some set of faults F , and may or may not fail after a fault in F has occurred depending on the timing of the fault. Again, the name of this reliability class and its characteristics mirror those of non-volatile memory resources such as magnetic and optical disks, drums, tapes, etc. A non-volatile object will remain intact provided that no faults (in F) occur while some operation is being performed on the object, i.e., while the object is "active". Analogously, a disk record can be expected to survive a repairable fault (in the power supply for instance) only if it was not actually being written upon at the time of the fault.

Resilient objects are guaranteed to be recoverable with respect to a fault class F provided the offending fault is eventually repaired. For objects in this class, if a fault occurs while an operation is in progress, then the operation is completely ignored and the state of the object after recovery is the same as it was before the operation was initiated.

Stable objects are the most reliable class of objects. Objects which are stable with respect to a fault class F are guaranteed to eventually recover from a failure regardless of whether the offending fault ever gets repaired. The distinction between resilient and stable objects is that the recovery of resilient objects is contingent upon the eventual repair of the underlying fault, while stable objects are guaranteed to recover even if the underlying fault is never repaired.

8.2 NETWORK-BASED RELIABILITY MODEL

This section describes a network-based approach for representing a system to evaluate its reliability. This model ideally suits for representing distributed system architectures. In the past a considerable amount of work has been done in the evaluation of reliability and availability of paths in network-based systems, particularly in the area of communication networks. Most of this work addresses the problem of pair-wise terminal reliability in communication networks. i.e., given a pair of nodes in the system, determine the availability of the communication path between these two nodes.

In distributed systems, an important generalization of the pair-wise terminal reliability problem considers the availability of paths from a set of nodes in the network to a different set of nodes. For example, a service execution in a network might require access to several resources that are located at different nodes. It is also possible for a service to require access to any one of the several resources distributed in the network. For example, a read operation on a replicated file can be successfully performed

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

if the node executing this operation can reach any one copy of the file. This is referred to as the multi-terminal reliability problem and has been addressed in a recent work [GRNA81]. In [GRNA81] an algorithm is presented which computes the multi-terminal availability from the availability of the network components.

NetRAT is a reliability analysis tool for network-based systems, which facilitates the evaluation of multi-terminal reliability characteristics. The NetRAT system is essentially based on the algorithms described in [GRNA81] [GRNA80]. However, the algorithm presented in [GRNA80] is incorrect. We have corrected this algorithm in [WANG83] and incorporated it into NetRAT. In addition to the availability calculation, NetRAT also permits the evaluation of other reliability measures, such as the reliability function, mean-time-to-failure (MTTF), and mission time. These extensions are described in the next section.

The reliability analysis model underlying the NetRAT system is network-based, and the evaluation procedures are combinatorial. In the network-based model, a system is represented as an interconnection of nodes. The nodes represent the functional units; and the links, which can be either directional or bidirectional, represent the communication paths. Reliability measures such as availability, reliability, MTTF, etc., are associated with these components. In the NetRAT model, a set of functions and resources are assigned to these nodes. Each function requires access to some resources, which can be physical resources or other functions. Functions in the NetRAT model correspond to activities which provide services in real systems; and physical resources in the NetRAT model correspond to data and hardware resources in real systems, such as processors, memory, disks, I/O devices, files, etc.

A node may contain more than one resource or service, and multiple copies of a resource may exist at several different nodes. In case of multiple copies of a resource, any one of these copies can be used to meet the resource requirements of a function. A function may be available at several different nodes. The resource requirements of a function can be combinatorial; for example, a function may require resources (A and B and C) or (B and D).

We illustrate this network-based model using a set of examples. Consider the network shown in Figure 8-1. This network model consists of four nodes 1, 2, 3, and 4. The availability data of these nodes and the interconnecting links are shown in the figure. A function (program) called FUN executes at node 1. This function requires access to resource R1 and R3. Resource R1 is located at two nodes, 2 and 4, and resource R3 is located at nodes 3 and 4. In this example, we are interested in computing the availability of function FUN. In the model shown in Figure 8-1, if a node is available (functioning correctly), then all the resources located at that node are available. Consider another scenario in which a node may be available, but the resources located at it may not all be available. At a given node, the availability of a local resource could be less than 1.0. For example, in the system of Figure 8-1, resource R2 at node 3 is available with probability 0.7 and R3 with probability 0.8. In order to represent this system in the NetRAT model, the network model in Figure 8-1 is changed to that in Figure 8-2. Here resources

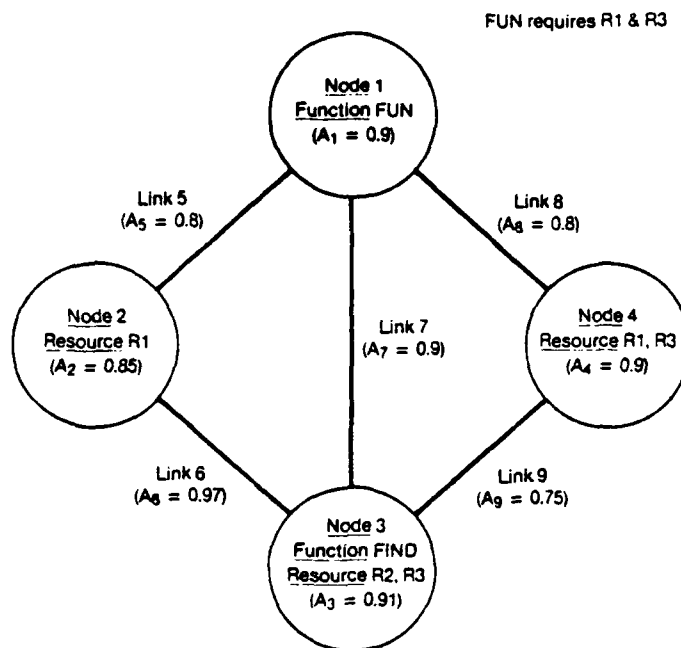


Figure 8-1

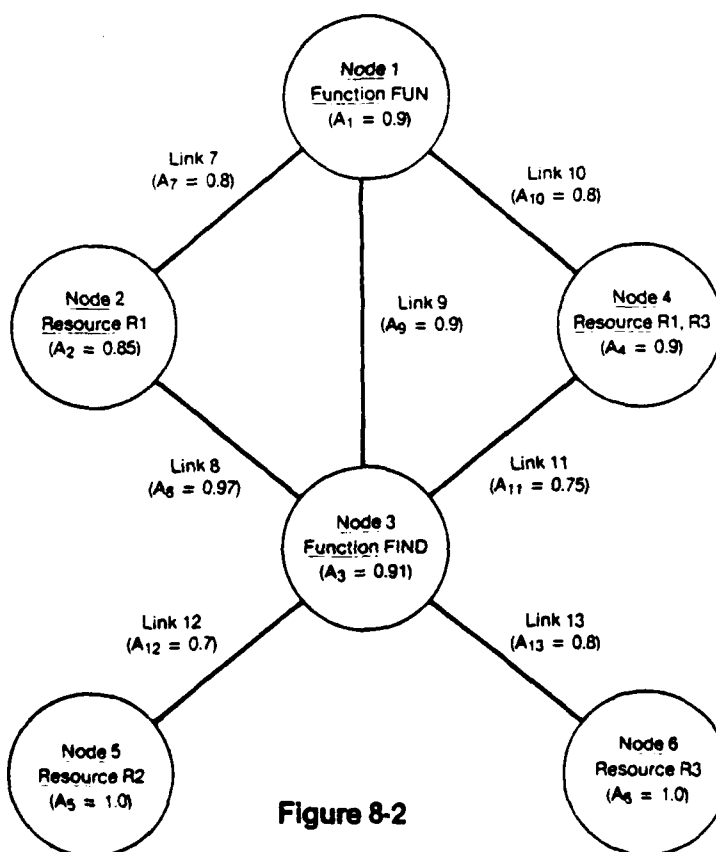


Figure 8-2

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

R2 and R3 are represented as separate nodes (shown as nodes 5 and 6) connected to node 3.

As mentioned earlier, it is possible to include in the resource requirements of the function FUN some other function names. The actual physical resources required for FUN include the union of the resources required by each of the functions whose names appear in the resource requirement of FUN. For example, in a modified scenario the function FUN requires resources R1, R2 and FIND, where FIND is a function that requires the resource R3. Hence the total resource requirement for FUN consists of R1, R2 and R3. Recursive references to function names in the resource requirements are permitted, as long as they can be resolved in terms of physical resource requirements.

8.3 NETWORK RELIABILITY COMPUTATION

In this section, we will describe the NetRAT approach for computing various reliability measures. Detailed derivation of the computation algorithms can be found in [GRNA80], [GRNA81], and [WANG83], and is thus omitted here. The computation approach described in this section is internal to the NetRAT system and invisible to the users. A brief description of this algorithm is given in Appendix I.

As described in Section 2, the network model of NetRAT consists of links, nodes, resource distribution in the network, function locations, and resource requirement of functions. In order for a function to access the required resources, at least one path must exist between the node where the function is executed and each of the required resources. Each of these paths is represented by a sequence of 1's and x's, which represents a cube in Boolean algebra [MILL65]. A '1' in the i-th position of the sequence means that the i-th element (either a link or a node) of the network is included in this path; and 'x' means that it is not included in the path and its availability is of no importance to this path.

Since a function may require more than one resource and each resource may exist in multiple locations, any one of which could be used by the function, the availability of a function is characterized by a sequence of OR's and AND's of paths. This expression represents the dependency of the availability of the system components. We will refer to such an expression as a dependency expression. For example, for the execution of the function FUN in the network shown in Figure 8-1, the alternate paths for FUN to access resources R1 and R2 are:

(e1e5e2 AND e1e7e3), e1e5e2e6e3, e1e8e4e9e3,
(e1e8e4 AND e1e7e3), e1e7e3e6e2, e1e7e3e9e4

where ei is the i-th component in the network. The corresponding dependency expression of the function FUN is

(11xx1xxxx AND 1x1xxx1xx) OR 111x11xxx OR 1x11xxx11 OR (1xx1xxx1x AND 1x1xxx1xx) OR 111xx11xx OR 1x11xx1x1.

Note that the paths in a dependency expression are not disjoint, and thus, the availability cannot be computed by substituting the 1's in the expression with the availability of each component. Therefore, we have to

convert the dependency expression so that it consists of a sequence of OR's of disjoint terms. First, we eliminate the AND operators in the original dependency expression as follows. For the AND of two paths, we AND the paths position-wise, according to the following rule:

$e_i \text{ AND } e_j = \{x, \text{ if both } e_i \text{ and } e_j \text{ are } x\text{'s}\}$
 $\{1, \text{ otherwise.}\}$

For example, $(1x1x \text{ AND } 11xx1)$ becomes $11x11$. For the AND of two OR sequences of paths, we replace the AND operation with an OR sequence of paths, which are generated by pairwise ANDing the paths from the original two OR sequences. For example, $((1xx1x \text{ OR } 1x1xx) \text{ AND } (xx1xx \text{ OR } 1xxx1))$ becomes $(1x11x \text{ OR } 1xx11 \text{ OR } 1x1xx \text{ OR } 1x1x1)$.

After eliminating the AND operations, the dependency expression consists of only a sequence of ORs of paths. However, the paths are still not disjoint. So the next step is to make the terms in the dependency expression disjoint. Suppose now the dependency expression is

$P_1 \text{ or } P_2 \text{ or } P_3 \text{ --- or } P_n$

The final dependency expression will be

$P_1 \text{ OR } (P_2 \$ P_1) \text{ OR } ((P_3 \$ P_1) \$ P_2) \text{ ... OR}$

$((P_n \$ P_1) \$ P_2 \text{ ... } \$ P_{n-1})$

where $\$$ is an "exclusive sharp" operation such that $A\$B$ generates disjoint subcubes that cover every point that is in A but not in B . The algorithm for the "exclusive sharp" operation was originally presented in [GRNA80]. However, [GRNA80] has described the algorithm incorrectly. A correct and more detailed description of the algorithm can be found in [WANG83].

The dependency expression is a symbolic representation of the reliability characteristics that are being calculated. Based on the final dependency expression, which consists of only disjoint terms, we can compute the numerical values of the availability, reliability, mean-time-to-failure, and mission time, from the reliability data of components. Here we only outline the basic steps for calculating various reliability measures in NetRAT. The details are given in [GRNA80] and [WANG83].

Availability: First, convert the dependency expression as follows: Replace each OR operator by a '+' operator. Replace each term by $(\sum_i P_i) \sum_k (1 - \sum_j P_j)$ for all such i 's that the i -th position of the term is a 1, and for all such j 's that the j -th position is 0^k , where 0^k represents a 0 generated in the k -th application of the "exclusive sharp" operation in evaluating a term in the dependency expression. This is explained in more detail in [GRNA80]. Next, substitute P_i by the availability data of the i -th component.

Reliability: In the following computation, we assume that the reliability function of each component is exponentially distributed with respect to time. The first step is the same as the above, except that the availability of each component is replaced by the reliability function of the component. The reliability of a component is a function of time, and therefore, so is the reliability of the system. For example, $(P_1 + P_2 - P_1 P_2)$ will result in

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

$(e^{-t/T_1} + e^{-t/T_2} - e^{-t/T_1} \cdot e^{-t/T_2})$ where e^{-t/T_1} and e^{-t/T_2} are the reliability functions of components 1 and 2 respectively; and T_1 and T_2 are the MTTF of component 1 and 2, respectively. The reliability function computed this way is correct only for non-repairable systems. A repairable system will always perform better than the reliability computed using this function.

Mean Time To Failure (MTTF): The first step here is the same as the first step for the availability calculation. Next, multiply out the expression so that the expression becomes the sum and difference of the products of P_i 's. For example, $P_1 + P_2(1 - P_1) + P_3(1 - P_1)(1 - P_2)$ becomes $P_1 + P_2 - P_2 \cdot P_1 + P_3 - P_3 \cdot P_1 - P_3 \cdot P_2 + P_3 \cdot P_1 \cdot P_2$. Next, replace each term, which is in the form of the product of P_i 's by the reciprocal of the sum of the reciprocals of the mean-time-to-failure of the components in the term. For example, $P_1 + P_2 - P_2 \cdot P_1 + P_3 - P_3 \cdot P_1 - P_3 \cdot P_2 + P_3 \cdot P_1 \cdot P_2$ will result in $MTTF =$

$$\frac{1}{T_1} + \frac{1}{T_2} - \frac{1}{\frac{1}{T_1} + \frac{1}{T_2}} + \frac{1}{T_3} - \frac{1}{\frac{1}{T_3} + \frac{1}{T_1}} - \frac{1}{\frac{1}{T_3} + \frac{1}{T_2}} + \frac{1}{\frac{1}{T_3} + \frac{1}{T_1} + \frac{1}{T_2}}$$

where T_i is the mean-time-to-failure for the i -th component. The above expression for MTTF is basically the time-integral of the reliability function from time 0 to infinity. Note that this calculation is correct only for non-repairable systems. A repairable system will always have a larger MTTF than the value computed this way.

Mission Time (MT): Equate the reliability expression with the required probability, then the solution to the equation is the mission time. Note that the reliability expression is a decreasing function of time. The equation can be solved by simple numerical methods such as the false-point method.

8.4 NetRAT USER INTERFACE

NetRAT is an interactive system. The user interface is designed such that a user can conveniently modify the configuration of the network and observe the changes in reliability characteristics. This is particularly useful in designing a network to satisfy some reliability requirements. The following functions are provided by NetRAT to change system configuration:

- o Nodes in the network
- o For each node, the neighboring nodes connected to it by its outgoing links
- o Resource and function locations
- o Resource requirements of each function
- o Availability of each node and link
- o Mean-Time-To-Failure of each node and link (exponential failure time assumed)

The resource requirements specifications are in the form of boolean expressions involving resource names. For example, the resource requirements for a function can be of the following form:

((A and B and C) or (A and D) or (B and D))

This states that this function can be performed successfully only if 1) all the three resources A, B and C are available, or 2) both A and D are available, or 3) both B and D are available. If all these three conditions are false, the function can not be performed. After the data is read into the system, the user can specify the function to be analyzed and the reliability measures of interest, such as availability, reliability, MTTF, and mission time. NetRAT then returns the numerical value of the specified reliability measures.

The user can modify the network configuration specification interactively. Such changes include:

- o Deleting a resource or function from a node
- o Adding a resource or function to a node
- o Deleting a link
- o Adding a link between two nodes
- o Deleting a node (implies deleting its links)
- o Adding a node
- o Changing the resource requirements of a function
- o Changing the availability or MTTF of a link or a node.

8.5 ANALYSIS OF SOME RELIABILITY TECHNIQUES

In this section, we present the reliability analysis of some reliability techniques presented in Chapter 4 of this volume of the guidebook. The first example analyzes the stable storage system described. The second example describes the analysis of ethernet-like systems. The next example deals with the analysis of some replication management techniques.

8.5.1 Analysis of Stable Storage

In this section we present an analysis of the stable storage mechanism described in Chapter 4. In the analysis we will consider two types of error conditions in a disk system: transient errors and crashes. In case of a crash, it is assumed that a set of pages on the disk may be destroyed. The transient error may affect only the page currently being written. Two operations, CarefulPut and CarefulGet, are defined on the disk pages which try to recover from the transient errors occurring during page read and write operations.

CarefulPut operations repeatedly writes and then reads a page until the read operation returns OKAY status. This ensures that the effects of transient errors during a page write operation are eliminated. If a disk crash occurs during a CarefulPut, the operation is aborted. A stable page is constructed from two ordinary disk pages and the CarefulPut operation. To write a StablePage, both disk pages are written, one after another, using the CarefulPut operation. See Chapter 4 for a detailed description of these techniques.

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

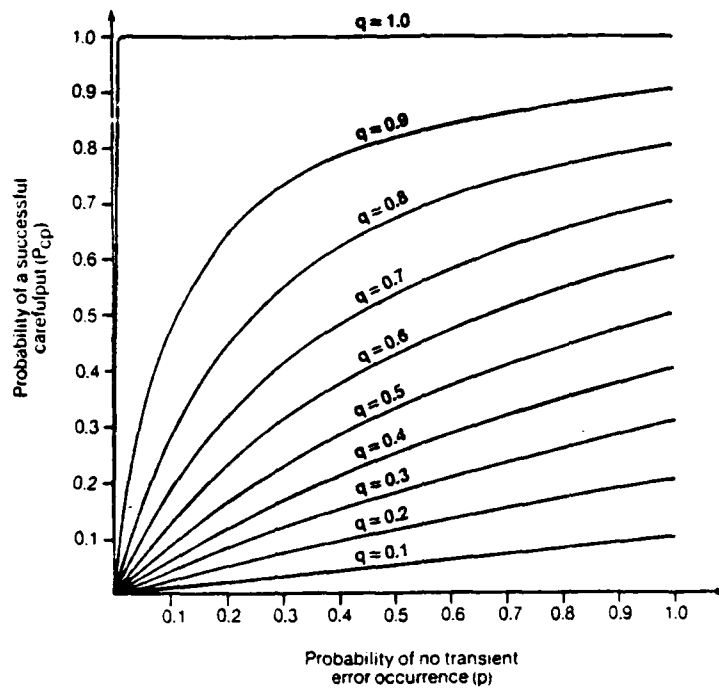


Figure 8-3

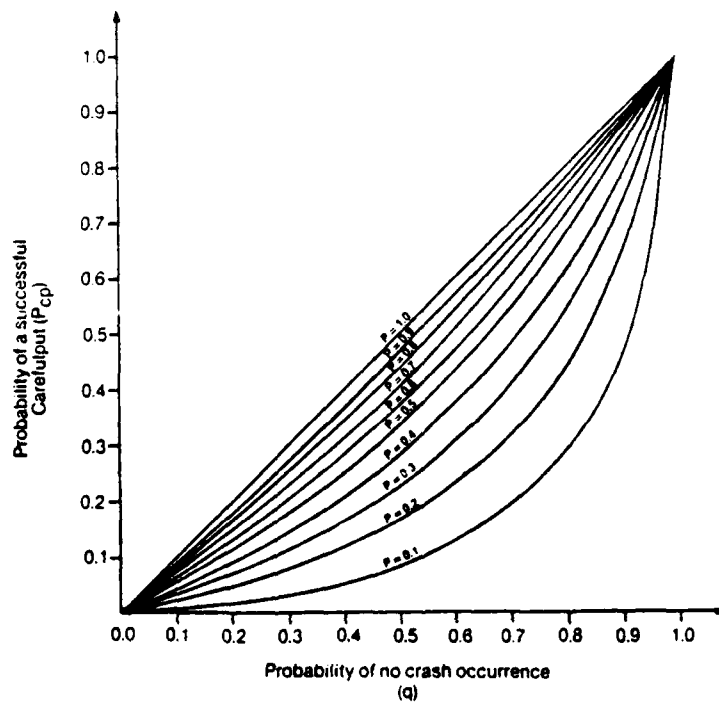


Figure 8-4

The following analysis is based on the assumption that both transient errors and disk crashes have Poisson distributions. Let T_{tr} and T_{cr} be the mean time to failures (MTTF) because of transient errors and disk crashes, respectively. In this analysis we will use

p = Probability of no occurrence of a transient error during a page write operation
and q = Probability of no occurrence of a disk crash during a page write operation.

Let T_w be the time required to write a page on the disk. The probabilities p and q are related to T_w as shown below:

$$p = e^{-T_w/T_{tr}}$$

and

$$q = e^{-T_w/T_{cr}}$$

First we will compute N_{cp} , the average number of page write attempts required to write a disk page successfully within the execution of a CarefulPut operation, and P_{cp} , the probability that a CarefulPut operation terminates successfully without encountering a disk crash. The probability that it takes n number of attempts to write a disk page successfully is given by $(1-p)^{n-1} \cdot p \cdot q^n$.

The term $(1-p)^{n-1}$ is the probability that the first $(n-1)$ attempts encounter some transient errors, p is the probability that the last write operation is successful, and q^n is the probability that no crash occurs during any of the n attempts. The average number of attempts is given by:

$$N_{cp} = pq + 2 \cdot (1-p)pq^2 + \dots + n(1-p)^{n-1}pq^n + \dots$$

$$N_{cp} = \frac{p \cdot q}{(1-q+pq)^2} \dots \dots \dots (8.1)$$

The probability of a CarefulPut operation terminating successfully is given by

$$P_{cp} = pq + (1-p)pq^2 + \dots + (1-p)^n pq^n + \dots$$

$$P_{cp} = \frac{p \cdot q}{(1-q+pq)} \dots \dots \dots (8.2)$$

One can easily see that for $q=1$, N_{cp} is and P_{cp} are equal to $(1/p)$ and 1, respectively. Figures 8-3 and 8-4 show the effects of p and q on P_{cp} . Figure 8-5 shows the effect of p and q on N_{cp} . The average time required to perform a CarefulPut operation is equal to $N_{cp} \cdot T_w$. Figure 8-3 shows the effect of q on the average number of attempts N_{cp} . As q increases this number decreases due to the fact that the probability of completing a CarefulPut operation decreases.

Next we analyze the reliability characteristics of a StablePut operation. For this purpose we will use NetRAT. The StablePut operation consists of

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

sequential execution of CarefulPut operations on the two disk pages comprising the Stable page. The StablePut operation is executed at Node 1 which represents CPU and volatile memory. Node 1 is connected to nodes 2 and 3 which represent two disk systems. The operation CarefulPut 1 is executed at Node 2, and CarefulPut 2 is executed at Node 3. The availability of Node 2 and Node 3 are P_{cp} computed as shown in equation (8.2) above. The MTTF for these nodes are the MTTF values for disk crashes. The resource requirements for the StablePut operation are (CarefulPut 1 and CarefulPut 2).

We will assume that the links connecting CPU to the disks never fail. For this example we will use the following hypothetical figures for the down-time of the disks and the CPU. For the disks, down-time of 3.6 hours in a month will be used. This gives $MTTF = 716.6$ $MTTR = 3.6$ and availability = 0.995. For the CPU, we will assume down-time equal to 0, meaning that its availability is one hundred percent. To compute the probability of successfully completing a StablePut operation, we compute its availability.

We will assume that the transient errors have a MTTF of 1 hour, and T_w , time required to write a page, equal to 50 msec. This gives $p=0.999999$. Using $MTTF=716.4$, we get $q=1.0$. This gives $P_{cp} = 1.0$. From NetRAT we obtain the symbolic reliability of StablePut as $p1.p2.p3$, which is equal to 0.990025. For all practical purposes, T_{cp} is equal to 50 msec.

Consider the same problem with a different set of reliability characteristics for the disks. Here we will assume down-time of one hour everyday, i.e., the MTTF for disk-crashes is 23 hours. This gives the availability of each of the disk systems equal to 0.95833. Using the same values for T_w and the MTTF of transient errors as before, $p=0.999999$ and $q=0.999999$. This gives $P_{cp}=0.999999$, which is quite close to 1 to have any significant effect on the availability of CarefulPut as dictated by the availability of the disk system which is equal to 0.95833. In this case, the availability of StablePut is 0.9184.

Now we examine the affect of the frequency of Cleanup procedure on survivability of a stable page. Suppose that a subset of pages gets completely destroyed on a disk crash. Let P_{loss} be the probability that a page gets destroyed on a disk-crash. We will use a hypothetical value of 0.4 (i.e., 40% of the pages are lost on a crash) for the purpose of presenting an example evaluation. Let T_c be the frequency of the cleanup procedure. The probability of losing a stable page is the probability of both disk systems crashing within the same interval between two consecutive cleanup operations and both pages getting destroyed on the crashes. The probability of the disks crashing within the same interval is $(1-e^{-T_c/T_{cr}})^2$ and the probability of both pages getting destroyed is $(P_{loss})^2$. This gives the probability of losing a stable page between two consecutive cleanup operations as

$$P(SS \text{ Loss}) = (P_{loss} * (1-e^{-T_c/T_{cr}}))^2$$

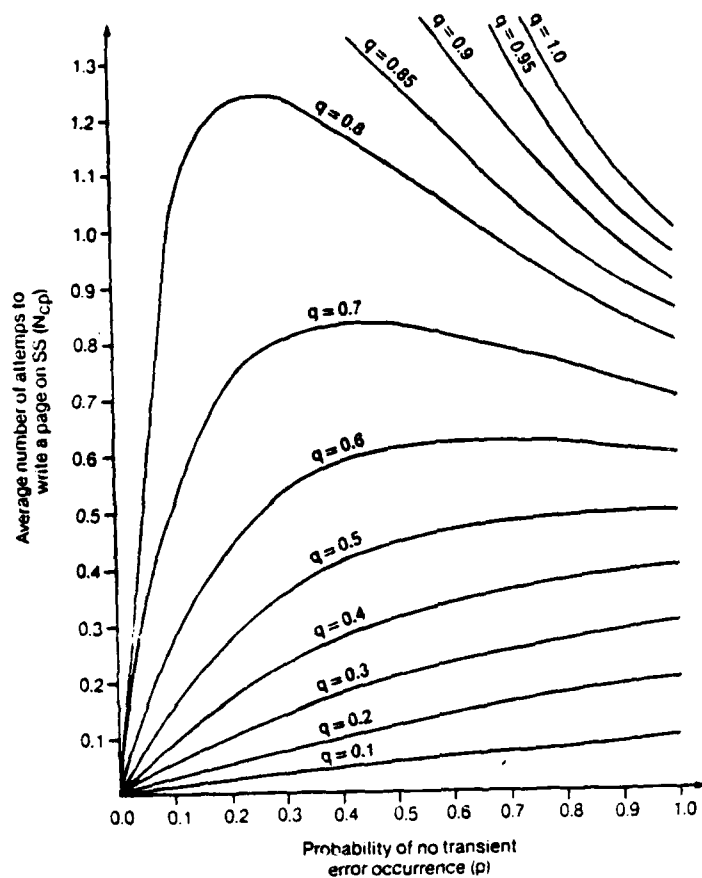


Figure 8-5

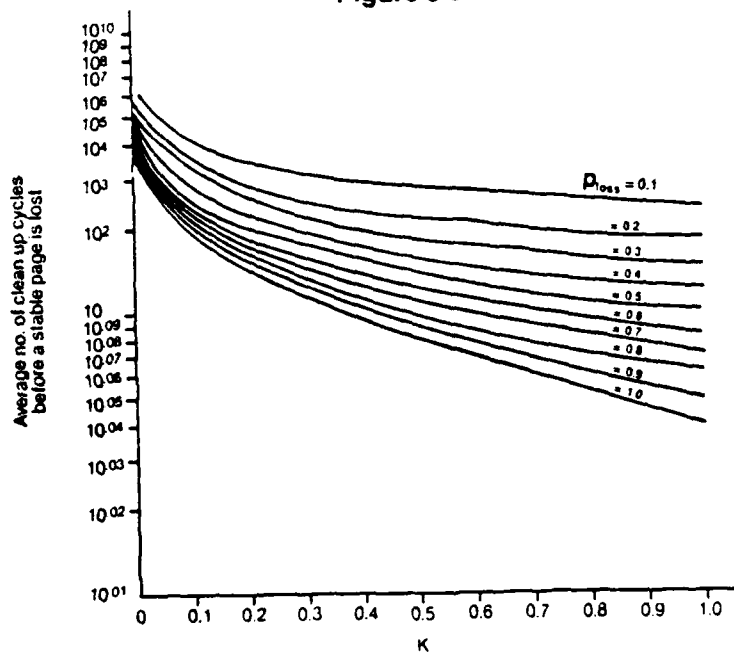


Figure 8-6

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

For a stable storage system with about 10^5 stable pages, the Cleanup procedure may take about 1 hour of CPU time. Consider a case where $T_c = 1$ day, $T_{cr} = 30$ days, and $P_{loss} = 0.6$. In this case $P(SS \text{ Loss}) = 3.8692 \cdot 10^{-4}$ and the average number of days for a stable page to get destroyed is about 7 years. This number is about 75 years if $T_{cr} = 100$ days.

Figure 8-6 shows mean time to failure (MTTF) for a stable page (in terms of the number of cleanup cycles) as a function of k , which is defined as the ratio T_c/T_{cr} . This relationship is shown for various values of P_{loss} .

8.5.2 Analysis of Replication Management

In this section we present an analysis of some replication management techniques. In general, the use of object replication is justified by an expected increase of system reliability and/or an improvement in system performance.

In the following example, we used NetRAT to measure the availability of the system in terms of degree of replication. The purpose of this example is to survey the changes in availability of the system with respect to the changes in degree of replication. As mentioned before, NetRAT provides quick first order evaluations of the system characteristics based on the characteristics of its components and these evaluations are less accurate as compared to simulation models due to the fact that they do not capture some of the dynamic operating conditions such as system load and resource contention.

Figure 8-7 shows the model used here for the analysis of replication management techniques. This is a simplistic model which represents each replicated copy as one node and the system user as another node connected to all the copies. In this simplified model, the unavailability of any copy due to either the communication links failures or the site crashes is modeled as the unavailability of the node representing that copy. Therefore, the links connecting the user node to the copies given availability of 1.0. The word of caution to remember here that if one wants to perform a more accurate analysis, then one must model the exact topology of the communication network using the reliability characteristics of the links and the sites. The user node has availability of 1.0. Read and write functions are executed from the user node on the replicated copies of the object.

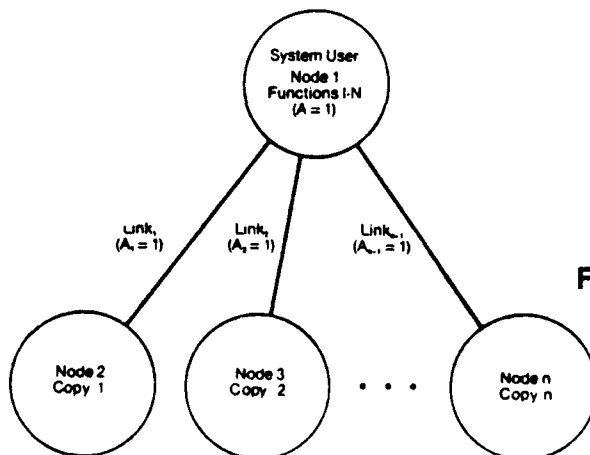


Figure 8-7. A Net RAT Model for Replication Analysis

The weighted voting algorithm [GIFF79] is evaluated with different sizes of read and write quorums to find the optimal sizes of read and write quorums.

The results obtained from different evaluations show that the majority consensus rule always gives the best result.

Figures 8-8 and 8-9 illustrate the system availability for read and write function, respectively. Tables 8-1 and 8-2 present the corresponding numerical data.

The followings are observations derived from these figures:

1. An even number of copies gives a better system availability, for read function, than the odd number of copies.
2. An odd number of copies gives better system availability, for write function, than the even number of copies.
3. System availability for the read function is always better than or the same as the availability of a single copy.
4. For both read and write functions system availability is an increasing function of the availability of a single copy.

These graphs can be used to determine the required degree of replication to achieve a given level of system reliability. It should be mentioned here that in this example we have assumed that all copies have the same availability.

Figures 8-10 and 8-11 depict the system availability improvement for read and write functions, respectively, here the system availability improvement for a function is computed as follows:

System availability improvement

$$\text{System availability improvement} = \frac{\text{Function availability} - \text{Node availability}}{1 - \text{Node availability}}$$

In these figures, the graphs for three different values of node availability are shown. The observations from these figures also agree with the ones mentioned previously, and since these graphs are illustrated as a function of degree of replication, the effect of this factor on the system availability can be seen more clearly. In Figure 8-11, the parts of the graphs which fall in the negative area correspond to those, in Figure 8-9, which are under the darker line representing the performance of a single copy system. These graphs represent the values of node availability and degree of replication for which the system availability would be less than the node availability.

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

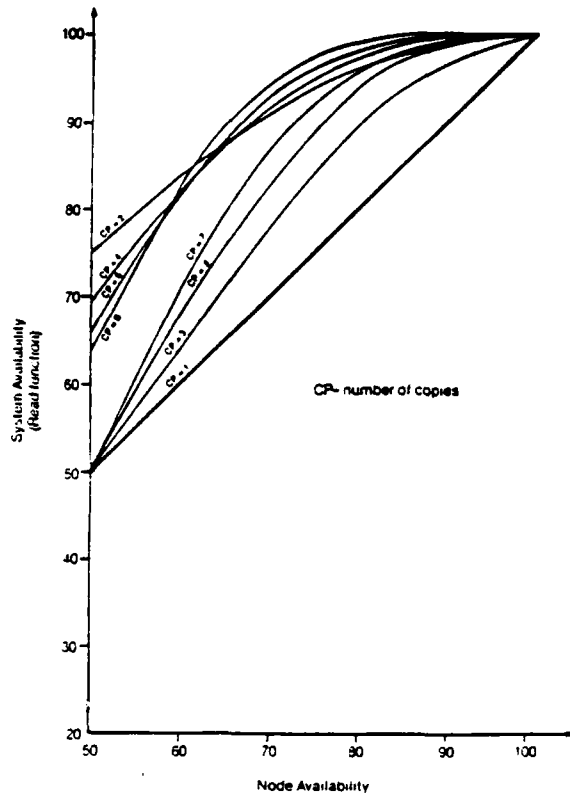


Figure 8-8.

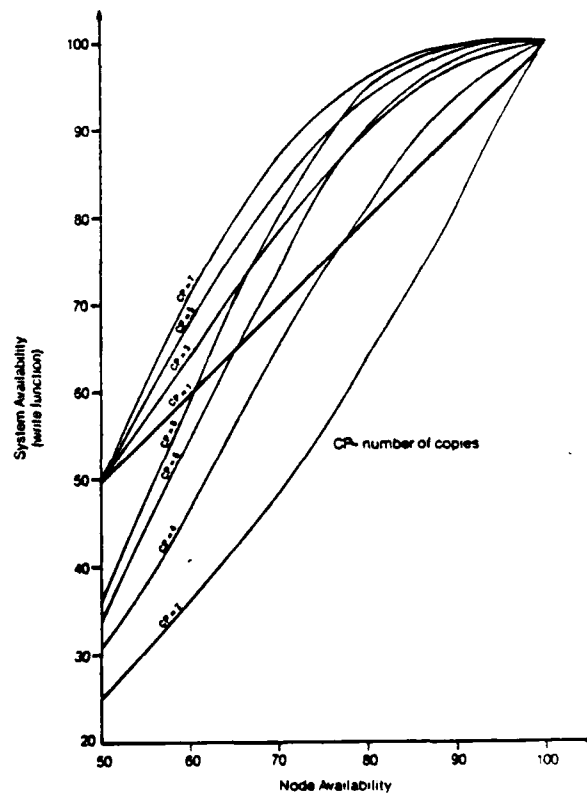


Figure 8-9.

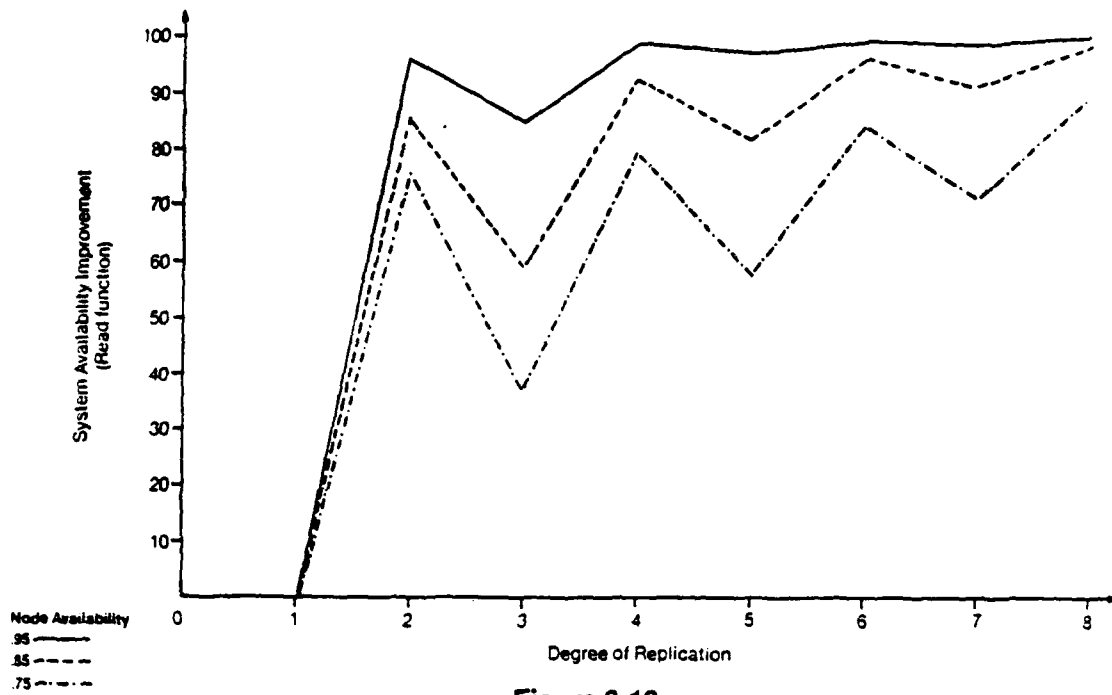


Figure 8-10.

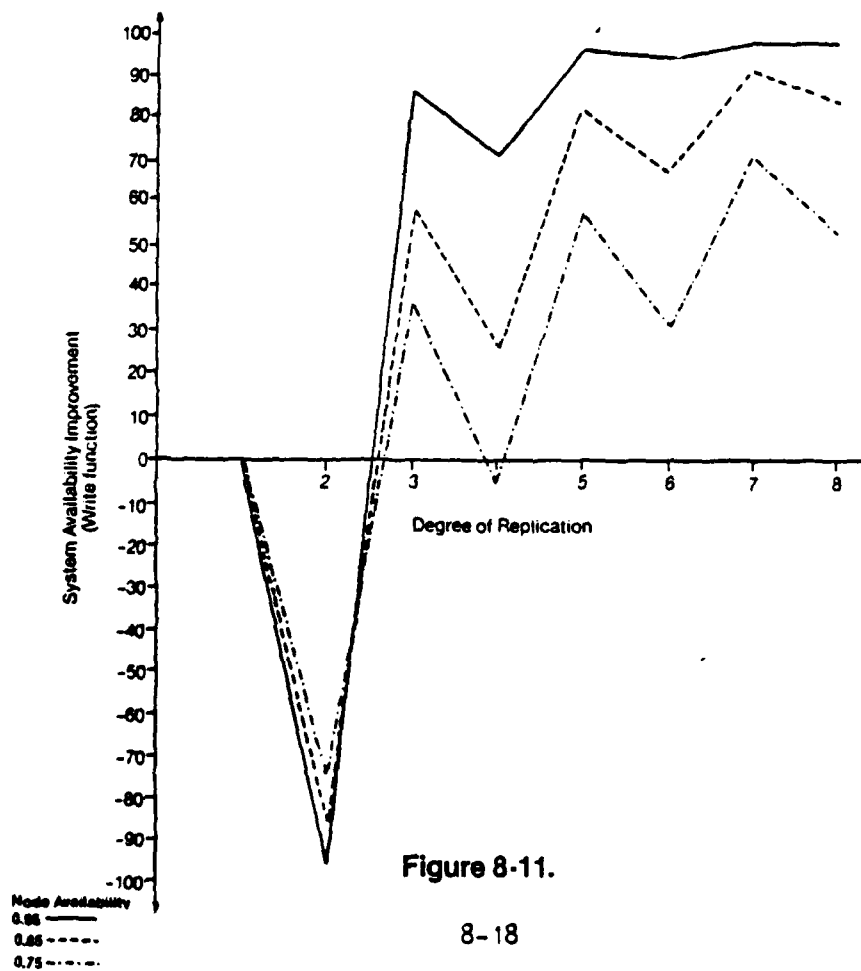


Figure 8-11.

Degree of Replication	Node Availability	70%	80%	90%	95%
1		70.00	80.00	90.00	95.00
2		91.00	96.00	99.00	99.750
3		78.400	89.600	97.200	99.275
4		91.630	97.280	99.630	99.951
5		83.692	94.208	99.144	99.884
6		92.953	98.304	99.873	99.991
7		87.396	96.665	99.727	99.980
8		94.203	98.959	99.995	99.998

System Availability
(Read Function)

Table 8-1

Degree of Replication	Node Availability	70%	80%	90%	95%
1		70.00	80.00	90.00	95.00
2		49.00	64.00	81.00	90.25
3		78.400	89.60	97.200	99.275
4		65.170	81.920	97.200	98.598
5		83.692	94.208	99.144	99.884
6		74.431	90.112	98.415	99.776
7		87.396	96.665	99.727	99.980
8		80.589	94.371	99.497	99.962

System Availability
(Write Function)

Table 8-2

	Pt-to-Pt	Broadcast
Quorum Collection: request votes vote responses	C q	1 q
Read operation Write operation Transaction Commitment	2 q	2 1
Read op	2q	q + 1
Write op: one-phase	2q	q + 1
two-phase	4q	2q + 2
Total messages: Read op Write op: one-phase two-phase	C + 3q + 2 C + 4q C + 6q	2q + 4 2q + 3 3q + 4

C: is the number of copies of the file

q: is the number of copies participating in the transaction

Table 8-3.

	Delay
Quorum collection: request votes voter responses	$\left. \begin{matrix} d \\ d \end{matrix} \right] = 2d$
Read operation Write operation	$d + t_r + d$ $d + t_w + d$
Transaction commitment: Read op: Write op: one-phase two-phase	2d $d + t_{wss} + t_{cs} + d$ $d + t_{wss} + t_{cs} + d + t_{cs} + d$
Total delay: Read op: Write op: one-phase two-phase	$6d + t_r$ $6d + t_w + t_{wss} + t_{cs}$ $7d + t_w + t_{wss} + 2t_{cs}$

Table 8-4.

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

We conclude this example by mentioning that even though the network model used here has a very simple configuration, it is a good demonstration of the manner in which more complex models can be studied.

In the rest of this section, we discuss the communication cost of a system with a set of replicated files when accessed with the weighted voting algorithm [GIFF79]. In this algorithm, a transaction has to collect at least Q_r votes to read a file and at least Q_w votes from current files to write a file. Q_r and Q_w are chosen such that $Q_r + Q_w$ is greater than the total number of votes in the system, and it is important to choose these quorum sizes to maximize the system availability and speed and minimize the cost. As mentioned earlier, the majority consensus rule was chosen to determine the read and write quorum sizes.

In general, the communication in weighted algorithms can be broken down into three phases:

1. Quorum Collection
2. Read and Write operation
3. Transaction Commitment

Table 8-3 contains information on the number of messages sent for operations in each phase and the total number of messages for them.

The number of messages transmitted in a point to point system is the number of messages from one host to another, not including intermediate hops that may be necessary; furthermore, if local copies of data are maintained and kept current, a read operation may be performed on the local copies without communication with the other copies.

To compute the communication delay for each operation, we defined the following variables:

- d : maximum delay for one way communication (host-to-host)
- t_r maximum time required to perform read operation on the object
- t_w maximum time required to perform write operation on the object
- t_{wss} : maximum time required to write the object on Stable Storage
- t_{cs} maximum time required to change the status of the object

The delay time for a remote operation is the sum of all delays for the three above mentioned phases in which each transaction is involved. Table 8-4 contains information on the communication delay for read and write operations in each phase and the total delay for them.

8.5.3 Analysis of the Broadcast-Bus Network

In this section we present an example analyzing reliability of some ethernet-like broadcast bus networks. The emphasis here is on modeling the system for reliability analysis.

The example presented in this section considers a set of computers connected by a broadcast bus. Each computer interfaces to the bus via a bus interface unit (BIU). One such system is shown in Figure 8-12. In this system we are interested in computing the availability of the communication path between two given computers. The communication path between two computers may be unavailable due to the failure of the BIUs or the broadcast bus.

A model for evaluating this system using NetRAT is shown in Figure 8-13. In this model each computer and the coaxial cable is represented by a node and a BIU is represented by the link connecting a computer node to the node representing the coaxial cable. The availability of the communication path from Node 1 to Node 2 is given by

$$P_{bus} \cdot P_{link1} \cdot P_{link2}$$

In order to increase the availability of the communication path, one might think of duplicating the communication bus. An example of this is shown in Figure 8-14, and the corresponding reliability analysis model is shown in Figure 8-15. The availability of the communication path from Node 1 and Node 2 is given by the following symbolic expression which was obtained using NetRAT:

$$P_{link1} \cdot P_{link2} \cdot P_{bus1} +$$

$$P_{link4} \cdot P_{link5} \cdot P_{bus2} (1 - P_{link1} \cdot P_{link2} \cdot P_{bus1}) +$$

$$P_{link1} \cdot P_{link3} \cdot P_{node3} \cdot P_{link6} \cdot P_{bus1} \cdot P_{link5} \cdot P_{bus2} \cdot q_{link2} \cdot q_{link4} +$$

$$P_{link4} \cdot P_{bus2} \cdot P_{link6} \cdot P_{node3} \cdot P_{bus1} \cdot P_{link2} \cdot q_{link1} \cdot q_{link5}$$

8.5.4 Analysis of Fault Tolerance

The goal here is to determine whether a system can withstand a given set of faults. This property is directly related to the discussion presented in Section 2.5 on reliability specification in terms of fault classes that a system can withstand.

We will use the example of Figure 2-5 to illustrate how one can use NetRAT to determine whether a system can tolerate a given set of faults. In the example of Figure 2-5, object OBJ6 is dependent on OBJ1, OBJ3, OBJ5, and DISK6 according to the following predicate:

RELIABILITY ANALYSIS TECHNIQUES AND TOOLS

(OBJ1 or OBJ3) and (OBJ5 or DISK6)

Suppose that we are interested in determining whether OBJ6 is stable under the failure of Node 1 only. To determine this, we assign availability of 0 to Node 1 and availability of 1.0 to all the other components. If the availability of OBJ6 evaluates to 1.0, then it means that OBJ6 is stable under that fault; otherwise, it means that OBJ6 is not stable in case Node 1 crashes. It turns out that the availability of OBJ6 is 1.0 under Node 1 failure, implying that OBJ6 is stable under this condition.

Consider another case where we want to determine if OBJ6 is stable under the failure of the communication link joining Node 2 and Node 4. For this purpose, we assign it availability equal to 0, and all the other components are assigned availability of 1.0. In this case we see that the availability of OBJ6 is 0, meaning that it is not stable with respect to this fault.

8.6 CONCLUSIONS

In this chapter, we have presented a modeling and analysis method to evaluate the reliability characteristics of systems. The analysis is combinatorial and it is not easy to use manually. It is advised that such procedures be automated as a general purpose tool. A system called NetRAT, which is based on such a procedure, has been described in this chapter. The modeling approach used in NetRAT is network based, i.e., the system is viewed as a collection of nodes connected by either bidirectional or unidirectional links. Reliability characteristics of the individual links and nodes are used to determine the reliability characteristics of the composite system and is particularly attractive from the viewpoint of hierarchical analysis of systems.

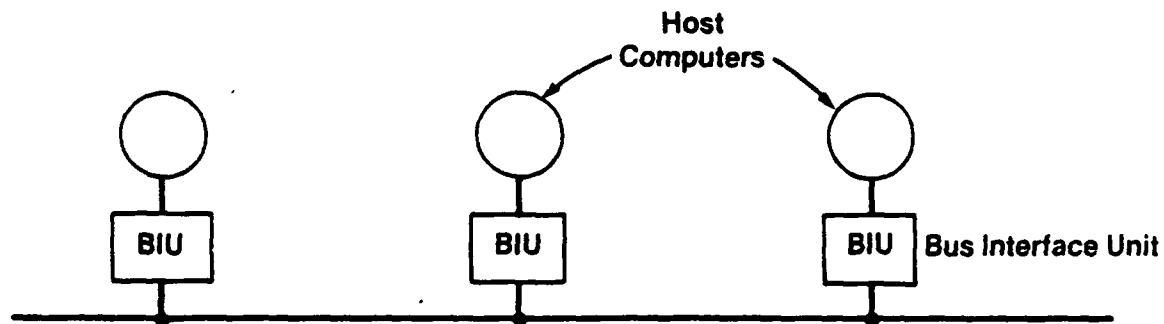


Figure 8-12

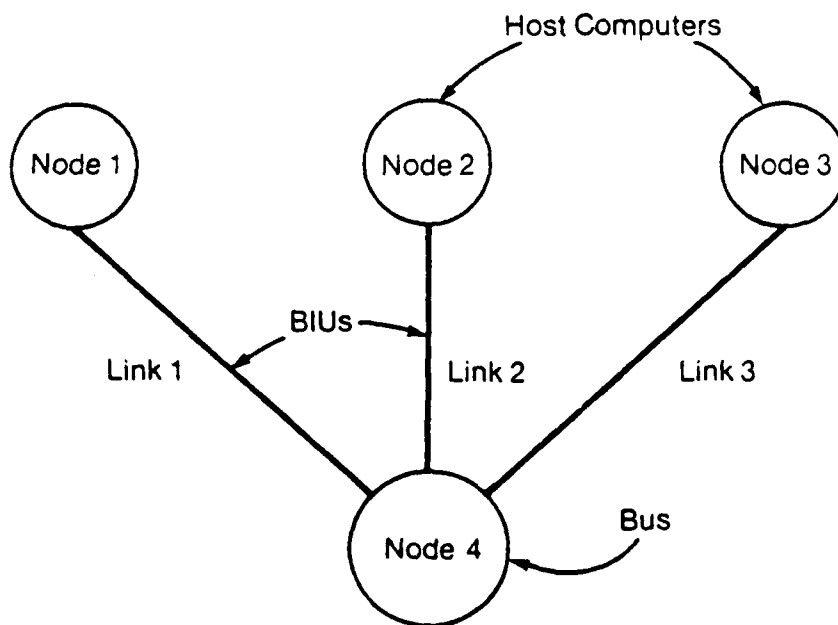
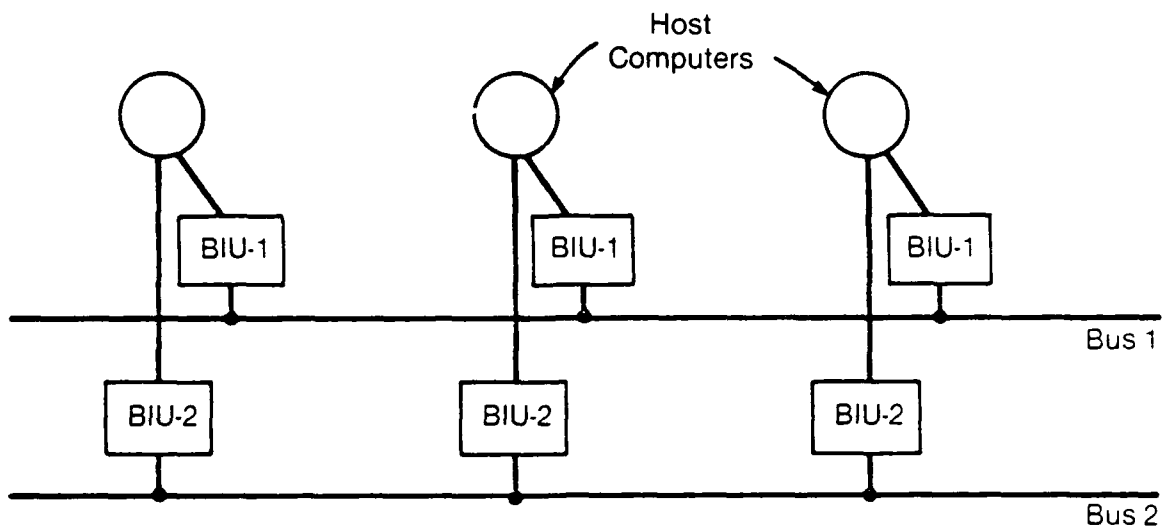
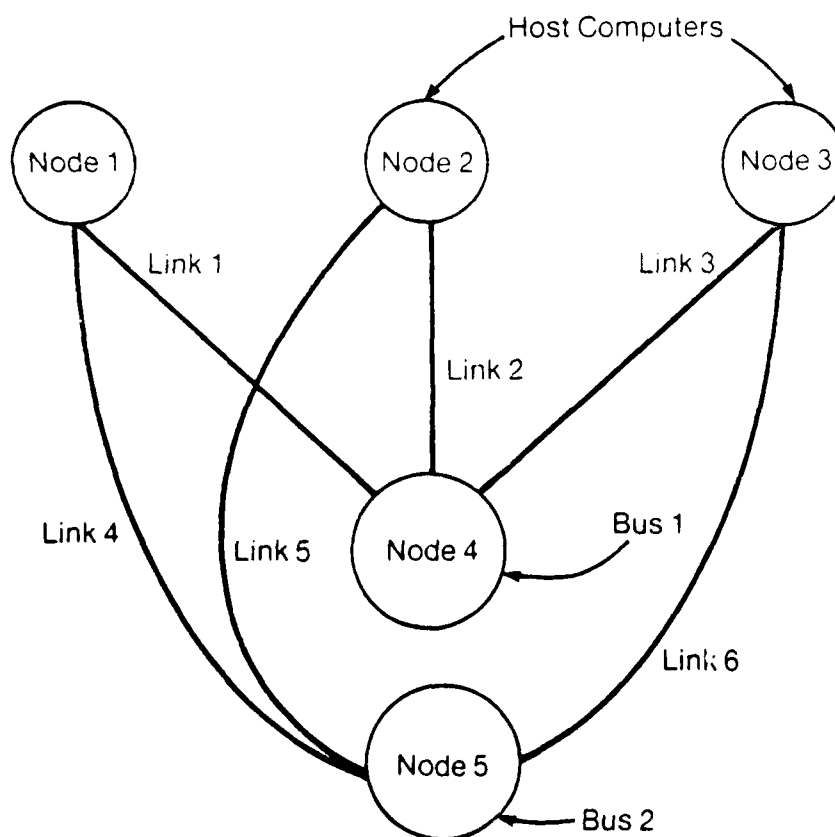


Figure 8-13



Duplication of Communication Bus
Figure 8-14



NetRAT Model for Duplicate Bus
Figure 8-15

CHAPTER 9

DESIGN VERIFICATION METHODS

(This Chapter was removed)

INTEGRATED DESIGN METHODS

CHAPTER 10

INTEGRATED DESIGN METHODS

10.1 INTRODUCTION

The previous chapters of this guidebook have introduced and discussed in some detail a number of concepts, procedures, tools, and techniques, all concerned with the design and development of high-integrity distributed systems. The purpose of the present chapter is to integrate these ideas and to offer a single, consistent methodology to guide those who would undertake such a development effort.

The proposed methodology is based on the classical software engineering paradigm which was originated by Barry Boehm [BOEH76] and which is commonly referred to as the software lifecycle. This view of software design treats the design evaluation as a more or less continuous process where a number of design phases are defined, each of which may actually be traversed several times in an iterative synthesis/analysis cycle. In the classical design strategy, the design is evaluated at each phase to determine if it is meeting the performance and functionality requirements set forth in the requirements statement. The principle extension to the software lifecycle paradigm contained in this chapter is the integration of reliability and performance considerations into the requirements specifications and design evaluation. It was mentioned in Chapter 2 that a system development activity consists of three phases - requirements specifications, design, and implementation. The design activity itself consists of a number of steps - concept level design, functional architecture design, and the detailed design. During each of these steps analyses are performed on the evolving design to ensure that it meets the goals set forth during that design activity. These steps are described in detail in Section 10.2.

Many references to previous sections of the guidebook have been included to indicate where the more detailed discussions of the concepts involved can be found.

10.2 DESIGN STEPS

The typical life-cycle of a software development process is shown in Figure 10-1. Here we describe the goals of the first four design steps - requirements specifications, concept level design, functional architecture design, and the detailed design. The goals of each of these phases are described in Table 10-1.

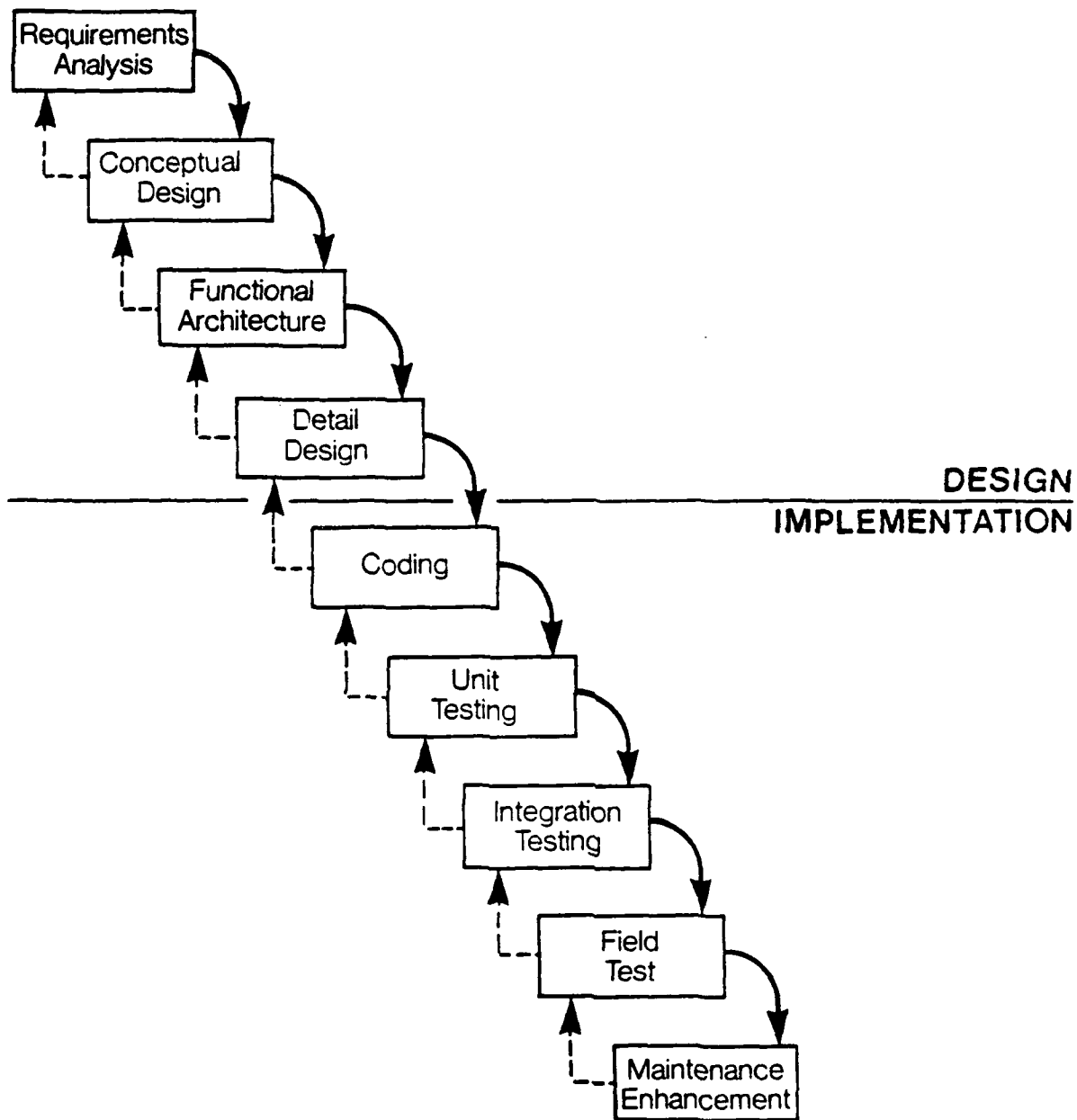


Figure 10-1 Software Life Cycle

INTEGRATED DESIGN METHODS

The requirements specification phase identifies various user-level operational scenarios to be executed by the system along with their performance and reliability requirements. In some cases the definition of such scenarios may involve conceptual definition of certain databases to be maintained by the system. This phase also defines the physical size (structure) of the system and the expected work-load.

The concept level design identifies the system functions and the command language provided to the users to support the scenarios defined in the first phase. This phase also defines the major databases maintained by the system and their relationship to the user-level functions (commands). The requirements statements for reliability and performance of these user functions are derived from the requirement statements of the first step. Also identified at this level are the consistency requirements stemming from the consistency requirements identified for the user-level scenarios. Additionally, the concept level design also defines the philosophy for introducing reliability in the system operations.

<u>Design Phase</u>	<u>Goal</u>
Requirements Specifications	Characterization of system size, user-level scenarios (transactions), performance, reliability, and security requirements.
Concept Level Design	Identification of major databases, functional definition of the system, system level philosophy for fault-tolerant operations, performance and reliability requirements for system functions.
Functional Architecture Design	Definition of various modules including the operating system modules; interactions among the modules, functions supported by the modules, error recovery protocols among the modules.
Detailed Design	Refinement of each module in terms of its internal data structures and algorithms. Selection of suitable recovery mechanisms into the detailed designs to realize the module functionality.

Table 10-1. Design Phase and Their Goals

The next step defines the functional architecture of the system which includes definition of various modules along with their inter-relationships.

A layered architecture such as the one used in the Zeus design is an ideal approach in this step. The functional architecture introduces some additional functions and objects to support the management of the application defined objects. We will refer to these as the operating system objects and functions. Typically these functions will include interprocess communication, process management, error recovery (using logs or shadows), and primary and secondary storage management.

The detailed design phase refines the internal structure of each module. This may involve partitioning or replicating an object across the network, which would introduce some additional new types of objects in the system. This phase also defines the data structures internally maintained by the modules and algorithms (or protocols) for managing them. In general, in this step the detailed designs of the application objects will be performed using the operating system functions defined in the concept level design phase. These detailed designs might identify modifications of the (operating system) functions or definition of some new functions.

10.3 REQUIREMENTS STATEMENT

Before beginning to design a high integrity distributed system (or any complex system for that matter), it is usually advisable to first write down the important characteristics which the finished system is expected to exhibit. Such a requirements statement almost always will contain some specifications concerning the required functionality of the finished system. An equally important part of the requirements statement is a specification of the expected system performance in terms of transaction response times and throughput. For high integrity systems, a third section of the requirements statement should specify the estimated reliability requirements of the various parts of the system. These three sections of the requirements statement - functional requirements, reliability requirements, and performance requirements - will serve to guide the design synthesis effort and to provide a basis for the subsequent analysis phases of the development. Before the requirements statements can be made the designer has to identify the application-level objects, the operation to be performed on such objects by the system users, and the physical structure of the system as implied by the application environment and the administrative constraints. The next step is to provide a detailed functional requirements and the expected workload in the system.

Typically the requirements specifications identify a set of user-level transactions and they also identify for each transaction the performance and reliability requirements under the different modes of system operations. Some examples of these modes are normal, alert and combat situations. The performance specifications identify the worst-case and the average response time for the transaction. Additionally the throughput requirements may be specified which would be directly related to the expected work-load in the system. These specifications may be made for performance expected at some given sites or for the overall system. Similar to the performance requirements, the reliability requirements are also specified for the different modes of operations. As discussed in Chapter 2, these requirements may be specified either in terms of availability and MTTF, or in terms of

INTEGRATED DESIGN METHODS

certain reliability levels that are to be maintained under a given set of component faults.

The important system features that a system designer has to know before starting the system design are described in Table 10-2.

<u>System Feature</u>	<u>Characterization</u>
1. Physical configuration, size, and environment	- o Multiprocessor, local area network (about 1KM span) - o long-haul network (10-100KM) - o User-population (Number of terminals and workstation) - o Limitations on the configuration and functionality of various sites due to organizational or environmental constraints - o Possible hazards at the various sites - o Modes of system operation (e.g., normal, alert, combat)
2. Main Databases	- o Intent of the databases - o Reliability and security requirements
3. User-Level Functions	- o Scenarios of operations - o Workload
4. Performance Requirements	o For each function, average and worst case response time (as a function of site and mode of system operation)
5. Reliability Requirements	- o Availability of each function - o For each function, the component failures it must withstand

Table 10-2. Requirements Characterization

10.3.1 Functional Requirements

The functional requirements of a high integrity command and control system should consist of a list of user-level functions and transactions which are to be performed. These functions and transactions are defined in terms of the operations on the application-level objects. Command and control systems in particular are well suited to this level of specification since the set of transactions which the system is expected to execute is a relatively fixed set and the transactions themselves are usually well defined at least in terms of high-level functionality. Such functional requirements will benefit greatly

from the use of a formal or semi-formal design language which includes some mechanisms to aid in subsequent functional analyses and correctness proofs.

For formally specifying the system functionality, one can use Concurrent System Definition Language (CSDL). CSDL actually consists of two languages, a specification language and a description language. The specification language is well-suited for the functional requirements specifications of user-level transactions which may then be extended and decomposed to provide specifications of lower level structures as the design proceeds. The data abstraction facility can be used to define the object types in the target system. These type definitions are then further refined as the design progresses through the subsequent phases. Furthermore, the description language part of CSDL may be used to define the preliminary and detailed designs (see sections 10.3 and 10.4) and thus provide a complete system definition (description plus specification) which can be formally verified to prove that the design meets its specifications.

A number of other languages exist which are also well suited to the specification of functional requirements and subsequent design documentation. In Chapter 9 we introduced Gypsy, a language which was specifically designed for the specification, definition, and subsequent verification of programs. Languages such as Path Pascal and Ada are also suitable due to the separation of specification and definition constructs in these languages.

10.3.2 Reliability Requirements

The second major part of the requirements statement is the reliability requirements section. In Chapters 2 and 8 we described various techniques for specifying such reliability requirements. The reliability requirements of the user-level transactions which are the focus of the requirements statement are probably best stated in terms of discrete levels of reliability such as those introduced in Chapter 2. Later refinements and analyses of the designs may then use more quantitative measures such as mean-time-to-failure and mean-time-to-repair and the automated reliability analysis tools like NetRAT described in Chapter 8.

Before any such reliability expectations can have meaning, however, it is first necessary to define the types of hardware and software faults which might occur and for which some protection is to be provided by the system. This also identifies the components which are most vulnerable in a combat situation and for which adequate redundancy must be provided. This constitutes the fault model which was discussed in Chapter 2.

10.3.3 Performance Requirements

The final part of the requirements statement deals with the performance expectations for the transactions for which the functional and reliability characteristics have already been specified. Chapter 7 discusses some of the techniques and tools which are available for the specification and evaluation of performance in a distributed, object-oriented system. At the level of the requirements statement, it is important to specify both the required average response times for the user transactions in a fault-free environment and the

INTEGRATED DESIGN METHODS

required average or worst case recovery times for each of the various fault classes defined in the fault model. Again, as the design progresses, the performance requirements of the user-level transactions will be decomposed and propagated downward to provide requirements for lower-level objects and operations in the system.

Obviously, these three parts of the requirements specifications are not totally independent. Both the reliability requirements and the performance specifications must be stated in terms of the user transactions which are initially defined as part of the functional specifications. Similarly, the reliability and performance of any given transaction cannot be expected to be totally independent. A highly reliable transaction, for example, is likely to utilize more resources and thus respond more slowly than a similar transaction for which the reliability requirements are not so high. Because of the interdependent nature of the various parts of the requirements statements, it will be difficult, if not impossible, to produce a consistent specification for a totally obtainable system on the first try. Instead, an iterative approach is advocated in which first the requirements statement is produced and then, as the design progresses, the functionality, reliability, and performance of the system are periodically re-evaluated using the tools and techniques of the previous chapters. At each point, changes in either the requirements or the design itself may be warranted if the measured characteristics of the design do not match the expected values of the requirements statement.

10.4 CONCEPTUAL DESIGN

The conceptual design phase identifies the functions supported by the system and the major databases (objects) maintained by it. The functional definitions identify all interfaces that are provided to the users for executing the scenarios of the requirements statements. The functional definition in most cases would be given as informal descriptions of the protocols for invoking the system functions and the effects of their executions. It is possible to provide a formal definition of the system functions; however, this may not be practical for a large system. The requirements for the performance and reliability characteristics of system functions are derived from the requirements specifications in the previous phase.

This phase also identifies the consistency requirements for the various operations and the database objects. Depending on these consistency requirements and the reliability requirements the philosophy underlying the system structuring is defined. There are several issues that are addressed within the global context during this phase. This phase develops a high-level approach to redundancy/replication management, error detection/recovery, object migration, remote operations, atomic actions, process checkpointing/rollback, etc. An important goal in this phase is to define a consistent and coherent approach to these design issues. Many of the functions provided to the users reflect the reliability features of the system, e.g., capability to checkpoint or rollback, etc.

In many instances certain lower level design decisions are also made during the conceptual design phase. For example in the Zeus design it was established at very early stages of the conceptual design that the underlying communication system will only be a simple datagram-like service which does not guarantee delivery of a message. The global level approach to reliability was strongly influenced by this lower level design decision.

In the Zeus design we had two options for our global level approach to reliable system structuring. The first was to integrate various recovery mechanisms directly within the object structures while the second was to divorce such mechanisms from the object structure and instead offer it as a service to the users. For example, the first approach would have a reliable object operation as an atomic action. In contrast, a transaction mechanism could permit different object operations to be coalesced into single atomic actions. Our choice in the design of the example system, Zeus, was to provide a general purpose atomic action facility that can be used by the other object managers.

10.5 FUNCTIONAL ARCHITECTURE

Functional architecture is the decomposition of a system into logical, functional units. It defines the functional relationships among the various modules. The functional architecture of a distributed system includes communication systems as well as the computational units. Ideally a functional architecture would follow certain hierarchical structure. In the Zeus design we proposed one such layered structure for reliable distributed systems. The functional architecture presents a more concrete embodiment of the philosophy for reliable operations as developed during the conceptual design phase. It includes provision for the redundancy necessary to implement survivability on both the hardware and the software.

We have espoused in Chapter 4 and 5 of this guidebook an object-orientated design technique for distributed systems. This technique can be used to decompose the required functions of the system into invocations against different typed objects. A functional requirement for the system can be decomposed into a number of object invocations. These invocations can be further decomposed into invocations on other objects and so on until the only objects depended on are hardware resources. These hardware resources can then be viewed as objects with some inherent performance and reliability characteristics. The hardware specifications must include definition of a suitable communication architecture.

In general, the definition of the functional architecture will start with the application visible functions and from them will define the object types required to support them. We refer to these objects the user-level type managers. Based on the conceptual design, some additional objects may be introduced to support the management of application derived objects. We will refer to these as the system-level type managers, as used in the Zeus design. These objects provide a set of generic services to the other type managers. These type managers depend directly on some other software or hardware modules. We call this set of software modules the kernel. For each of the modules, which include the user/system type managers and the kernel, detailed

INTEGRATED DESIGN METHODS

functional specifications are given during this phase. For this purpose a language such as CSDL or Gypsy can be used.

10.5.1 System-Level Type Managers

The Zeus type managers which are discussed at length in Volume 2 of this report are examples of the system type managers. In addition to providing low-level services to the other type managers, these system type managers are also characterized by the fact that they generally must be resident on each and every site (host) in the system. Some of the objects which would typically be managed by system type managers include:

- o Processes and transactions, examples of active objects which can change their own state,
- o Message objects which provide inter-process communication between and among the other type managers and user processes,
- o Principal objects which represent the users of the system and facilitate such system-level activities as authentication and protection from unauthorized use,
- o Symbolic name contexts, directories of user-supplied symbolic names for objects.

10.5.2 Kernel Functions

Another component which must be dealt with to some degree of detail during the functional architecture design is the kernel. This is the program or programs which serve to interface the type managers with the underlying host; either at the hardware level as in the Zeus example or at the level of the native operating system executing on the site. Although we must stress that the approach taken in Zeus is merely one possible way to partition the various tasks among the type managers and kernel, it does serve to illustrate the types of tasks which will almost certainly be required in one form or another. The services which might be handled at the kernel level include,

- o Communication services such as remote procedure call facilities among the type managers,
- o Storage management services which might serve to virtualize the volatile and non-volatile storage devices,
- o Process scheduling, allowing the type managers to time-share the physical host processor and scheduling them with the appropriate priority,
- o Unique identifier generation, providing very fast generation of object identifiers which are globally unique and, as in Zeus, non-repeating.

10.5.3 Design Analysis and Verification

The evaluation of a system design begins here with the functional architecture. Given a functional architecture one can establish its completeness with respect to its intended functionality and with respect to its ability to handle error detection, fault diagnosis, and recovery/restart/reconfiguration. In Chapters 7, 8, and 9 we presented methods to evaluate the performance, reliability and functionality of the designs. In Chapter 2, we described how to extend object specification to include reliability and availability specifications.

The functional architecture must be evaluated for its functional correctness, performance and reliability of the entire system using these characteristics for each of the modules. This step will set forth the functional, performance and reliability requirements for each of the modules. The functional correctness can be established using Gypsy or some other methods such as interval logic. Simulation and reliability measurements tools such as PAWS and NetRAT can be used at this level. This is an iterative procedure requiring modifications to the functional architecture to meet the desired goals. For example, a processor upgrade can be used to ensure that a performance measure is met, or a multiple copy scheme can be tried out to ensure that object reliability is met.

10.6 DETAILED DESIGNS

Detailed design resolves the functional architecture into software units. It gives, for each software unit, specifications which precisely, and where possible formally, define its properties. The requirements identified for each module in the previous phase are used as the design goals during this step.

After the requirements of the various modules have been clearly identified in the requirements statement and the general architecture has been defined in terms of object interfaces, the next step is to begin designing the internal structure of the type managers themselves. This process should probably start first with the detailed design of the user-level followed by the designs of the system-level type managers and the kernel.

In designing a module, a preliminary design should be constructed first and subjected to an early evaluation. Using the tools/techniques described in Chapter 7, 8, and 9, estimates of how well the design is currently satisfying the requirements statement may be obtained. If it is found that some area of the design is indeed not adhering as well as expected to the requirements, then either a new set of mechanisms may be substituted or the requirements themselves may be modified to make them more realistic. In either case, it is of course much easier to make modifications at this stage than after the design has already proceeded to the point of rendering the mechanisms in some programming language.

In the detailed design phase, the detailed designs of the user-level type managers should be constructed first using the functions provided by the system-level type managers and the kernel. This might require definition of new kernel/system functions or modification to the existing functions. In

INTEGRATED DESIGN METHODS

this phase, mechanisms should be identified which are suitable for carrying out the various operations that have been defined for each of the type managers. The problems involved with mapping the mechanisms to a real programming language and programming environment are also addressed during this phase. The detailed designs are constructed using CSDL; these designs are then implemented in the chosen programming language.

Many of the recovery mechanisms required to perform the specified actions of the type managers are well known and well documented in the literature. A particular concern of this report has been to identify, define, and evaluate the various mechanisms which are available for providing high integrity and reliability in object-oriented systems. Chapter 4 constitutes a detailed and complete survey of these kinds of mechanisms.

The detailed design of a user-level type manager consist of the following steps:

1. For each object type, define the data structures and storage operations associated with the data structures.
2. For each object operation, define the algorithms it will use and the operations it will invoke against other objects.
3. For each object type describe the algorithms necessary to implement reliable storage for that type. For example, stable storage algorithms, write-ahead logs, shadow page algorithms, etc.
4. For each object type, identify how the system/kernel functions are to be used for constructing reliable operations for that type definition.
5. For each object operation, use its interface to determine the size of the calls and responses to it. This is used to determine the message handling and communication system design for messages sent to and from each type manager.
6. For each object type, define its role in sending and receiving calls and responses during failure and recovery.

Taking the above considerations into account, the design can be re-evaluated in terms of functionality, reliability, and performance. The choice of algorithms for the operations and for object manipulation provides better estimates for resource usage by the object types.

The design of the system type managers is carried out on similar lines. The kernel designs are influenced by the host hardware/software. Once all modules have been designed a detailed evaluation can be performed before implementing the system. For performance evaluation this requires building the PAWS models of the detailed designs, integrating the PAWS models into one simulator, and then mapping the user-level work-load by simulating scenarios as PAWS transactions in the simulator. If a verified system is the prime goal of the design, then the detailed designs are expressed in Gypsy and verified using its verification system. Functional correctness can also be established by using Path Pascal simulations of the parts of the system or the whole

system. Reliability evaluations can be done in several ways, e.g., using analytical techniques such those used in NetRAT or using simulation techniques. NetRAT is more appropriately applied in a hierarchical fashion, i.e., parts of the the detailed designs are evaluated independently, and then these results are used in the evaluation of the functional architecture. In contrast, reliability analysis using the simulation models is more appropriate when applied to the entire design.

10.7 SUMMARY

Presentation of a set of guidelines and steps for designing reliable distributed systems to meet certain desired levels of performance has been the prime goal of this guidebook. Our discussion has focussed on the development of systems for command and control applications ranging from real-time controls and signal processing to database management systems. Survivability and continued operation despite the loss of resources is the key requirement; such systems must keep operating even on the "best effort" basis. This guidebook advocates object-oriented system design principles because they facilitate creation of well-bounded domains of error confinement and recovery. An example system was presented in Chapter 5 which showed integration of various recovery mechanisms in an object-oriented design framework. A major emphasis is placed on design evaluation before building the real system. Design evaluation and verification is a continuous process which is carried out during almost every design phase. Such design evaluations help us to identify design errors in the early stages, thus reducing the life-cycle costs significantly.

Security, which is a very important and crucial aspect of command and control systems, is not addressed in this guidebook. We feel that the integrated set of methods presented here can be properly extended by including security specifications in the requirements statements and a set of security analysis methods in the various design phases.

Our discussion of the communication network design has been limited only to network survivability. In Chapter 6 we presented a discussion on selecting communication network topology. The technology related to communication protocol design is very well-developed and understood at this point; therefore, any discussion of this aspect has been ommitted from the guidebook.

In designing a large distributed system several cost optimization problems are encountered, such as allocation of a given number of file copies to certain sites in the system. This guidebook does not address such optimization problems; however, once a system configuration has been selected, it can be analyzed thoroughly for performance and reliability using the techniques presented here. Addition of optimization techniques will definitely be a significant and useful enhancement to the integrated set of design methods presented here.

Appendix A

PERFORMANCE EVALUATION OF THE ZEUS SYSTEM

This appendix describes the performance evaluation of the Zeus distributed operating system. We present an overview of what parts of the system were modeled, what measures about which recovery mechanisms were collected, and the results of the performance evaluation. A discussion of the Zeus distributed operating system and of the PAWS performance evaluation system is contained in Chapters 5 and 7 respectively.

1. Model Overview

There are three components of a performance model -- environment, system structure, and workload. The environment captures the standard hardware and software as well as the effect of the physical environment. The Zeus environment included the following: configuration of the system, the performance attributes of its components, and operational conditions such as failures and their rates. The system structure captures the architecture of the parts of the model, both hardware and software, that are being analyzed. For example, the Zeus object managers with their consistency and recovery mechanisms are part of the system structure as are the command and control object managers. Finally, the workload captures the pattern and frequency of usage of various resources in the system as derived from the execution of the application systems. This includes the definition of the classes of command and control jobs. The components of the model used for this effort are described below.

1.1 Model Environment

The model environment consisted of the system configuration and fault injection functions. The system configuration consisted of seven sites interconnected by a local area network. Rather than model the intricacies of transmission on a local network (e.g., link control, medium access control, physical control, etc.), interrupt handling, and process scheduling, a delay that approximated the time of a remote procedure call (rpc) was used. We assumed the existence of a special purpose integrated rpc protocol that provided for the efficient delivery of a service request and response, rather than an rpc protocol implemented on top of a generic layered architecture. The delay used to model message transfer from one process to another had an exponential distribution with a mean value of 1 msec. Finally, we assumed that the communications subsystem provided reliable communication and that messages were neither duplicated nor lost. This and several other assumptions were made to make the tasks of building a model and analyzing the resulting data more tractable.

The hardware configuration at each site was identical. It consisted of one cpu and five disks. Two of the disks were configured to be a stable disk (i.e., the contents of the two physical disks were identical and a page was

accessed sequentially from one disk and then from the other). A service request consisted of cpu service followed by disk service (e.g., a central server model). The amount of service received depended on the complexity of the object manager operation (e.g., to double the speed of the cpu, the size of the cpu service requests would be divided in half). Disk service requests had an erlang distribution with a mean access time of 20 msec and a standard deviation of 10 msec for a single disk. Stable storage requests had an erlang distribution with a mean access time of 40 msec and a standard deviation of 15 msec.

All failures were assumed to be clean. The types of faults injected were site crashes. A site failure caused all processing at a site to be suspended until the site had recovered. A failure was modeled as a 1500 msec delay. While this is an unrealistically short period of time, it is necessary in order to make the execution of the models tractable (e.g., not take days).

Five fault injection rates were defined. They included no faults, 1 fault/100 seconds, 3.3 faults/100 seconds, 8.3 faults/100 seconds, and 12.8 faults/100 seconds. (The fault rates are given in faults/milliseconds in the details of the statistics. The last four rates correspond to 1/100000 msec, 1/30000 msec, 1/12000 msec, and 1/7812.5 msec.) While these values may seem excessively high and perhaps unrealistic, they are justified for several reasons. First, it is expected that future distributed C2 systems will have to operate in extremely hostile as well as benign environments. The higher fault rates accurately describe extremely hostile environments while the low fault rates describe benign environments. Second, it was an explicit goal to determine for what environment a particular level of reliability is needed and justified, and if a reliability mechanism that is adaptive to the fault rate of an environment is useful. Finally, because of the expense in computer resources for running complex simulations it was necessary to increase the occurrence of the desired phenomena to a smaller than desired time interval.

This range of fault injection rates provided a base case of operation in a fault free environment to compare with the expected case of a few faults and an extreme case of many faults. Originally, we wanted to quantify the extra burden that a recovery mechanism imposes on an application by quantifying its differential overhead. The differential overhead of a recovery mechanism was to be determined by comparing the performance of the model in a fault free environment with the recovery mechanisms operating and without the recovery mechanisms operating. This would have measured the extra overhead of the recovery mechanisms. Because of the major modifications required to be made to the model this one comparison was not done. It can be addressed in future work.

1.2 Model System Structure

The system structure consisted of a number of object managers assigned to different sites and a number of transactions that could be initiated from different sites. Each site had a process/transaction manager (PTM) to handle operations such as begin transaction, end transaction, and abort transaction. The object managers that performed C2 operations were instantiations of the generic object manager (GOM) for exemplary C2 objects. Each object manager

performed operations such as concurrency control, commit processing, transaction undo, and object manager specific operations. The amount of time to do an object manager specific operation was determined based on the number of objects accessed by an operation and the amount of reading and writing done (see workload discussion). In addition, each site had the equivalent of the operating system support for providing transparency of object and object manager location.

Table 1 describes the configuration of the C2 object managers for the performance evaluation. For each object manager the following information is listed: the name of the object manager (e.g., the type of object managed), the number of instances of objects, and the sites where an instance of the object manager exists. The names of the sites have the following meaning: TACC is tactical air command center, CRC is control reporting center, AS1 is air squadron 1, and AS2 is air squadron 2. There are three additional sites that have none of the listed C2 object managers. They are FACP1, FACP2, and FACP3 (forward area control posts). Transactions may originate from any of the sites.

Table 1. C2 Object Manager Configuration

Object Manager	Number Objects	Sites
Intelligence	80	TACC, CRC
Navigation	80	TACC, CRC
Supplies	80	AS1, AS2
Mission Plans	120	TACC, CRC, AS1, AS2
Squadron	40	TACC, AS1, AS2
Weather	80	TACC, CRC

There were a number of simplifying assumptions made to constrain the scope of the problem and allow us to concentrate on the critical parts. The most important assumption was to target the evaluation primarily at commit protocols. This allowed the simplification of the PTM and GOM. The notions of nested transactions and checkpoints were excluded from the model. The details of the algorithm by which recoverable objects are maintained was deemed less important than capturing the resource usage of any such algorithm. Site failures were assumed to be clean; furthermore, there was sufficient state information after a site failure to continue correct operations (e.g., either finish commit processing or abort the transaction). Message delivery was assumed to be perfect. The model was limited to executing top-level, sequential transactions. The PTM model included the following operations: create process, begin transaction, object manager operation invocation, and end transaction. The GOM model included the following operations: object manager operation execution, concurrency control, and transaction commitment. The execution of the model is outlined below.

A new unit of work is generated every 125 msec. The work is generated deterministically based on the distribution of types of transactions and what percentage of transactions of each type have been generated. A unit of work may also describe a failure. The following discussion assumes that the unit of work is a user transaction.

The unit of work causes a process to be created by the PTM. This is modeled as the consumption of a set amount of memory and cpu resources. Following the execution of "create process" a set amount of cpu and disk is consumed to model "begin transaction" and an entry is made in the PTM database (PMDB). The PTM then executes a loop in which the script of the operations described for the particular type of transaction being executed (see Section 1.3) is modeled. On each iteration of the loop either an rpc is initiated, some user computation is executed, a transaction is aborted, or an end transaction operation is initiated. For an rpc, an object manager is selected, the request sent to the object manager, a timer is set for 1200 msec, and a response is awaited. If the response is received before a timeout occurs, the object is added to the transaction's accessed object list (AOL) in the PMDB and the loop is executed again; otherwise, the transaction is aborted. A user computation is modelled as a consumption of cpu and disk resources followed by another execution of the loop. An abort is modelled as a consumption of cpu and disk resources that results in the PMDB being cleared, the transaction's resources freed (e.g., local resources are deallocated and object managers receive an abort message), and the execution of the transaction script terminated.

The way the PTM handles "end transaction" is more complex because this is the part of the model whose output was analyzed in detail and because there were multiple commit protocols modelled. The protocols modelled include one phase and two phase commit protocols both using presumed abort and presumed commit options. (The protocols and options are described in Chapter 4.) When a message in a commit protocol is sent to the participating object managers it is sent to each object manager one at a time and then a timeout is set (e.g. 1200 msec). If the timeout expires for a time interval for which no response is received, an appropriate action is taken. The action would be to abort the transaction if it is a prepare message for a two phase commit, or to retransmit the message if it was a commit message with the presumed abort option or an abort message with the presumed commit option. If a response is received between the time the timer was set and expired and there are more responses outstanding, the timer is reset. All commit messages are preceded by a write to stable storage.

The GOM performs three kinds of operations -- object operations, end transaction, and abort transaction. All object accessing was modelled as well formed two phase locking. For operations on objects, the request is checked to see if this is the first access to the object. If it is, a simple resource allocation mechanism is used that models the 80/20 rule -- 80% of all requests are to 20% of all objects. Rather than allocate a specific instance of an object type, tokens that represent objects within either the 80% reference class (e.g., 20% of the tokens) or the 20% reference class (e.g., 80% of the tokens) are allocated. If an object cannot be allocated then the request is queued. Requests indicate an operation to be performed by the GOM. An operation was modelled as a service request consisting of cpu service followed by disk service (e.g., a central server model). There was a .3 probability that a process made a subsequent service request. CPU service requests had an exponential distribution with a mean that varied from .75 msec to 3.75 msec. The mean was determined based on the object manager operation being implemented. Typically, update operations had higher means than read operations. A response was then sent to the PTM and an inactivity timer of

4500 msec was set for the object if a two phase commit protocol was being used. If the timer expires before a subsequent request was received, the object was freed and an entry was made in the object manager database indicating this action. When a subsequent request was received for an object whose timer had expired, an object timeout abort message was returned to the requesting PTM.

A GOM aborted an operation either due to its site failing or a PTM sending an abort message. To abort a transaction the GOM freed all allocated objects and cleared its database of all records of the transaction. In addition for site failures, a site failure abort message was sent to the PTM.

For "end transaction" a GOM that was executing a one phase commit has followed every operation on the object by writing the object on stable storage. For a two phase commit, a prepare message causes the objects to be written to stable storage and a response returned to the PTM. For either commit protocol, a commit message causes an end transaction record to be written to stable storage and for a presumed abort option a message to be sent to the PTM.

1.3 Model Workload

In evaluating the performance of the Zeus design, we were interested in examining the characteristics of the system with a wide variety of different job types. We were particularly interested in the effect of the different recovery mechanism design options on the performance of different job classes. However, since no application software was available, a number of generic scenarios were defined in terms of several performance-affecting attributes. Four job attributes and two possible values for each attribute were defined as follows:

- o Duration (short or long) - The total number of type manager operations executed by a scenario. The difference between "short" and "long" scenarios is about an order of magnitude.
- o Number of Objects Accessed (few or many) - The total number of objects which are either read, written, or read and written by a scenario. The difference in magnitude between "few" and "many" object accesses is about an order of magnitude.
- o R/W Ratio (R/O or update) - This indicates whether or not the scenario does any update operations on ANY of the objects that it accesses. R/O jobs do not do any update operations whereas "update" jobs do at least one.
- o Object Distribution (single- or multi-site) - If all the objects accessed by a job reside on a single host (not necessarily the same one that the scenario is running on), then the value of this attribute is "single-site"; otherwise, it is "multi-site."

Of the sixteen possible generic jobs classes that may be obtained by substituting values for these four attributes, eight were chosen based on information about existing C2 applications. Table 2 summarizes the attributes

of these eight jobs and defines an instruction mix distribution for them. The percentage figures in the job mix column indicate the percentage of the total number of jobs resident in the system at any given time (after a steady-state has been reached). By way of justification for the job mix given here, notice the following things:

- o 80% of the concurrently executing jobs are short - that is, they perform relatively few operations,
- o 75% of the jobs have a relatively small working set (access only a few objects),
- o Many of the jobs (30%) are read-only.

Table 2. Job Mix Description

Job Number	Job Mix (%)	Duration	# Objects Accessed	R/W Ratio	Object Distrib.
1	15	short	few	R/O	multi-site
2	15	short	few	R/O	single-site
3	15	short	few	update	single-site
4	15	short	few	update	multi-site
5	20	short	many	update	multi-site
6	5	long	few	update	multi-site
7	10	long	few	update	single-site
8	5	long	many	update	multi-site

Although this method of selecting an example workload may seem somewhat ad hoc, it is expected that it will provide valuable performance data that is sufficiently accurate to guide the design process. More importantly for our present purposes, it provides a concrete example of the use of instruction mixes in real design situations.

For each job description in the mix, a number of exemplary jobs were required. Synthetic jobs were used since we were modeling a pre-operational system. These are artificial jobs for which the resource usage is similar to the expected characteristics of some future real jobs or to existing jobs being run on other systems.

Synthetic jobs are easier to obtain than the real applications because they summarize the resource utilization of the jobs that they are characterizing. Local CPU usage, for example, is usually represented by a simple idle loop in a synthetic job and remote requests are abstracted so that the parameters of the calls are simplified or left out entirely.

As an example of a synthetic user-level scenario from the Zeus performance analysis, consider the following job:

```

Begin Scenario
  Read Intelligence
  Read Navigation

```



```

Read Weather
Read Mission-Plan
Computation
Update Mission-Plan
End Scenario

```

It is assumed that "Intelligence", "Navigation", "Weather", and "Mission-Plan" are objects (or groups of objects) in the command and control system. The semantics of the scenario is meant to resemble a "Mission Control" job which is typical (although much simplified) of jobs run on C2 systems. The scenario reads the appropriate data, does some local computation to determine how the plan of some in-progress mission should be changed, and then updates that mission plan.

Notice how this synthetic job represents the basic performance-affecting features of the scenario in a very stylized and simplified way. The meaning and functionality of the local computation is not specified and neither are the parameters of the remote calls.

The following guideline was used to determine values for the transaction attributes duration and number of objects accessed: there should be an order of magnitude difference between short and long for duration and few and many for the number of objects accessed. The values used for these attributes were the following:

- (1) "short" transactions have 1 to 4 operations;
"long" transactions have 30 to 40 operations.
- (2) "few" objects means 1 to 6 objects;
"many" objects means 30 to 60 objects.

The synthetic jobs (transactions) that make up the job mix for the performance evaluation are listed below. For each job the following information is given: name of the job, its attributes, the sites from which it may be initiated, the type and number of objects accessed, and the order in which objects are accessed.

1. Squadron Status 15%, short, few, R/O, multi-site
 Sites: TACC, AS1, AS2, FACP1, FACP2, FACP3
 #Objects: Squadron1 2
 Squadron2 2

Begin Transaction	#Objects
Read Squadron1	2
Read Squadron2	2
End Transaction	

2. Weather Check 15%, short, few, R/O, single-site
 Sites: TACC, CRC, FACP1, FACP2, FACP3
 #Objects: 1

	#Objects
Begin Transaction	
Read Weather	1
End Transaction	

3. Supplies Check 15%, short, few, update, single-site
 Sites: TACC or AS1 or AS2
 #Objects: 1

	#Objects
Begin Transaction	
Read Supplies	1
Update Supplies	1
End Transaction	

4. Mission Check1 15%, short, few, update, multi-site
 Sites: TACC and CRC
 #Objects: Intelligence 4
 Mission Plans 4

	#Objects
Begin Transaction	
Read Intelligence	4
Read Mission Plans	4
Update Mission Plans	2
End Transaction	

5. Mission Check2 20%, short, many, update, multi-site
 Sites: TACC, CRC
 #Objects: Mission Plans 15
 Intelligence 15

	#Objects
Begin Transaction	
Read Mission Plans	15
Read Intelligence	15
Update Intelligence	5
Update Mission Plans	4
End Transaction	

6. Air Control Report 5%, long, few, update, multi-site
 Sites: TACC, CRC, FACP1, FACP2, FACP3
 #Objects: Weather 3
 Navigation 3

	#Objects
Begin Transaction	
Read Weather	1
Update Weather	1
Read Weather	1
Update Weather	1
Read Weather	1
Update Weather	1
Read Weather	1
Update Weather	1
End Transaction	

Read	Navigation	1
Update	Navigation	1
Read	Navigation	1
Update	Navigation	1
Read	Navigation	1
Update	Navigation	1
Read	Navigation	1
Update	Navigation	1
Read	Weather	1
Update	Weather	1
Read	Weather	1
Update	Weather	1
Read	Weather	1
Update	Weather	1
Read	Weather	1
Update	Weather	1
Read	Navigation	1
Update	Navigation	1
Read	Navigation	1
Update	Navigation	1
Read	Navigation	1
Update	Navigation	1
Read	Navigation	1
Update	Navigation	1
End Transaction		

7. Intelligence Report 10%, long, few, update, single-site
 Sites: CRC
 #Objects: Intelligence 4
 Mission Plans 1

Begin Transaction	#Objects
Read	Intelligence 1
Read	Intelligence 1
Read	Intelligence 1
Insert	Intelligence 1
Read	Mission Plans 1
Update	Intelligence 1
Update	Intelligence 1
Update	Intelligence 1
Update	Mission Plans 1
Read	Intelligence 1
Update	Intelligence 1
Read	Intelligence 1
Update	Intelligence 1
Read	Intelligence 1
Update	Intelligence 1
Read	Intelligence 1
Update	Intelligence 1
Read	Mission Plans 1
Update	Mission Plans 1
Read	Intelligence 1

Update	Intelligence	1
Read	Intelligence	1
Update	Intelligence	1
Read	Intelligence	1
Update	Intelligence	1
Read	Mission Plans	1
Update	Mission Plans	1
Read	Intelligence	1
Read	Intelligence	1
Read	Intelligence	1
Read	Intelligence	1

End Transaction

8. Mission Planning 5%, long, many, update, multi-site

Sites: TACC, CRC, AS1, AS2

#Objects: Intelligence 10
Mission Plans 10
Navigation 10
Squadron1 5
Squadron2 5
Supplies 10
Weather 10

Begin Transaction		#Objects
Read	Intelligence	10
Read	Squadron1	5
Read	Squadron2	5
Read	Supplies	8
Read	Weather	5
Read	Mission Plans	5
Insert	Squadron1	1
Insert	Squadron2	1
Insert	Supplies	1
Insert	Mission Plans	3
Read	Squadron1	1
Read	Squadron2	1
Read	Supplies	2
Read	Mission Plans	3
Update	Squadron1	4
Read	Squadron2	4
Update	Supplies	4
Update	Mission Plans	2
Read	Navigation	6
Delete	Navigation	4
Read	Navigation	4
Update	Navigation	2
Insert	Navigation	4
Read	Mission Plans	1
Read	Navigation	2
Read	Squadron1	1
Read	Squadron2	1

End Transaction

2. Model Output

The model output consists of statistics about the performance in terms of the workload and system structure, and statistics about the failures. The first set of data includes traditional measures such as response time and throughput as well as measures about the system structure such as the in-doubt period of a commit protocol. The second set of data includes measures such as the number of faults of a given type and the number of transactions effected by that kind of fault. This section is organized as follows. First, a discussion of the traditional measures is given. This is immediately followed by a set of data for each run of the model. Next, the failure statistics are discussed and then presented for each run of the model. Finally, some analysis of the data is made. This is supported by graphs describing some interesting phenomena.

2.1 Response Time Statistics

This section presents the response time statistics. For each run of the model (e.g., a particular commit protocol and fault injection rate) a number of statistics are collected. They are organized into the following tables:

- o summary statistics,
- o distribution of committed transactions,
- o in doubt periods,
- o response times, and
- o throughput.

The contents and use of each of these tables is described below.

The summary statistics list for a specific run the number of transactions generated, number of transactions committed, time elapsed, average in-doubt period for a process/transaction manager, average in-doubt period for a generic object manager, and the average response time. Each run is executed until 1000 transactions have successfully committed. Because transactions may be aborted due to failures and because various commit protocols require different operations, the total number of transactions that must be generated and the total amount of simulation time required to have 1000 transactions successfully complete will vary. Also, there are some transactions still active when the one thousandth transaction completes. The number of uncompleted transactions may be determined by subtracting the number of aborted transactions and committed transactions from the total number of transactions generated (see failure statistics, Section 2.2). The average in-doubt period and response time represent a measure in milliseconds for the 1000 successfully committed transactions.

The distribution of committed transactions table describes the workload of the successfully completed transactions in terms of the number of transactions of a given type. The workload is generated deterministically to produce the job mix described in Table 2. In the absence of faults and resource contention, the distribution of the committed transactions in percentage should equal that described in Table 2. Analyzing the percentage of the types of different transactions allows the determination of whether the

performance of a particular class of applications is favored or penalized by a specific commit protocol in a given fault environment.

The in-doubt period table lists for the successfully completed transactions of each type of transaction the average in-doubt period in milliseconds and the number of transactions. For the PTM the number equals the distribution of committed transactions. The in-doubt period measures the amount of time that an object manager is vulnerable to the failure of a site that another object manager resides on and will be forced to keep a resource locked until that site recovers. A PTM is vulnerable to the failure of an OM and may be required to keep a commit/abort record indefinitely in its database. An OM is vulnerable to the failure of a PTM and may be required to keep an object locked indefinitely. These measures are elaborated on in Chapter 4 of the Guidebook. Ideally, the in-doubt measures would allow the determination of the effect of a commit protocol and fault environment on the concurrency level of an object manager and thereby the transaction throughput and response time. Unfortunately, the in-doubt periods are not as useful as we would like them to be for two reasons. The measure only includes successful and not failed transactions, but more importantly the down time of a site (1500 msec) is unrealistically short. (This is due to the cost of running the simulations.)

The response time table lists the mean, minimum, and maximum response time for all successfully completed transactions by transaction type in milliseconds. This measure includes the time interval from when a PTM receives a "begin transaction" command until the PTM completes its part of the commit protocol. Note that this point will vary depending on whether a presumed commit or presumed abort protocol is followed.

The throughput tables describe the effect of failure rates and different commit protocols on the number of transactions successfully completed for an interval of time. The throughput is measured in transactions committed per 100 seconds and the failure rate is measured in site failures injected per 100 seconds. The throughput measure is normalized for a fixed experiment length of 130,000 milliseconds with a fixed transaction generation rate of one transaction every 125 milliseconds. Contrasting the throughput figures allows the determination of what commit protocol is appropriate for an expected fault environment with a given class of applications.

The first four tables are organized by commit protocol and presented in the following order: one phase presumed abort, one phase presumed commit, two phase presumed abort, two phase presumed commit. The data for each commit protocol is organized by fault rate with the no fault case presented first, followed by the next higher fault rate, and so on. This data is preceded by the throughput tables which are organized slightly differently. The throughput tables for all of the different runs are summarized in Table 3. This is followed by Tables 4-7 which each describe the effect of one of the commit protocols on all of the different transaction types for all of the different fault rates.

Table 3. Summary of effect of different commit protocols and failure rates on throughput.

Protocol	Failure Rate				
	0.0	1.0	3.3	8.3	12.8
One Phase Presumed Abort	791.8	786.0	770.8	765.9	752.1
One Phase Presumed Commit	794.9	788.4	773.2	761.6	745.6
Two Phase Presumed Abort	773.3	772.8	756.9	726.8	724.7
Two Phase Presumed Commit	782.7	770.9	761.1	733.3	734.6

Table 4. Effect of one phase commit protocol with presumed abort on transaction type throughput.

Transaction Type	Failure Rate				
	0.0	1.0	3.3	8.3	12.8
1	119.6	119.5	114.9	114.9	114.3
2	119.6	120.2	119.5	118.7	117.3
3	119.6	117.9	118.7	118.7	117.3
4	119.6	118.7	118.0	117.2	116.5
5	159.1	159.4	157.3	157.0	155.6
6	38.8	37.0	36.3	36.0	34.6
7	77.6	75.5	71.7	70.4	69.2
8	38.0	37.7	34.7	32.9	27.1

Table 5. Effect of one phase commit protocol with presumed commit on transaction type throughput.

Transaction Type	Failure Rate				
	0.0	1.0	3.3	8.3	12.8
1	120.0	119.9	116.7	115.8	112.7
2	120.0	119.9	119.1	117.3	118.0
3	119.2	118.1	119.1	119.6	117.2
4	119.2	119.1	117.5	117.3	116.5
5	159.8	159.2	157.7	156.2	155.3
6	38.9	37.1	35.6	35.1	33.6
7	78.7	77.2	74.2	69.3	65.7
8	38.9	37.8	33.4	31.2	27.6

Table 6. Effect of two phase commit protocol with presumed abort on transaction type throughput.

Transaction Type	Failure Rate				
	0.0	1.0	3.3	8.3	12.8
1	119.1	119.0	115.8	113.4	113.8
2	119.8	119.7	118.8	117.0	118.8
3	119.1	118.3	118.1	117.0	113.8
4	119.1	119.0	117.3	116.3	115.2
5	159.3	159.2	156.7	166.1	154.3
6	31.6	34.0	32.5	29.8	29.0
7	68.8	68.8	65.9	53.8	57.2
8	36.3	34.8	31.8	25.5	22.5

Table 7. Effect of two phase commit protocol with presumed commit on transaction type throughput.

Transaction Type	Failure Rate				
	0.0	1.0	3.3	8.3	12.8
1	119.8	119.5	115.7	113.0	112.4
2	119.8	120.2	120.4	117.4	117.5
3	119.8	117.9	118.7	118.2	114.5
4	119.8	117.2	117.2	115.9	116.8
5	156.2	158.8	156.8	153.4	154.2
6	35.2	36.3	35.8	30.1	30.1
7	71.8	65.6	67.7	58.0	63.2
8	37.5	35.5	31.2	27.9	25.7

CASE: One phase commit protocol using presumed abort.
 No failures.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1010
 Total number of transactions committed = 1000
 Total time elapsed = 126299.758
 Mean process/transaction manager in-doubt period = 799.402
 Mean generic object manager in-doubt period = 464.942
 Mean response time = 972.354

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	151	151	151	151	201	49	98	48

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	88.736	302	155.952	151
2	51.127	151	88.431	151
3	58.253	151	231.083	151
4	153.386	302	282.885	151
5	188.539	606	384.924	201
6	1301.156	196	3409.161	49
7	1183.712	392	3285.523	98
8	646.707	294	2442.917	49

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	282.772	212.684	306.153
2	190.980	72.693	195.894
3	350.311	218.022	405.608
4	447.932	250.819	471.812
5	596.125	464.554	602.826
6	3677.650	3026.021	3718.187
7	3535.672	2811.551	3591.510
8	2786.650	2647.311	3100.684

CASE: One phase commit protocol using presumed abort.
 One failure every 100000 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1018
 Total number of transactions committed = 1000
 Total time elapsed = 127642.820
 Mean process/transaction manager in-doubt period = 792.798
 Mean generic object manager in-doubt period = 466.248
 Mean response time = 969.462

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	152	153	150	151	203	47	96	48

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	94.883	304	163.559	152
2	52.995	153	90.632	153
3	58.361	150	231.251	150
4	157.259	304	287.757	151
5	188.067	609	385.758	203
6	1320.781	188	3440.090	47
7	1188.312	388	3297.581	96
8	673.283	289	2445.166	48

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	287.512	212.684	306.153
2	192.365	72.693	195.252
3	344.660	218.022	405.608
4	458.479	250.819	471.812
5	598.589	464.554	602.826
6	3733.995	3026.021	3734.189
7	3539.663	2811.551	3591.510
8	2887.054	2647.311	3100.684

CASE: One phase commit protocol using presumed abort.
 One failure every 30000 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1038
 Total number of transactions committed = 1000
 Total time elapsed = 130965.914
 Mean process/transaction manager in-doubt period = 784.491
 Mean generic object manager in-doubt period = 461.660
 Mean response time = 967.260

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	149	155	154	153	204	47	93	45

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	88.129	298	157.521	149
2	55.528	155	91.639	155
3	52.353	154	222.594	154
4	156.464	308	278.746	153
5	190.209	614	375.062	204
6	1331.146	192	3547.088	47
7	1178.922	372	3304.996	93
8	699.440	270	2600.878	45

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	298.368	212.684	306.153
2	224.434	72.693	229.263
3	324.725	218.022	405.608
4	451.868	250.819	471.812
5	600.541	464.554	611.401
6	3825.680	3026.021	3835.838
7	3545.344	2811.551	3598.698
8	3040.774	2647.311	3100.684

CASE: One phase commit protocol using presumed abort.
 One failure every 12000 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1045
 Total number of transactions committed = 1000
 Total time elapsed = 134317.469
 Mean process/transaction manager in-doubt period = 722.428
 Mean generic object manager in-doubt period = 429.148
 Mean response time = 915.722

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	150	155	155	153	205	47	92	43

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	94.772	301	157.648	150
2	62.746	156	96.787	155
3	53.628	155	219.200	155
4	146.463	308	259.760	153
5	172.492	621	343.390	205
6	1266.037	188	3264.101	47
7	1097.319	371	3087.325	92
8	642.693	268	2348.108	43

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	306.964	212.684	335.186
2	210.046	72.693	227.830
3	327.548	218.022	333.478
4	439.623	250.819	455.547
5	562.575	464.554	583.444
6	3515.644	3026.021	3573.613
7	3411.287	2811.551	3483.715
8	2900.768	2794.617	3209.838

CASE: One phase commit protocol using presumed abort.
 One failure every 7812.5 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1064
 Total number of transactions committed = 1000
 Total time elapsed = 137319.188
 Mean process/transaction manager in-doubt period = 728.339
 Mean generic object manager in-doubt period = 431.300
 Mean response time = 912.774

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	152	156	156	155	207	46	92	36

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	90.106	304	161.012	152
2	49.645	157	94.588	156
3	54.943	156	209.495	156
4	155.669	310	278.348	155
5	170.045	621	347.819	207
6	1250.079	184	3300.606	46
7	1144.510	374	3236.177	92
8	674.998	216	2495.917	36

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	306.357	212.684	315.964
2	201.901	72.693	203.256
3	334.136	218.022	568.243
4	465.254	250.819	471.597
5	551.782	464.554	587.813
6	3612.139	3026.021	3756.093
7	3546.670	2811.551	3658.463
8	2883.278	2777.916	3718.584

CASE: One phase commit protocol using presumed commit.
 No failures.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1006
 Total number of transactions committed = 1000
 Total time elapsed = 125805.188
 Mean process/transaction manager in-doubt period = 754.071
 Mean generic object manager in-doubt period = 437.494
 Mean response time = 807.652

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	151	151	150	150	201	49	99	49

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	93.527	302	165.665	151
2	45.930	151	85.871	151
3	48.885	151	217.414	151
4	144.358	300	263.784	150
5	165.376	603	336.028	201
6	1241.082	196	3257.549	49
7	1096.351	396	3071.698	99
8	625.577	294	2309.891	49

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	216.012	175.522	255.230
2	134.867	97.191	153.309
3	267.694	192.301	270.959
4	321.980	205.683	323.235
5	390.801	341.004	403.602
6	3313.576	2037.916	3383.755
7	3124.476	2799.705	3171.879
8	2366.884	2203.594	2570.161

CASE: One phase commit protocol using presumed commit.
 One failure every 100000 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1015
 Total number of transactions committed = 1000
 Total time elapsed = 127337.625
 Mean process/transaction manager in-doubt period = 738.241
 Mean generic object manager in-doubt period = 429.065
 Mean response time = 791.597

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	152	152	150	151	202	47	98	48

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	94.302	304	164.858	152
2	48.483	152	90.942	152
3	51.254	150	215.102	150
4	145.810	302	267.011	151
5	160.804	609	328.262	202
6	1266.253	188	3263.858	47
7	1069.048	392	3020.535	98
8	626.752	288	2322.078	48

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	216.233	175.522	255.230
2	142.576	97.191	153.309
3	262.866	192.301	263.287
4	326.244	205.683	326.244
5	384.280	341.004	403.602
6	3322.018	2037.916	3383.755
7	3064.402	2799.705	3127.916
8	2381.110	2203.594	2570.161

CASE: One phase commit protocol using presumed commit.
 One failure every 30000 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1034
 Total number of transactions committed = 1000
 Total time elapsed = 130672.633
 Mean process/transaction manager in-doubt period = 723.725
 Mean generic object manager in-doubt period = 424.228
 Mean response time = 776.149

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	151	154	154	152	204	46	96	43

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	91.227	302	159.371	151
2	49.898	154	92.195	154
3	48.338	154	205.702	154
4	134.964	306	252.381	152
5	165.724	612	335.993	204
6	1230.346	184	3188.353	46
7	1089.573	382	3075.293	96
8	658.050	258	2452.472	43

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	211.697	175.522	255.230
2	144.714	97.191	153.309
3	254.700	192.301	260.336
4	305.216	205.683	317.129
5	387.069	341.004	403.602
6	3237.919	2037.916	3257.707
7	3130.953	2799.705	3145.667
8	2506.989	2203.594	2570.161

CASE: One phase commit protocol using presumed commit.
 One failure every 12000 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1050
 Total number of transactions committed = 1000
 Total time elapsed = 133611.453
 Mean process/transaction manager in-doubt period = 705.495
 Mean generic object manager in-doubt period = 418.081
 Mean response time = 756.784

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	152	154	157	154	205	46	91	41

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Generic Object Manager			Process/Transaction Manager	
Type	Period	Number	Period	Number
1	90.875	303	159.159	152
2	49.056	155	86.585	154
3	58.324	157	207.803	157
4	132.254	308	250.798	154
5	174.052	615	347.704	205
6	1226.575	184	3245.397	46
7	1101.662	362	3059.680	91
8	641.401	245	2398.504	41

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	204.150	175.522	255.230
2	136.512	97.191	149.434
3	254.017	192.301	261.661
4	302.966	205.683	312.387
5	401.440	341.004	417.665
6	3294.577	2037.916	3441.325
7	3115.566	2799.705	3170.627
8	2459.222	2258.350	2570.161

CASE: One phase commit protocol using presumed commit.
 One failure every 7812.5 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

SUMMARY STATISTICS

Total number of transactions generated = 1073
 Total number of transactions committed = 1000
 Total time elapsed = 136285.453
 Mean process/transaction manager in-doubt period = 699.446
 Mean generic object manager in-doubt period = 415.515
 Mean response time = 752.347

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	151	158	157	156	208	45	88	37

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	87.902	301	152.731	151
2	54.399	158	94.734	158
3	55.453	157	220.131	157
4	139.426	313	251.058	156
5	170.865	624	344.273	208
6	1261.332	178	3330.991	45
7	1135.088	348	3182.785	88
8	641.084	223	2339.113	37

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	203.440	175.522	255.230
2	145.915	97.191	149.596
3	264.590	192.301	270.458
4	306.507	205.683	310.224
5	397.761	341.004	403.602
6	3392.625	2037.916	3415.730
7	3236.483	2756.480	3269.166
8	2405.470	2272.771	2463.375

CASE: Two phase commit protocol using presumed abort.
 No failures.

Total number of transactions generated = 1034
 Total number of transactions committed = 1000
 Total time elapsed = 129312.820
 Mean process/transaction manager in-doubt period = 202.894
 Mean generic object manager in-doubt period = 171.780
 Mean response time = 1286.113

DISTRIBUTION OF COMMITTED TRANSACTIONS								
Type	1	2	3	4	5	6	7	8
Number	154	155	154	154	206	41	89	47

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	118.613	308	154.445	154
2	62.019	155	127.044	155
3	61.406	155	140.489	154
4	124.795	308	202.683	154
5	171.885	618	246.119	206
6	206.454	164	278.846	41
7	227.325	356	301.584	89
8	311.623	282	375.654	47

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	503.886	253.765	509.091
2	387.813	237.982	412.946
3	549.424	344.435	550.105
4	720.295	480.030	720.295
5	954.843	532.933	977.222
6	4525.685	3160.785	4557.085
7	4488.261	2826.835	4554.664
8	3641.677	3461.139	3791.274

CASE: Two phase commit protocol using presumed abort.
One failure every 100000 units of time.

Total number of transactions generated = 1035
Total number of transactions committed = 1000
Total time elapsed = 130126.195
Mean process/transaction manager in-doubt period = 205.315
Mean generic object manager in-doubt period = 175.333
Mean response time = 1285.480

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	154	155	153	154	206	44	89	45

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	120.733	308	156.610	154
2	64.298	155	130.653	155
3	62.084	153	135.500	153
4	118.784	310	192.915	154
5	177.640	618	254.688	206
6	232.061	176	304.761	44
7	230.032	360	307.027	89
8	315.245	270	382.519	45

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	493.276	253.765	494.384
2	388.441	237.982	412.946
3	525.606	344.435	535.485
4	700.364	480.030	702.522
5	974.414	532.933	979.056
6	4591.208	3160.785	4596.767
7	4413.169	2826.835	4515.355
8	3678.159	3461.139	3791.274

CASE: Two phase commit protocol using presumed abort.
One failure every 30000 units of time.

Total number of transactions generated = 1057
Total number of transactions committed = 1000
Total time elapsed = 133054.984
Mean process/transaction manager in-doubt period = 194.571
Mean generic object manager in-doubt period = 170.215
Mean response time = 1214.016

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	153	157	156	155	207	43	87	42

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	124.757	306	161.643	153
2	60.448	157	121.969	157
3	55.710	156	124.921	156
4	113.631	310	175.481	155
5	167.971	621	237.542	207
6	214.844	172	288.606	43
7	230.251	352	306.127	87
8	318.418	264	365.117	42

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	495.778	253.765	498.230
2	349.261	237.982	385.410
3	520.189	344.435	521.696
4	672.788	480.030	674.927
5	876.578	532.933	902.461
6	4492.702	3160.785	4525.576
7	4352.421	2826.835	4367.053
8	3442.740	3330.643	3791.274

CASE: Two phase commit protocol using presumed abort.
One failure every 12000 units of time.

Total number of transactions generated = 1101
Total number of transactions committed = 1000
Total time elapsed = 140130.469
Mean process/transaction manager in-doubt period = 203.809
Mean generic object manager in-doubt period = 170.285
Mean response time = 1189.204

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	156	161	161	160	212	41	74	35

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	114.461	314	148.847	156
2	69.526	161	127.038	161
3	68.138	161	153.372	161
4	136.656	324	214.878	160
5	179.361	636	250.381	212
6	246.205	164	331.066	41
7	226.354	296	298.466	74
8	295.361	210	353.558	35

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	497.045	253.765	513.899
2	374.450	237.982	385.715
3	568.256	344.435	585.193
4	812.561	480.030	852.145
5	927.987	532.933	932.426
6	4643.237	3160.785	4720.393
7	4326.673	2826.835	4359.419
8	3502.781	3042.464	3791.274

CASE: Two phase commit protocol using presumed abort.
One failure every 7812.5 units of time.

Total number of transactions generated = 1104
Total number of transactions committed = 1000
Total time elapsed = 141976.188
Mean process/transaction manager in-doubt period = 194.532
Mean generic object manager in-doubt period = 168.387
Mean response time = 1175.476

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	157	164	157	159	213	40	79	31

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	117.188	316	149.414	157
2	54.095	164	112.275	164
3	63.042	158	140.934	157
4	130.942	318	205.349	159
5	166.916	640	234.447	213
6	221.699	160	292.193	40
7	207.014	316	278.569	79
8	403.195	186	462.883	31

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	519.012	253.765	576.231
2	376.634	237.982	379.246
3	532.446	338.873	555.910
4	709.942	480.030	730.489
5	917.672	532.933	929.151
6	4566.727	3160.785	4791.098
7	4252.830	2826.835	4320.531
8	3923.846	3233.491	3970.353

CASE: Two phase commit protocol using presumed commit.
No failures.

Total number of transactions generated = 1022
Total number of transactions committed = 1000
Total time elapsed = 127767.258
Mean process/transaction manager in-doubt period = 201.942
Mean generic object manager in-doubt period = 177.419
Mean response time = 1142.024

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	153	153	153	153	203	47	90	48

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	120.229	306	151.490	153
2	66.507	153	130.202	153
3	65.159	153	141.032	153
4	122.139	306	192.798	153
5	189.068	612	251.213	203
6	207.277	188	286.412	47
7	213.925	360	279.575	90
8	325.585	288	377.009	48

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	389.286	208.517	400.589
2	298.271	144.348	298.468
3	457.769	277.998	512.286
4	584.906	339.713	598.307
5	732.846	526.659	799.793
6	4189.423	3250.596	4333.606
7	4151.547	2873.040	4314.868
8	3291.392	2812.532	3485.440

CASE: Two phase commit protocol using presumed commit.
One failure every 100000 units of time.

Total number of transactions generated = 1038
Total number of transactions committed = 1000
Total time elapsed = 130339.305
Mean process/transaction manager in-doubt period = 200.876
Mean generic object manager in-doubt period = 173.733
Mean response time = 1097.376

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	155	156	153	152	206	47	85	46

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	112.925	310	145.284	155
2	64.791	156	126.849	156
3	69.590	153	146.974	153
4	130.524	306	200.956	152
5	194.791	618	256.761	206
6	217.489	188	287.541	47
7	213.880	340	279.706	85
8	282.811	276	333.742	46

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	379.658	208.517	400.589
2	289.023	144.348	290.696
3	468.997	277.998	512.286
4	583.858	339.713	594.244
5	732.068	526.659	799.793
6	4197.764	3250.596	4333.606
7	4007.942	2873.040	4314.868
8	3133.927	2812.532	3485.440

CASE: Two phase commit protocol using presumed commit.
One failure every 30000 time units.

Total number of transactions generated = 1051
Total number of transactions committed = 1000
Total time elapsed = 132661.094
Mean process/transaction manager in-doubt period = 194.676
Mean generic object manager in-doubt period = 170.731
Mean response time = 1100.072

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	152	155	156	154	206	47	89	41

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	102.833	304	137.947	152
2	57.647	156	111.679	155
3	63.441	156	138.242	156
4	129.010	310	192.859	154
5	187.859	618	249.754	206
6	194.011	188	271.381	47
7	221.950	356	292.273	89
8	311.989	246	367.148	41

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	375.550	208.517	391.563
2	269.760	144.348	276.017
3	454.030	277.998	503.580
4	579.095	339.713	579.753
5	745.909	526.659	798.376
6	4161.456	3250.596	4328.029
7	4055.513	2873.040	4258.543
8	3194.554	2812.532	3485.440

CASE: Two phase commit protocol using presumed commit.
One failure every 12000 units of time.

Total number of transactions generated = 1091
Total number of transactions committed = 1000
Total time elapsed = 139558.656
Mean process/transaction manager in-doubt period = 197.404
Mean generic object manager in-doubt period = 177.132
Mean response time = 1038.545

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	154	160	161	158	209	41	79	38

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	115.021	307	148.229	154
2	64.585	161	120.012	160
3	58.050	161	133.493	161
4	119.954	314	189.989	158
5	178.980	628	240.076	209
6	280.662	164	368.864	41
7	229.508	317	294.607	79
8	350.664	228	404.395	38

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	386.593	208.517	388.622
2	291.636	144.348	296.283
3	455.473	277.998	457.231
4	560.284	339.713	584.626
5	720.036	526.659	723.447
6	4459.612	3250.596	4485.259
7	4008.714	2873.040	4008.714
8	3170.241	2812.532	3205.617

CASE: Two phase commit protocol using presumed commit.
One failure every 7812.5 units of time.

Total number of transactions generated = 1089
Total number of transactions committed = 1000
Total time elapsed = 138978.266
Mean process/transaction manager in-doubt period = 192.401
Mean generic object manager in-doubt period = 167.611
Mean response time = 1026.134

DISTRIBUTION OF COMMITTED TRANSACTIONS

Type	1	2	3	4	5	6	7	8
Number	153	160	156	159	210	41	86	35

MEAN IN DOUBT PERIOD FOR OBJECT MANAGERS BY TRANSACTION TYPE

Type	Generic Object Manager		Process/Transaction Manager	
	Period	Number	Period	Number
1	131.445	306	164.766	153
2	58.137	160	127.204	160
3	71.027	156	127.104	156
4	129.044	320	184.268	159
5	165.459	634	229.569	210
6	244.099	165	298.619	41
7	213.013	342	284.172	86
8	308.685	207	365.208	35

RESPONSE TIMES FOR COMMITTED TRANSACTIONS BY TYPE

Type	Mean	Minimum	Maximum
1	403.499	208.517	408.504
2	287.113	144.348	298.014
3	429.594	277.998	436.213
4	553.346	339.713	582.821
5	676.315	526.659	683.577
6	4214.406	3250.596	4234.385
7	3919.276	2873.040	4068.864
8	3188.158	2711.962	3288.440

2.2 Failure Statistics

The statistics about the failures that occurred for each run of the model are presented in this section. For each run of the model (e.g., a particular commit protocol and fault injection rate) a number of statistics are collected. They are organized into the following four tables:

- * failure and fault event totals and distributions,
- * causes of ptm timeouts,
- * state of transactions aborted due to a site crash, and
- * state of transactions aborted due to a PTM timeout.

The contents and use of each of these tables is described below.

The failure and fault event table summarizes the number of events and their effect on a run of the model. The events considered are site failures, timeouts, and retransmissions. All of the events are counted from the perspective of the PTM in terms of the number of transactions effected. Since there is a PTM on every site, all instances of failures are detected. The effect of a failure on a transaction is to abort the transaction unless the transaction is in the second phase of a two phase commit protocol in which case the effect is to repeat the unsuccessful operation. The timeout events can be further divided into those that occur during a normal operation (e.g., a remote procedure call) and those that occur during the first phase of a two phase commit protocol (e.g., a prepare message). For each event the total number of transactions effected by the event is listed as well as the number of transactions of each type. This allows the susceptibility of a given type of transaction to a particular event while executing a particular commit protocol to be determined.

The causes of PTM timeout table elaborates on why timeouts occurred. There are three reasons for the PTM to timeout -- there may be resource contention on the server's site, the remote site may crash during a transaction, or the remote site may be down when the transaction is initiated. The table lists what percentage of the timeouts are due to which reason.

Unfortunately, most of the statistics that measure the effect of site crashes are somewhat questionable due to the short down period assumed for each site (necessitated by the cost of executing a simulation). For example, it is inappropriate to assume that because there is a greater number of transactions aborted due to timeouts than site failures that timeout values are of greater importance than site availability in increasing system throughput and reliability. This may or may not be true, but the model cannot justifiably aid in such conclusions.

The last two tables describe the progress of transactions that were aborted due to either a PTM site crash or a PTM remote procedure call (RPC) timeout. The progress is measured in terms of the percentage of object managers of a transaction that have been successfully accessed when a failure occurs. For example, if a transaction is to access ten object managers and has successfully issued an RPC to five of them, the transaction would be considered to be 50% complete. Note that this does not measure progress in terms of the number of operations within a transaction that are completed.

This can be estimated, however, by correlating the percentage of object managers accessed with the transcript of the transaction type (listed in section 1.3).

Not every run has a complete set of tables. For instance in the no fault cases there will be no statistics describing the progress of transactions failed due to site crashes since there were no site crashes. The data is organized by commit protocol and presented in the following order: one phase presumed abort, one phase presumed commit, two phase presumed abort, and two phase presumed commit. The data for each commit protocol is organized by fault rate with the no fault case presented first followed by the next higher fault rate and so on.

	FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE							
	Transaction Type							
Event	1	2	3	4	5	6	7	8 Total
PIM Site Failure	0	0	0	0	0	0	0	0
PIM RPL Timeout	0	0	0	0	0	0	0	0
PIM Retransmission	0	0	0	0	0	0	0	0

CASE: One phase commit protocol using presumed abort.
 One failure every 100,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	0	0	1	0	0	0	0	1	2
PTM RPC Timeout	0	0	1	1	0	2	2	1	7
PTM Retransmission	0	0	0	0	0	0	0	1	1

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	28.6
Server Site Crash During Transaction	28.6
Server Site Down At Start Of Transaction	42.9

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Percent of object managers accessed (<=)

Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Percent of object managers accessed (<=)

Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					

CASE: One phase commit protocol using presumed abort.
 One failure every 30,000 msec.
 PTM Timeout = 1200 msec.
 GUM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	1	0	1	1	2	1	4	1	11
PTM RPC Timeout	5	1	0	2	2	4	4	1	20
PTM Retransmission	0	2	0	1	1	1	0	1	6

CAUSES OF PTM TIMEOUT		Percent
Cause		
Resource Contention		40.0
Server Site Crash During Transaction		25.0
Server Site Down At Start Of Transaction		35.0

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Percent of object managers accessed ($\leq$)

Transaction type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Percent of object managers accessed ($\leq$)

Transaction type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					

CASE: One phase commit protocol using presumed abort.
 One failure every 12,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	1	0	0	0	2	1	3	2	9
PTM RPC Timeout	6	2	1	3	3	3	7	6	31
PTM Retransmission	2	2	0	2	1	0	7	5	19

CAUSES OF PTM TIMEOUT		Percent
Cause		
Resource Contention		29.0
Server Site Crash During Transaction		32.3
Server Site Down At Start of Transaction		38.7

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

DISTRIBUTION OF PROCESS REQUESTS BY TRANSACTION TYPE																							
Transaction Type	Percent of object managers accessed (<=)																				99	100	
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94				
1																						1	
2																						2	
3																							
4																							
5																						2	
6																						2	
7																						1	
8																							

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ACCORDING TO PERCENT OF OBJECT MANAGERS ACCESSED (<=)																							
Transaction type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100		
1																							
2	2										4										1		
3	2																						
4	3																						
5	3																						
6						2															2		
7	2					1					2										2		
8	1			2																			

CASE: One phase commit protocol using presumed abort.
 One failure every 7812.5 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	1	0	0	0	1	2	6	2	12
PTM RPC Timeout	8	3	3	4	5	4	6	14	47
PTM Retransmission	0	0	2	2	0	1	3	1	9

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	25.5
Server Site Crash During Transactions	24.0
Server Site Down At Start Of Transaction	40.4

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Percent of object managers accessed (<=)

Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Percent of object managers accessed (<=)

Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					

3

[illegible]

CASE: One phase commit protocol using presumed commit.
 One failure every 100,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PIM Site Failure	0	0	0	0	0	1	2	0	3
PIM RPC Timeout	0	0	1	2	0	1	0	1	5
PIM Retransmission	0	0	0	0	0	0	0	0	0

CAUSES OF PIM TIMEOUT

Cause	Percent
Resource Contention	60.0
Server Site Crash During Transaction	20.0
Server Site Down At Start Of Transaction	20.0

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Transaction Type	Percent of object managers accessed (<=)															
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79
1																
2																
3																
4																
5																
6																
7																
8																

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Transaction Type	Percent of object managers accessed (<=)															
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79
1																
2																
3																
4																
5																
6																
7																
8																

CASE: One phase commit protocol using presumed commit.
 One failure every 30,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	0	0	1	0	1	5	2	2	11
PTM RPC Timeout	4	1	0	2	1	0	3	5	16
PTM Retransmission	0	0	0	0	0	0	0	0	0

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	18.8
Server Site Crash During Transaction	31.3
Server Site Down At Start Of Transaction	50.0

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ASSOCIATED WITH OBJECT MANAGERS																					
Transaction Type	Percent of object managers accessed (<=)																			99	100
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94		
1																					
2																					
3																					
4																					
5														1							
6											1					2					
7						2										1					
8																	1				

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

DISTRIBUTION OF PROGRESS OF TRANSACTIONS REPORTED FOR THE YEAR 1983																					
Transaction Type	Percent of object managers accessed (<=)																			99	100
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94		
1																					
2	2										2										
3	1																				
4	2																				
5	1																				
6						1					2							2			
7																				1	
8																					

CASE: One phase commit protocol using presumed commit.
 One failure every 12,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	0	0	0	0	0	3	6	2	11
PTM RPC Timeout	5	3	1	3	5	2	6	8	33
PTM Retransmission	0	0	0	0	0	0	0	0	0

Cause	CAUSES OF PTM TIMEOUT								Percent
	1	2	3	4	5	6	7	8	
Resource Contention									36.4
Server Site Crash During Transaction									27.3
Server Site Down At Start Of Transaction									36.4

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Transaction Type	Percent of object managers accessed (<=)																99	100		
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	
1																				
2																				
3																				
4																				
5																1				
6																				1
7																				2
8																	1			1

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Transaction Type	Percent of object managers accessed (<=)																99	100		
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79			84	89
1																				
2	2																			
3	3																			
4	1																			
5	3																			
6	4													1						
7	1																			
8	1																			

CASE: One phase commit protocol using presumed commit.
 One failure every 7812.5 units of time.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	0	0	0	1	1	3	7	1	13
PTM RPC Timeout	10	3	4	3	5	10	15	1	55
PTM Retransmission	0	0	0	0	0	0	0	0	0

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	32.7
Server Site Crash During Transaction	29.1
Server Site Down At Start Of Transaction	38.2

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PGM SIZE																						
Transaction Type	Percent of object managers accessed (<=)																					
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1																					1	
2																					1	
3																					5	
4																					1	
5																					1	
6																						
7																						
8																						

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT																							
Transaction type	Percent of object managers accessed (<=)																						
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100		
1											6										1		
2	4																						
3	3																						
4	3																						
5	5										1					1					1		
6	1										5										2		
7	3							2			1										5		
8	2																3						

TABLE 10 EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
PIM Site Failure	0	0	0	0	0	0	0	0	0
PIM RPC Timeout	0	0	0	0	0	0	0	0	0
PIM Prepare Timeout	0	0	0	0	0	9	10	2	21
PIM Retransmission	0	0	0	0	0	0	0	0	0

CAUSES OF PTM TIMEOUT	
Cause	Percent
Resource Contention	100.0
Server Site Crash During Transaction	0.0
Server Site DownAt Start Of Transaction	0.0

CASE: Two phase commit protocol using presumed abort.
 One failure every 100,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	0	0	0	0	1	1	2	0	4
PTM RPC Timeout	0	0	1	0	0	0	1	1	3
PTM Prepare Timeout	0	0	1	0	0	5	7	4	17
PTM Retransmission	0	0	0	0	0	0	0	0	0

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	80.0
Server Site Crash During Transaction	15.0
Server Site Down At Start Of Transaction	5.0

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Transaction Type	Percent of object managers accessed (<=)															
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79
1																
2																
3																
4																
5																
6																
7																
8																

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Transaction type	Percent of object managers accessed (<=)															
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79
1																
2																
3																
4																
5																
6																
7																
8																

CASE: Two phase commit protocol using presumed abort.
 One failure every 30,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	0	0	1	1	1	1	3	0	7
PTM RPC Timeout	4	1	1	2	2	1	5	7	23
PTM Prepare timeout	1	0	0	0	0	6	8	1	16
PTM Retransmission	0	0	0	0	0	0	0	2	2

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	46.2
Server Site Crash During Transaction	30.8
Server Site Down At Start Of Transaction	23.1

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

DISTRIBUTION OF PROGRESS OF TRANSACTIONS RECORDED																						
Transaction type	Percent of object managers accessed (≤)																					
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1																						
2																						
3																						
4																						
5																						
6																						
7																						
8																						

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

DISTRIBUTION OF PROGRESS OF TRANSACTIONS REPORTED OVER PERCENT OF OBJECT MANAGERS ACCESSED (<=)																						
Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1											2											
2																					1	
3																						
4																						
5																						
6																						
7											2						2				1	
8																					3	

CASE: Two phase commit protocol using presumed abort.
 One failure every 12,000 msec.
 PIM Timeout = 1200 msec.
 GDM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PIM Site Failure	0	0	1	3	1	4	6	1	16
PIM RPC Timeout	7	4	3	3	6	2	8	11	43
PIM Prepare Timeout	1	0	0	0	0	8	18	6	33
PIM Retransmission	2	0	0	3	0	1	0	2	8

CAUSES OF PIM TIMEOUT		Percent	
Cause			
Resource Contention			54.5
Server Site Crash During Transaction			26.0
Server Site Down At Start Of Transaction			19.5

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM SITE CRASH

Percent of object managers accessed (<=)

Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					1
3																					3
4																					
5											2										1
6										3											3
7																					
8																					

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM RPC TIMEOUT

Percent of object managers accessed (<=)

Transaction Type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					1
3																					
4																					
5																					
6																					1
7																					2
8																					

(CASE: Two phase commit protocol using presumed abort.
 One failure every 7812.5 units of time.
 PIM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.)

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PIM Site Failure	1	1	5	2	1	0	9	5	26
PIM RPC Timeout	8	2	3	4	4	5	12	14	52
PIM Prepare Timeout	0	0	0	0	0	9	8	4	21
PIM Retransmission	3	0	0	0	0	2	2	0	7

CAUSES OF PIM TIMEOUT		Percent	
Cause			
Resource Contention			47.9
Server Site Crash During Transaction			30.1
Server Site Down At Start Of Transaction			21.9

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM SITE CRASH

		Percent of object managers accessed (%)																					
Transaction type		4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1																						1	
2																						5	
3																						2	
4																							
5															3								
6																						6	
7							1											2				3	
8																							

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM RPC TIMEOUT

		Percent of object managers accessed (%)																				
Transaction type		4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																						
2																						
3																						
4																						
5																						
6																						
7																						
8																						

CASE: Two phase commit protocol using presumed commit.

No failures

P1M Timeout = 1200 msec.

GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
P1M Site Failure	0	0	0	0	0	0	0	0	0
P1M RPC Timeout	0	0	0	0	0	0	0	0	0
P1M Prepare Timeout	0	0	0	0	0	0	0	0	0
P1M Retransmission	0	0	0	0	0	3	6	1	10

CASE: Two phase commit protocol using presumed commit.
 One failure every 100,000 msec.
 PIM Timeout - 1200 msec.
 GOM Timeout - 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
PIM Site Failure	0	0	0	0	1	2	2	1	6
PIM RPC Timeout	0	0	2	1	0	0	3	1	7
PIM Prepare Timeout	0	0	0	1	0	0	0	0	1
PIM Retransmission	0	0	0	1	0	2	6	1	10

CAUSES OF PIM TIMEOUT

Cause	Percent
Resource Contention	25.0
Server Site Crash During Transaction	50.0
Server Site Down At Start Of Transaction	25.0

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM SITE CRASH

Transaction type	Percent of object managers accessed (%)																				
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					
2																					
3																					
4																					
5																					1
6																					2
7																					
8																					

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM RPC TIMEOUT

Transaction type	Percent of object managers accessed (%)																					
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1																						
2																						
3																						
4																						
5																						
6																						
7																						
8																						

CASE: Two phase commit protocol using presumed commit.
 One failure every 30,000 msec.
 PTM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	1	0	0	1	0	1	4	3	10
PTM RPC Timeout	4	2	1	2	2	1	4	5	21
PTM Prepare Timeout	0	0	0	0	1	1	0	0	2
PTM Retransmission	0	0	0	0	1	0	6	1	8

CAUSES OF PTM TIMEOUT

Cause	Percent
Resource Contention	34.8
Server Site Crash During Transaction	34.8
Server Site Down At Start Of Transaction	30.4

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Transaction Type	Percent of object managers accessed (<=)																					
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1																						1
2																						1
3																						1
4																						1
5																						1
6																						3
7																						1
8																						1

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Transaction Type	Percent of object managers accessed (<=)																						Total
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100		
1																						1	
2	2										2											1	
3																							
4	2																						
5	1													1									
6																1							
7	1										2											1	
8																						3	

CASE: Two phase commit protocol using presumed commit.
 One failure every 12,000 msec.
 PTM Timeout - 1200 msec.
 GOM Timeout - 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction type								Total
	1	2	3	4	5	6	7	8	
PTM Site Failure	1	0	2	2	5	4	6	3	23
PTM RPC Timeout	8	3	0	3	4	4	10	11	43
PTM Prepare Timeout	0	0	0	0	0	2	2	1	5
PTM Retransmission	0	0	0	0	0	1	7	0	8

CAUSES OF PTM TIMEOUT		Percent
Cause		
Resource Contention		33.3
Server Site Crash During Transaction		29.2
Server Site Down At Start Of Transaction		37.5

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM SITE CRASH

Percent of object managers accessed (<=)

Transaction type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					1
2																					2
3																					2
4																					1
5																					3
6																					3
7																					1
8																					1

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PTM RPC TIMEOUT

Percent of object managers accessed (<=)

Transaction type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1	4																				
2	3																				
3	3																				
4	4																				
5	1																				
6	1																				
7	2																				
8																					

CASE: Two phase commit protocol using presumed commit.
 One failure every 7812.5 units of time.
 PIM Timeout = 1200 msec.
 GOM Timeout = 4500 msec.

FAILURE/FAULT EVENT TOTALS AND DISTRIBUTION BY TRANSACTION TYPE

Event	Transaction Type								Total
	1	2	3	4	5	6	7	8	
PIM Site Failure	2	0	1	2	2	5	6	4	22
PIM RPC Timeout	8	3	5	3	5	6	8	12	50
PIM Prepare Timeout	0	0	1	0	0	1	2	1	5
PIM Retransmission	0	0	1	0	0	1	4	1	7

CAUSES OF PIM TIMEOUT

Cause	Percent
Resource Failures	23.6
Server Site Crash During Transaction	32.7
Server Site Down At Start Of Transaction	43.6

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM SITE CRASH

Transaction Type	Percent of object managers accessed (<=)																				
	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100
1																					2
2																					1
3																					2
4																					2
5											1					2					2
6											4						1				2
7																					3
8																					3

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO PIM RPC TIMEOUT

DISTRIBUTION OF PROGRESS OF TRANSACTIONS ABORTED DUE TO THE PERCENT OF OBJECT MANAGERS ACCESSED (<=)																						
Transaction type	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79	84	89	94	99	100	
1	3	1	1	1	1	1	1	1	1	1	4	1	1	1	1	1	1	1	1	1	1	
2	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
3	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
4	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
5	4	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1	1	1	1	1	1	
6	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
7	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
8	1	1	1	3	1	2	1	1	1	1	1	1	1	1	1	1	3	1	1	1	2	

2.3 Data Analysis

The system structure allowed the evaluation of four design alternatives for commit protocols. These included one phase versus two phase in combination with presumed abort versus presumed commit.

The presume abort protocol implies that in the absence of any information about a transaction's commitment at its coordinator, it is presumed that the transaction was aborted. This requires that for a committed transaction a coordinator must keep the transaction's commit status information until it is certain that it will not receive future status queries. Analogously, presumed commit implies that in the absence of a commit record a transaction is presumed to have committed. When a transaction is aborted the coordinator must maintain its abort status until it is certain that it will not receive future status queries. In an environment where failures are rare and the majority of the transactions are committed, presumed commit should provide superior performance to that of presumed abort. Similarly, in the presence of a large number of failures and many transactions being aborted presumed abort should have superior performance. As the failure rate increases there should exist a point of inflexion where it is more advantageous to use the presumed abort protocol. This point was determined by varying the fault rate.

The other major design option investigated was one phase versus two phase commitment. The one phase commit protocol implies that in response to every update operation on an object, its type manager creates a new commit pending version of the object on stable storage. The object remains in the commit pending state until the type manager receives the decision about the outcome of the transaction. An object that is in the commit pending state is unavailable to other transactions until the transaction is terminated. A coordinator failure while an object is in the commit pending state causes the object to become unavailable until the coordinator recovers. The period during which the object is in the commit pending state is called the in-doubt period or window of vulnerability. The two phase commit protocol tends to reduce the window of vulnerability. In this protocol, each update operation creates an uncommitted version of the object. At the end of a transaction, the coordinator attempts to cause the objects accessed by the transaction to transition into a commit pending state during the first phase of the protocol. A commit or abort decision is then made identical to that of a one phase commit protocol. The two phase commit protocol reduces the window of vulnerability at the expense of extra messages. One would expect that two phase would be desirable for long transactions and one phase for short transactions.

The model attempted to investigate under what failure environments for what kinds of applications it is appropriate to select which options for a commit protocol. The transaction throughput summary figure (page J-60) shows the effect of a commit protocol on the throughput of the overall workload as the failure rate increases. It demonstrates the performance degradation due to the increasing occurrence of faults. The next four figures (pages J-61 - J-64) depict the effect of a specific commit protocol on the different types of transactions. Each figure shows how a protocol may favor one type of transactions over another. This is further visible by contrasting the effect of different protocols on a single transaction type.

There are three main points to note -- the effect of presumed abort versus presumed commit, of two phase versus one phase, and of timeout periods. Presumed commit protocols outperform presumed abort protocols for low fault rates as expected. But for one phase protocols, the presumed abort protocol outperforms the presumed commit protocol when the fault rate exceeds 5 faults/100 seconds. This indicates that it may be desirable to have an adaptive commit algorithm that uses a presumed commit protocol when the environment is not faulty and switches to a presumed abort protocol when a fault rate surpasses a given threshold.

One phase protocols outperform two phase protocols. This is not surprising for environments with a low fault rate. It is somewhat surprising for environments with a very high fault rate. This can be explained by two observations. First, the slope of the curve of a two phase protocol tends to decrease more rapidly than that of one phase protocols indicating that there may exist some fault rate at which two phase protocols do indeed outperform one phase protocols. Second, the model of the time duration of a device failure is unrealistically short. This is due to the excessive time and resources that would be required to run a simulation that accurately modelled a device's downtime. The effect on a one phase commit protocol of a longer downtime is to increase the size of the window of vulnerability (or in-doubt period) of a server to the failure of a coordinator. This would increase the period during which a set of objects would be indefinitely blocked, thereby reducing the potential system concurrency and therefore throughput.

The timeout period which an object manager holds a lock for a transaction using a two phase commit protocol can unduly effect the throughput. A short period may result in many transactions being aborted unnecessarily. This is explained as follows. As the multiprogramming level increases, the concurrency level of the object managers increases. When the objects are reliably manipulated there are extra disk accesses to store stable states. The increased number of disk accesses results in a bottleneck at the stable storage device. This results in a longer period of time that a transaction waits for a response from an object manager and that an object manager holds a lock for a transaction. As the stable storage request queue grows, the number of aborted transactions increases. Note that this does not happen for one phase commit protocols because objects are placed in a commit pending state following their first access, nor does it happen for environments with a high fault rate because the multiprogramming level is reduced. This problem can be avoided for a two phase protocol, if either the multiprogramming level is reduced or the timeout period increased. This explains why the throughput of the two phase presumed abort run with no faults is as low as the few faults run.

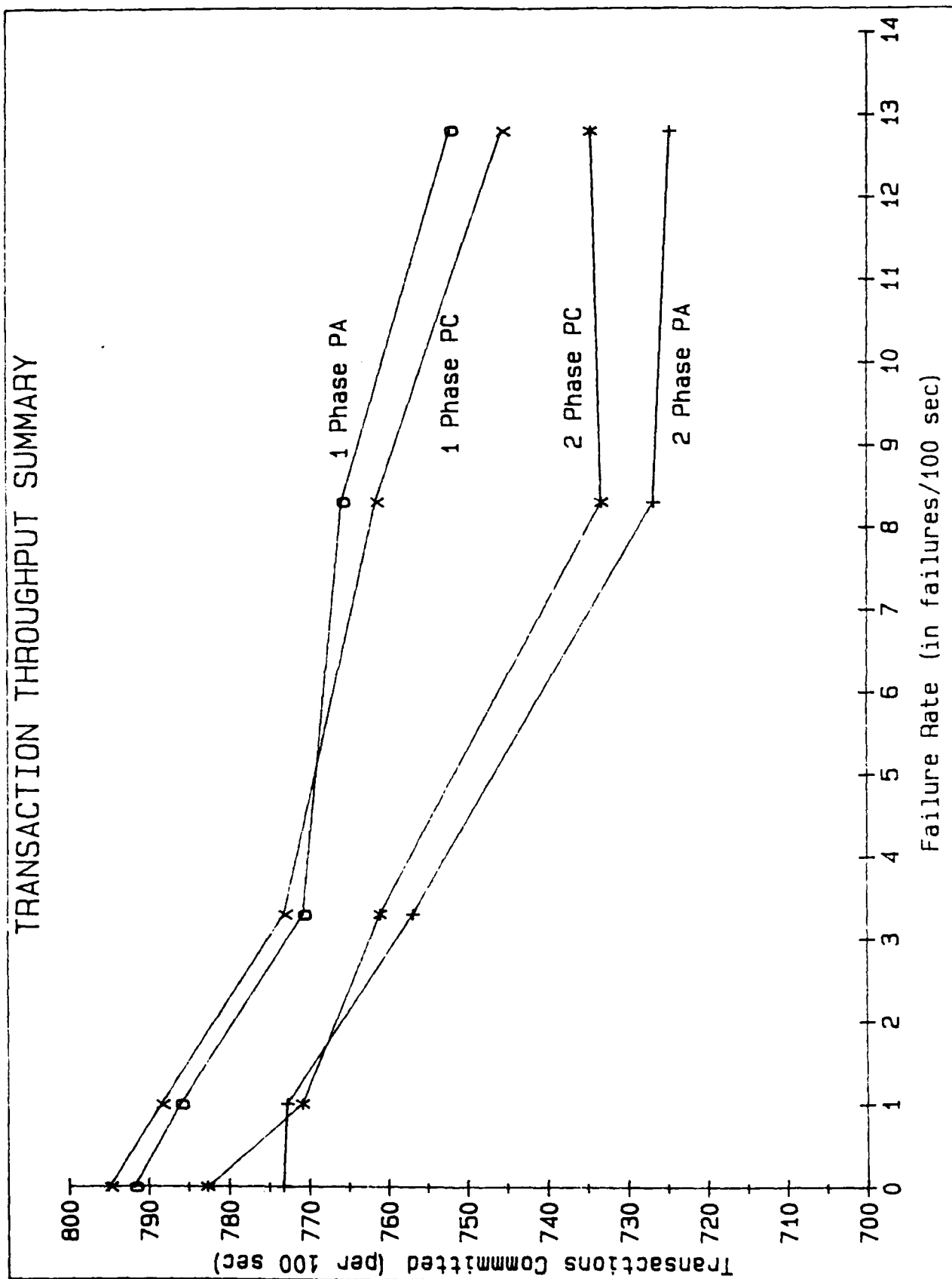
The effect of the commit protocol with varying fault rates on the workload mix is shown from a slightly different viewpoint in the next four figures (pages J-65 - J-68). It demonstrates that as the fault rate increases shorter transactions tend to dominate the mix of successful transactions. Short appears not to be sensitive to the number of objects accessed, but to the number of operations on the set of objects. Further, the overhead of a two phase protocol did not seem to be warranted for short transactions under any fault rate.

The next set of figures (pages J-69 - J-77) show the response time for successfully committed transactions. The response time summary figure shows the average response time for a transaction executing a given commit protocol as the rate of failures increases. This is followed by eight figures, each showing the effect of the different commit protocols on an individual transaction type. The average response time for all transactions decreases as failures increase even though the average response time for each transaction type tends to increase or stay the same as failures increase. This is due to two facts. As the number of failures increases the number of large transactions that fail increase thus resulting in a proportionately larger number of small transactions succeeding. Since the response time only reflects successfully committed transactions and the proportion of transactions with lower response times increases with an increase of failures, the overall average response time decreases. Another contributing factor is that as the number of failures increase the number of free objects increase because the effective concurrency level (e.g. objects held by successful transactions) falls.

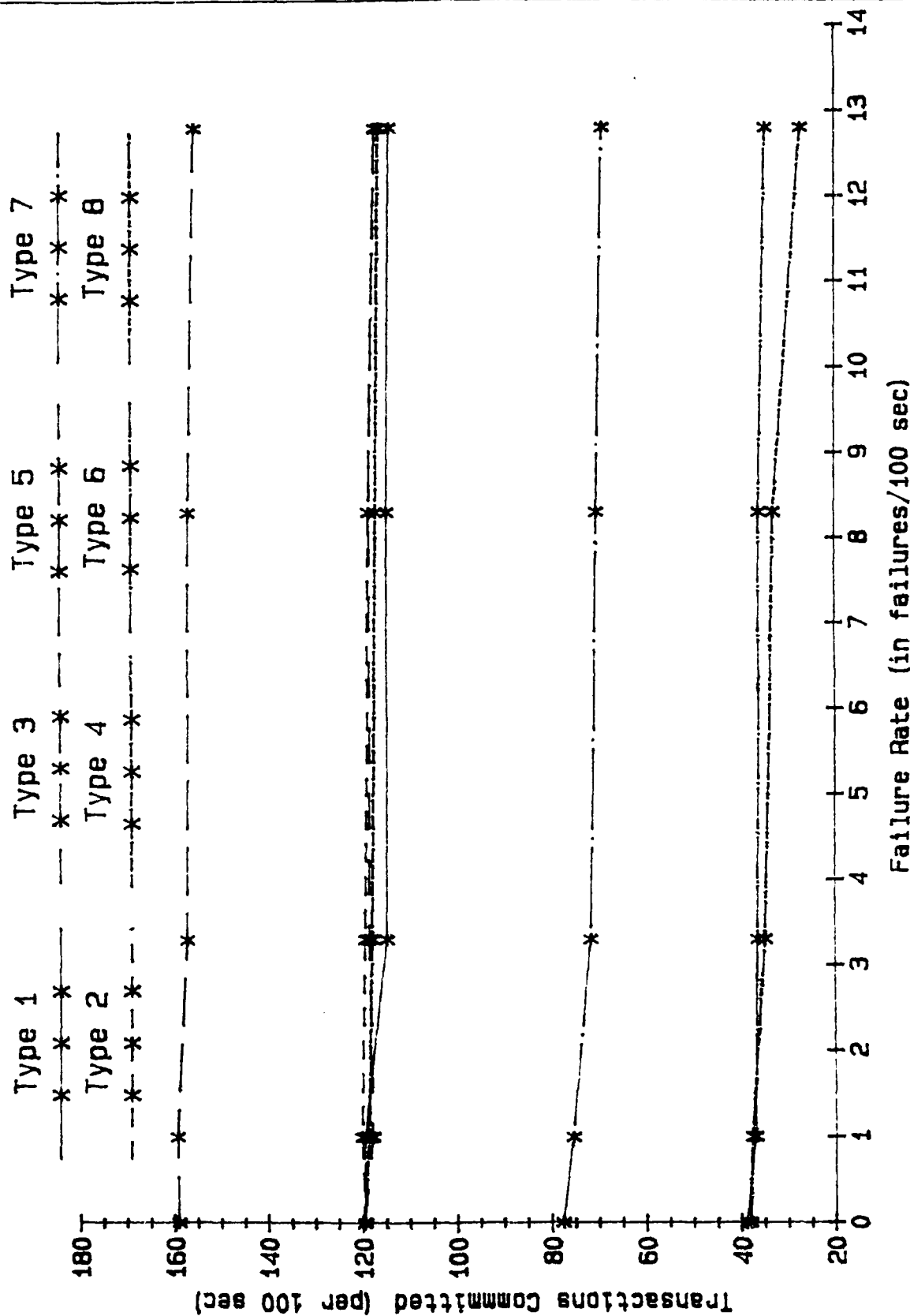
One other interesting point is that the response time of presumed abort protocols did not fare as well as that of presumed commit protocols. This is easily explained by noting that presumed commit is optimized for successful transaction and that we are measuring only successful transactions. The added response time of presumed abort represents the extra acknowledgment that must be sent for a successful transaction. The advantage of presumed abort versus presumed commit is depicted in the transaction throughput summary figure. It is best measured in the number of transactions completed rather than the time for a transaction to complete.

The final figure (page J-78) shows the average size of the object manager in-doubt period for all transactions as failures increase for the four protocols. Unfortunately, this only includes successful transactions so it does not really show the effect of the increase of failures and commit protocol choice on object availability. It does clearly show that the average period during which an object is in a commit pending state, and thereby vulnerable to the failure of a coordinator, is much smaller for a two phase commit protocol than for a one phase commit protocol.

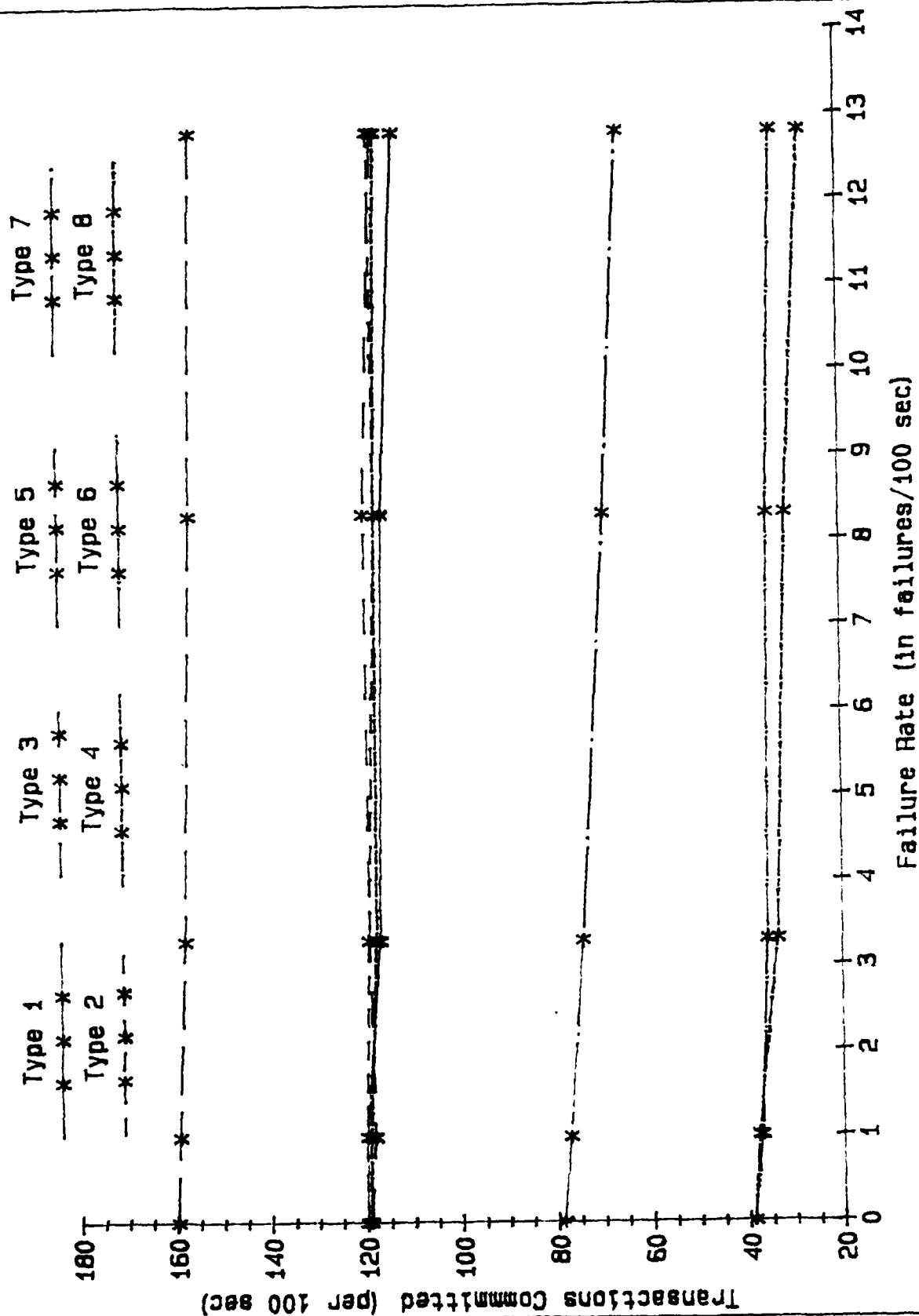
TRANSACTION THROUGHPUT SUMMARY



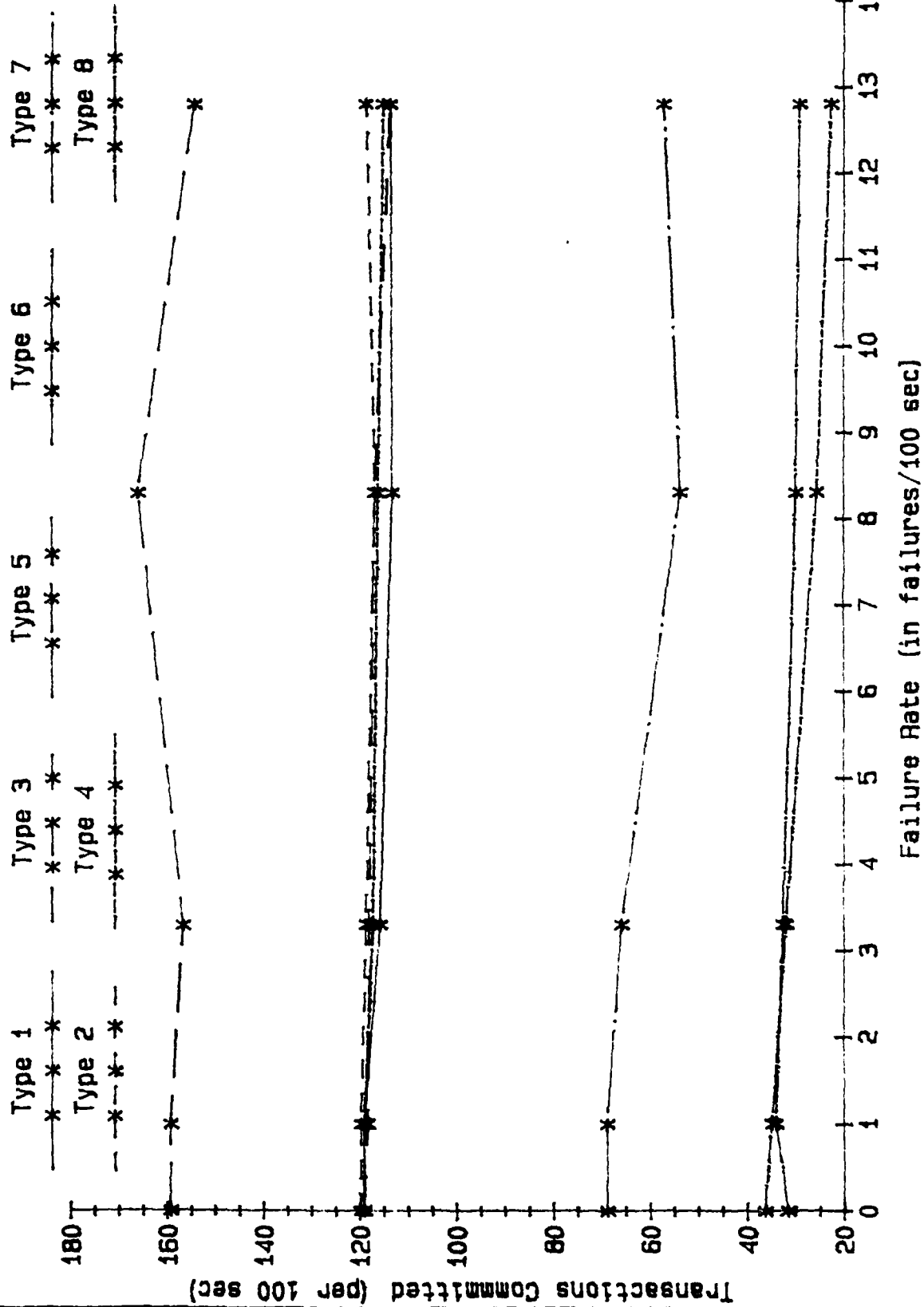
EFFECT OF ONE PHASE PRESUMED ABORT ON TRANSACTION TYPE THRUPUT



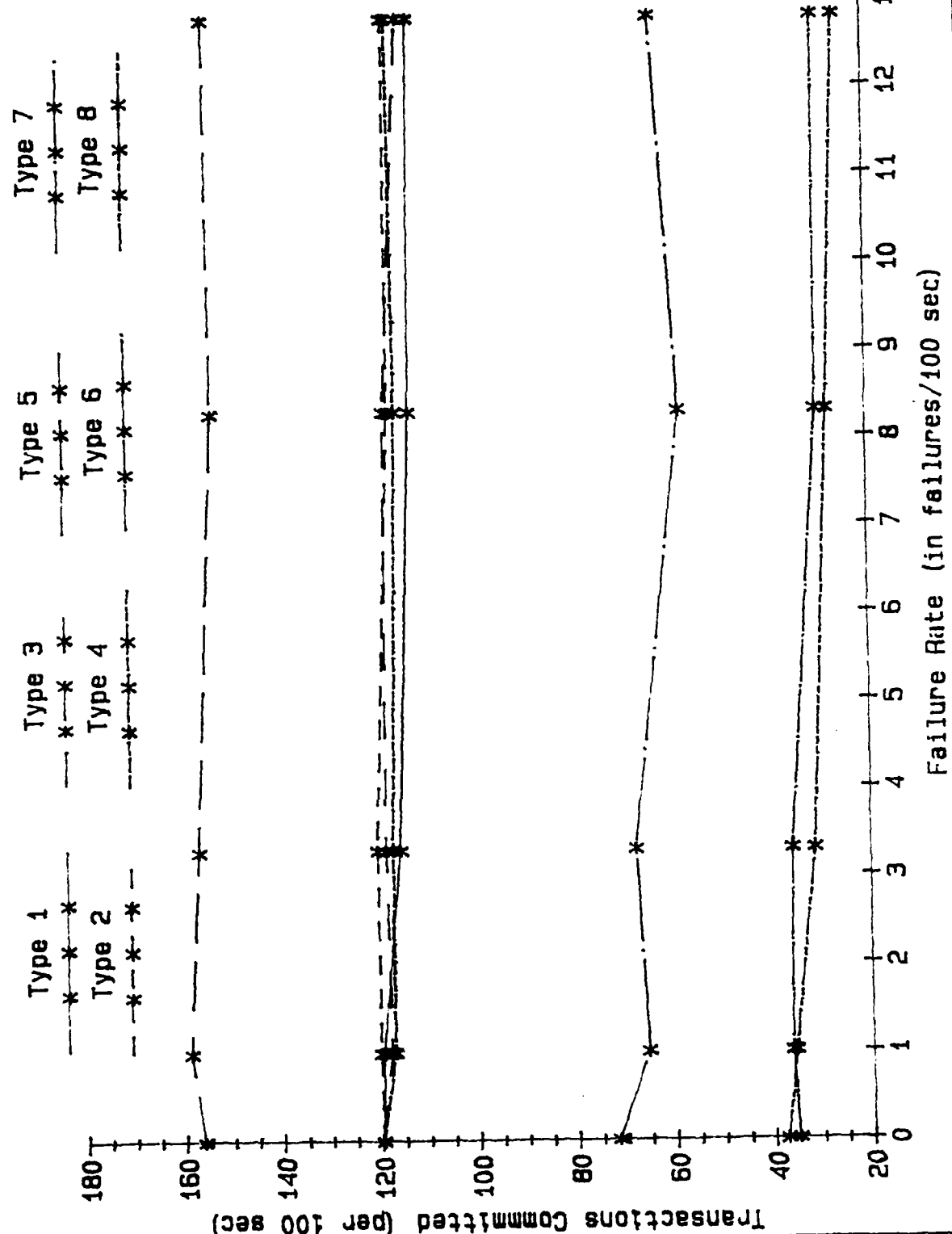
EFFECT OF ONE PHASE PRESUMED COMMIT ON TRANSACTION TYPE THRUPUT



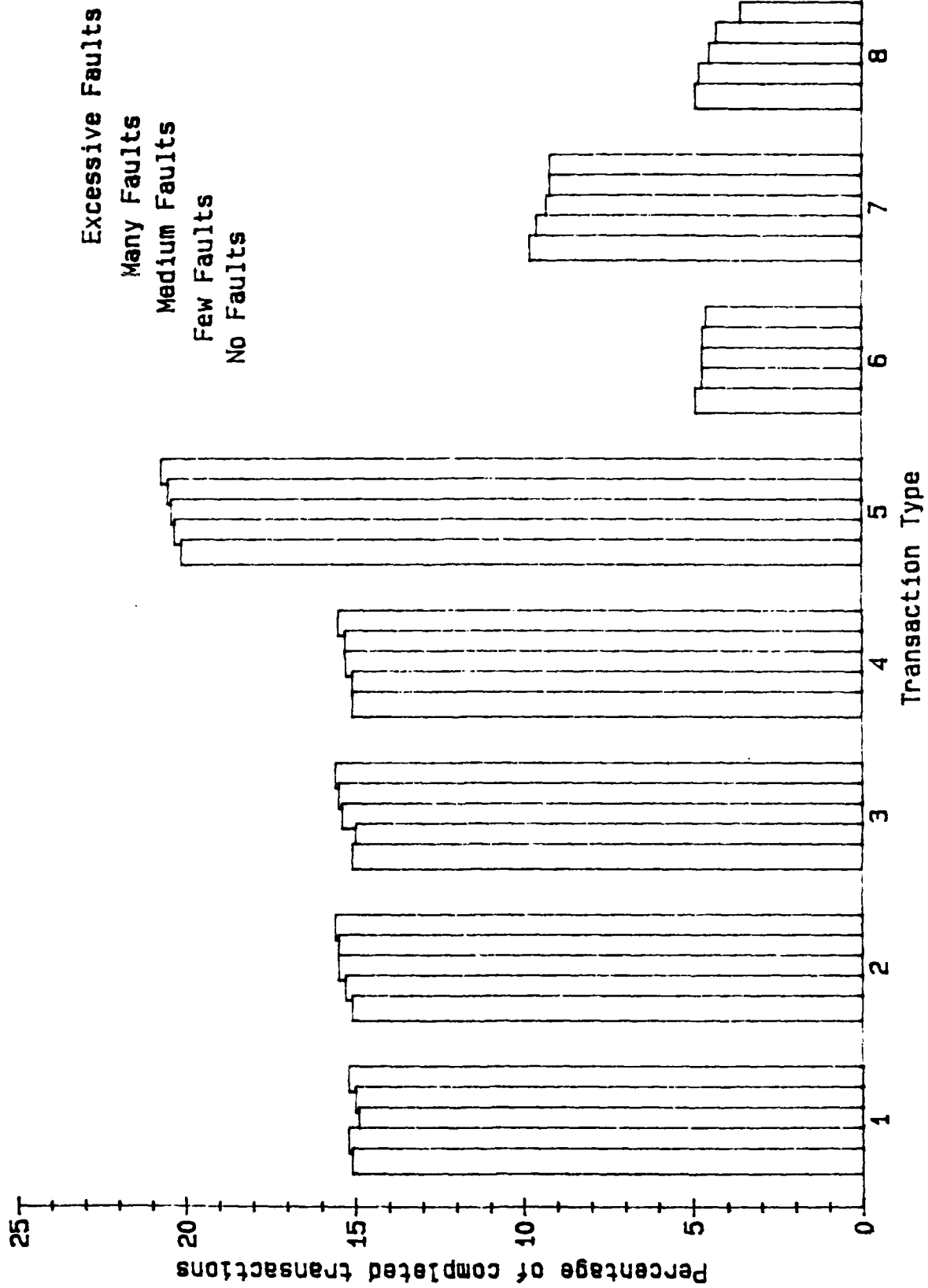
EFFECT OF TWO PHASE PRESUMED ABORT ON TRANSACTION TYPE THRUPUT



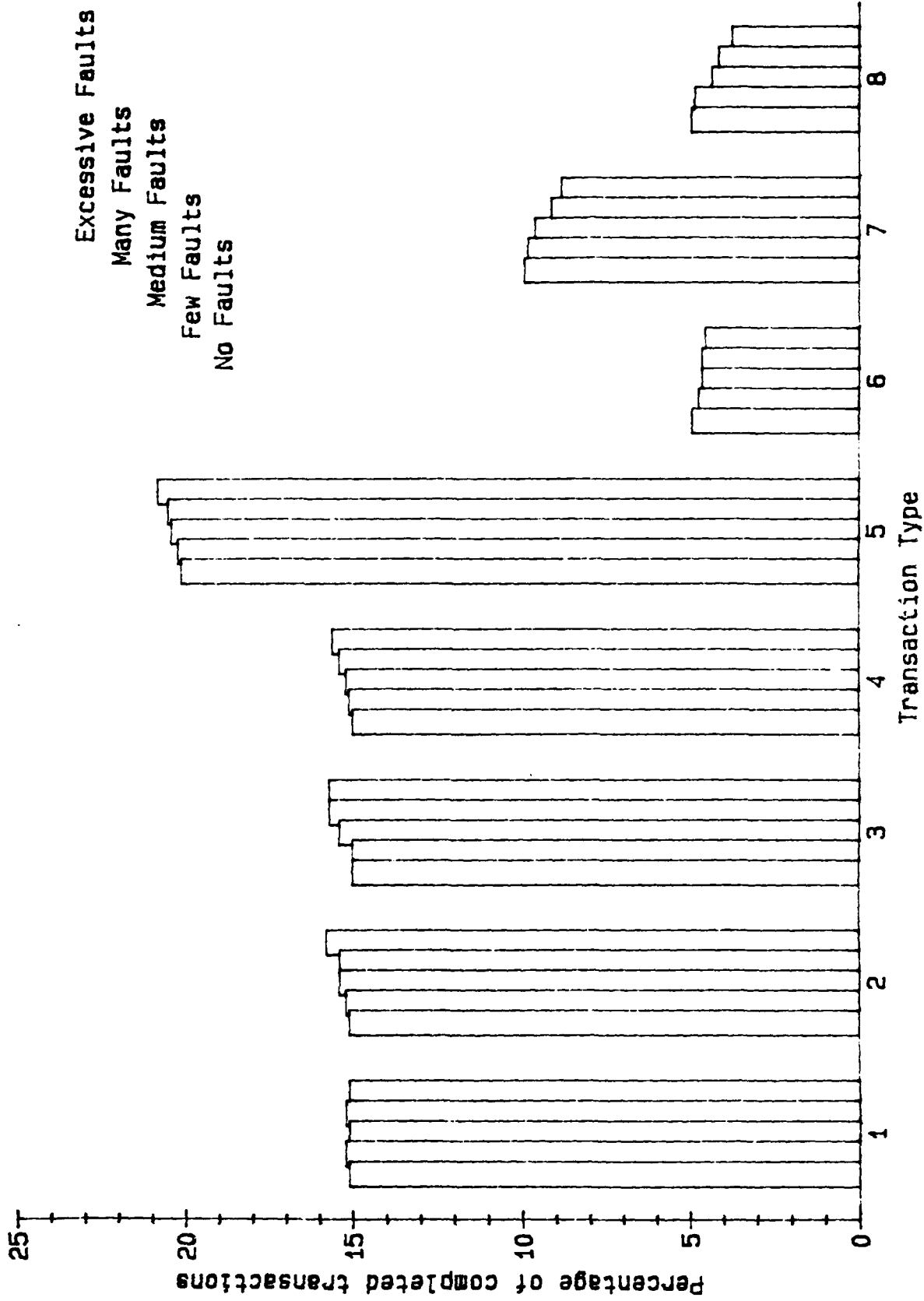
EFFECT OF TWO PHASE PRESUMED COMMIT ON TRANSACTION TYPE THRUPUT



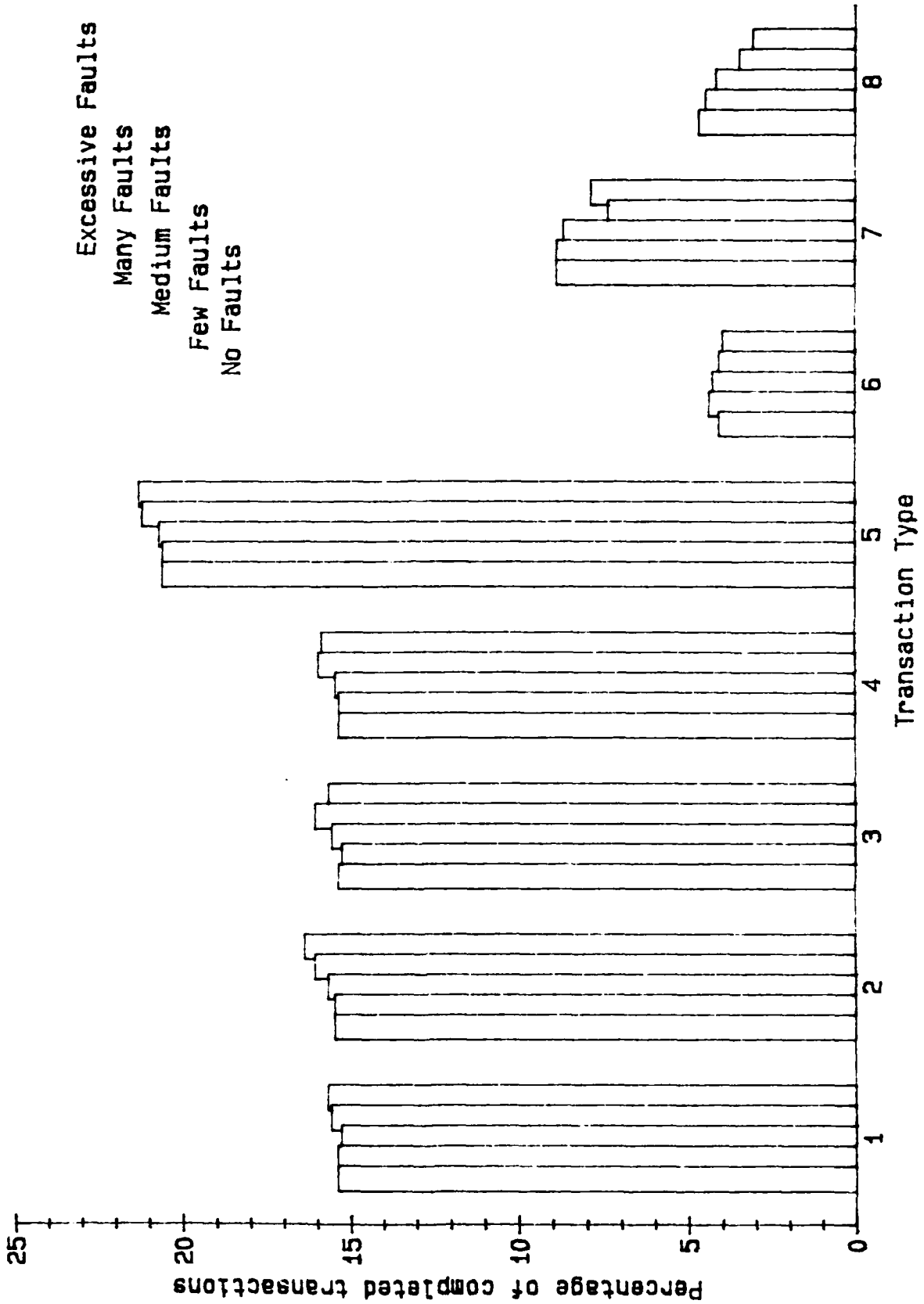
Effect of 1 Phase Presumed Abort on Transaction Mix



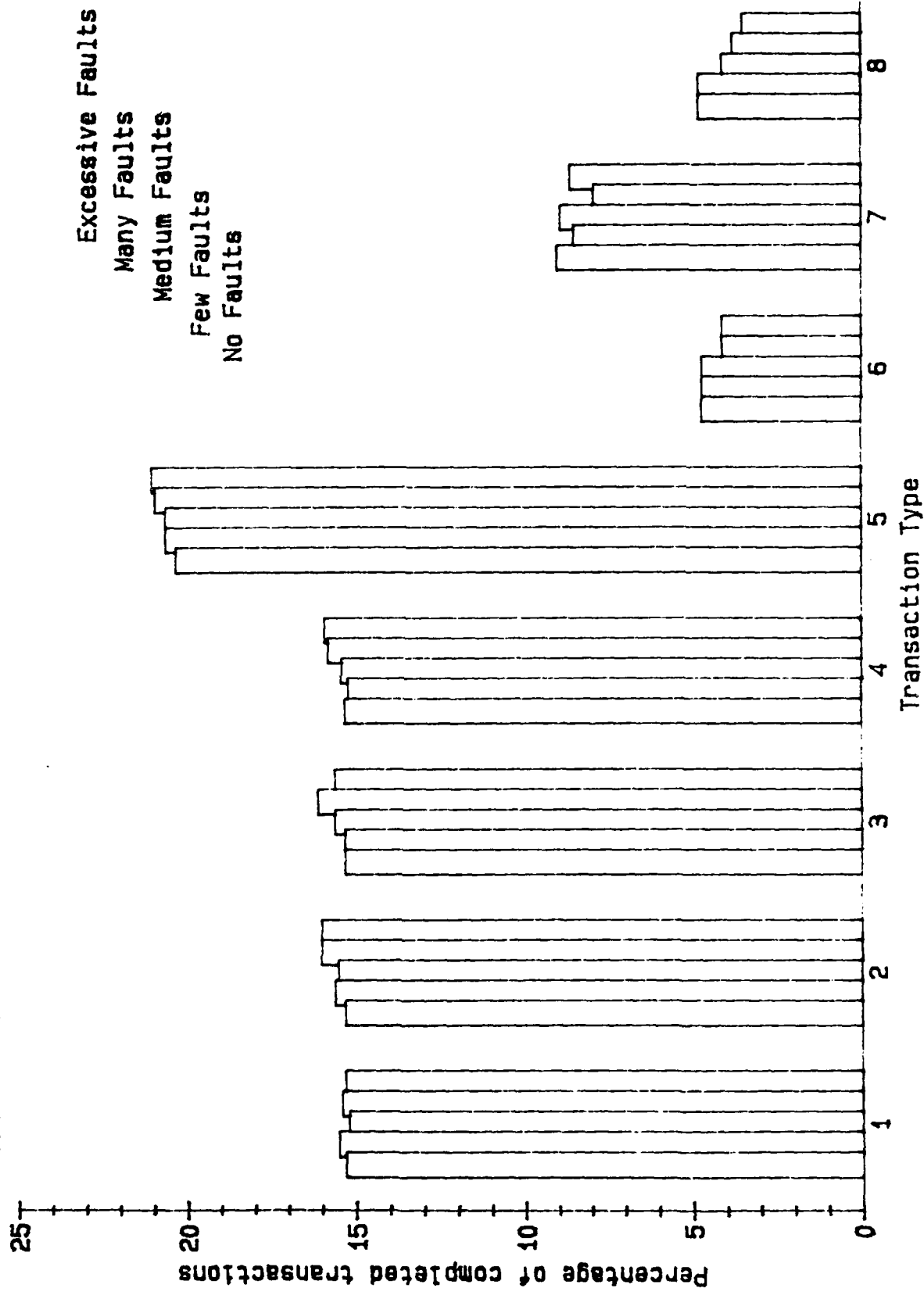
Effect of 1 Phase Presumed Commit on Transaction Mix



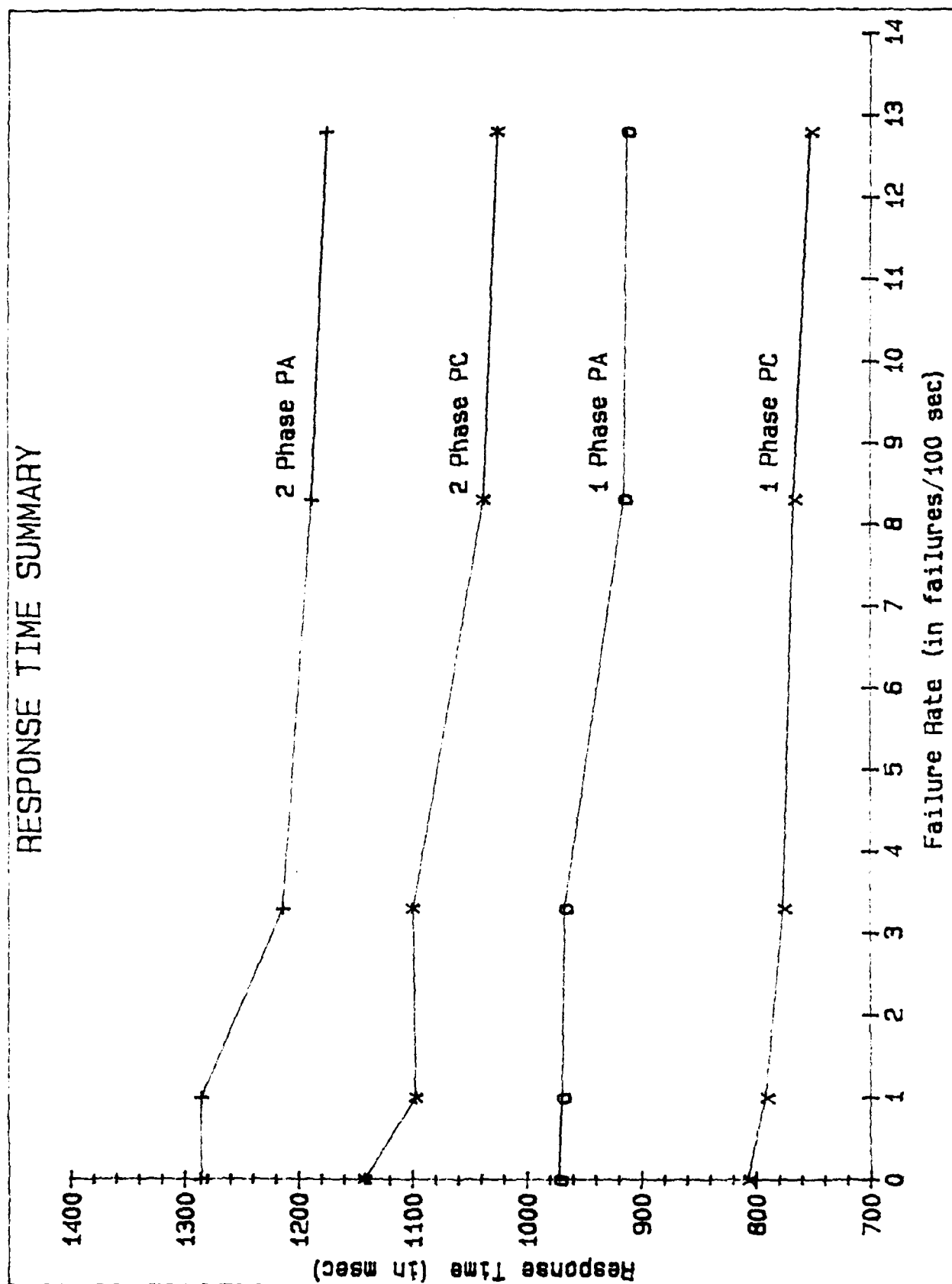
Effect of 2 Phase Presumed Abort on Transaction Mix



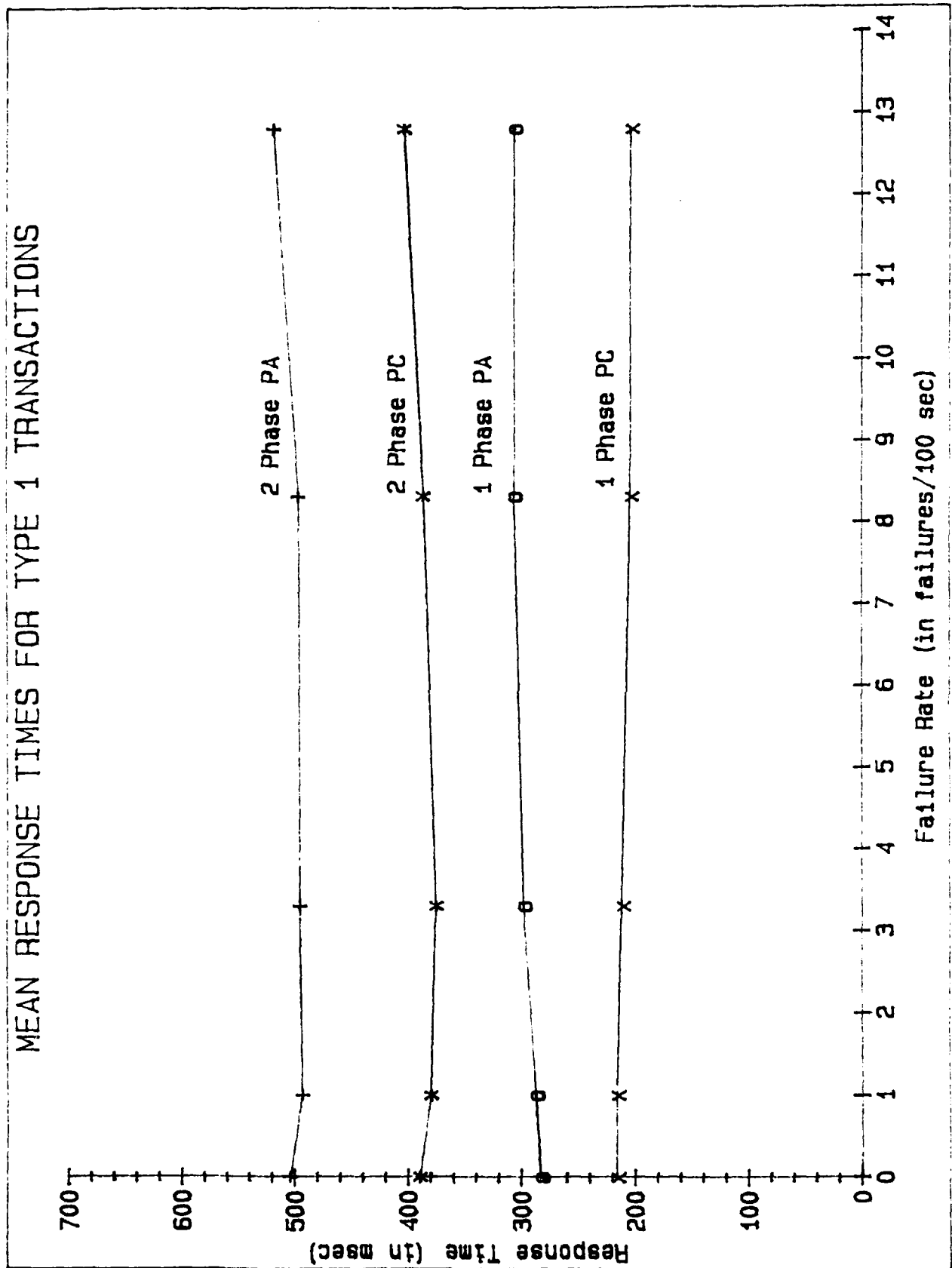
Effect of 2 Phase Presumed Commit on Transaction Mix



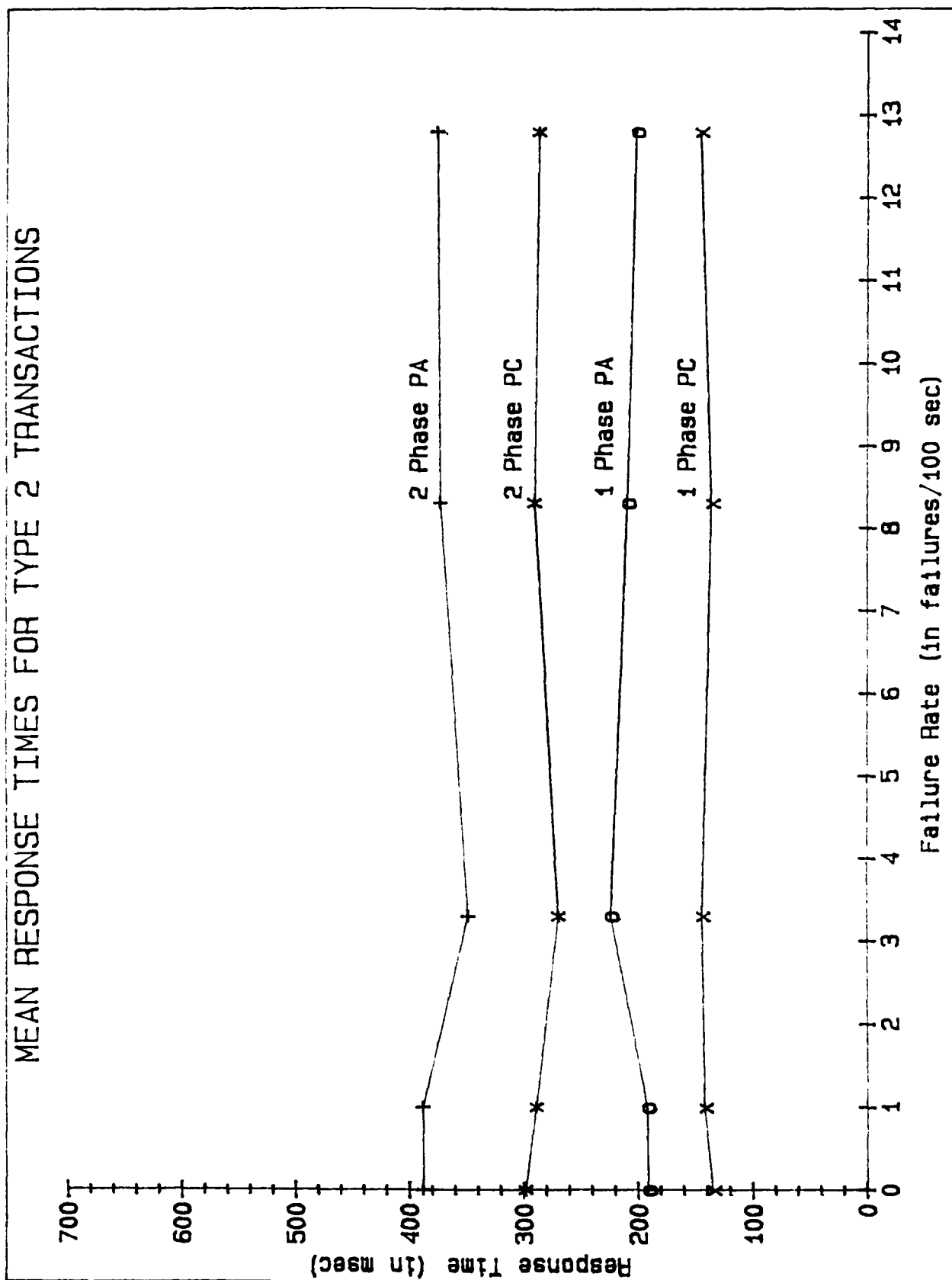
RESPONSE TIME SUMMARY



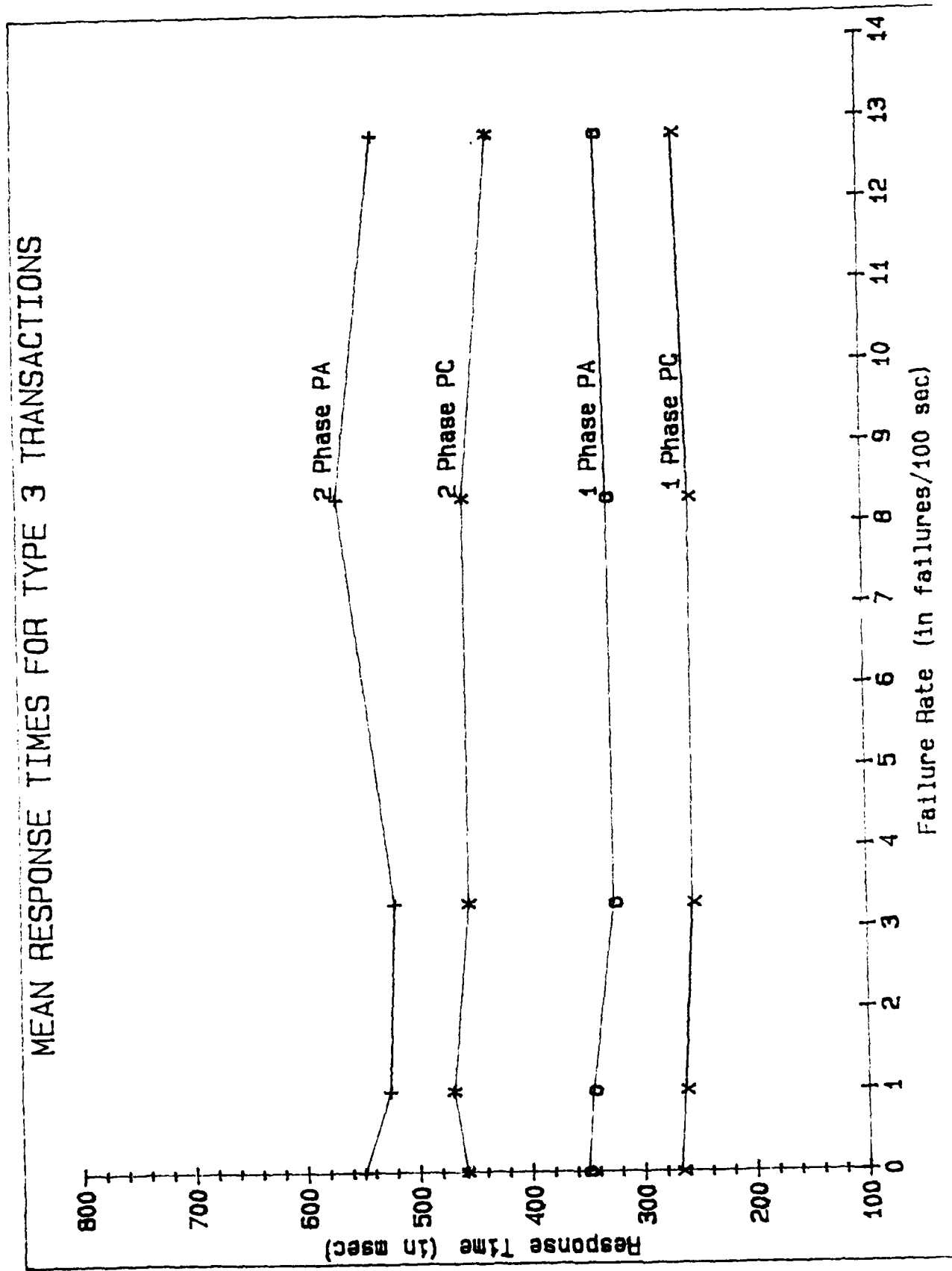
MEAN RESPONSE TIMES FOR TYPE 1 TRANSACTIONS



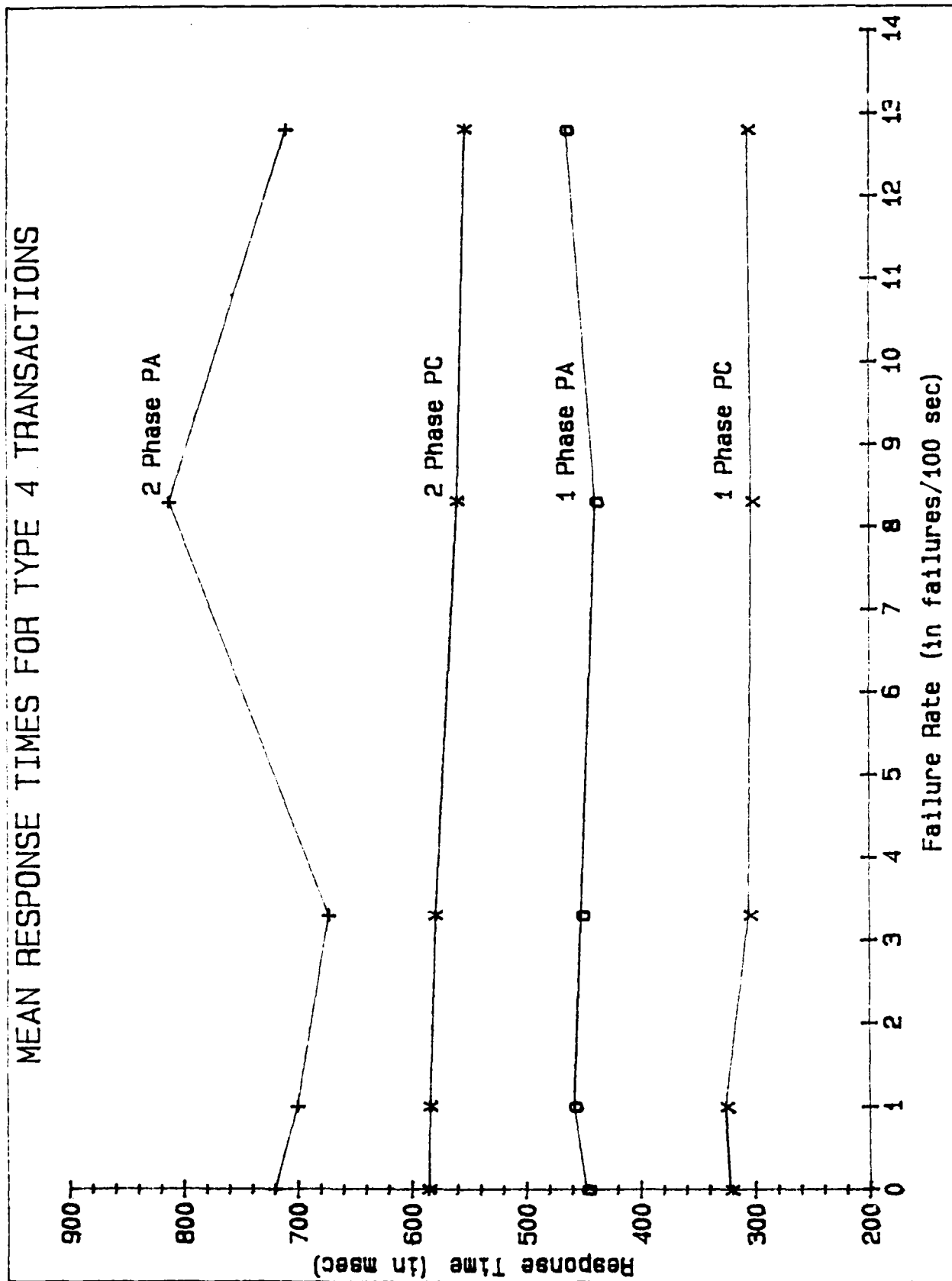
MEAN RESPONSE TIMES FOR TYPE 2 TRANSACTIONS



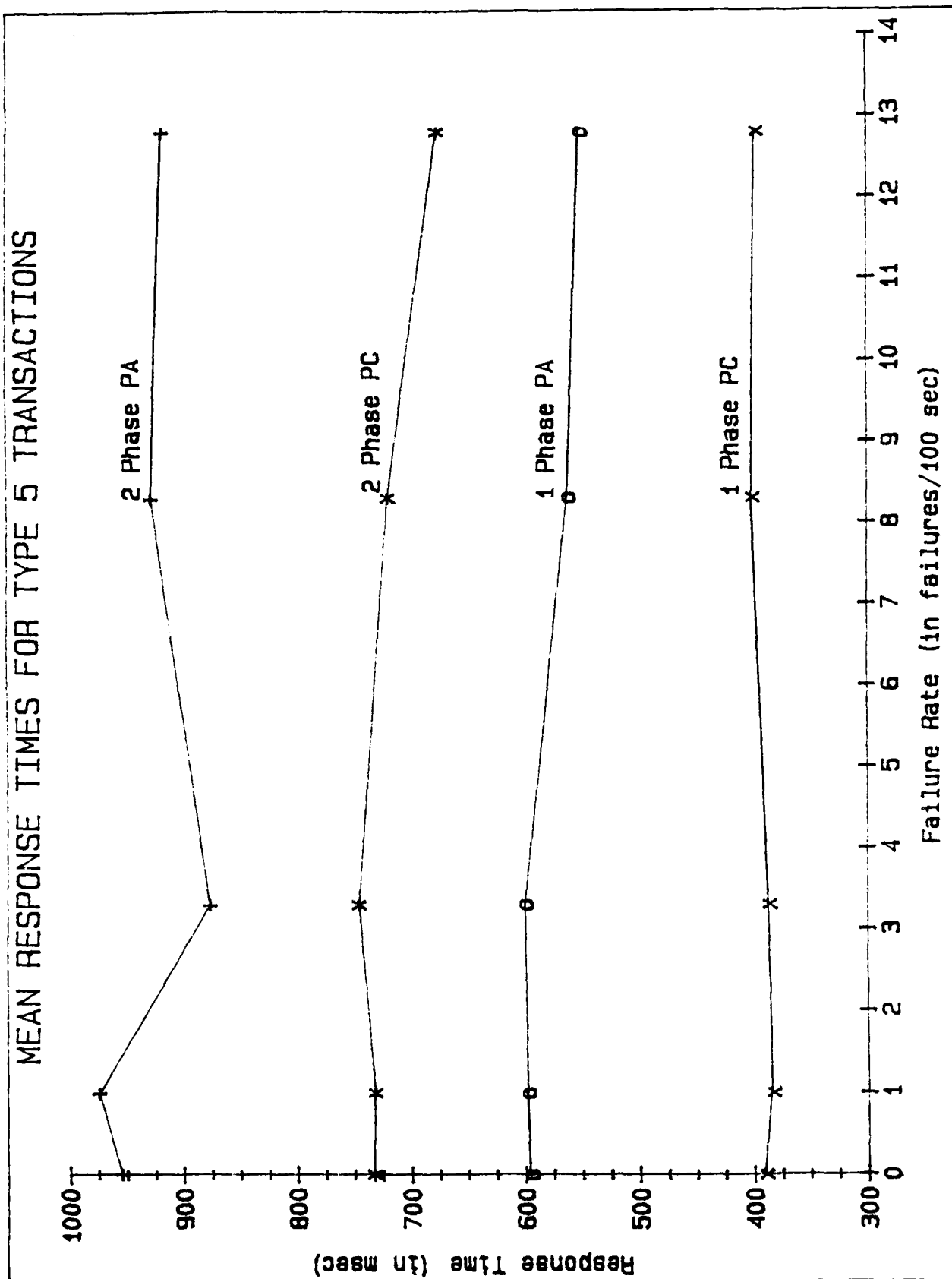
MEAN RESPONSE TIMES FOR TYPE 3 TRANSACTIONS



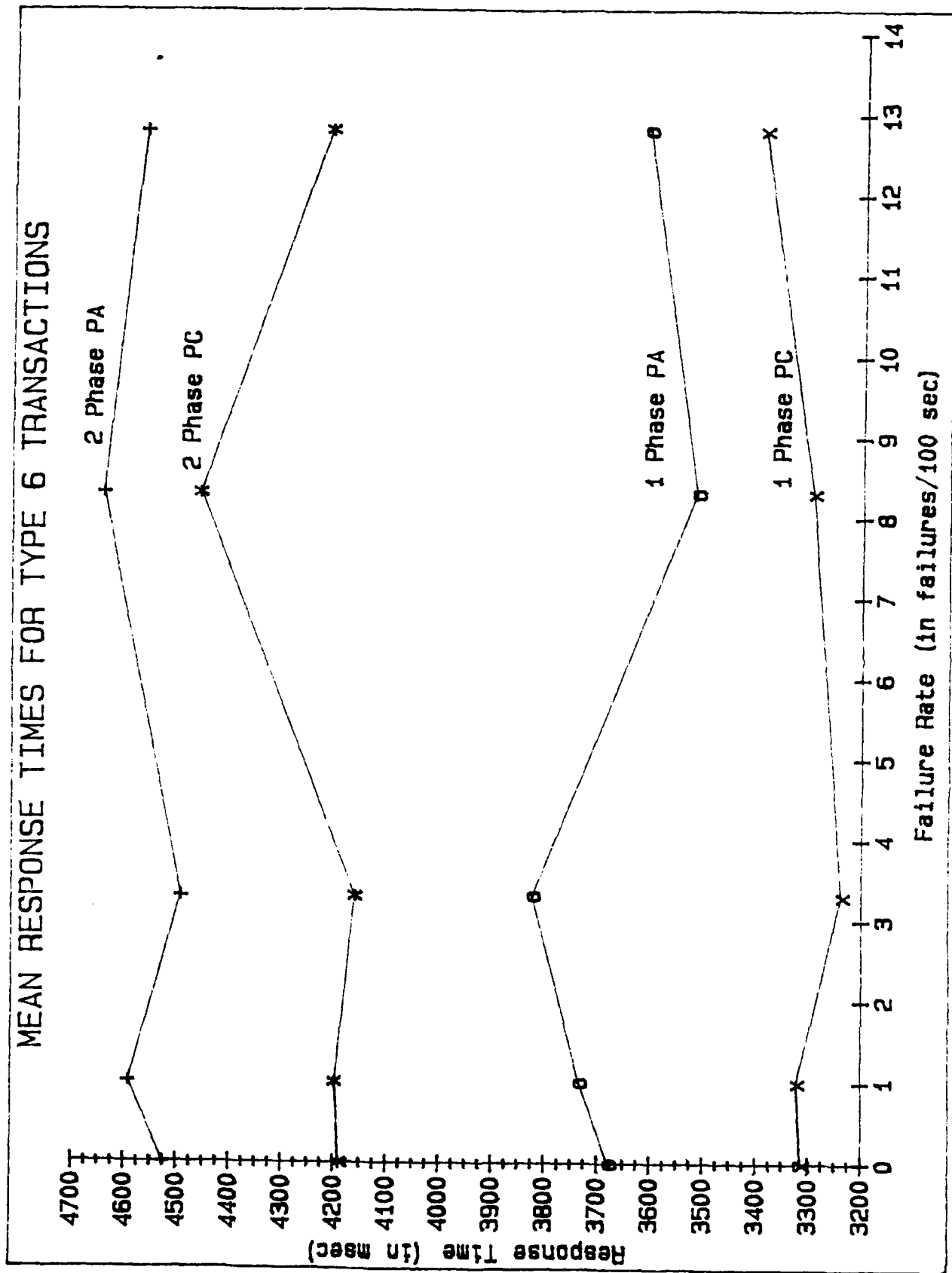
MEAN RESPONSE TIMES FOR TYPE 4 TRANSACTIONS



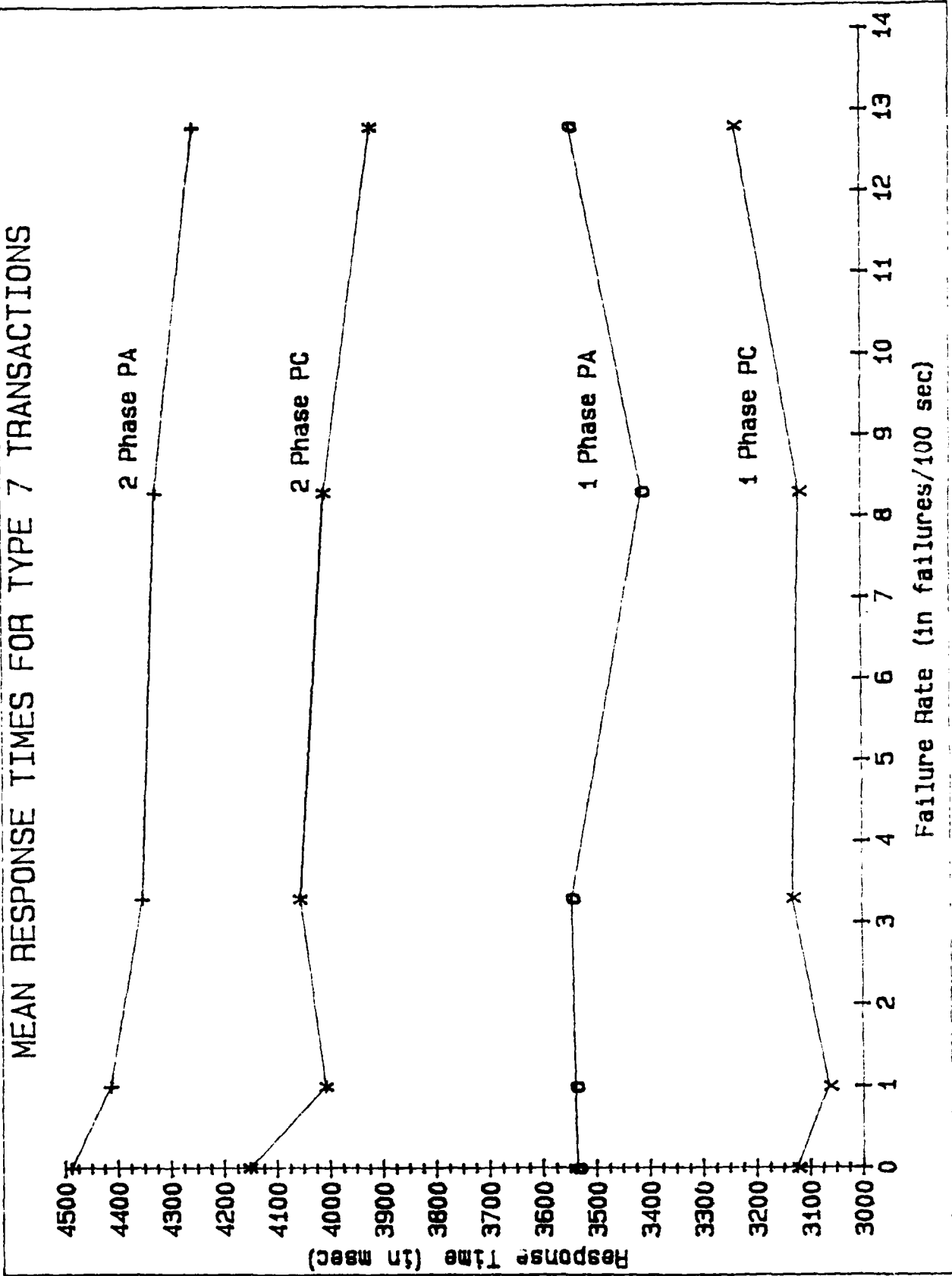
MEAN RESPONSE TIMES FOR TYPE 5 TRANSACTIONS



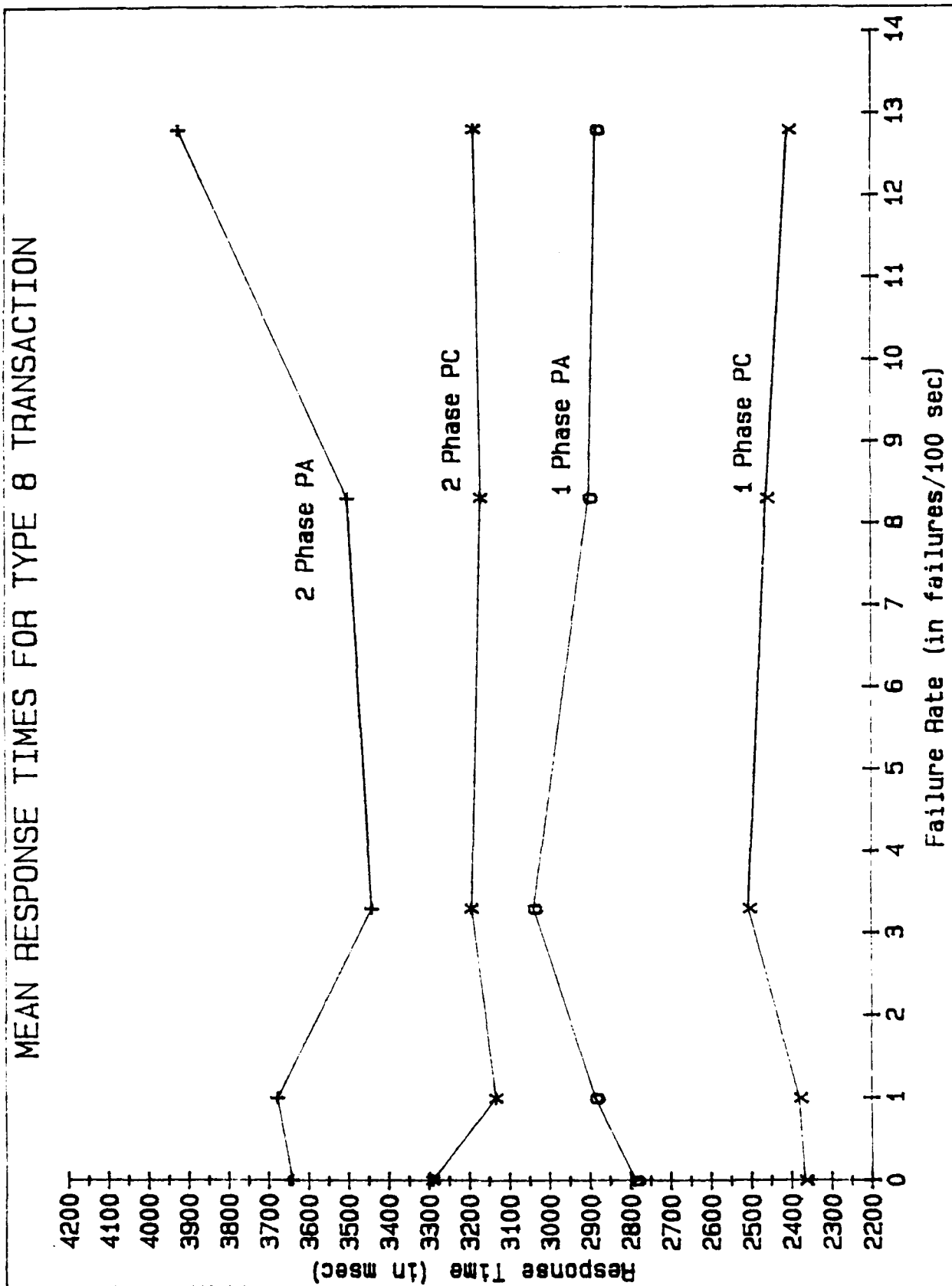
MEAN RESPONSE TIMES FOR TYPE 6 TRANSACTIONS



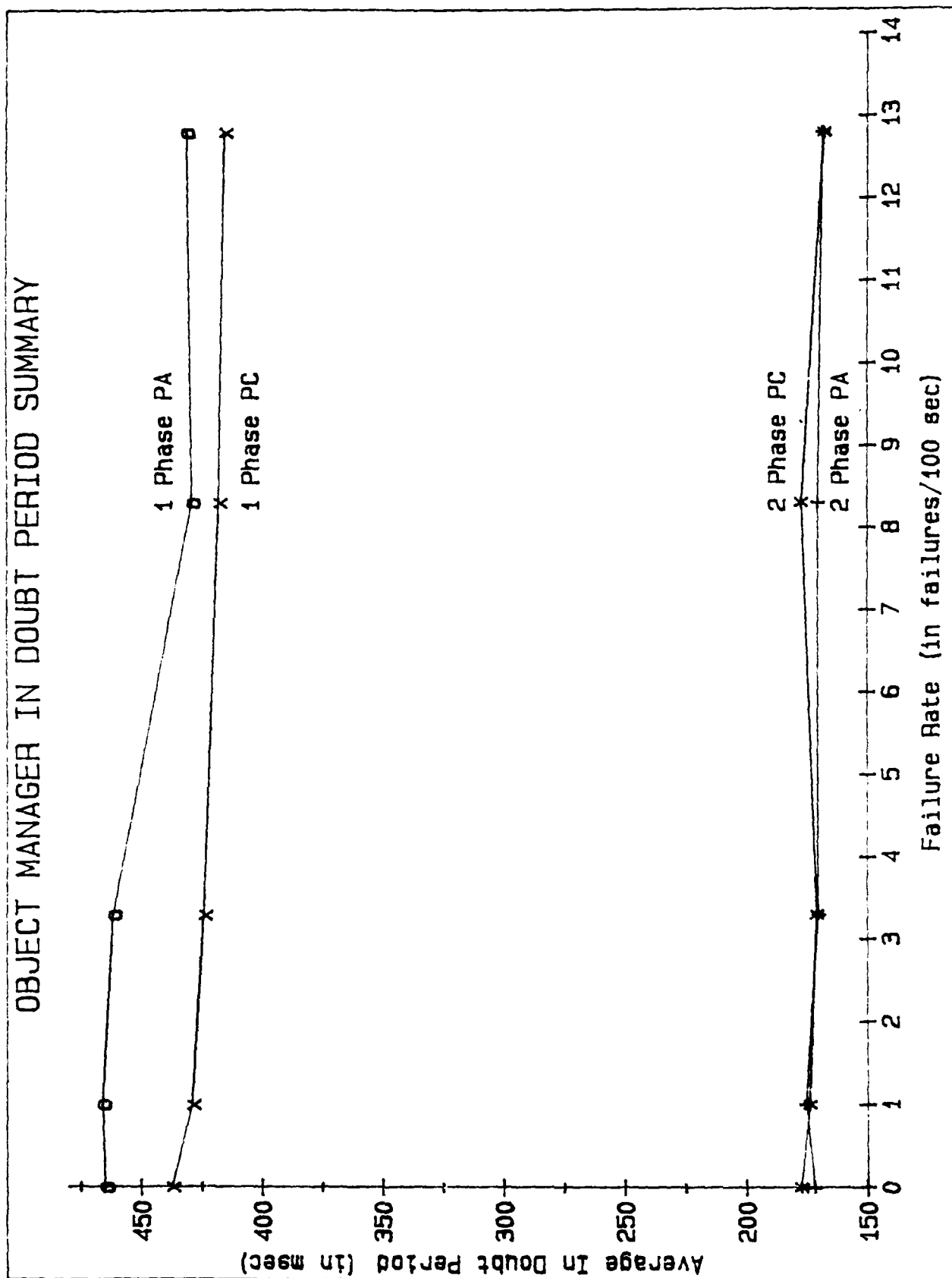
MEAN RESPONSE TIMES FOR TYPE 7 TRANSACTIONS



MEAN RESPONSE TIMES FOR TYPE 8 TRANSACTION



OBJECT MANAGER IN DOUBT PERIOD SUMMARY



BIBLIOGRAPHY

- [ABRA73] Abramson, N., "The ALOHA System," in Computer Communication Networks, N. Abramson and F. F. Kuo (Eds.), 1973.
- [ADRI82] Adrion, W., et al., "Validation, Verification, and Testing of Computer Software," Computing Surveys, 14(2), pp. 159-192, June 1982.
- [AGRA83] Agrawal, R., DeWitt, D. J., "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," Computer Sciences Technical Report 497, University of Wisconsin, Madison, February 1983.
- [AKER65] Akers, S. B., "On the Construction of (d,k) Graphs," IEEE Transactions on Electronic Computers, Vol. EC-14, June 1965.
- [AKER83] Akers, L., W. Bevier, R.M. Cohen, D.I. Good, L.M. Smith, M.K. Smith, "Formal Proof of Recovery Mechanisms," Institute of Computing Science, University of Texas at Austin, Austin, Tx 78712, October 1983.
- [AKER83] Akers, L., L. Smith, "Proof Logs of Recovery Mechanisms," Internal Note 118, Institute for Computing Science, October 1983.
- [AMAR83] Amar, D., "On the Connectivity of Some Telecommunications Networks," IEEE Transactions on Computers, Vol. C-32, No. 5, May 1983, pp. 512-519.
- [ANDE79] Anderson, T., Lee, P.A., Shrivastava, S.K., "System Fault Tolerance," Computing Systems Reliability, Cambridge, England, 1979, pp.152-209.
- [ANDR83] Andrews, G., and Schneider, F., "Concepts and Notations for Concurrent Programming," Computing Surveys, 15(1), pp. 3-43, March 1983.
- [ARDE82] Arden, Bruce W., Hikyu Lee, "A Regular Network for Multicomputer Systems," IEEE Transactions on Computers, Vol. C-31, No. 1, January 1982, pp. 60-69.
- [BALT81] Balter, R., "Selection of a Commitment and Recovery Mechanism for a Distributed Transactional System," Proc. First Symposium on Reliability in Distributed Software and Database Systems, 1981, pp. 21-26.
- [BALT82] Balter, R., P. Bernard, P. Decitre, "Why control of the concurrency level in distributed systems is more fundamental than deadlock management," Proc. of the First Symposium on Principles of Distributed Computing, August 1982, pp.183-193.
- [BERN81] Bernstein, P.A., Nathan Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13, 2, June 1981, pp.185-222.

- [BERN83] Bernstein, P. A., Goodman, N., "The Failure and Recovery Problem for Replicated Database," Proceeding of the Second ACM Symposium on Principles of Distributed Computing, August 1983, pp. 114-122.
- [BIRR82] Birrell, Andrew D., Roy Levin, Roger Needham, Michael Schroeder, "Grapevine: An Exercise in Distributed Computing," Communications of the ACM, April 1982, pp. 260-273.
- [BLAU83] Blaustein, B., et. al, "Maintaining Replicated Databases Even in the Presence of Network Partitions", Computer Corporation of America, 1983.
- [BOEB78] W.E. Boebert, W.R. Franta, E.D. Jensen and R.Y. Kain, "Kernel Primitives of the HXDP Executive," COMPSAC 78, Nov. 1978.
- [BOYD78a] Boyd, D.L., A. Pizzarello, and S.C. Vestal, The Rational Design Methodology - Final Report, RADC Contract No. F30602-77-C0043, Honeywell Inc., June 1978.
- [BOYD78b] Boyd, D.L. and A. Pizzarello, "Introduction to the WELLMADE Design Methodology," IEEE Trans. on Software Engineering., Vol. SE-4, No. 4, July 1978.
- [BROW75] Browne, J. C., K. M. Chandy, R. M. Brown, T. W. Keller, D. F. Towsley, and C. W. Dissly, "Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems," Proceedings of IEEE, Vol. 63, No. 6, 1975.
- [BROW83] Browne, James C., "Structuring Strategies for the Design and Implementation of High Integrity Systems: A Vertically-Structured Approach," Research Report, Information Research Associates, Austin, Texas 78705, 1983.
- [BROW84] Browne, J.C., V. Fernandes, J.E. Dutton, A.R. Tripathi, J. Silverman, P. Wang, "Zeus: An Object-Oriented High Integrity Distributed Operating System," ACM National Conference, October 1984.
- [BURS81] M. Burstin, Y. Forscher, Y. Maimon and I. Rutbard, "SuperPDL - A Software Design Tool," Proceedings, SOFTWARE - A Conference on Software Development Tools, Techniques, and Alternatives, IEEE Comp. Society, July 1983.
- [CAMP79] Campbell, R. and Kolstad, R. "Path Expressions in Pascal," Department of Computer Science, University of Illinois, Urbana, Illinois, July 1979.
- [CHAN81] Chandy, K. M., and C. H. Sauer, Computer Systems Performance Modeling, Prentice-Hall, 1981.
- [COHE75] Cohen, Ellis, David Jefferson, "Protection in the Hydra Operating System," Fifth Symposium on Operating Systems Principles, 1975, pp. 141-160.

- [DAHL66] Dahl, D., and K. Nygaard, "SIMULA - An ALGOL-Based Simulation Language," CACM, Vol. 9, No. 9, 1966.
- [DAVI73] Davies, C.T., "Recovery Semantics for a DB/DC System," ACM National Conference, 1973, pp.136-146.
- [DIJK76] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
- [DINN80] Dinneen, G.P., "A Systems Approach to Command, Control, and Communication" Signal, February 1980.
- [DoD83] United States Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815 A, 1983.
- [DRUF83] L. Druffel, S. Redwine, Jr. and W. Riddle, "The STARS Program: Overview and Rationale," IEEE Computer, Vol. 16, No. 11, November 1983.
- [DUTT83] Dutton, Jim, Fernandes, Vincent, "A Scheme for Reliable Generation of Unique Sequence Numbers in a Local Area Network," Technical Report, Information Research Associates, Austin Tx, September 1983.
- [ESWA76] Eswaran, K.P., et. al., "The Notion of Consistency and Predicate Locks in Database Systems," Communications of the ACM, 19, 11, November 1976, pp.624-633.
- [FERR78] Ferrari, D., Computer Systems Performance Analysis, Prentice-Hall, 1978.
- [FERR83] Fernandes, V. B., "Performance Evaluation of Systems Functionally Specified in ADA," Information Research Associates, Austin, Tx., October 1983.
- [FISH82] Fischer, M., Michael, A., "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," ACM Symposium on Principles of Database Systems, March 1982, pp. 70-75.
- [FRAN83a] E. Frankowski, W. Wood and R. Orgass, "Concurrent System Definition Language, Volume One, Description Language," Tech. Report CTC-R-83-17; 8213, Honeywell Computer Sciences Center, September 1983.
- [FRAN83b] E. Frankowski and W. Wood, "Concurrent System Definition Language, Volume Two, Specification Language," Tech. Report CTC-R-83-18; 8213, Honeywell Computer Sciences Center, December 1983.
- [FREI80] Feiertag, R.J., "A Technique for Proving Specifications are Multilevel Secure," Technical Report CSL-109, Computer Science Lab, SRI International, Menlo Park, Calif., January 1980.
- [GARC83a] Garcia-Molina, H., et. al, "Data-Patch: Integrating Inconsistent Copies of a Database After a Partition," Third Symposium on Reliability in Distributed Software and Database Systems, October 1983, to appear.

- [GARC83b] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed System," ACM Transactions on Database Systems, Vol. 8, No. 2, June 1983, pp. 186-213.
- [GERH80] Gerhart, S.L., et al., "An Overview of AFFIRM: A Specification and Verification System, In Proceedings IFIP 80, pages 343-348, October 1980.
- [GIFF79] Gifford, D. K., "Weighted Voting for Replicated Data," Seventh Symposium on Operating Systems Principles, 1979, pp.150-162.
- [GOOD75] Goodenough, J. B., "Exception Handling: Issues and Proposed Notation," Communications of the ACM 18, 12, December, 1975, pp.683-696.
- [GOOD78] Good, D. I., Cohen, R. M., Hoch, C. G., Hunter, L. W., Hare, D. F., "Report on the Language Gypsy, Version 2.0," Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September 1978.
- [GOOD82a] Good, D. I., "The Proof of a Distributed System in Gypsy," in Formal Specification - Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar - M. J. Elphick)Ed.. September 1982. Also Technical Report #30, Insititute for Computing Science, The University of Texas at Austin.
- [GOOD82b] Good, D. I., Siebert, A E., Smith, L. M., "OSIS Message Flow Modulator - Status Report," Internal Note #36A, Institute for Computing Science, The University of Texas at Austin.
- [GRAY79] Gray, J.N., "Notes on Database Operating Systems," in Operating Systems: An Advanced Course, ed. R. Bayer, R.M. Graham, and G. Seegmuller, Springer-Verlag, 1979, pp.393-481.
- [GRIF79] Griffith, David A., "Flexible Intraconnect, A New Approach for Configuring Command and Control Systems," Proc. of the LACN Symposium, May 1979, pp. 263-277.
- [GRNA80] Grnarov, A., Kleinrock, L. Gerla, M., "A New Algorithm for Symbolic Reliability Analysis of Computer Communication Networks," Proc. Pacific Telecommunications Conference, January 1980, pp. 1-9.
- [GRNA81] Grnarov, A., Gerla, M., "Multi-terminal Reliability Analysis of Distributed Processing Systems," Proc. 1981 International Conf. on Parallel Processing, August 1981, pp. 79-86.
- [HALP83] J. Halpern, Z. Manna and B. Moszkowski, "A Hardware Semantics Based on Temporal Intervals," Tech. Report STAN-CS-83-963, Stanford University, March 1983.
- [HOAR78] C.A.R. Hoare, "Communicating Sequential Processes", Com. of the ACM, Vol. 21, No. 8, August 1978.

- [HONE83] Honeywell, Inc., "Distributed C(2) System Recovery Mechanisms," Interim Scientific Report, Corporate Technology Center, Honeywell, Inc., September 1983.
- [HORN74] Horning, J.J., et. al., "A Program Structure for Error Detection and Recovery," Computer Science, Springer Verlag Lecture Notes in "Operating Systems Techniques", Volume 16, .
- [IBM71] IBM, General Purpose Simulation System, IBM Corporation, 1971.
- [IBM72] IBM, SIMPL/1: Program Reference Manual, IBM Corporation, 1972.
- [IMAS83] Imase, Makato, Masaki Itoh, "A Design for Directed Graphs with Minimum Diameter," IEEE Transactions on Computers, August 1983, pp. 782-784.
- [IRA83] Information Research Associates, Performance Analyst's Workbench System - Introduction and Technical Summary, Information Research Associates, Austin, Tx., July 1983.
- [IRA84] Information Research Associates, Performance Analyst's Workbench System (PAWS) - User's Manual, Information Research Associates, Austin, Tx., 1984.
- [JANS77] T.M.V. Janssen, and P. Van Emde Boas, "On the Proper Treatment of Referencing, Dereferencing and Assignment," Proceedings, Fourth Colloquium on Automata, Languages and Programming, A. Salomaa and M. Steinby eds., in Lecture Notes in Computer Science, Vol. 52, 1977.
- [KEMM80] Kemmerer, R., "FDM - A Specification and Verification Methodology," In Proc. 3rd Seminar on the Department of Defense Computer Security Initiative Program, Nat. Bur. Stand., November 1980.
- [KIM79] Kim, K.H., "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems," First International Conference on Distributed Processing, 1979, pp.284-295.
- [KIVI79] Kiviat, P. J., and R. Villanueva, The SIMSCRIPT II Programming Language Reference Manual, Prentice-Hall, 1969.
- [KNUT74] Knuth, D. E. "Structured Programming with GOTO Statements," Computing Surveys, Vol. 6, No. 4, December 1974.
- [KOBA78] Kobayashi, H., Modeling and Analysis: An Introduction to Performance Evaluation Methodology, Addison Wesley, 1978.
- [KCHL81] Kohler, W. H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.
- [KOLS80] Kolstad R. and Campbell R., "Path Pascal User Manual," Department of Computer Science, University of Illinois, Urbana, Illinois, January 1980.

- [KUNG83] Kung, H.T., John T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, pp.231-226.
- [LAMP76] Lampson, B.W., Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System," Technical Report, Xerox, Parc, April 1976, pp..
- [LAMP81a] Lampson, Butler W., "Atomic Transactions," in Lecture Notes in Computer Science Vol. 105, ed. B.W. Lampson, M. Paul, and H.J. Siebert, Springer-Verlag, 1981, pp.246-265.
- [LAMP81b] Lampson, Butler W., "Remote Procedure Calls," in Lecture Notes in Computer Science Vol. 105, ed. B.W. Lampson, M. Paul, and H.J. Siebert, Springer-Verlag, 1981, pp.365-370.
- [LAMP82] Lamport, L., Shostak, Pease, M., "The Byzantine Generals Problems," ACM Transactions on Programming Languages and Systems, July 1982, Vol. 4, No. 3, pp. 382-401.
- [LEVI79] Levitt, K. N., Robinson, L., Silverberg, B. A., "The HDM Handbook," Computer Science Lab, SRI, Menlo Park, California, 1979.
- [LISK75] B. Liskov and S. Zilles, "Specification Techniques for Data Abstractions," IEEE Trans. on Software Engineering," Vol. SE-1, No. 1, March 1975.
- [LISK79a] B. Liskov, "Modular Program Construction Using Abstractions," in Abstract Software Specifications, D. Bjorner, ed., Springer-Verlag, 1979.
- [LISK79b] B. Liskov and V. Berzins, "An Appraisal of Program Specifications," in Research Directions in Software Technology, P. Wegner, ed., MIT Press, 1979.
- [LISK79c] B. Liskov and A. Snyder, "Exception Handling in CLU," IEEE Trans on Software Engineering, Vol. SE-5, No. 6, November 1979.
- [LISK81] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," CSG Memo 210-1, MIT LCS, November 1981.
- [LISK82a] Liskov, B., "On Linguistic Support for Distributed Programs," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp.203-210.
- [LISK82b] Liskov, B., Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Ninth Annual Symposium on Principles of Programming Languages, January 1982, pp.7-19.
- [LORI77] Lorie, R.A., "Physical Integrity in a Large Segmented Database," ACM Transactions on Database Systems, March 1977, pp. 91-104.
- [LUCA71] Lucas, H. C., Jr., "Performance Evaluation and Monitoring," Computing Surveys, Vol. 3, No. 3, 1971.

- [MARI80] Mariani, M.P., R. Turn, B.R. Johnson, "Architecture Study for Tactical C3 Construct," RADC-TR-80-304, September 1980.
- [MARC82] Marciniak, John J., "Technology Needed for C3I Evolution," *Astronautics/Aeronautics*, July/August 1982.
- [MASC80] The Official Handbook of Mascot, Final Draft, Mascot Suppliers Association, February 1980.
- [MCHU83] McHugh, J., "Towards the Generation of Efficient Code from Verified Programs," PhD Thesis, The University of Texas at Austin, December 1983.
- [McMI82] McMillan, David R., "Introducing C3I," *Astronautics/Aeronautics*, July/August 1982.
- [MELL77] Melliar-Smith, P.M., Randell, B., "Software Reliability: The Role of Programed Exception Handling," ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, March 1977, pp.95-100.
- [MEMM82] Memmi, Gerard, Yves Raillard, "Some New Results about (d,k) Graph Problem," *IEEE Transactions on Computers*, August 1982, pp. 784-791.
- [METC76] Metcalf, R., and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM*, July 1976
- [MILL65] Miller, R.E., Switching Theory, John Wiley & Sons, Inc., 1965.
- [MINO82] Minoura, T., Wiederhold, G., "Resilient Extended True-Copy Token Scheme for a Distributed Database System," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3, May 1982, pp..
- [MOHA83] Mohan, C., Lindsay, B., "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, August 1983, pp. 76-88.
- [MOSS81] Moss, J. Elliot B., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, MA 02139, April 1981, pp..
- [MOSZ83] B. Moszkowski, "A Temporal Logic for Multi-Level Reasoning about Hardware," Computer Hardware Description Languages and their Application, T. Uehara and M. Barbacci, eds, North-Holland Pub. Co., 1983.
- [MUSS80] Musser, D. R., "Abstract Data Type Specification in the AFFIRM System," *IEEE Transactions on Software Engineering* SE-6(1), January, 1980.
- [MYER78] G.J. Myers, Composite/Structured Design, Van Nostrand Reinhold Company, 1978.

- [NELS81] Nelson, B.J., "Remote Procedure Call," Technical Report-Department of Computer Science, Carnegie-Mellon University, May 1981, pp..
- [OPPE81] Oppen, Derek C., Yogen K. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," Xerox Office Products Division, Palo Alto, California 94304, October 1981.
- [PALM83] Palmer, A.S., V. F. Fernandes, J. Hwang, "Representing the Functional Specification and Performance Specifications of the ZEUS Operating System in the Evolutionary Database Management System," Information Research Associates, Austin, Tx., October 1983.
- [PARK80] Parker, Scott, D., "A Distributed File System Architecture Supporting High Availability," Proc. of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, 1982, pp. 161-184.
- [PAXT79] Paxton, W.H., "A Client-Based Transaction System to Maintain Data Integrity," Seventh symposium on Operating System Principles, December, 1979, pp.18-23.
- [REDE80] Redell, David D., et al., "Pilot: An Operating System for a Personal Computer," Communications of the ACM, February 1980, pp. 81-91.
- [REED78] Reed, D.P., "Naming and Synchronization in Decentralized Computer Systems," Technical Report MIT/LCS/TR205, September 1978, pp..
- [REMP83] T. K. Remple, "A Text Oriented Tool Set for the Concurrent System Definition Language", Proceedings of the Symposium on Application and Assessment of Automated Tools for Software Development, IEEE Computer Society, November 1983.
- [RIES82] Ries, Daniel, R., Gordon C. Smith, "Nested Transactions in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No.3, May, 1982, pp.167-172.
- [ROBI75] Robinson, L., Levitt, K. N., Neumann, P. G., Saxena, A. R., "On Attaining Reliable Software for a Secure Operating System," In International Conference on Reliable Software, pp.267-284, IEEE, April 1975.
- [ROBI77] Robinson, L. and O. Roubine, "Special - A Specification and Assertion Language," Tech. Report CSL-46, Stanford Research Institute, January 1977.
- [ROCH83] Rochfeld, A. "PROTEE 6000: An Automated Development Support System for MERISE Methodology," Proceedings of the Symposium on Application and Assessment of Automated Tools for Software Development, IEEE Computer Society, November 1983.

- [ROSE78] Rosenkrantz, D.J., R.E. Stearns, P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," ACM Trans. on Database Systems, June 1978, pp.178-198.
- [RUSS80] Russell, D.L., "State Restoration in Systems of Communicating Processes," IEEE Transactions on Software Engineering, Vol SE-6, No.2, March 1980, pp.183-194.
- [SALT81] Saltzer, J.H., Reed, D.P., Clark, D.D., "End-To-End Arguments in System Design," Second International Conference on Distributed Computing Systems, 1981, pp.509-512.
- [SCHA83] Schantz, R., et al., "Cronus: A Distributed Operating System - Interim Technical Report No. 2, "RADC Report No. 5261, February 1983.
- [SEVE76] Severance, D.G., Lohman, G.M., "Differential Files: Their application to the Maintenance of Large Databases," ACM Transactions on Database Systems, Vol. 1, No. 3, September, 1976, pp.256-267.
- [SILB80] Silberschatz, A., Kedem, Z., "Consistency in Hierarchical Database Systems," J.ACM 27, 1, January 1980, pp.72-80.
- [SHRI81] Shrivastava, S.K., "Structuring Distributed Systems for Recoverability and Crash Resistance," IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, July 1981, pp.436-447.
- [SHRI82a] Shrivastava, S.K., Panzieri, F., "The Design of a Reliable Remote Procedure Call Mechanism," July 1982, pp.692-697.
- [SHRI82b] Shrivastava, S.K., "A Dependency, Commitment and Recovery Model for Atomic Actions," Second Symposium on Reliability in Distributed Software and Database Systems, July 1982, pp.112-119.
- [SKEE82] Skeen, D., "A Quorum-Based Commit Protocol," Proc. 6th Berkeley Workshop on Distributed Databases and Computer Networks, Berkeley, Calif., 1982, pp.69-80.
- [SMIT79a] Smith, C. U., and J. C. Browne, "Modeling Software Systems for Performance Predictions," Proceedings of the Conference on Simulation Measurement and Modeling of Computer Systems, Boulder, Colorado, August 1979.
- [SMIT79b] Smith, C. U., and J. C. Browne, "Modeling Software Systems for performance Predictions," Proceedings of the Computer Measurement Group X, Dallas, December 1979.
- [SMIT80a] Smith, C. U., and J. C. Browne, "Aspects of Software Design Analysis; Concurrency and Blocking," Proceedings of Performance 80, Toronto, May 1980.
- [SMIT80b] Smith, C. U., "The Prediction and Evaluation of the Performance of Software from Extended Design Specifications," Ph.D. Dissertation, The University of Texas, Austin, Tx., August 1980.

- [SPEC82] Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM*, Vol. 25, No. 4, April 1982, pp.246-259.
- [STON81] Stonebraker, Michael, "Operating System Support for Database Systems," *Communications of the ACM*, July 1981, pp.412-418.
- [SVOB81] Svobodova, L., "A Reliable Object-Oriented Data Repository for a Distributed Computer," *Eighth Operating Systems Principles Conference*, 1981, pp. 47-58.
- [THOM79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, June 1979, pp.180-209.
- [THOM80] Thompson, John R., "TAC C3 Distributed Operating System Study," *RADC-TR-79- 360*, January 1980.
- [THOM81] Thompson, D. H., Sunshine, C. A., Erikson, R. W., Gerhart, S. L., Schwabe, D., "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models," *Technical Report ISI/RR-81-88*, USC/Information Sciences Institute, March 1981.
- [TRIP83a] Tripathi, A. R., P. S. Wang, "An Object-Oriented Design Model for Reliable Distributed Systems," *Third Symposium on Reliability in Distributed Software and Database Systems*, October 1983.
- [TRIP83b] Tripathi, A. R., W.T. Wood, "A Formal Model to Prove Global Properties in Distributed Systems," *Technical Report*, Corporate Computer Sciences Center, Honeywell Inc., Bloomington, MN 55420, September 1983.
- [WALK83] Walker, B., et al., "The LOCUS Distributed Operating System," *Ninth ACM Symposium on Operating System Principles*, October 1983, pp. 49-70.
- [WANG83] Wang, P.S., Tripathi, A.R. "An Algorithm for Network Reliability Computations," *Research Report*, Honeywell Computer Sciences Center, Bloomington, MN 55420.
- [WEEL80] J.A. Weeldreyer, "The Entity-Category-Relationship Model of Data," *Honeywell Computer Sciences Center, Technical Report No. HR-80-250*: 17-38, 1980.
- [WULF76] W. Wulf, R. London and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 4, December 1976.
- [WULF82] Wulf, William A., Roy Levin, Samuel P. Harbison, HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, 1981.

INDEX TABLE

access control:4-65 5-3 5-5 5-7 9-4
 accessed object list:5-36 5-42 5-44
 affirm:9-3
 allocate node:7-15 7-25
 alter:7-15 9-5 9-6 9-7 9-8 9-11 9-13
 aol:5-36 5-39 5-44 5-45 5-47
 argus:5-2 5-3
 atomic action:4-24 4-54 4-68 4-76 5-6 5-32 5-49 10-8
 authentication:3-5 3-6 5-3 5-10 5-21 5-24 10-9
 authentication manager:5-21 5-24
 availability:2-1 2-4 2-5 2-11 2-15 2-17 2-18 3-6 3-7 3-8 4-4 4-5 4-34 4-36
 4-38 4-39 4-46 4-50 4-52 4-65 5-28 5-51 7-7 8-1 8-2 8-3 8-4 8-5 8-7 8-8 8-9
 8-10 8-13 8-15 8-16 8-21 8-22 8-23 10-4 10-5 10-10
 availability improvement:8-16

 backward error recovery:4-64
 backward log:4-23 9-10
 begin transaction:4-48 4-74 5-32 5-38 5-42 5-51 5-52 9-16 9-18 9-19
 branch:7-15 7-16

 c2 systems:2-1 2-7 3-2 7-36 7-37
 careful replacement:4-19 4-21 9-40
 central server model:7-30
 centralized control:4-26 4-38 4-39 4-42 4-50
 certifier:4-7
 change node:7-16 7-25 7-27 7-32
 checkpoint:4-17 4-18 4-68 4-69 4-70 5-22 5-23 5-34 5-38 5-41 5-48 7-9 9-39
 10-7
 checkpointing:4-17 4-69 4-70 5-2 5-3 5-22 5-23 5-34 5-41 5-48 10-7
 circulating token:4-11
 cleanup:4-13 4-19 4-20 8-13 8-15
 cobegin:9-16 9-17 9-18
 cohort:4-25 4-26 4-27 4-28 4-29 4-30 4-31 4-33 4-34 4-33 4-34 4-35 4-36
 command and control:2-1 2-5 2-6 2-8 2-12 2-17 3-1 3-2 3-3 3-4 3-6 3-7 3-8
 3-9 3-10 3-12 6-1 5-1 5-3 7-4 7-10 7-35 7-37 8-2 9-2 9-21 10-5 10-12
 commit pending:4-26 4-29 4-33 4-35 5-45 5-46 5-52 9-42 9-55
 compute node:7-16 7-21 7-33
 conceptual design:10-7 10-8
 concurrency:2-7 4-1 4-2 4-4 4-6 4-7 4-9 4-12 4-14 4-15 4-38 4-39 4-42 4-44
 4-51 4-52 4-53 5-3 5-6 5-23 5-32 5-37 5-53 7-9 7-20 7-24 9-3 9-12 9-17
 concurrency control:2-7 4-1 4-2 4-4 4-6 4-7 4-12 4-15 4-39 4-42 4-44 4-51
 5-3 5-37 5-53 7-9 7-20 7-24 9-12
 concurrent system definition language:10-6
 connectivity:4-76 6-1 6-2
 consistency:2-6 2-7 2-9 2-11 2-12 2-13 2-18 3-6 3-7 3-8 3-10 3-12 4-1 4-2
 4-4 4-7 4-8 4-11 4-12 4-15 4-16 4-25 4-36 4-37 4-38 4-39 4-41 4-42 4-46
 4-47 4-52 4-53 4-55 4-59 4-60 4-61 4-67 5-7 5-9 5-22 5-26 5-37 5-38 5-42
 5-51 7-28 8-1 8-3 10-3 10-7

coordinator:4-25 4-26 4-27 4-28 4-29 4-30 4-31 4-33 4-34 4-35 4-36 4-42
4-44 4-45 4-50 4-51 5-39 7-9 7-26 7-27 7-28 9-42
crasher:9-48 9-49 9-50 9-51
create_process:5-30
cronus:5-3
csdl:10-6 10-9 10-11
cube:8-7
current operation table:5-39 5-49

data patch:4-60
datagram:4-7 4-76 5-15 10-8
deadlock detection:4-11 4-14 4-15 7-9
deadlock prevention:4-10 4-14 4-15 5-37 9-40
degree:6-1 6-2 6-3 6-4 6-5 6-6
delete_process:5-30
dependency expression:8-7 8-8
descendent transaction list:5-36 5-39
detailed design:2-19 5-26 5-53 10-1 10-3 10-10 10-11
diameter:6-1 6-2 6-3 6-4 6-5 6-6
differential file:4-23 4-72 4-73 5-37 5-48 5-49
domino effect:4-1 4-18 5-3 5-31 5-34
dtl:5-36 5-44 5-45 5-46 5-47 5-48

edge connectivity:6-1
encryption:4-7
end transaction:4-48 4-74 4-75 5-32 5-33 5-36 5-39 5-42 5-44 5-46 5-52
9-16 9-17 9-19
error detection:2-2 4-16 4-17 10-10
error recovery:4-1 4-15 4-16 4-17 4-55 4-61 4-64 4-69 4-70 4-77 5-22 5-23
ethernet:5-12 5-13 7-21 8-10 8-22
exception handling:4-77
extended uid:5-6 5-11 5-12 5-21

fault:2-1 2-2 2-5 2-6 2-9 2-11 2-12 2-13 2-14 2-15 2-16 2-17 3-4 5-7 7-5
7-26 7-29 7-30 7-32 7-33 7-34 8-3 8-4 8-22 8-23 9-38 9-40 9-41 9-42 9-43
9-45 9-48 9-49 9-50 9-51 10-3 10-6 10-7
fault diagnosis:2-2
fault injector:9-48 9-49 9-50 9-51
finite state machine:9-23 9-25 9-31
force write:4-28 4-29 4-33 5-40 5-47 5-48
fork:7-16 7-25 7-28
forward error recovery:4-17 4-55 4-61 4-64 4-77
forward log:4-23 4-29 4-67 4-68 9-9
fsm:9-29 9-30 9-32
functional architecture:10-1 10-3 10-4 10-8 10-9 10-10 10-12
functional simulation:5-1 9-38

generic object manager:5-34 5-51 9-38 9-39 9-47 9-48
get_response:5-36 5-39
grapevine:5-3

gve:9-10
gypsy:5-1 9-2 9-3 9-4 9-5 9-9 9-10 9-11 9-12 9-14 9-17 9-18 9-21 10-6 10-9
10-10 10-11

hydra:4-5 5-5 5-6 5-22

idempotent:4-67 4-69 4-71 5-23
immutable:4-6
in-doubt period:4-27 4-30 4-31 4-32
intention list:4-67 4-68
interactive consistency:4-2 4-4
interval:4-20 4-46 4-59 5-14 5-32 8-3 8-13 9-23 9-24 9-25 9-27 9-32 9-42
9-45 9-47 9-50 9-55 10-10
interval logic:9-27 9-32 10-10
ipg:7-11 7-13 7-14 7-15 7-17 7-18 7-21 7-23 7-25 7-26 7-27 7-28 7-30 7-33

join:7-16 7-25 7-29

kernel:2-8 5-4 5-8 5-10 5-11 5-16 5-18 5-19 5-20 5-21 5-30 5-36 5-39 7-4
9-38 9-40 9-45 9-47 9-49 9-51 10-8 10-9 10-10 10-11

lemma:9-8 9-10
local area network:3-10 5-12 7-5 4-2 4-9 4-10 4-11 4-13 4-14 4-30 4-33
4-40 5-33 5-37 5-41 5-42 7-9 7-20 7-24 7-28 9-4 9-12 9-13 9-40
lock:4-9 4-11 4-14 4-26 4-40 4-44 4-47
lock mode:4-9
locking:4-2 4-9 4-10 4-11 4-13 4-14 4-30 4-33 4-40
log transformation:4-60 4-62 4-63
logs:2-19 4-22 4-23 4-24 4-29 4-30 4-55 4-60 4-62 4-63 4-67 4-68 5-1 5-3
9-2 9-9 9-10 10-4 10-11

majority consensus:4-42 4-45 4-46 4-50 4-51 4-77 5-32 8-16 8-21
make call:5-20 5-36 5-39
masking redundancy:4-16
merge log:4-63
message type manager:5-28
mission time:8-2 8-5 8-8 8-9 8-10
module:7-11 7-12 9-39 9-48 9-49 10-3 10-4 10-10
moore bound:6-2 6-5
mtm:3-12 5-28
mttf:2-5 2-17 2-18 7-32 8-1 8-2 8-9 8-10 8-13 10-4
mttr:8-13
multiple objects:9-9
multitree structured:6-2

nested transaction:4-70 4-71 4-72 4-73 4-75 4-77 5-23 5-33 5-39 5-41 5-42
5-43 5-44 5-46 5-48 5-51 5-52 9-40

netrat:2-17 8-2 8-4 8-5 8-7 8-8 8-9 8-13 8-15 8-22 8-23 10-6 10-10 10-12
network:2-2 2-7 2-8 2-12 2-19 3-3 3-4 3-6 3-7 3-10 4-2 4-7 4-30 4-31 4-37
4-38 4-41 4-42 4-52 4-53 4-55 4-60 4-61 4-63 4-76 5-3 5-11 5-12 5-13 5-14
5-15 5-18 5-19 5-26 5-28 5-49 7-5 7-7 7-11 7-14 7-21 7-23 7-24 7-29 8-2 8-3
8-4 8-5 8-7 8-9 8-10 8-15 8-21 8-23 9-22 9-24 9-40 9-42 9-45 9-49 10-4 10-5
10-12

network partition:4-41 4-52 4-63 7-29

object oriented:2-11 4-42 9-3 9-4
one-phase commit:4-26 4-31 4-35 4-36
optimistic:4-6 4-8 4-12 4-13 4-14 4-42
orphan: 4-76

path expression:9-44 9-49 9-50 9-51
path pascal:5-1 8-2 9-38 9-40 9-44 9-49 9-55 10-6 10-11
paws:7-7 7-10 7-12 7-17 7-18 7-24 7-26 7-27 7-32 8-2 10-10 10-11
performance evaluation:7-1 7-3 7-34 7-38 8-2 10-11
performance requirements:2-4 2-18 10-4 10-6
perseverance:9-30 9-31
pmdb_log buffer:5-37 5-38 5-41
poisson:8-12
presumed:4-34 5-39 5-46 9-22
presumed abort:4-34
presumed commit:4-34
principal:5-11 5-24 5-25 5-26
process manager:4-33 4-34 5-30 5-32 5-33 5-34 5-36 5-38 5-46 5-53 9-25
9-26 9-27 9-28 9-29 9-30 9-31
program type manager:5-28
ptm:5-40 9-40 9-47 9-54

recoverable object:9-4 9-5 9-6 9-7 9-9 9-12
recovery block:4-77
redo:4-23 4-24
redundancy:2-2 2-6 2-7 2-8 4-1 4-4 4-16 4-77 5-7 10-6
reliability:2-1 2-4 2-5 2-6 2-8 2-11 2-12 2-13 2-14 2-15 2-17 2-18 2-19
4-1 4-4 4-5 4-16 4-36 4-39 4-64 4-68 4-76 4-77 5-1 5-2 5-3 5-9 5-26 5-28
5-29 5-31 7-1 7-2 7-7 7-8 7-9 7-25 7-26 7-28 8-1 8-2 8-3 8-4 8-5 8-7 8-8
8-9 8-10 8-13 8-15 8-16 8-21 8-22 8-23 9-38 10-1 10-3 10-4 10-5 10-6 10-7
10-8 10-10 10-11 10-12
reliability function:8-3 8-5 8-8 8-9
reliability requirements:2-4 2-5 2-11 2-14 2-18 5-2 8-2 8-4 8-9 10-3 10-4
10-6 10-7 10-10
reliability specification:2-17 8-22
remote procedure call:4-6 4-7 4-76 5-4 5-5 5-11 5-19 5-20 5-53 7-4 7-21
7-22 7-24 10-9
replication:2-8 2-9 2-13 2-18 2-19 3-8 4-4 4-36 4-38 4-47 4-60 4-77 5-3
5-9 5-26 5-28 5-48 5-51 7-8 7-10 7-25 7-28 8-10 8-15 8-16
replication management:4-60 5-51 8-10 8-15
resilient:2-13 2-14 2-16 2-17 4-42 4-44 4-51 5-2 8-4
resilient object:2-13 2-14
response time:2-4 2-11 2-18 3-2 4-14 4-28 4-31 4-33 4-36 4-39 7-2 7-4 7-8

7-21 10-4 10-5
rpc:5-11 5-18 5-49 5-50 5-51 7-22 7-23 7-29

scenario:2-6 2-17 3-1 3-8 3-10 4-70 7-36 7-37 8-5 8-7 9-2 9-15 9-17 9-19
security:3-2 3-4 3-5 4-64 5-8 9-2 9-3 9-4 9-5 9-38 9-41 10-3 10-5 10-12
serializability:4-2 4-7 4-9 4-12 4-33 4-36 4-51 5-42
serialization:4-8 4-12 4-13 4-53
service node:7-14 7-16
shadow:4-19 4-21
sibling:7-16 7-23 7-24 7-27
simula:7-7
simulation:4-14 5-1 7-6 7-7 7-8 7-10 7-11 7-16 7-17 7-18 7-34 7-38 8-1 8-2
8-15 9-38 9-40 9-42 9-43 9-44 9-45 9-48 9-53 9-55 10-10 10-12
sink:7-16 7-24 7-27 7-32 7-34
snm:5-25 5-26
source:7-15 7-16 7-23 7-30 7-33 9-48
sphere of control:5-31 5-34 5-41 5-44
split:7-15 7-16 7-23 7-24 7-27 7-33 9-18
stable:2-13 2-14 2-16 2-17 4-4 4-5 4-6 4-17 4-20 4-21 4-23 4-24 4-25 4-26
4-28 4-29 4-30 4-34 4-64 4-65 4-67 4-68 4-69 4-76 5-9 5-11 5-12 5-15 5-18
5-20 5-22 5-28 5-33 5-36 5-38 5-41 5-45 5-52 5-53 7-27 8-4 8-10 8-13 8-15
8-21 8-23 9-4 9-5 9-8 9-11 9-12 9-21 9-22 9-26 9-28 9-47
stable object:5-18 9-4 9-8 9-11
stable storage:4-5 4-6 4-17 4-20 4-21 4-23 4-24 4-25 4-26 4-28 4-29 4-30
4-34 4-64 4-65 4-67 4-68 4-69 4-76 5-9 5-11 5-12 5-18 5-22 5-33 5-36 5-38
5-41 5-45 5-52 5-53 7-27 8-10 8-15 8-21 9-26 9-28 9-47
stable_get:5-21
stable_put:5-21
stablepage:4-20 4-21 8-10
status queries:5-46
strong consistency:4-36 4-38 4-41 4-60 5-51
survivability:2-1 2-7 3-2 3-3 3-7 3-9 3-10 6-1 8-13 10-8 10-12
symbolic name manager:5-24 5-25

temporal:9-22
throughput:2-4 2-9 2-11 2-18 4-14 4-15 4-39 7-2 7-9 10-4
timestamp:4-8 4-13 4-43 4-44 4-56 4-59 5-38 5-39 5-41 5-49 5-50
token:4-11 7-24 7-28
top level:4-71 5-41 5-42 5-44 5-46 9-3 9-14 9-15
trace:5-22 7-34 9-38 9-40 9-53 9-54
transaction context list:5-43
transaction tree:5-42 5-44 5-46
transient error:8-10 8-12
two phase commit:4-27 4-31 4-32 4-33 4-36 4-42 4-44 5-44 5-46 9-48
two phase locking:4-9

undo:4-19 4-23 4-24 5-44 5-50 9-10
unique identifier:4-5 4-6 5-6 5-12 5-18 5-19 5-20 5-21 5-24 5-29 5-30 5-36
9-17 9-53

validation:2-4 4-12 4-13 7-5 7-6 8-1 8-2 9-3 9-38 9-39 9-41
verification:2-19 9-3 9-4 9-5 9-15 9-18 9-22 9-32 9-38 10-6 10-11 10-12
virtual machine monitor:9-47 9-48
vmm:9-48 9-49 9-50 9-51 9-52
volatile storage:4-23 4-24 4-25 4-28 4-33 4-34 5-7

weak consistency:4-37 4-38 4-39 4-41 4-52 4-59
weighted voting:4-50 4-51 5-32 5-51 8-16 8-21
window of vulnerability:9-42
workload:2-18 2-19 3-6 7-2 7-4 7-5 7-10 7-11 7-18 7-34 7-35 7-37 9-38 10-4
10-5

zeus:3-8 5-1 5-2 5-3 5-4 5-5 5-6 5-9 5-10 5-12 5-23 5-25 5-26 5-28 5-30
5-31 5-32 5-33 5-34 5-41 5-50 5-51 5-52 5-53 7-1 7-10 7-26 7-29 7-30 7-32
7-33 7-35 7-36 7-37 9-38 9-39 9-43 9-45 9-51 10-4 10-8 10-9



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 (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.