CRITICAL ISSUES IN ULTRA-RELIABLE PARALLEL PROCESSING

by

Rick Harper

SUBMITTED IN PARTIAL FULLFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1987

© Richard Edwin Harper

Signature of Author_____
May 11, 1987

Approved by *Wallace E VanderVelde*
Professor W. E. VanderVelde, Committee Chairman
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology

Approved by
Dr. J. J. Deyst
Associate Department Head, Manufacturing Automation Department
Charles Stark Draper Laboratory

Approved by
Professor T. F. Knight
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Approved by
Dr. J. H. Lala, CSDL Technical Supervisor
Division Leader, System Architectures Division
Charles Stark Draper Laboratory

Accepted by
Professor H. Y. Wachman
Chairman, Departmental Graduate Committee
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAY 27 1987
LIBRARIES

ACKNOWLEDGEMENT

This work would not have been possible without the outstanding personal and technical support of the staff of the Charles Stark Draper Laboratory, especially that of Jay Lala and John Deyst. I also appreciate and respect the willingness of my thesis committee to go with me into an uncharted area. Most importantly, I would like to acknowledge the steadfast love and support of my wife Ruth.

....

This report was prepared at the Charles Stark Draper Laboratory, Inc. under Internal Research and Development Technical Plan Task 246.

Publication of this report does not constitute approval by the Draper Laboratory or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc. Cambridge, Massachusetts.

Richard E. Harper

Permission is hereby granted by the Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

Table of Contents

Page

ABSTRACT 13

CHAPTER 1. INTRODUCTION 14

1.1 Problem Statement 14

1.2 Prior Work 17

1.3 Prerequisite 2: Connectivity 18

1.4 Prerequisite 4: Synchronization 19

1.5 Objective 20

1.6 Approach 21

CHAPTER 2. THE COMPUTATIONAL PROBLEM 24

2.1 Description of the Computational Problem 24

2.2 One Parallelization of the Algorithm 29

2.3 Architectural Implications 33

CHAPTER 3. RELIABILITY ANALYSIS PRELIMINARIES 36

3.1 Reliability Requirements 36

3.2 Approach and Assumptions 37

3.3 Fault Taxonomies 49

3.4 The Fault Detection and Recovery Process 54

CHAPTER 4. RELIABILITY ANALYSIS OF CONVENTIONAL APPROACHES 59

4.1 Simplex Analysis 59

4.2 Duplex Analysis 67

4.3 Fault Masking Analysis 79

4.3.1 Analysis of Quadruply Redundant Fault Masking Architecture 86

4.3.2 Analysis of Triply Redundant Fault Masking Architecture 95

CHAPTER 5. CONNECTIVITY 101

5.1 Existing Approaches to Connectivity Provision 102

5.2 Embedding Regions of Connectivity in Existing Parallel Topologies..... 102

5.3 Existence of an Embedding 103

5.4 Efficiency of the Embedding 106

5.5 Reconfiguration of the Embedding After Failures 107

5.6 Diagnosability of the Embedded System 107

5.7 Conclusions 108

CHAPTER 6. RELIABILITY ANALYSIS OF FULLY-CONNECTED CLUSTER ARCHITECTURE 110

6.1 Fully-Connected Cluster Description 110

6.2 Approximate Formulation of the Probability of Ensemble Loss 118

6.3 Simultaneous Failures 118

6.4 Optimization of Attrition Resilience 122

6.5 Analytical Results and Conclusions 123

CHAPTER 7. RELIABILITY ANALYSIS OF NETWORK ELEMENT-BASED CLUSTER ARCHITECTURE 127

7.1 Network Element-Based Cluster Description 127

7.2 Assumptions and Approach 136

7.3 Network Element Aggregate.Markov Model 136

7.4 Probability of Ensemble Loss Due to Simultaneous Failures 137

7.5 Overestimation of Simultaneous Processor Failure Probability............ 139

7.6 Underestimation of Simultaneous Processor Failure Probability 141

7.7 Linear Approximation to Overestimation 142

7.8 Probability of Ensemble Loss Due to Attrition 143

7.9 Probability of Ensemble Loss due to Cluster Isolation..................... 146

7.10 Total Probability of Ensemble Loss 151

7.11 Relative Magnitudes of Failure Modes 154

7.12 Expected Value of Number of Triads in Ensemble 158

7.13 Mean Time to System Failure 160

7.14 Architectural Optimization 161

7.15 Optimization of Simultaneous Failure Resilience 162

7.16 Optimization of Attrition Resilience 164

7.17 Results for the Optimized Architecture 167

CHAPTER 8. RELIABILITY ANALYSIS CONCLUSIONS 170

8.1 Summary of Quantitative Analysis 170

8.2 Conclusions From Simplex Analysis 170

8.3 Conclusions from Analysis of Duplex Ensemble 172

8.4 Conclusions From the Fault Masking Analysis 174

8.5 Conclusions from the Fully Connected Cluster Analysis 175

8.6 Conclusions from the Network Element-Based Cluster Analysis 176

8.8 General Conclusion 177

CHAPTER 9. CONSISTENCY MAINTENANCE 179

9.1 Introduction 179

9.2 Consistency Abstraction 180

9.2.1 Consistency in Data 181

9.2.2 Consistency in Ordering 183

9.2.3 Consistency in Time 185

9.2.4 Definition of Consistency Maintenance 192

9.3 Requirements for Consistency Maintenance 193

9.4 Existing Approaches to Consistency Maintenance 194

9.4.1 Consistency in Time 195

9.4.1.1 SIFT Synchronization Methodology 195

9.4.1.2 AIPS FTP Synchronization Methodology 199
9.4.2 Consistency in Data 204
9.4.2.1 SIFT Data Consistency Maintenance Methodology 204
9.4.2.2 AIPS FTP Data Consistency Maintenance Methodology...... 205
9.4.3 Consistency in Ordering 206
9.4.3.1 SIFT Order Consistency Maintenance Methodology 206
9.4.3.2 AIPS FTP Order Consistency Maintenance Methodology 206
9.5 Byzantine Resilient Virtual Circuit Abstraction 208
9.6 Desired Features of a BRVC Implementation in a Parallel System 208
CHAPTER 10. PROPOSED APPROACH TO CONSISTENCY MAINTENANCE 211
10.1 Cluster Architecture and Operation 211
10.1.1 Organization of the Cluster 211
10.1.2 Definitions 213
10.1.3 Fault Containment Region Architecture 215
10.2 Functional Synchronization 219
10.3 Exchange Classes 228
10.3.1 Class 1 Exchange 228
10.3.2 Class 2 Exchange 236
10.4 Operation of Network Element Core 243
10.4.1 Network Element Core Cycle 243
10.4.2 Achieving Frame Synchrony 251
10.5 Provision of BRVC Abstraction 255
10.6 Cluster Expansion 258
10.6.1 Inter-Cluster Message Transmission 258
10.6.2 Inter-Cluster Message Reception 260
10.7 Failure Detection, Identification, and Reconfiguration 263

10.7.1 Distributed Failure Detection and Identification 263

10.7.2 Distributed Reconfiguration 264

10.7.3 Distributed Bootstrap and Reintegration 267

CHAPTER 11. PERFORMANCE ANALYSIS 272

11.1 Goal and Approach 272

11.2 Detailed Description of Basic Cycle 273

11.3 Simulation Description 279

11.4 Simulation Results 284

11.5 Conclusions 290

CHAPTER 12. CONCLUSIONS AND RECOMMENDATIONS

FOR FUTURE WORK 297

12.1 Conclusions 297

12.2 Recommendations for Future Work 300

REFERENCES 302

APPENDICES 307

Appendix A Ensemble Loss Curves for Network Element-Based Clusters..... A-1

Appendix B Relative Complexities of Processing, Network, and IO Elements B-1

List of Figures

FIGURE — Page

1.1 Application Domains of Fault Tolerant Computers 15
1.2 Performability of Existing Fault Tolerant Computers 16
2.1 Perturbation of a Plan 28
2.2 Interaction of Goalpoint Planner and Waypoint Planner 30
2.3 Example A* Search of Terrain Database by Waypoint Planner 32
2.4 Task Flow Diagram for Simple Parallelized Goalpoint Planner 33
2.5 Simplified Architecture of a MIMD System 35
3.1 Processing Element Architecture 38
3.2 Overall Algorithmic Structure 41
3.3 Probability of Result Corruption versus Iteration for Synchronous Iterative Algorithm 43
3.4 MTTSF vs. Number of PEs in the Ensemble 46
3.5 Domino-Effect Rollback 57
4.1 Markov Model of Simplex Processor Behavior 61
4.2 Ensemble Loss Probability for 64-PE Simplex Ensemble 64
4.3 Expected Value of Number of PEs vs. Mission Time for 64-PE Simplex Ensemble 65
4.4 Self-Checking Pair Processing Element 73
4.5 Externally Checked Pair 74
4.6 Markov Model of Self Checking Pair 75
4.7 Ensemble Loss Probability for 64-PE Duplex Ensemble 76
4.8 Expected Value of Number of Groups vs. Mission Time for 64-Group Duplex Ensemble 77

4.9 Simple TMR Architecture....80

4.10 Minimal Byzantine Resilient Architecture....83

4.11 CSDL Fault Tolerant Processor Architecture....85

4.12 SRI Software Implemented Fault Tolerance (SIFT) Computer....86

4.13 Markov Model for Symmetric 4-Processor Fault Masking Architecture....87

4.14 Ensemble Loss Probability for 64-PE Quadruplex Ensemble....88

4.15 Markov Model to Demonstrate Linear Approximation to System Loss Probability....89

4.16 Comparison of Linear Approximation to Full Markov Model Result....91

4.17 Comparison of Unadjusted Combinatorial Model to Full Markov Model Results....92

4.18 Comparison of Adjusted Combinatorial Model to Full Markov Model Results....93

4.19 Comparison of Composite System Loss Curve to Full Markov Model Results....94

4.20 Expected Value of Number of FMGs vs. Mission Time for 64-FMG Quadruplex Ensemble....94

4.21 Markov Model of Triplex Fault Masking Architecture....97

4.22 Ensemble Loss Probability for 64-PE Triplex Ensemble....98

4.23 Expected Value of Number of FMGs vs. Mission Time for 64-FMG Triplex Ensemble....98

5.1 Impossibility of Embedding a Byzantine Resilient Group in the Minimal Square of Planar Mesh....104

5.2 One Possible Embedding of Byzantine Resilient Group in Planar Mesh....105

5.3 Tessellation of Embedded Byzantine Resilient Sites In a Planar Mesh....105

5.4 One Possible Embedding of Byzantine Resilient Group in a Toroidal Mesh....106

5.5 Embedding of Four 1-Byzantine Resilient Computing Sites in a Hypercube 107

5.6 Single Byzantine Resilient Processing Site Embedded in a Hypercube 108

6.1 Parallel Architecture Composed of Small Byzantine Resilient Sites Connected by Diagnosible Links 111

6.2 Two 8-Processor Clusters 112

6.3 8 8-Processor Clusters Connected into a Hypercube 114

6.4 Markov Model for 7-Processor Cluster 116

6.5 Comparison Between Linear Overapproximations for Quad and Cluster ... 119

6.6 Comparison of Linear Approximation and Full Markov Model Result for 8-Processor Cluster 120

6.7 Bus-Oriented Inter-Cluster Topology 121

6.8 Fully Connected Clusters 123

6.9 Ensemble Loss Probability for 64-FMG Fully Connected Cluster-Based Ensemble 124

6.10 Expected Value of Number of FMGs vs. Mission Time for 64-FMG Fully Connected Cluster-Based Ensemble 125

7.1 16-Processor Cluster Using 4 Network Elements 131

7.2 Possible 16-Processor Cluster Configuration 132

7.3 Reconfiguration of a Network Element-Based Cluster 133

7.4 Inter-Cluster Connection Using IO Elements 135

7.5 Markov Model of Network Element Aggregate 138

7.6 Overestimation Model for a Cluster of 7 Processors 140

7.7 Markov Model of Underestimate Model 142

7.8 Comparison of Approximation Methods For Ensemble Loss Probability For Network Element-Based Clusters 143

7.9 Constituents of Ensemble Loss Probability for Network Element-Based Cluster $\lambda_{IOE0} = 10^{-4}$/hour 155

7.10 Sensitivity of MTTSF to λ_{IOE0}........ 156

7.11 Architecture of Simplified IO Element........ 157

7.12 Constituents of Ensemble Loss Probability for Network Element-Based Cluster $\lambda_{IOE0} = 10^{-5}$/hour 158

7.13 Sensitivity of Normalized Near-Simultaneous NE Failure Probability to $N_{PE/NE}$ 164

7.14 MTTSF Versus Architectural Design Parameters 167

7.15 Ensemble Loss Probability for 64-FMG Network Element-Based Ensemble........ 168

7.16 Expected Value of Number of FMGs vs. Mission Time for 64-FMG Network Element-Based Ensemble........ 168

8.1 Comparison of Ensemble Loss Probabilities vs. Mission Time........ 171

8.2 Comparison of Expected Number of Processing Sites vs. Mission Time .. 171

8.3 Comparison of Short-Term Failure Rate........ 172

8.4 Comparison of Mean Time to System Failure........ 172

9.1 Input Data Consistency Abstraction 181

9.2 Guaranteed Message Delivery Abstraction........ 182

9.3 Input Ordering Consistency Abstraction 184

9.4 Output Ordering Consistency Abstraction 184

9.5 Consistency in Time Abstraction........ 185

9.6 Framewise Synchrony 186

9.7 Minimal Byzantine Resilient Processing Group........ 189

9.8 Markov Model for Symmetric 4-Processor Fault Masking Architecture 190

9.9 Consistency Abstraction 192

9.10 Synchronization-Specific SIFT Architectural Details........ 195

9.11 SIFT Synchronization Scheme 198
9.12 Architectural Details of FTP Synchronization Technique 200
9.13 Adjustment of Frame Length in FTP Synchronization Scheme 202
9.14 Simplified AIPS FTP Architecture 207
9.15 Byzantine Resilient Virtual Circuit Abstraction 209
10.1 FTPP Cluster 212
10.2 Configuration Table 214
10.3 Fault Containment Region Architecture 216
10.4 Frame Definition by Reading Input Buffer 220
10.5 Functional Synchronization 224
10.6 Scooping 225
10.7 Phase 1 of a Class 1 Exchange 229
10.8 FIFO Map Following Phase 1 of Class 1 Exchange 230
10.9 Phase 2 of a Class 1 Exchange 231
10.10 FIFO Map Following Phase 2 of Class 1 Exchange 232
10.11 Phase 3 of a Class 1 Exchange 233
10.12 FIFO Map Following Phase 3 of Class 1 Exchange 234
10.13 Phase 1 of Class 2 Exchange 236
10.14 FIFO Map Following Phase 1 of Class 2 Exchange 237
10.15 Phase 2 of Class 2 Exchange 238
10.16 FIFO Map Following Phase 2 of Class 2 Exchange 239
10.17 Phase 3 of Class 2 Exchange 240
10.18 FIFO Map Following Phase 3 of Class 2 Exchange 241
10.19 FIFO Map Following Phase 4 of Class 2 Exchange 242
10.20 Basic Network Element Aggregate Cycle 243
10.21 Exchange Request Format 244
10.22 Local Exchange Request Pattern for Network Element 1 ($LERP_1$) 245

10.23 System Exchange Request Pattern (SERP) 246

10.24 FIFO Map Obtaining After LERP Broadcast 247

10.25 Network Element Synchronization Algorithm 255

10.26 Three Clusters 259

10.27 Inter-Cluster Message Order Ambiguity 261

11.1 Basic Cycle of Cluster 274

11.2 Sourcing an Inter-Cluster Message Transmission 281

11.3 Forwarding of an Inter-Cluster Message 282

11.4 Delivery of an Inter-Cluster Message 283

11.5 Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 4$ 285

11.6 Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 8$ 285

11.7 Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 16$ 286

11.8 Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 32$ 286

11.9 Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 64$ 287

11.10 Comparison of Mean Intra-Cluster Delay vs. t_{FMG_period} for all Cluster Sizes 287

11.11 Comparison of Mean Inter-FMG Delay vs. t_{FMG_period} for all Cluster Sizes 288

11.12 Convergence of Mean Intra-Cluster Delay vs. Number of Monte Carlo Trials 289

11.13 Convergence of Mean Inter-Cluster Delay vs. Number of Monte Carlo Trials 289

11.14 Convergence of Mean Inter-FMG Delay vs. Number of Monte Carlo Trials 290

11.15 Sensitivity of Mean Delay to t_{IOE_period} 294

11.16 Sensitivity of Mean Delay to Locality of Reference Parameter ϕ 295

12.1 Performance and Reliability of Proposed Architecture 299

List of Tables

Table

Page

3.1 General Performance Parameters....47

3.2 Simplified Fault Taxonomy....54

4.1 Performance Parameters for 64-PE Simplex Ensemble....66

4.2 Performance Parameters for 64-PE Duplex Ensemble....78

4.3 Performance Parameters for 64-PE Quadruplex Ensemble....95

4.4 Performance Parameters for 64-PE Triplex Ensemble....99

6.1 Performance Parameters for 64 FMG Fully Connected Cluster-Based Ensemble....126

7.1 MTTSF Versus Architectural Design Parameters....166

7.2 Performance Parameters for 64 FMG Network Element-Based Ensemble....169

ABSTRACT

Future mission-critical computing systems will have throughput requirements that current fault tolerant computers cannot meet, while currently available high-throughput computers have inadequate reliability for such applications. This thesis attempts to bring together the disciplines of fault tolerant computing and parallel computing by formulating and addressing some issues central to the design of a computer meeting these throughput and reliability needs. Specifically, the problem to be discussed is the design of a Byzantine Resilient Multiple Instruction stream Multiple Data stream computer. Such an architecture must meet lower bounds on the number of fault containment regions (FCRs), the number of disjoint paths between FCRs, the amount of communication among FCRs, and the degree of asynchrony allowable between FCRs.

This thesis shows that the addition of Byzantine Resilience to existing parallel ensembles requires embedding the requisite connectivity into the existing interprocessor topology, incurring serious diagnosis, reconfiguration, efficiency, and anisotropy penalties. In the proposed architecture the connectivity requirements are met by treating connectivity as a resource which is shared among many processors, allowing flexibility in their configuration and reducing its relative cost. Full interprocessor connectivity is replaced by the interposition of specialized *Network Elements* between multiple processors and a much smaller region of shared connectivity. An quantitative comparison of different fault tolerance techniques applied to a parallel ensemble shows that this approach reduces the ensemble's short-term failure rate by an order of magnitude and increases its Mean Time to System Failure by a factor of four over an equivalent ensemble constructed from conventional Byzantine Resilient fault masking processing sites.

The synchronization requirements are met by defining *functional synchrony*, in which redundant sites are synchronized solely by message transmissions and receptions arising from synchronizing acts occurring naturally in the course of the applications program's execution. Functional synchronization removes many constraints imposed upon the ensemble's components by prior synchronization techniques, allowing ease and efficiency of implementation, heterogeneous redundant sites, and an easily extendable and open system. Calculations of message transmission delays incurred in a shared region of connectivity show that less than 5% of the processor's throughput is consumed in performing fault tolerance-specific message transmissions. This figure compares favorably with similar performance figures of other ultra-reliable architectures and shows that the proposed architecture meets the throughput requirements of the application.

To mask the complexity of the underlying redundant parallel system, the Byzantine Resilient Virtual Circuit Abstraction (BRVC) is presented to the programmer of the parallel ensemble and supported by the proposed architecture. BRVC is a set of guarantees that messages are delivered uncorrupted, in the order sent, in identical order at different members of a recipient redundant site, and within a bounded time window at different members of a recipient redundant site. This abstraction has vastly facilitated the development of the programming model for the prototype of this architecture.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Over the past decade various ultra-high reliability computers have been conceived, fabricated, tested, and used in several applications. These computers include the CSDL Fault Tolerant Processor (FTP) [Smith83], the CSDL Fault Tolerant MultiProcessor (FTMP) [Hopkins78], an FTP with attached processors [Houtchens83], and the Software Implemented Fault Tolerance (SIFT) computer [Wensley78]. The throughput and short-term failure probability of these computers range between 1 to 10 Million Instructions Per Second (MIPS) and 10^{-6}/hour to 10^{-9}/hour, respectively. When such computers are interconnected by a fault and damage tolerant circuit switched network, as in the CSDL Advanced Information Processing System (AIPS) [CSDL84], the system's performance and reliability can be increased to an extent.

However, there are problems in the area of intelligent systems that require orders of magnitude more processing power than can be mustered by a system constructed with these currently available building blocks. Such an application is the problem of planning a mission in the presence of a changing external environment to meet certain objectives subject to certain practical considerations[Adams86]. Traditional solutions to such problems employ dynamic programming algorithms, but the application of dynamic programming to systems of even moderate dimension quickly becomes impractical. Other algorithms are being developed that are more efficient than dynamic programming, but even these algorithms are computationally intensive. It is therefore reasonable to conclude that the most advanced fault tolerant computer architectures currently being developed will fall short of the required throughput by a factor of 10 to 100. Similarly, currently existing computers that have the requisite throughput do not have the required reliability for mission or life-critical applications.

A useful framework for viewing this problem is presented in Figure 1.1. The requirements of a given application can be located on a two-dimensional graph, with the abscissa of the graph representing the throughput requirement of the application and the ordinate representing its reliability requirement.

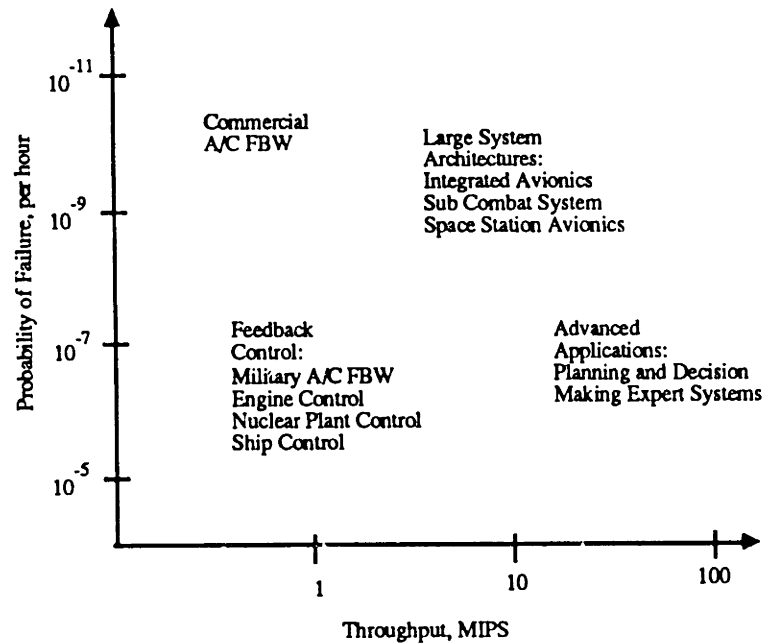


Figure 1.1

Application Domains of Fault Tolerant Computing Systems

Figure 1.2 positions existing fault tolerant computers and some existing high-speed computers on this graph. It can be seen that existing fault tolerant computers are capable of meeting the needs of commercial aircraft fly-by-wire, feedback control, and large system architectures, but are not capable of meeting the throughput needs of mission-critical planning and decision making expert systems. It can also be seen that existing high-speed computers such as the hypercube and the Cray-1 are incapable of meeting the reliability requirements of this application. The architecture developed in this thesis is to occupy the shaded area in Figure 1.2.

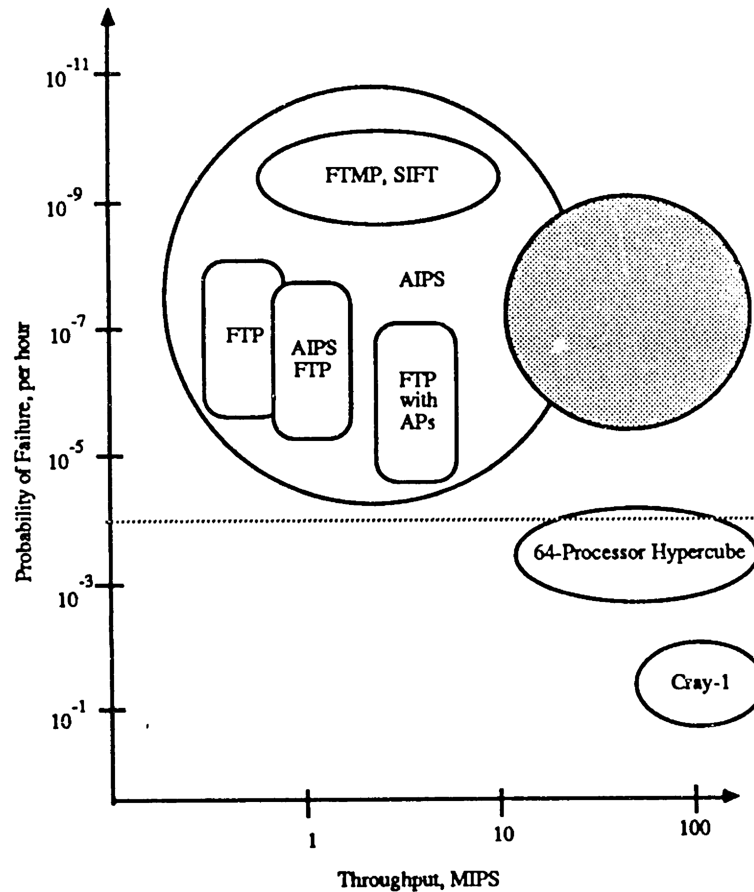


Figure 1.2

Performability of Existing Computers

A parallel processing computer has the potential to bring significant benefits to the above problem. First, to the extent that the addition of processors appreciably increases the net system throughput (e.g., average speedup of n/ln n considered typical) [Hwang84], the raw computational capability of a parallel computer makes the problem computationally feasible. Study of the algorithms most likely to be executed in the intelligent systems applications of interest indicates that adequate intrinsic parallelism exists to make relatively efficient use of a parallel computer capable of supporting parallel execution of many coarse granularity loosely coupled tasks. Second, the existence of a number of identical computing components in such a computer leads to the hope that a highly reliable implementation is possible.

For such a computer to be considered adequately reliable for life or mission critical applications, it must be capable of surviving a specified number of component failures with a probability approaching unity [Bouricius71]. In addition, a conservative but not unrealistic model of failure behavior is to consider failures as consisting of arbitrary behavior on the part of failed components [Lala83], [Martin78]. Examination of how the above requirement and failure hypothesis combine to impact the design and implementation of a parallel computer reveals several critical issues that must be explicitly addressed in a plausible design. These issues arise from the requirement that, in the face of arbitrarily malicious or "Byzantine" failure behavior on the part of some of the processors, the system must be capable of obtaining bitwise consensus on input information as a necessary precursor to obtaining bitwise congruent redundant computation and subsequent high-coverage fault detection and isolation via bitwise output comparison.

1.2 Prior Work

Techniques for achieving high-coverage tolerance of malicious faults for a relatively small number of functional components are well understood [Rennels84]. Such fault tolerance schemes generally employ component replication. To achieve high-coverage failure detection by bitwise comparison of the outputs of redundant components, consensus protocols must be implemented when members of a redundant group need to agree on something, e.g., input data or diagnostic information. These protocols must also be resilient to arbitrary failure modes. Consensus protocols have been extensively studied, resulting in the well-known and theoretically demonstrable prerequisites for consensus algorithms which are capable of correctly functioning in the face of arbitrary failure behavior on the part of "f" of the participants in the algorithm. One form of these requirements may be expressed as:

1. There must be at least 3f+1 participants in the algorithm [Pease80].
2. Each participant must be connected to at least 2f+1 other participants through disjoint

communication paths [Dolev82].

3. The algorithm must consist of a minimum of f+1 rounds of communication among the participants [Fischer82].

4. The participants must be synchronized to within a known skew of each other [Dolev84].

A system which meets these prerequisites and is capable of executing the appropriate algorithm is called "f-Byzantine Resilient". The first and third prerequisites are readily achieved in a parallel environment, but methods to achieve the second and fourth are not obvious.

1.3 Prerequisite 2: Connectivity

The connectivity of prerequisite two is absent in most current and proposed parallel computers. For example, consider a parallel system which consists of a number of computational nodes connected in a planar mesh, a common interconnection topology. To form a redundant computational site which would be 1-Byzantine Resilient, four nodes are needed. But if one tries to form a redundant site from the elements at the corners of a minimal square of the mesh, it proves impossible to provide each node with three disjoint paths to the other members of the redundant site because the paths through the plane must cross. It is possible to embed a 1-Byzantine Resilient topology into a planar mesh in other ways by using intermediate routing nodes, but issues such as anisotropy of communication, tessellation of the sites in the plane, efficiency, reconfiguration, and diagnosability are difficult to address, especially under random site failures. This is true of many other common interconnection networks, such as the hypercube, the tree, the star, rings, and others.

This "embedding" problem is equivalent to the subgraph isomorphism problem, which is known to be NP-complete [Tsai82]. While it may be possible to find an initial embedding by trial and error, finding an embedding in the random graph corresponding to

a system suffering failures is difficult and time-consuming. It must also be noted here that the PMC fault diagnosis model [Preparata67] does not fully capture malicious failure behavior, so the results of this body of literature are not directly applicable to the diagnosis of malicious faults in an embedding.

Byzantine Resilient computers have traditionally taken one of two approaches to solving the connectivity management problem. The first approach is to provide tiny islands of theoretically sufficient connectivity [Lala86b]. Expansion of systems such as these usually consists of connecting these islands to each other through diagnosable links [CSDL84]. The problem with this approach to expansion is that Byzantine Resilient sites cannot be formed from members of different islands, thus limiting the configuration options of the system and hence its attrition resilience. The second approach is to provide enough connectivity to allow a redundant site to be formed from any subset of members in the system [Wensley78]. This approach works well for a small system, but the cost of providing such connectivity increases quadratically with the number of members. Therefore it is worthwhile to investigate new schemes for providing connectivity in a parallel system.

1.4 Prerequisite 4: Synchronization

The fourth prerequisite for Byzantine Resilient computation is that redundant sites must be synchronized to within a known skew. Mutual synchrony is necessary to guarantee termination of any deterministic distributed algorithm in the face of faulty behavior, as well as to allow detection of a halted or excessively slow processor. This fundamental problem in fault tolerant computers is to a large extent responsible for many of the constraints imposed on their design and operation. Moreover, current means for synchronizing fault tolerant processors and multiprocessors do not tractably extend to synchronizing redundant sites in a distributed, parallel computer.

Frame synchronous computers constrain the programmer by forcing applications code segments, iterative or otherwise, to fit within an arbitrarily imposed time frame, which

must in turn be sized to accommodate worst-case execution time of all tasks executing in the frame. This cognitive overhead coupled with that of programming a parallel computer would make a frame synchronous parallel system exceedingly difficult to program. In addition, frame synchronous systems utilize an undue proportion of the processor's capability in executing the synchronization and scheduling activity [Palumbo86].

Tightly synchronized systems require the dissemination of a high-speed fault tolerant clock signal, an achievement which increases in difficulty with the size of the system and is incompatible with the precepts of a distributed system. Furthermore, this method requires that the processors composing the redundant groups are *clock-deterministic*, in that each member of the redundant group must execute exactly the same number of instructions in a given number of clock cycles. This constraint typically requires that the designers of such computers weed out all asynchronies and nondeterminacies that might possibly exist in the components to be run in synchrony; i.e., they must design their own functional-level components (e.g., processor cards) from scratch. Otherwise perfectly acceptable processor behavior, such as correcting a memory error, would take one member of a redundant group longer than another and cause that member of the group to lose synchronization and be declared faulty. Again, a new approach must be developed to provide the requisite processor synchrony in a parallel system.

1.5 Objective

The primary objective of this thesis is to combine the disciplines of fault tolerant computing and parallel processing to yield theoretically sound design principles for a high throughput highly reliable computer. Specifically, this includes finding efficient and implementable ways to satisfy the theoretical requirements for the tolerance of arbitrary failure behavior in a parallel computer, of which the most difficult to satisfy are connectivity and synchronization. The second objective is to quantify the benefits of the proposed solutions by the analysis of the reliability and performance of a computer

fabricated according to these solutions.

1.6 Approach

To motivate the problem, an application area of interest is selected and used to define throughput and reliability goals. A class of algorithms likely to be executed in this candidate application is used to determine the architecture's functional characteristics. This process results in a Multiple Instruction Multiple Datastream architecture consisting of an ensemble of highly integrated Processing Elements connected via an intercomputer network.

Reliability analysis is then used as a framework for the introduction and comparison of different fault tolerance approaches applied to a parallel processor ensemble. Reliability analysis of several candidate arrangements of parallel processors is carried out. These are:

1. An ensemble of processors each of which operates in simplex mode;
2. A number of processors which are grouped into unreconfigurable duplex processing groups;
3. A number of processors which are grouped into unreconfigurable triplex processing groups;
4. A number of processors which are grouped into unreconfigurable quadruplex processing groups;
5. A number of variable-sized fully connected groups of processors in which any processor may be combined with any others to form a redundant group;
6. A number of variable-sized regions of partial connectivity having the theoretical requirements for Byzantine Resilience provided by a set of specialized

The results of these analyses show that near-unity coverage of random faults is required if the desired reliability goals for the parallel computer are to be met. The reliability analyses also lead to the conclusion that the use of Network Elements to provide a region of shared connectivity results in a significant enhancement of the system's short and long-term reliability.

A proposed solution to providing an efficient and unconstraining synchronization mechanism using these Network Elements is presented and compared to existing methods. This new method, denoted functional synchronization, uses naturally-occurring breakpoints in the applications code to serve as synchronization instances. At these breakpoints members of a given redundant site transmit a message to themselves and await the reception of a congruent copy of that message. Upon reception of this message, the redundant sites are guaranteed by the operation of the underlying Network Element aggregate to be in synchrony. Functional synchronization is suitable for use in a distributed parallel system, in which the distribution of a high speed fault tolerant clock is difficult. It also has the potential for an efficient and programmer-transparent implementation, and does not obviate the use of other more conventional synchronization techniques where desired. Finally, the use of the functional synchronization technique yields advantages arising from the removal of severe constraints on the homogeneity of the redundant sites composing the system. Such advantages include the capability to implement a system possessing temporally disparate, nondeterministic, and heterogeneous Processing Elements, and hardware and software design diversity.

Members of several redundant processing sites share the Network Element aggregate to perform their synchronization exchanges. This sharing of a resource to perform a throughput-critical function raises the question of contention-induced latency which may degrade throughput. The overhead involved in executing the synchronization transmissions in the proposed architecture is therefore calculated using a Monte Carlo simulation. This simulation provides an estimate of the average delay incurred by a synchronization message as a function of the frequency with which the redundant sites perform their synchronizing acts and the number of redundant sites which must share a set of Network Elements. The message delay calculations indicate that the overhead involved in the fault tolerance-specific synchronization message transmissions is on the order of a few percent of the processor's raw throughput.

The thesis terminates with conclusions and recommendations for future work.

CHAPTER 2
THE COMPUTATIONAL PROBLEM

2.1 Description of the Computational Problem

The approach to be taken in determining the overall functional characteristics of an appropriate architecture will be to select a particular application which is representative of a large class of computationally intensive, parallelizable problems. The characteristics of this particular example will be extracted and used to help in pinpointing major features of the architecture. It is necessary to ground the architecture generation process in the context of a particular application to help resolve ambiguities, demonstrate utility, and smoke out otherwise hidden requirements which could have major impact on the architecture. This is not to imply that the result of this process is necessarily an architecture that is useful for only a small class of problems; the architectural concepts so derived should be general enough to be applicable to a large class of problems, which of course includes the original point of reference. Indeed earlier fault tolerant computer architectures (i.e., FTP), which were initially designed for flight control applications, have demonstrated broad applicability in areas such as nuclear power plant trip monitoring (Argonne National Laboratory FTP), turbine control (Woodward Governor FTP), circuit-switched network control (CSDL IR&D FTP, Dryden FTP), and general purpose computation (IR&D FTP, AIPS FTP). A similar architecture has been commercially implemented for process control applications (August Systems CS3000). It is intended that the architecture described in this thesis will be similarly applicable to a large class of computational problems.

For the application of interest, we have selected the computational problem that an autonomous or semi-autonomous system faces when it must plan its own actions in the face of a changing internal and external environment. To this end, this section presents a generalized description of a methodology under development at Draper to solve mission and trajectory planning problems for autonomous or semi-autonomous systems. A recent

summary of this work is presented in [Adams86]. We have selected a computationally-intensive algorithm which is central to this methodology, cast this algorithm into a form which exposes its potential for parallelism, and used the features of this representative parallel algorithm to motivate the design of a computer which will be useful in solving problems of this class. In addition, this algorithm will serve as a reference point when application-specific information is needed for analytical reasons.

Generally speaking, the planning problem is to determine adaptively and in real-time the ordered sequence of actions required to best achieve stated objectives, while remaining within operational constraints and taking into account the system's current estimate of the state of its world [Adams86]. Planning algorithms can be viewed as constrained statistical optimization problems which are carried out over a very large search space. The optimization and constraints can in general be very complex, including energy consumption, threats, scheduling constraints, system failure status, and other such parameters.

The problem can also in some cases correspond to a search of a tree structure. Many papers have appeared on the parallelization of such algorithms ([Karnin84], [Mohan83], [Deo80], [Wah84]) and on the design of specialized computer architectures to carry out these algorithms efficiently ([[Browning80], [El-Dessouki80], [Sherron82]).

For example, [El-Dessouki80] discusses a method by which the work of evaluating such a tree structure can be divided among a number of Processing Elements (PEs). Essentially, the method assigns each of a number of PEs a disjoint branch of a search tree, which each PE then searches in a depth-first branch-and-bound algorithm, independently of the activities of the other processors of the ensemble. The simulation of [El-Dessouki80] showed that a speedup proportional to the number of PEs was possible when searching trees of large size. The overhead involved in concurrency management was estimated to be equal to the throughput of one PE, and inter-PE communication was reduced to one interchange every time a solution was found by one of the PEs. This optimistic result can

be traced to the assumed disjoint nature of the search space.

Searching of a tree structure is a part of the mission and trajectory planning problem, but the latter is much more complicated, in that at one level of the planning methodology significant parallelism exists in that it is possible to search many trees in parallel. This point will be expanded upon later.

The approach to solving the planning problem presented in [Adams86] will now be excerpted. Define a *plan* P to be an ordered sequence of actions A_i, each with a relative worth v_i,

$$P=\{A_1,A_2,...A_N\}.$$

The *expected utility* of a plan P is

$$U_P=\sum_{A_i \in P} \text{Prob(success of } A_i) \; v_i \; PF ,$$

where Prob(success of A_i) is the probability that there is sufficient energy, vehicle capability, and time to perform action A_i, and PF is a penalty function that acts to reduce the utility of a plan when any constraint is violated. In the particular instance addressed in [Adams86], an action A_i is defined to be the reaching of a *goal*, a particular geographic location which it is desired to visit, possibly before, after, or during a given time window.

It is desired to generate a plan P, i.e., an ordered sequence of goal visitations, that maximizes this expected utility. In [Adams86] the maximization process is divided into a hierarchical set of levels, each with a different problem abstraction, planning horizon, and planning methodology. These levels are called the *goalpoint* , *waypoint* , *modal*, *trajectory*, and *planning executive* levels. The goalpoint planner's purpose is to generate the ordered sequence of goals and their geographic locations. It uses a heuristically-guided simulated annealing algorithm to determine this sequence. The goalpoint planner utilizes the waypoint planner to generate maximum survivability routes between selected goals. The goalpoint planner calls the waypoint planner with a pair of goals and the waypoint planner, using a heuristically-guided A* search, provides the desired path along with the needed cost

parameters for that path. The modal planner provides an ordered sequence of subsystem actions and associated flight modes to the waypoint planner, using a hybrid system of backwards chaining and a blackboard system. The trajectory planner generates a time history of vehicle position and a detailed plan of subsystem actions using a combination A* search and scripts. Finally, the planning executive coordinates, focuses, monitors, and prioritizes activities at each of these levels using a set of interacting rule-based systems.

Since this set of interacting functions is extremely complex, a salient computationally-intensive phase will be extracted, simplified, and used for the present purpose. Specifically, consider the problem of the goalpoint planner. It is given a set of goals which it is desired to visit. In the applications currently under investigation, N, the number of goals, is between ten and one hundred. The goalpoint planner must determine from the O(N!) possible tours through the goals a sequence of visitations which provides a near-optimal cost functional. It is computationally infeasible to exhaustively evaluate all these paths for a moderately sized problem, so the goalpoint planner first uses heuristic screening of goals to reduce the number of goals to be considered in the problem. Goals whose visitation violates survivability, energy, or scheduling constraints are removed from the problem. With the possibly reduced set of goals, the goalpoint planner then enters a perturbation loop in which it perturbs an existing plan using a few well-defined rules, evaluates the consequences of the perturbation, and either accepts or rejects the resulting perturbed plan based on a simulated annealing schedule. This process is repeated until a termination condition is met.

A plan perturbation consists of one of three possible rearrangements of goals in the plan. A goal may be *added* when sufficient energy, time, and survivability margin exist. A goal may be *removed* when the energy, time, and survivability parameters of the plan are unacceptable. Finally, goals may be *reordered* when the energy, time, and survivability parameters are acceptable but uncomfortably stressed. The type of rearrangement to be performed on a given iteration is probabilistically chosen using a distribution which is

scheduled in accordance with the planning situation.

Given that the goalpoint planner has determined the perturbation class, the appropriate perturbations must be performed and the resulting plans analyzed. In a problem having N goals, there are O(N) possible goal insertions and deletions. The goal reordering follows a "2-opt" algorithm [Lin65], in that perturbations consist of swapping the positions of pairs of goals in the plan. For example, suppose that a plan consists of the goal sequence $A_1A_8A_5A_4A_3A_9A_7A_2A_6A_{10}$ and it is desired to perform a 2-opt permutation of all pairs of goals. One of these permutations is the swapping of the goals 4 and 3, such that the permuted plan is $A_1A_8A_5A_3A_4A_9A_7A_2A_6A_{10}$ (Figure 2.1). There are $O(N^2)$ such permutations.

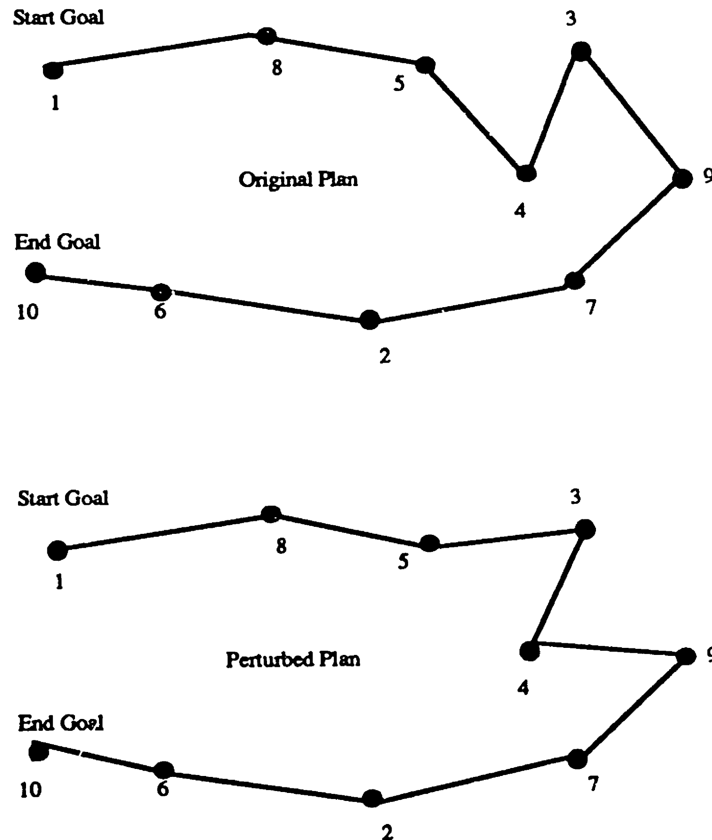


Figure 2.1

Perturbation of a Plan

Assuming that a perturbation class has been selected, the goalpoint planner determines the energy consumption increment or decrement involved in implementing each possible perturbation of that class. Even though this may take $O(N^2)$ work in the case of goal reordering, the energy calculation requires minimal computation. The goalpoint planner then sorts the perturbed plans based on energy, and rejects out of hand those requiring more than a threshold energy. Again, sorting of the possible moves is not very computationally intensive. A plan is selected from those falling below the energy threshold and is a candidate for more involved utility evaluation. Following this utility evaluation, which requires Monte Carlo estimation of Prob(success of A_i) for all A_i in the plan, the perturbed plan is either accepted as the new plan for the next iteration of the algorithm, or rejected and the old plan retained. Perturbed plans are accepted or rejected probabilistically according to a simulated annealing algorithm, in which plans exhibiting decreases in utility have a higher probability of being accepted early on in the planning process in an attempt to avoid local minima in the utility manifold, while as the search progresses this probability is gradually decreased on the grounds that a global maximum in utility is being approached and it is desirable to not stray too far from it.

This completes the informal description of the hierarchical mission and trajectory planning methodology. We now discuss the inherent parallelism in at least one phase of this methodology.

2.2 One Parallelization of the Algorithm

The methodology currently under development is designed to operate on a single processor. Therefore the planning process is serial in nature, in that the goalpoint planner generates, selects, and analyzes one plan perturbation at a time. However, the plans resulting from the many possible perturbations could be analyzed in parallel in an appropriate computational medium. As mentioned above, in the case of goal insertion or removal there are $O(N)$ possible perturbations, and in the case of 2-opt goal reordering

there are $O(N^2)$ possible perturbations. Given that these are uncoupled, independent analyses we see that significant potential parallelism exists in the perturbation analysis.

When the goalpoint planner requires the utility increment or decrement corresponding to a plan perturbation, it passes the goals for which the perturbation has changed the ordering to the waypoint planner which calculates a minimum lethality path between the goals subject to the constraints on energy, time, and velocity. The waypoint planner calculates this path using an A* search of the geographical grid underlying the possible paths between the two goals. Figure 2.2 illustrates the interaction between the goalpoint planner and the waypoint planner.

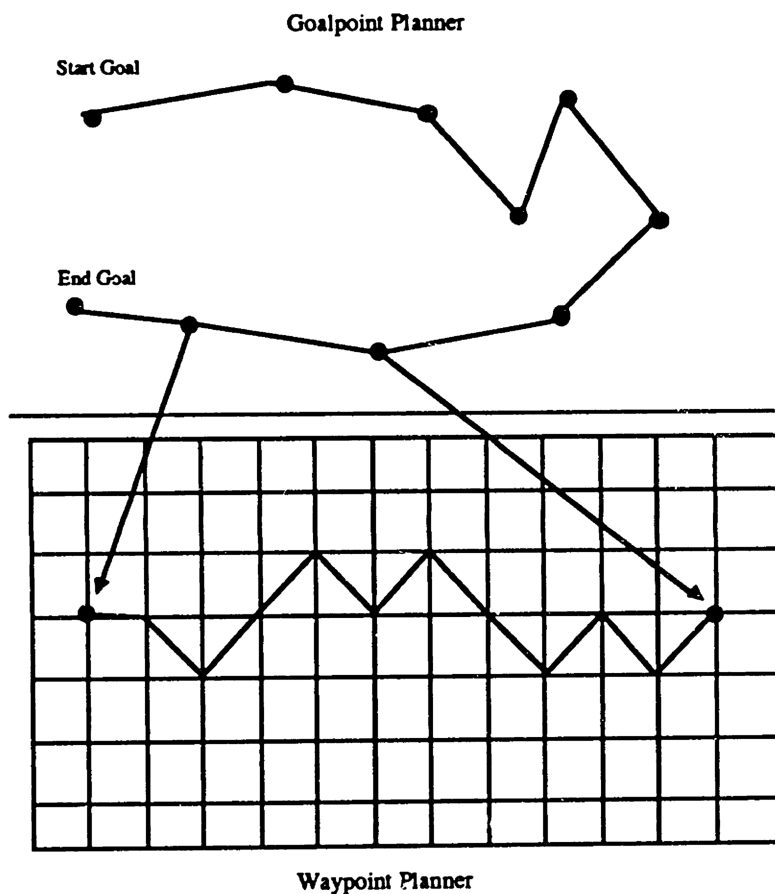


Figure 2.2

Interaction of Goalpoint Planner and Waypoint Planner

Figure 2.3 shows an example of an A* search of a terrain database containing a start goal, A, and a terminal goal, B, and an obstacle. It is the significant number of searches of this type which compose the parallelizable activity which the proposed architecture will be capable of supporting. According to [Cervantes86], each of these searches takes approximately one to two seconds on a 7 MIPS Symbolics 3670 Lisp Machine. As mentioned before, on any given iteration, between O(N) and O(N^2) such searches can be performed. Therefore we will modify the existing planning methodology such that all these searches are carried out in parallel. Figure 2.4 shows the task flow diagram for the parallelized goalpoint planner. Such a synchronous iterative parallel algorithm can be expected to exhibit significant speedup if the time taken by the parallel activity is significantly longer than the time taken by the computation required to generate the goal pairs, sort the results, and select the new plan. Thus this parallelization is subject to Amdahl's observation that the amount of inherently serial activity in a parallelizable algorithm ultimately limits the obtainable speedup. We will note this important observation in passing, and then carry on with the determination of an architecture capable of executing these searches in parallel.

To summarize the parallelization strategy, the analysis of the many possible perturbations represents a significant part of the computing load of the planning problem. It is this computation which will be accelerated by the proposed architecture by taking advantage of the high degree of coarse granularity parallelism which is inherent in the many independent analyses of the possible perturbations, i.e., the existence of tens to hundreds of A* searches of a terrain database. It appears that the analysis of the many possible candidate plans represents a set of loosely coupled complex tasks which can be efficiently executed in a parallel manner. It should be stressed that this example of the parallel activity which is available in the mission and trajectory planning problem is presented mainly to point out that enough parallel activity exists in problems of this type to justify parallelization efforts and the concomitant investigation of architectures which can efficiently exploit such

parallelism, as well as to serve as a reference point when one is needed. Other parallelization schemes exist, but are beyond the scope of the present discussion.

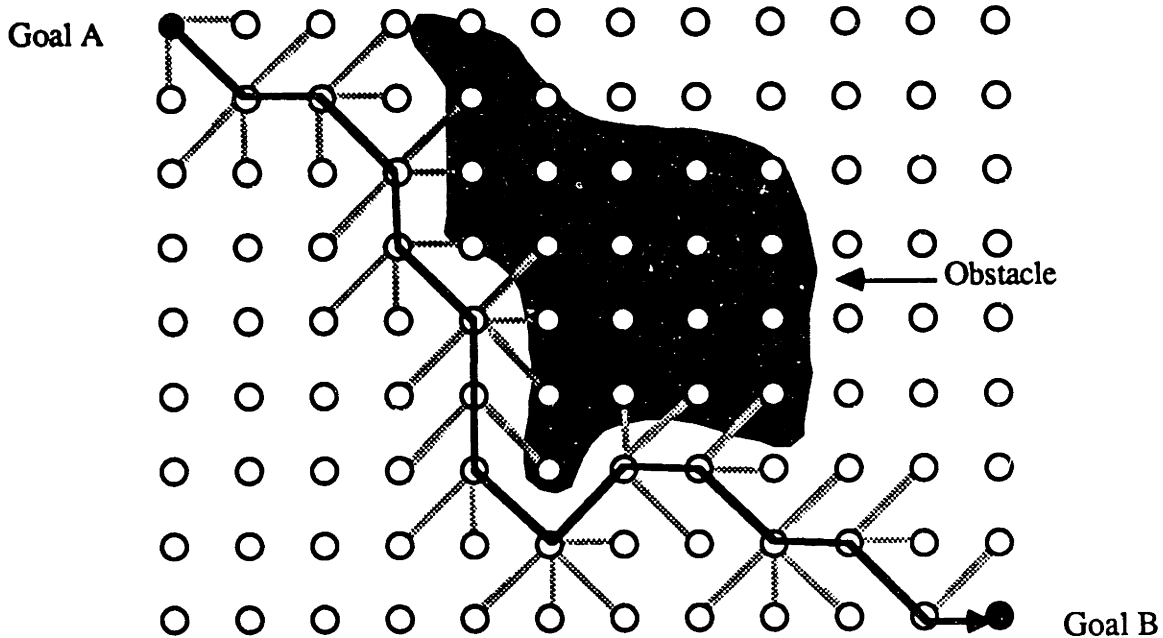


Figure 2.3

Example A* Search of Terrain Database by Waypoint Planner

In consonance with the desire that the architecture be useful for a wide range of problems of which the example application is representative, one of the goals of the architecture's design is that it should allow the sequence to consist of arbitrary entities, such as a series of cities in a traveling salesman problem, a sequence of goals in a route planning problem, a sequence of discrete actions in a control problem, or a resource assignment sequence in a resource allocation problem. The cost metric is likewise not to be constrained by the architecture. It can include time considerations, difficulty considerations, threats, resource contention levels, scheduling conflicts, and in fact need not even be a scalar quantity.

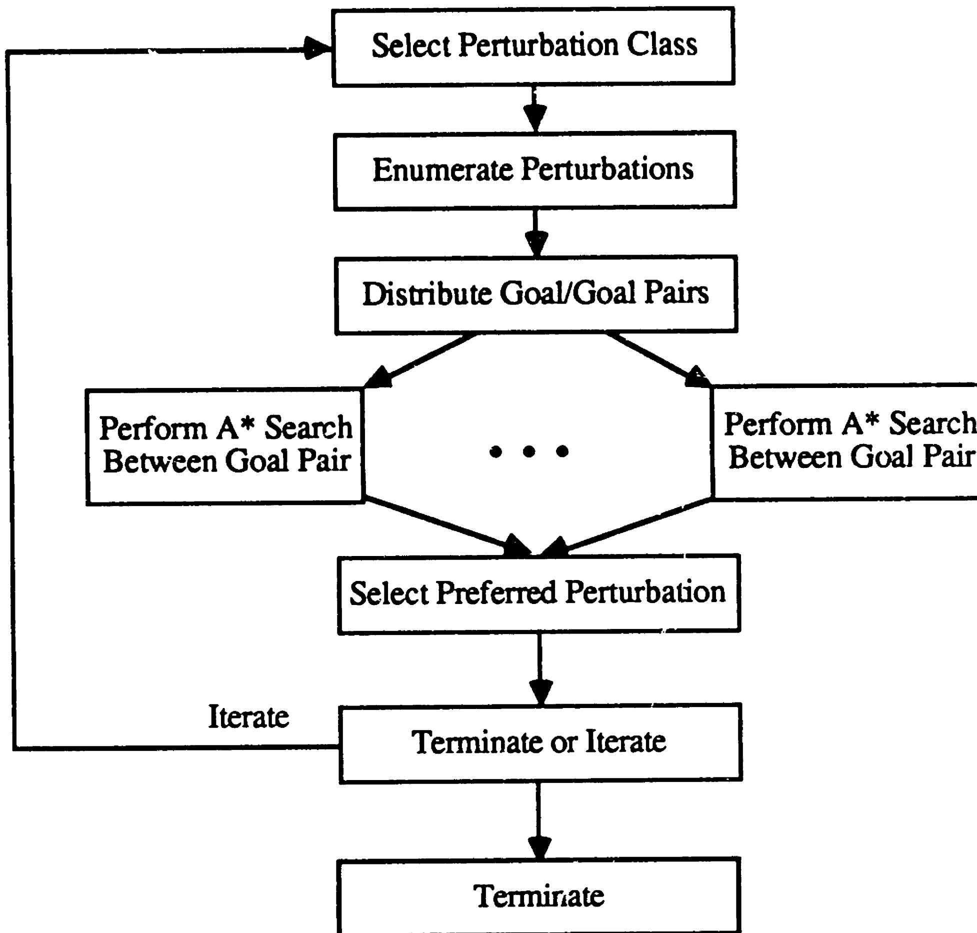


Figure 2.4

Task Flow Diagram for Simple Parallelized Goalpoint Planner

The above approach represents a fairly general view of and partitioning method for the planning problem. Solutions to such problems can be implemented on fairly conventional and realizable multicomputer architectures which are not so esoteric as to be difficult to program, (ILLIAC, MPP), special purpose (signal processors, systolic arrays), or of an extremely radical design (neural nets, Boltzmann machines).

2.3 Architectural Implications

These algorithmic characteristics influence the architecture of the computer system in several ways.

Observation 1: The salient characteristic of the hypothetical parallel algorithm is the existence of a moderate degree of inherent parallelism, i.e., the inter-goal waypoint searches of the perturbed plans, which the architecture must have the capability of efficiently exploiting. For example, for a problem having ten to one hundred goals, the number of potentially simultaneous A* searches ranges from tens in the case of goal additions/removals to thousands in the case of 2-opt goal reordering.

Observation 2: The parallelizable activity consists of arbitrarily structured code having a bounded execution time. In other words, we do not know *a priori* what exact form the A* search will take. This implies that, however this potential parallelism is exploited, the entities that actually do this analysis must have the capability of executing such unstructured code sequences. Therefore the parallel functional modules should be general purpose computational devices. This is in contrast to signal processing or similar highly structured problems in which functional modules perform highly specialized operations such as FFT butterflies, and which can therefore be hardwired to optimize execution speed.

Observation 3: Internal to each A* search exists significant additional parallelism which has a finer granularity than the search itself. This parallelism may be exploited via appropriate Processing Element design and using parallel tree-searching techniques described in, for example, [Karnin84], [Mohan83], [Deo80], [Wah84].

Observation 4: A desirable goal is to match the degree of parallelism which the architecture is capable of supporting with that of the algorithms which will run on the machine. For example, the algorithm may consist of N pieces of code that can be executed in parallel. A large disparity between this degree of parallelism and that supported by the architecture would result in either idle processors on one hand or inadequate exploitation of the algorithm's intrinsic parallelism on the other hand. Due to the multitasking capability of most available PEs, it is better to have more parallel tasks than PEs. This would allow the PEs to context switch to a ready task when a running task invokes a long-latency transaction or blocks for some purpose.

Observation 5: According to the current work at Draper, the memory requirements for the evaluation of a plan are not too severe; the code is under one thousand lines of Lisp, while the data is on the order of 50-100KWords [Cervantes86]. Therefore it is plausible that each PE can possess not only the code required to perform the perturbation analyses, but the data as well. The implication of this observation is that the parallel activity is decoupled enough that a tightly coupled shared memory implementation is unnecessary and a message-passing implementation would suffice.

The above considerations imply that the architecture must be able to perform N arbitrarily complicated and loosely coupled operations in parallel. An architecture capable of supporting these needs is a Multiple Instruction stream Multiple Data stream (MIMD) computer [Flynn66]. Such an architecture achieves parallelism by the use of many highly integrated, general purpose processors with onboard memory, communications ports, ROM, timers, and other features. Each such Processing Element (PE) is capable of executing its own program independently of the activities of the other PEs, and each PE may operate on different copies of data. The PE aggregate, often called an *ensemble*, is interconnected by some means such as a network or a bus. Figure 2.5 shows a simplified view of such an architecture.

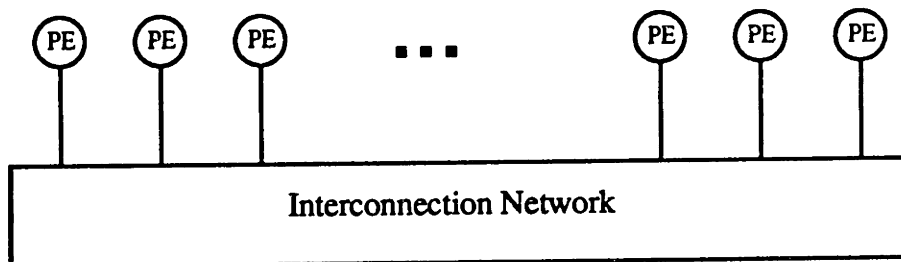


Figure 2.5

Simplified Architecture of a MIMD System

CHAPTER 3
RELIABILITY ANALYSIS PRELIMINARIES

3.1 Reliability Requirements

The target application of the architecture under study includes autonomous and semi-autonomous systems. In such applications, the results of the planning process are critical to the survival of the system. For example, an autonomous vehicle which is planning its own activities must correctly take into account fuel constraints, otherwise the vehicle may be unable to return to its host vehicle due to inadequate fuel reserves. Therefore the confidence in the computational result of the planning process is in many cases an upper bound on the probability that the vehicle successfully completes its mission, assuming that part of its mission specification is to return to its home base.

Reliability requirements for short-term missions have been well defined. They tend to be expressed in terms of the probability of failure per hour, or the integrated probability of failure over a given mission time. Therefore the short-term reliability of the computer which executes the planning problem described above will be of interest, and effort will be expended to maximize this quantity. This short-term reliability target is displayed in Figure 1.2.

An intriguing potential of parallel systems is the capability of operating in a variety of modes depending on the immediate needs of the mission. For example, to achieve high reliability over the short term, a fault masking mode could be entered in which the processing elements are triplicated and results are voted to mask arbitrary faults, assuming that the appropriate architectural mechanisms are in place to support this computational mode. To achieve maximal throughput with loss of the guarantee of the integrity of the computational result, the triplex groups could be disbanded into three times as many simplex processors. It must be stressed that the capability to achieve this flexibility must be designed into the system, as opposed to being a happy coincidence which can occur in any

parallel system.

It is possible that a parallel system meeting the above short-term reliability goal may also meet a long-term longevity goal. There are several applications in which it is desired to reliably execute a form of the planning problem under a longevity specification. One example is that of a space platform, where there is a specification that the system have a probability of failure no greater than 0.05 at the end of a 1-year mission [Lala86c]. At the same time, such a platform may be called upon at any time in its mission to have the capability to execute a computation which is highly resilient with respect to hardware faults. Therefore, in addition to the study of ways to fulfill the short-term reliability specification, we will also study the impact of various fault tolerance schemes on the longevity of the system.

Finally, the superfluity of redundancy of parallel systems leads to the expectation that such systems would exhibit graceful degradation as the system's components fail. It is of interest to quantify this property of graceful degradation, so this thesis will compute the expected value of the number of processing elements available for computation in a parallel ensemble as failures occur, as a function of time and hence the number of failures.

3.2 Approach and assumptions

The goal of the next four chapters is to quantitatively examine the applicability of different fault tolerance schemes to a parallel processor ensemble. The results of the analysis will be used to generate and compare figures of merit for each approach. Once a suitable approach has been identified, further chapters will be devoted to finding an appropriate implementation.

The system under consideration will be referred to as a *processor ensemble*, an aggregate consisting of processing elements interconnected via a communications network. An example of such an ensemble is the JPL hypercube series of experimental parallel processors [Seitz85]. The architectural concept has been commercially implemented by

Intel, Ametek, Ncube, and others.

Each processing element consists of a general-purpose programmable CPU, RAM, ROM, onboard timers, I/O, and interprocessor communications ports (Figure 3.1).

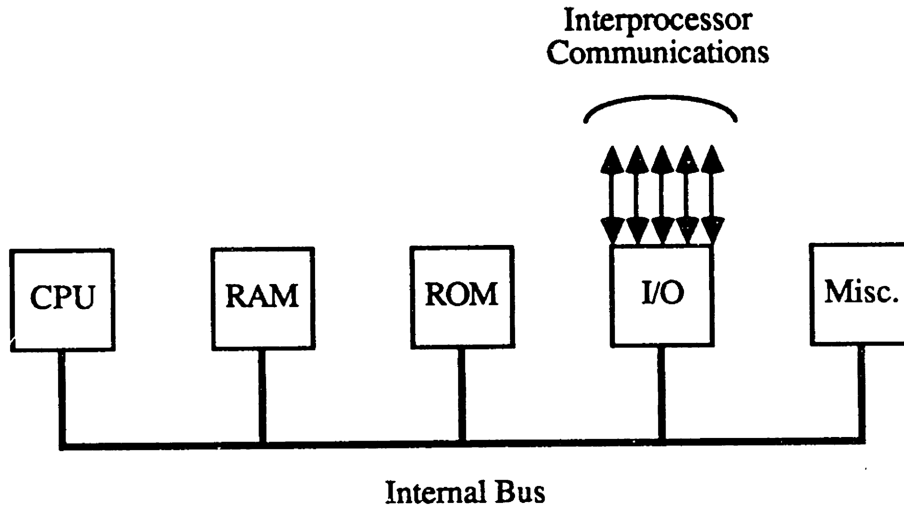


Figure 3.1

Processing Element Architecture

The processor ensemble is assumed to be executing a single parallelized computation, such as the mission planning algorithm discussed earlier. Since the ensemble is operated as an embedded computer in an autonomous or semi-autonomous system, the reliability calculations will be performed assuming that no repair of failures is possible. The overall analytical approach will be to calculate the state occupancy probabilities for a given processing site or a group of processing sites using a continuous-time Markov model. Generally, numerical results will be calculated using discrete-time integration of the matrix differential equation representing the temporal evolution of the state occupancy probabilities. Successive squaring of the single-step transition matrix is used to obtain solutions with reasonable computational effort. In some cases approximations will be formulated and used. The resultant expressions for the temporal variation of state occupancy probabilities will be combinatorially aggregated to yield the system performance parameters of interest. Specifically, these are the probability of system loss due to

uncovered processor failure or attrition, the Mean Time To System Failure (MTTSF), and the expected value of the number of non-faulty processing elements.

The Mean Time to System Failure figure of merit will be primarily of use in comparing the resilience of the different architectural approaches to attrition, i.e., the exhaustion of the ensemble's computing resources as a result of failures. In a real application it is more likely that the upper limit of infinity in the formulation for MTTSF will be replaced by the actual mission time to reflect the fact that the reliability of the aggregate is unimportant beyond this mission time. However, the MTTSF will be used in this analysis because it is more general.

To facilitate the analysis, additional assumptions will be made about the nature of the processor ensemble.

1. The ensemble's throughput, represented by the number of processing sites, is assumed to be specified at some minimum value. The analysis below will use the specification that 64 nonfaulty processing sites are available at the start of the mission.

2. Component failures, such as processor failures, are assumed to be independent, with constant failure rate, typically denoted by λ. In the cases where reconfigurations can occur, reconfiguration rates, typically denoted by μ, are also assumed to be independent and constant. Note that the mean time to achieve reconfiguration $1/\mu$ includes the fault latency time, which for analytical tractability is assumed to be exponentially distributed [Shin84].

3. Interprocessor communication link failures will be ignored. Instead of considering such failures explicitly, the complexity of supporting interprocessor communication will be modeled by increasing the failure rate of the hardware component responsible for interprocessor communication by a constant fraction f_{port} for each interprocessor port hosted by the component. Thus, in the case where the processor itself hosts the interprocessor communications hardware, the processor failure rate, λ_p, is $\lambda_p = \lambda_{p0}(1+\#ports * f_{port})$, where λ_{p0} is the "core" failure rate of a processing element. The

net result of this approximation is to upper-bound system failure probability by assuming that a processor is failed when one or more of its ports is failed.

4. The processor failure rate will be assumed to include transient, intermittent, and permanent failures. Each failure class may include inert, active, or Byzantine failure modes. These failure modes will all be defined shortly.

5. Processors exist and function only in the context of the ensemble. Thus system loss must be defined in terms of the effect of processor faults upon the capability of the ensemble to meet its specifications. Two distinct ensemble failure modes can be identified, in general. System loss of the first kind is defined to occur when no processors are available because all have suffered covered failures, i.e., all processors that have failed have been safely shut down. This failure mode will be denoted "system loss due to attrition". In an application-specific situation this failure mode could be redefined to be the existence of fewer than a minimum complement of usable processors. System loss of the second kind occurs when a processor or other component failure corrupts the ensemble's computational result by communicating erroneous information to the remainder of the system. This event can occur when any processor enters an uncovered failure mode, and will be denoted "system loss due to uncovered processor failure."

The models presented below give the probability that the processor fault goes undetected by both the processor and the remainder of the ensemble. In such a case, information from the processor propagates into the ensemble, possibly corrupting the computational result. However, the effect of this erroneous information on overall system reliability is not immediately apparent. Undetected processor faults do not necessarily result in immediate system corruption because algorithmic robustness may exist in the parallelized algorithm under execution. For example, the corrupted information from the faulty processor may or may not be used in the final result, so system integrity may or may not be affected.

Some feel for this effect may be obtained by calculation of the probability that system

integrity is corrupted given that a processor has failed uncovered. For such a calculation to be possible, an algorithmic structure must be hypothesized. The overall structure of the mission planning algorithm described in Chapter 2 will be used. Specifically, assume a synchronous iterative parallel algorithm whose structure is given in Figure 3.2.

A control task t_c dispatches one worker task t_{wi} to each of the N processors p_i of the ensemble, where $1 \leq i \leq N$. After the worker tasks are completed the worker processors transmit the results back to the control task. The control task then makes some decision based on the results of the worker tasks and starts another iteration. In particular, it is assumed that the control task selects <u>one</u> of the workers' results to form the basis of the next iteration. If the selected result happens to emanate from a faulty processor and the processor fault is uncovered, then system corruption and hence system loss will occur.

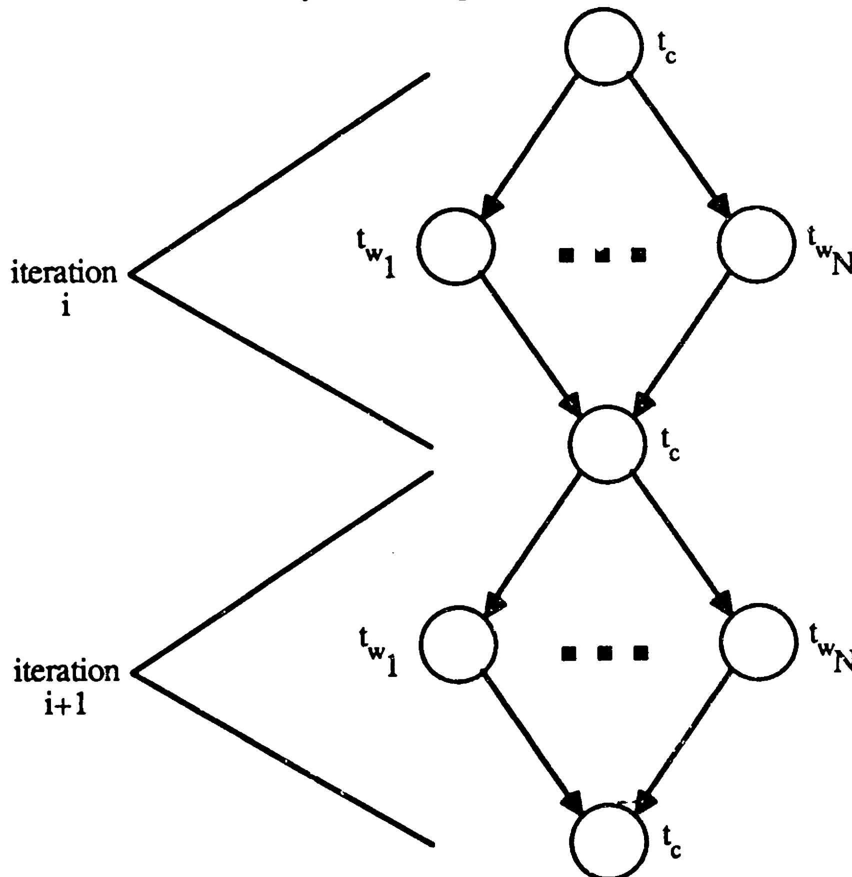


Figure 3.2

Overall Algorithmic Structure

Assume that the result of each worker task is equally likely to be selected by the control task at the end of an iteration. The probability that the result produced by p_i is consumed by the control task on a given iteration is 1/N, and the probability that the result is not consumed on a given iteration is 1-1/N. The control task is assumed to cover the fault with probability c. Assume that a permanent failure of p_i occurs during iteration 1. It is desired to compute the probability that the erroneous result of t_{wi} is selected and not covered on iteration M. This is the probability that the fault has gone undetected up to and including iteration M-1, times the probability that the faulty result is selected and uncovered on iteration M. The probability that the fault has gone undetected up to and including iteration M-1 is the probability that the result emanating from the faulty processor has not been selected on any of the M-1 iterations,

prob(fault undetected up to and including iteration M-1 | p_i failed during iterations 1)

=prob(t_{wi} not selected on any of the M-1 iterations)

=(prob(t_{wi} not selected on an iteration))$^{M-1}$

=(1-prob(t_{wi} selected on an iteration))$^{M-1}$

=$(1-1/N)^{M-1}$.

Processor fault coverage does not enter into the calculation yet because if the faulty result were selected, then either it would be undetected, resulting in system loss prior to iteration M, or it would be detected. By construction, neither event is assumed to occur on any of the M-1 iterations.

The probability that the system is corrupted due to selection of the result of the worker task running on p_i on iteration M is 1/N, the probability that the result t_{wi} is selected, times 1-c, the lack of coverage of the processor fault. Therefore the probability that the system is corrupted by selection of a result emanating from a faulty processor on iteration M given that the fault occurred on iteration 1 and remained undetected on iterations 1 through M-1 is

prob(result t_{wi} selected on iteration M | p_i failed during iteration 1 and t_{wi} unselected

on iterations 1 through M-1)

$=(1-1/N)^{M-1}(1-c)/N$.

The probability that the erroneous result has been selected by iteration I is the discrete integral of the above density function over iterations 1 to I,

prob(result t_{wi} selected by iteration I and the fault uncovered | p_i failed during iteration 1)

$$=\sum_{M=1}^{I}\left(1-\frac{1}{N}\right)^{M-1}\frac{1-c}{N}.$$

Figure 3.3 shows that, for 64 PEs, this probability approaches a steady-state value approximately equal to 1-c in a few hundred iterations. Therefore in the sequel we will assume with some basis in reality that algorithmic robustness does not increase the coverage c by any appreciable amount, and uncovered processor failures result in ensemble loss due to eventual corruption of the computational result.

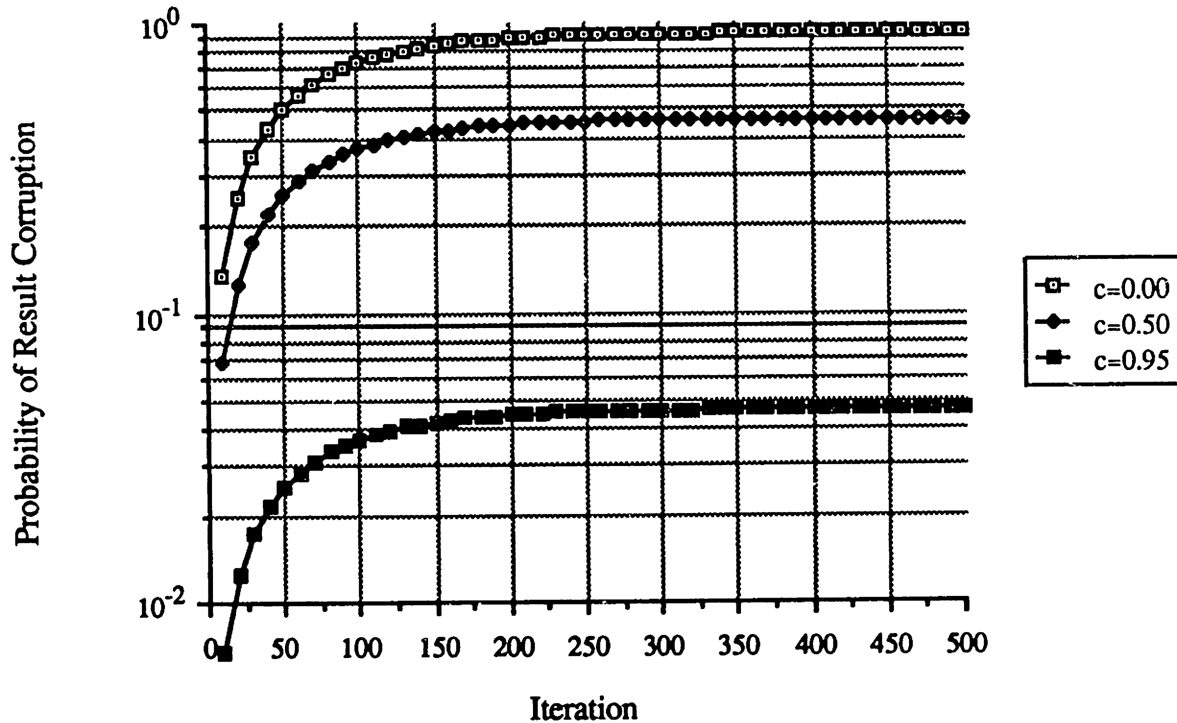


Figure 3.3

Probability of Result Corruption versus Iteration for

Synchronous Iterative Algorithm

In such algorithms there is a third failure mode: that of the processor hosting the central control task t_c. If p_{uf} is the probability that the central processor suffers an uncovered failure, then the system loss probability is at least as large as p_{uf}. Since would probably use a highly reliable central controller in this application, this probability can be neglected compared to the probability of result corruption.

It is also of interest to compute the attrition resiliency of a parallel ensemble as a function of the number of PEs in the ensemble. Intuitively, it would seem that as the number of PEs is increased, attrition could be staved off indefinitely. A simple calculation shows this not to be the case. Assume an ensemble having N PEs, unity fault coverage, and constant failure rate for each PE of λ. Define system loss due to attrition to occur when there are no operational PEs left in the ensemble. Thus the probability of system loss due to attrition $p_{SL/E(ATTR)}$ is

$$p_{SL/E(ATTR)}(t) = (1-e^{-\lambda t})^N .$$

The reliability of the ensemble is then

$$R_E(t) = 1 - p_{SL/E(ATTR)}(t) = 1 - (1-e^{-\lambda t})^N$$

and the Mean Time to System Failure MTTSF is

$$
\begin{aligned}
MTTSF &= \int_0^{\infty} R_E(t)\,dt = \int_0^{\infty} 1 - (1-e^{-\lambda t})^N\,dt \\
&= \int_0^{\infty} dt - \int_0^{\infty} (1-e^{-\lambda t})^N dt \\
&= \int_0^{\infty} dt - \int_0^{\infty} (1-e^{-\lambda t})(1-e^{-\lambda t})^{N-1} dt \\
&= \int_0^{\infty} dt + \int_0^{\infty} e^{-\lambda t}(1-e^{-\lambda t})^{N-1} dt - \int_0^{\infty} (1-e^{-\lambda t})^{N-1} dt \\
&= \int_0^{\infty} dt + \frac{1}{N\lambda}(1-e^{-\lambda t})\Big|_0^{\infty} - \int_0^{\infty} (1-e^{-\lambda t})^{N-1} dt \\
&= \int_0^{\infty} dt + \frac{1}{N\lambda} + \int_0^{\infty} e^{-\lambda t}(1-e^{-\lambda t})^{N-2} dt - \int_0^{\infty} (1-e^{-\lambda t})^{N-2} dt \\
&= \int_0^{\infty} dt + \frac{1}{N\lambda} + \frac{1}{(N-1)\lambda} - \int_0^{\infty} (1-e^{-\lambda t})^{N-2} dt \\
&= \int_0^{\infty} dt + \frac{1}{N\lambda} + \frac{1}{(N-1)\lambda} + \ldots + \frac{1}{2\lambda} - \int_0^{\infty} (1-e^{-\lambda t}) dt \\
&= \int_0^{\infty} dt + \frac{1}{N\lambda} + \frac{1}{(N-1)\lambda} + \ldots + \frac{1}{2\lambda} - \int_0^{\infty} dt + \frac{1}{\lambda} \\
&= \frac{1}{\lambda}\sum_{i=1}^{N} \frac{1}{i} \\
&= \frac{1}{\lambda}\, O(\lg N).
\end{aligned}
$$

This function is plotted versus N in Figure 3.4 for $\lambda = 10^{-4}$/hour.

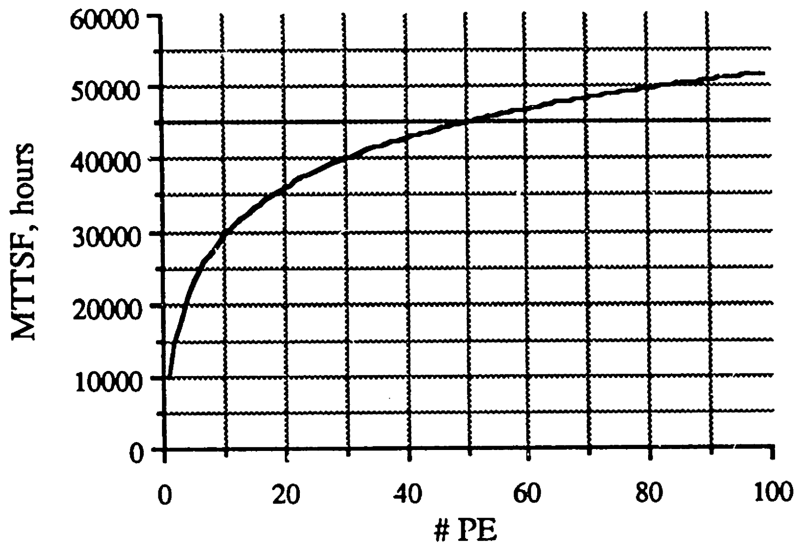


Figure 3.4

MTTSF vs. Number of PEs in the Ensemble

This calculation shows that the MTTSF measure of attrition resiliency increases logarithmically with the number of PEs in the ensemble, with the attrition benefit of adding the ith PE to the ensemble being only 1/i. From the inverse linear dependence on λ it is equally apparent that minimization of the failure rate of the processors has a stronger effect on MTTSF, and therefore effort should be expended in achieving this minimization.

6. The processors are assumed interconnected in a topology in which the number of ports on a processor scales logarithmically with the number of processors in the ensemble. In the ensemble under analysis, the number of ports per processing site is lg64=6. An example of such a topology is the binary hypercube topology. This common processor interconnection topology will be used as a baseline in future calculations.

Several parameters are of interest to the designers of systems into which such an ensemble of processors may be embedded. Table 3.1 lists these parameters in the most general case.

- # PE
- #ports
- system loss probability
    - 10 hours
    - 100 hours
    - 1000 hours
    - 10000 hours
- $E_{PE}(t)$
    - 10 hours
    - 100 hours
    - 1000 hours
    - 10000 hours
- mean time to system failure
- intra-group communication
    - distance
    - diagnosability
    - anisotropy
    - inter-group interactions
- inter-group communication
    - distance
    - diagnosability
    - anisotropy
- transparency of FT technique to programmer
- efficiency of FT technique

Table 3.1

General Performance Parameters

The entry "# PE" represents the number of processing elements required to achieve a usable throughput of 64 processing sites at the beginning of the mission. For example, a fault tolerance scheme requiring processor duplication would superficially require 128 processing elements to achieve a usable throughput of 64 processing elements. (Inefficiency factors combine to reduce the effective throughput of duplicated processors, resulting in the need for more than 64 duplexes to obtain the specified throughput.)

The entry "# ports" indicates the total number of interconnection ports required by the particular architecture, and includes not only inter-processing site ports but also ports between redundant processors forming a processing site. The minimization of this parameter is very important from the point of view of system implementation.

The entries listed under "system loss probability" are the probability that the system has

failed by time T given that it was operational at time zero. Entries are given for 10, 100, 1,000, and 10,000 hours. Following the system loss probabilities is the "Mean Time To System Failure", or MTTSF, of the ensemble. Recall that this parameter is of use for comparison only and that in an actual implementation one would be more interested in the integrated reliability over the actual mission time.

Under the entry "$E_{PE}(t)$" reside the expected values of the number of usable processing sites versus mission time. Again, entries are given for 10, 100, 1,000, and 10,000 hours. Another form of this parameter, possibly of more interest in a particular application, would be the probability that there are more than "n" processing sites at a given time. We will stay with the former parameter because it is more general and easier to calculate.

Many fault tolerance schemes require replication of the processing elements. In this thesis, the term *group* will be used to denote a redundant processing site composed of two or more processing elements. Since a group member must constantly communicate with other members of its group, it is logical to study the parameters of intra-group communication. The parameter "max distance" is defined as the maximum of the minimum number of communications links which must be traversed by an intra-group message. "Diagnosability" is a qualitative indication of the ease of diagnosis of a group member by other members of the same group. "Anisotropy" is defined as the difference between the maximum and the minimum distance between any two group members. "Inter-group interactions" denotes how failures of members of one group may affect the behavior of another. A case in which this might be expected is when groups use other group members to relay intra-group messages.

Groups must communicate with each other because of the holism of the single parallel algorithm under execution by the ensemble. Therefore "inter-group communication" is an important performance parameter. The definitions of the terms "distance", "diagnosability", and "anisotropy", when applied to inter-group communications, are identical to those given above.

Finally, the "transparency of the FT technique to programmer" is a qualitative measure of the degree to which the incorporation of a given fault tolerance technique into a parallel processor ensemble exacerbates the task of programming the ensemble. This parameter is of extreme importance when it comes to use of a system. Programming fault tolerant systems which use inadequately transparent paradigms can be intractable. Similarly, the "efficiency of FT technique" is a measure of the cost of implementing the particular form of fault tolerance. It includes the cost of hardware redundancy, software and temporal redundancy, software execution overhead, etc.

3.3 Fault Taxonomies

This thesis will be mainly concerned with techniques for tolerance of "random" hardware faults. Specification, verification, validation, software, and design faults will not be discussed, although in the conclusion the effectiveness of the techniques proposed in the thesis in addressing some of these important issues will be pointed out. The net effect of the fault tolerance techniques available to the system designer is to achieve a non-zero coverage of certain processor failure modes. The coverage can vary depending on the type of fault suffered, so it is necessary to enumerate the fault categories and characterize the associated coverages. In this section a simplified taxonomy of random hardware faults is presented. Two orthogonal classification schemes will be presented. In the first, faults are classified according to the severity of their behavior. In the second, they are classified according to the frequency of their activity.

The first, most benign, type of processor fault is the *inert* fault. In this failure mode, the processor simply halts. Causes of this type of fault include loss of power to the processor, a spurious jump from execution of valid code to the execution of data or the wrong code, a double bus error[1], or any other conceivable cause. Let the probability that the processor

[1] For example, in the Motorola 680x0 class of microprocessors, a double bus error forces a processor halt.

fault is inert, given that a processor fault has occurred, be denoted by f_I. Associated with an inert fault is a coverage, denoted c_I, which is the probability that the processor ensemble survives the inert fault. This coverage is not necessarily unity, since the faulty processor could be an I/O processor, a control processor, or a central diagnostic processor whose survival is necessary to system operation [Intel86], or the faulty processor could crash while accessing a common bus, making that bus unavailable.

The second type of processor fault under consideration is the *active* fault. Such faults may arise from noisy communication interfaces, a processor stuck in a loop in which it keeps sending a message, or similar causes. Denote the probability that a processor's failure mode is active, given that the processor has failed, by f_A. Define the coverage of active faults as the probability that the ensemble survives an active fault given that one occurred, and denote it by c_A. The active fault coverage c_A is generally less than the inert fault coverage c_I because active faults consume system bandwidth, impeding diagnosis and reconfiguration efforts.

The third and most general type of faulty behavior is one in which a failed processor may manifest arbitrary behavior. This type of failure model is known as a *Byzantine fault*[2]. Byzantine behavior may include stopping and then restarting execution at a future time, sending conflicting information to different destinations, usurping control functions from control processors, and any other physically conceivable and even malicious behavior. The assumption of arbitrarily malicious failure behavior is commonly used as a failure model in distributed computing theory, where specific requirements for the capability of a computing system to survive such faults have been rigorously demonstrated. These requirements will

[2]This nomenclature refers to an analogy between the participants in the consensus algorithm and a group of generals of the realm of Byzantium who are trying to agree upon a plan of attack when some of them may be traitors.

be discussed in greater detail later.

The occurrence of a Byzantine failure is not so farfetched as it may seem. For example, in [Lala83], it was shown that certain failure modes caused a redundancy management program in computer A to tell computer B that computer C is failed, while it tells computer C that computer B is failed. In [Martin78], "at least one inflight failure of a triplex digital computer system was traced to a Byzantine fault and the lack of appropriate architectural safeguards against such faults" [Lala86a]. In the distributed circuit switched network studies in progress at CSDL, a failure mode has been observed in which a network node responds erroneously to commands to another network node. The resulting garbled response would of course result in identification of the innocent node as failed unless more sophisticated tests are carried out specifically with this failure mode in mind. Finally, in recent tests at CSDL, Byzantine behavior of one channel of a quad FTP caused the other three channels to diagnose one of themselves as faulty. The problem was traced to a marginal power supply on the faulty channel, a likely failure mode in practice.

Unless significant effort and appropriate architectural features are combined to survive, categorize, and analyze failure behavior, Byzantine failure is difficult to identify. It is likely that other undiagnosed cases of Byzantine failure mo [illegible] have occurred unbeknownst to the unlucky victims of the failure. It can be concluded that such seemingly malicious failure modes exist in practice and mechanisms must be in place to tolerate them in an ultra-high reliability system.

Let the probability that a processor fault is Byzantine, given that one occurred, be denoted by f_B, and let the probability that the processor ensemble survives a Byzantine fault be denoted by c_B . Since a Byzantine fault may consist of arbitrary behavior, it is fairly clear that such behavior can defeat all fault tolerance techniques based upon restrictive models of failure behavior. Therefore it may be concluded that in such cases c_B=0.

Faults may also be classified according to the temporal duration and periodicity of their activity. In this classification scheme, faults may be classified as *permanent, intermittent,* or

transient faults [Siewiorek82]. In permanent faults, the physical malfunction occurs and remains in the system thereafter. Examples of permanent faults are a bit in memory "stuck at" a fixed value, a signal transmission line shorted to a rail voltage level, and a signal transmission line which is held between logical threshold voltage levels. Note that the latter example can be viewed as a Byzantine failure because different recipients of the marginal signal may perceive different logical levels due to slightly different logic thresholds and thus draw different conclusions about the source. In intermittent faults, the physical malfunction repetitively and indefinitely appears and disappears, possibly as a function of system loading or environmental stress. Examples of intermittent faults include a memory sense amplifier which is in oscillation, periodically outputting a 0, 1, or the correct value, a watchdog timer whose "clear" signal is periodically triggered by a logic fault, and a signal transmission line which is in oscillation. Many permanent faults are initially manifested as intermittent faults which increase in frequency until they are considered to be permanent. Both are caused by a persistent malfunction of the underlying hardware implementation. Therefore the difference between intermittent and permanent faults is largely a matter of semantics.

A transient fault is one in which the fault-inducing mechanism appears, persists for a finite time period, and then vanishes. Examples include a noise spike on a signal transmission medium, a memory bit flip caused by an alpha particle, and a power surge which triggers the power-up reset circuitry. Transient faults are not considered to be symptomatic of persistently faulty hardware, but result instead from external influences and unfortunate combinations of environmental and operational circumstances.

Faults must be classified into these three categories because the results of the fault detection and recovery procedures depend upon the type of fault which occurred. A permanent fault will be present during the rollback attempt, so the resulting error will recur. An intermittent fault may be active during the rollback, or it may disappear during rollback only to reappear during later operation. A transient fault, depending upon its duration, can

be tolerated by repeating rollback until the fault goes away, assuming that the system has not already been irretrievably corrupted.

These three classes of faults occur with different relative probabilities. Permanent and intermittent faults comprise roughly 10 to 20 percent of all physical faults and transient faults comprise the remaining 80 to 90 percent [Siewiorek82]. Therefore, assuming that the fault can be detected before system corruption occurs, there is a significant probability that a retry or a rollback attempt will succeed. Many techniques in common use are targeted to the survival of transient faults, thus reflecting the predominance of this failure mode. In the analysis to follow, the conditional probability that the fault is transient, given that a fault occurred, will be denoted by f_t. Since the difference between them is largely a matter of definition, intermittent and permanent failures will be lumped together and be assumed to occur with a conditional probability $1-f_t$.

The above fault taxonomy can be concisely represented in tabular format (Table 3.2). The horizontal direction represents the temporal nature of the fault, and the vertical direction represents the degree of malice of the fault. The majority of past efforts at fault tolerant computer system design and analysis have focused on tolerance of inert failure modes of all temporal characteristics, and/or transient faults of all degrees of severity.

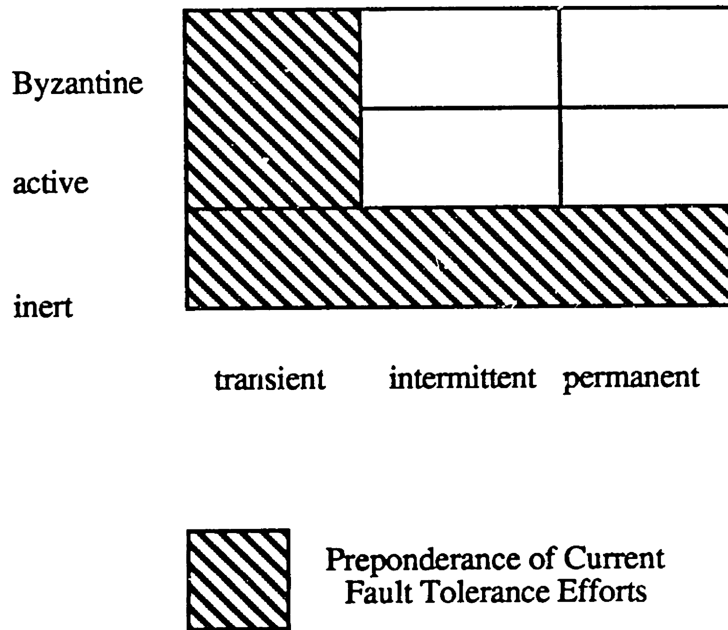


Table 3.2

Simplified Fault Taxonomy

3.4 The Fault Detection and Recovery Process

All fault tolerance techniques rely for their effectiveness on the detection of and recovery from faults. The fault detection and recovery process can consist of up to 10 stages [Siewiorek82], some subset of which must be successfully completed before the fault can be said to be covered. Below, a simplified sequence is presented.

The first event is the detection of the fault. It has been shown that the probability of fault detection, p_{det}, can be made equal to unity only if the mechanism used to detect the fault is at least as complex (in the sense of having at least as many states) as the unit under test [Sundstrom74]. Therefore for a given functional unit p_{det} can be made equal to unity only if duplication of the unit is employed. Comparison of duplicated outputs may be augmented or supplanted by other common fault detection techniques such as watchdog timers, consistency and capability checking, reasonableness checks, and error detecting codes.

In addition to fault detection by the above means, error-correcting codes may be used to

provide both fault masking and detection. Different parts of a computer are amenable to different forms of encoding with varying effectiveness. Error correcting encoding has been most effective in data storage and transmission components, such as memories and transmission links, because the operations performed by these components do not permute the data so the coding scheme does not have to be invariant with respect to any permutation operations upon the data. On the other hand, data permutation devices such as arithmetic/logical units must be equipped with special codes whose properties remain invariant with respect to the arithmetic and logical operations. These codes are complex and it is therefore generally more effective to simply replicate the small amount of data permutation hardware in a computer.

The techniques used to detect or mask the fault may not provide sufficient information to unambiguously diagnose the source of the fault. Therefore a diagnosis mode may be entered in an attempt to identify the faulty processor. Fault diagnosis means include self-test, testing by neighbors or diagnostic processors, failure signature analysis, and others. Successful identification of the faulty component is necessary for the ensemble to survive the processor fault or further faults. Let p_{diag} denote the probability that a correct diagnosis is made, given that a failure occurred and was detected.

Following successful fault detection and diagnosis, a retry at an operation may be performed. The retry may be performed by the faulty unit or possibly another processor entirely. This second attempt may be successful if the fault was a transient fault, but a critical prerequisite to the successful completion of this step is the capability of backing up the execution to a fault-free state, or checkpoint, which was known to exist or could logically have existed prior to the occurrence of the fault. Many issues combine to make this an extremely difficult condition to satisfy. For example, the fault may have occurred long before its detection and/or the storage of a rollback point, so the undetected effects of the fault may be unknown. Moreover, it is often up to the applications programmer to decide when checkpoints should occur and how much data should be saved, exacerbating

the difficulty of the programming task. Finally, rollback points must be globally coordinated in a distributed system to prevent the occurrence of "domino rollback", in which a rollback in one processor triggers a rollback in another, and so forth [Hosseini83]. As described in [Kuhl86],

> "To understand this effect, consider the example of three communicating tasks, shown in Figure 3.5. The solid lines between tasks indicate points of communication...between tasks. ...The dashed lines represent recovery points established for the tasks.
>
> Now, suppose that the processor on which Task A is executing suffers a fault subsequent to establishment of recovery point *ra2*. Then Task A can be restarted on another processor from point *ra2*. But since Task A communicated with Task B subsequent to establishing *ra2*, Task B must be rolled back to maintain proper synchronization with Task A. But rolling back Task B to point *rb2* results in the need to roll Task C back to *rc1*, which requires Task A to roll back to *ra1*, etc., until eventually all three tasks are rolled back to their starting points. This admittedly contrived example illustrates the fact that if the setting of recovery points for the tasks...is not done in a coordinated fashion, the need to roll a task back can trigger a cascade of rollback activity among the tasks... . This is a serious problem not only in that it results in the need to redo a potentially large amount of work, but also because recovery points may need to be saved for an arbitrarily long time to allow for the possibility for a domino-effect rollback."

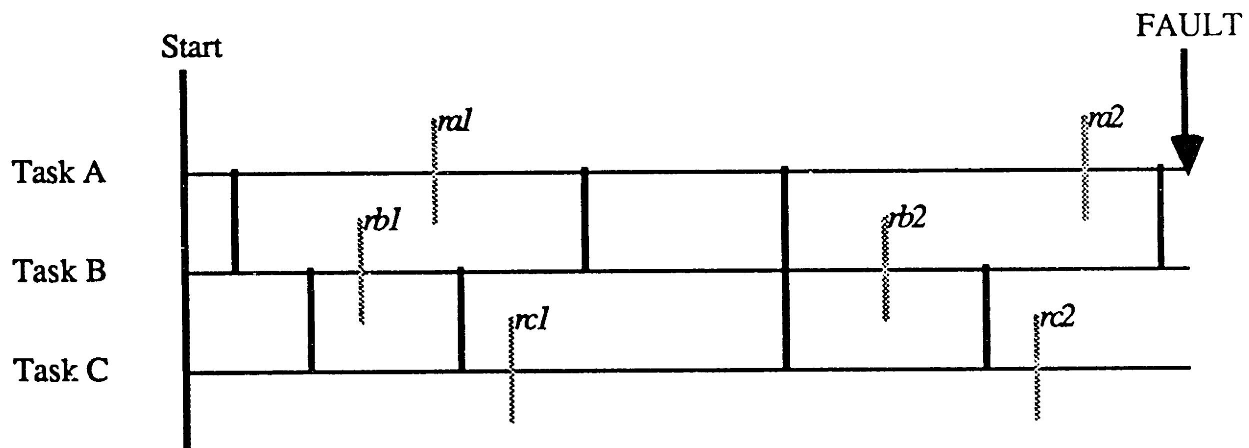


Figure 3.5

Domino-Effect Rollback

It should be mentioned in passing that papers are beginning to appear on functional programming models for fault tolerant distributed systems which have the potential for overcoming the need for global checkpoint coordination [Jagannathan86], [Lin86].

In addition, experience has shown that even in systems composed of a single processor, validation of the correctness of rollback schemes is very difficult. Because of these difficulties, the probability of successful retry of an operation or rollback of a segment of computation, denoted p_{retry}, is less than unity.

If the fault is detected again during the rollback attempt, the faulty unit must be disabled or otherwise isolated from the ensemble. Under certain failure modes such as a stopping failure, this is easily accomplished. Other failure modes such as active failures may require more extensive reconfiguration to isolate the failed processor. Therefore the probability of failure isolation, p_{iso}, is less than unity. If no fault occurs during the rollback procedure, either the fault was transient and will not recur, or the fault was intermittent and will recur as some future time. In the latter case, the fault must be identified eventually or the system will be corrupted and the computational results will be suspect.

As the last line of defense against faults, the parallel algorithm under execution by the

processor ensemble may exhibit a degree of algorithmic robustness to processor faults. This was shown above to be ineffective, in at least one simple algorithmic structure.

In a computer where this train of events must occur for it to survive processor faults, the failure detection, isolation, and recovery processes are complex and difficult to quantitatively analyze. For this reason the probability that the entire set of events is successfully completed is usually lumped into a parameter called *coverage*. Fault coverage, denoted "c", is defined as the probability that the system recovers from the fault and continues to meet its operational specifications given that a component of that system has suffered a fault, i.e., c=Pr[system recovers|system component fails] [Bouricius71]. Because the parameters which contribute to fault coverage are so difficult to quantify, it is usual to parameterize system reliability by the coverage and perform a sensitivity analysis with respect to coverage to observe the effects of varying the fault detection mechanisms used by the system [Rosch84]. This technique of sensitivity analysis will be used in certain analyses to follow.

CHAPTER 4
RELIABILITY ANALYSIS OF CONVENTIONAL APPROACHES

4.1 Simplex Analysis

This section presents a computation of the reliability and performance parameters for an ensemble composed of *simplex* processing elements, i.e., those which do not employ processor replication to aid in detecting, isolating, or masking processor faults. Survival of a processor fault in the simplex ensemble requires the successful completion of the chain of events enumerated in the previous section. Since the simplex processor's primary means of fault detection is a combination of self-test, encoding, reasonableness tests, temporal checks, and possibly periodic testing by the processor's neighbors, the arguments in the preceding section show that the probability that a fault is detected is not equal to unity. In addition, since faults may propagate into the system prior to detection, contaminating checkpoint and control data, the probability of successful rollback or restart is also less than unity.

The Markov model to be used for the reliability analysis of a single simplex processor is shown in Figure 4.1. State 1 is the unfailed state. In this state either the processor has suffered no faults, or a transient fault was suffered, covered, and the resulting logical damage repaired and the processor brought back into the pool of available processors. State 2 is defined to be the uncovered failure or System Loss state. In this state the failure detection and recovery mechanisms have failed to prevent a permanent, intermittent, or transient processor fault from causing erroneous information to propagate into the remainder of the system. State 3 is defined to be the covered failure or Shut Down state. In this state, a permanent or intermittent fault has been correctly detected and the recovery effort has resulted in the identification, isolation, and shutdown of the faulty unit, coupled with the repair of any damaged logical structures in the system. State 4 is denoted the Rollback state, in which the processor has detected a transient failure and is in the process

of successfully rolling back a segment of computation to a known safe state. Note that we do not model the effects of incorrect diagnosis of transient faults as permanent/intermittent or vice-versa.

The probability that the failure is covered is equal to the probability that the failure is inert times the probability that an inert failure is covered, plus the probability that the failure is active times the probability that an active failure is covered, plus the probability that the failure is Byzantine times the probability that the Byzantine failure is covered. According to the above arguments, this latter probability is equal to zero. Therefore define coverage c as $c=(f_I c_I+f_A c_A)$. Since $f_I+f_A+f_B=1$, the lack of coverage 1-c is equal to $1-c=(f_I(1-c_I)+f_A(1-c_A)+f_B)$. Note that the lack of coverage is lower-bounded by the conditional probability of Byzantine failure f_B. The transition rate from State 1 to 2 is the processor failure rate times the probability that the failure is uncovered times the probability that the fault is permanent, intermittent, or transient. The probability that the fault is permanent, intermittent, or transient is unity. Thus

$\lambda_{12} = \lambda_p\,(1\text{-}c)\,1.$

The transition rate from State 1 to 3 is the processor failure rate times the probability that the failure is covered times the probability that the fault is not transient,

$\lambda_{13} = \lambda_p\,c\,(1\text{-}f_t).$

The transition rate from State 1 to State 4 is equal to the processor failure rate times the coverage times the probability that the failure is transient,

$\lambda_{14} = \lambda_p\,c\,f_t.$

Assuming that the probability of completing a successful rollback is constant with respect to time, the transition rate from State 4 to State 1 is equal to the rollback rate μ,

$\lambda_{41} = \mu.$

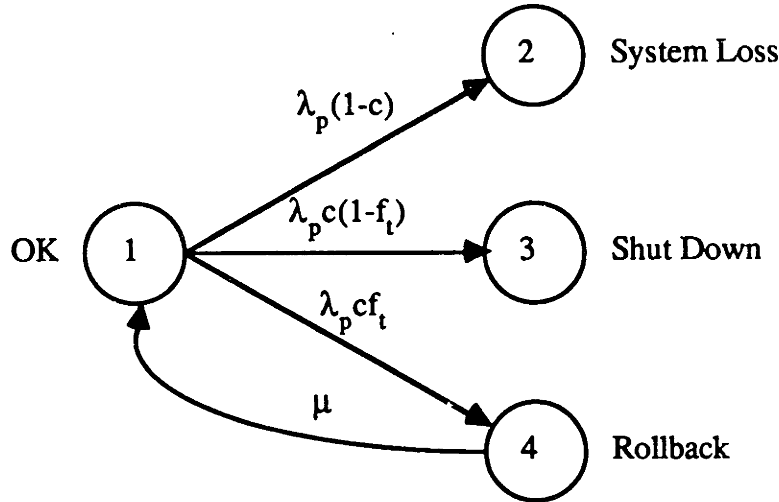


Figure 4.1

Markov Model of Simplex Processor Behavior

According to the arguments in the previous section, if a processor enters an uncovered failure state, ensemble loss due to a corrupted computational result or data structure will soon follow. Therefore, to a close approximation, the probability of ensemble loss due to uncovered processor failures, $p_{SL/E(UPF)}(t)$, is equal to the probability that any processor is in State 2, which is in turn equal to one minus the probability that no processor is in State 2,

$$P_{SL/E(UPF)}(t)=1-(1-p_2(t))^N,$$

where N is the number of processors in the ensemble.

The probability of system loss due to attrition is equal to the probability that all processors are in State 3,

$$P_{SL/E(ATT)}(t)=p_3(t)^N.$$

Define the total probability of system loss as the probability of occurrence of uncovered processor failure or attrition. Since the two events are mutually exclusive, the probability of their intersection is zero, so

$$P_{SL/E(TOTAL)}(t) = P_{SL/E(UPF)}(t) + P_{SL/E(ATT)}(t).$$

$P_{SL/E(UPF)}(t)$ and $P_{SL/E(ATT)}(t)$ have been defined in terms of the following

parameters:

λ_{p0}="core" processor failure rate;

f_{port}=port failure rate as a fraction of λ_{p0};

ports=the number of ports hosted by a processor;

$\lambda_p=\lambda_{p0}(1+\#\,ports*f_{port})$, the aggregate processor failure rate;

μ=reconfiguration or rollback rate;

f_I=probability of inert failure given processor failure;

c_I=coverage of inert failure;

f_A=probability of active failure given processor failure;

c_A=coverage of active failure;

f_B=probability of Byzantine failure;

c_B=coverage of Byzantine failure;

c=effective coverage, $c=(f_I c_I+f_A c_A)$;

f_t=probability of transient failure given processor failure;

N=number of processors in an ensemble.

Because of the large number of parameters listed above, a full parametric analysis is not feasible. In addition, with the exception of the fraction of transient failures f_t, the parameters describing the relative fractions of inert, active, and Byzantine failures are unknown, as are the coverages of all but the Byzantine failure modes. Since these parameters are unknown, the ensemble's reliability will be parameterized on overall coverage c, which will be varied over the range 0.9 to 0.99. The baseline parameters are:

$\lambda_{p0}=10^{-4}$ per hour [Hopkins78] [Wensley78];

f_{port}=0.10 [Rennels84];

ports=lg(64)=6;

$\lambda_p=\lambda_{p0}(1+\#\,ports*f_{port})=1.6*10^{-4}$;

$\mu=10^3$ per hour[1];

f_I=unknown;

c_I=unknown;

f_A=unknown;

c_A=unknown;

f_B=unknown;

$c_B=0$;

$f_t=0.5$ [Siewiorek82];

$c=0.9$ to 0.99 [Rosch84];

$N=64$.

Using these parameter values, the total probability of ensemble loss due to uncovered processor failures or attrition, $p_{SL/E(TOTAL)}(t)$, was calculated and is plotted in Figure 4.2.

The Mean Time To System Failure of the ensemble is calculated as

$$MTTSF = \int_0^\infty R(t)\,dt = \int_0^\infty (1-p_{SL/E(TOTAL)}(t))\,dt\,.$$

It is presented in Table 4.1.

---

[1] This value varies with the synchronization technique and fault recovery technique in use. The extremely conservative value of 10^3/hour will be used in all cases to remove it as a parameter in the following reliability analyses, which have the purpose of comparing *structural* arrangements of processors as opposed to *operational* details. Note that this value includes the fault latency time, which is also assumed to be exponentially distributed [Shin84].

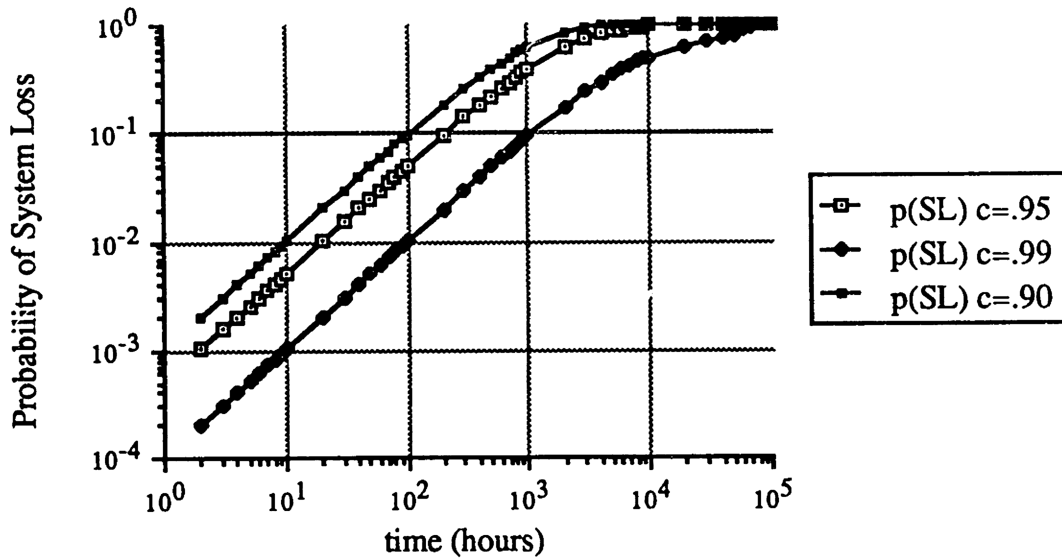


Figure 4.2

Ensemble Loss Probability for 64-PE Simplex Ensemble

The next system performance parameter to be computed for the simplex parallel system is the expected number of operational processing elements in the ensemble versus mission time, $E_{PE}(t)$. This may be calculated as the summation over all possible combinations of system operation of the number of operational processors times the probability that this number of processors is operational, times the probability that the remainder of the processors have suffered covered failures,

$$E_{PE}(t) = \sum_{i=1}^{N} i\ C(N,i)\ p_1(t)^i\ p_3(t)^{N-i}\ , \text{ where } C(N,i) = \frac{N!}{(N-i)!\ i!}\ .$$

This quantity is plotted in Figure 4.3.

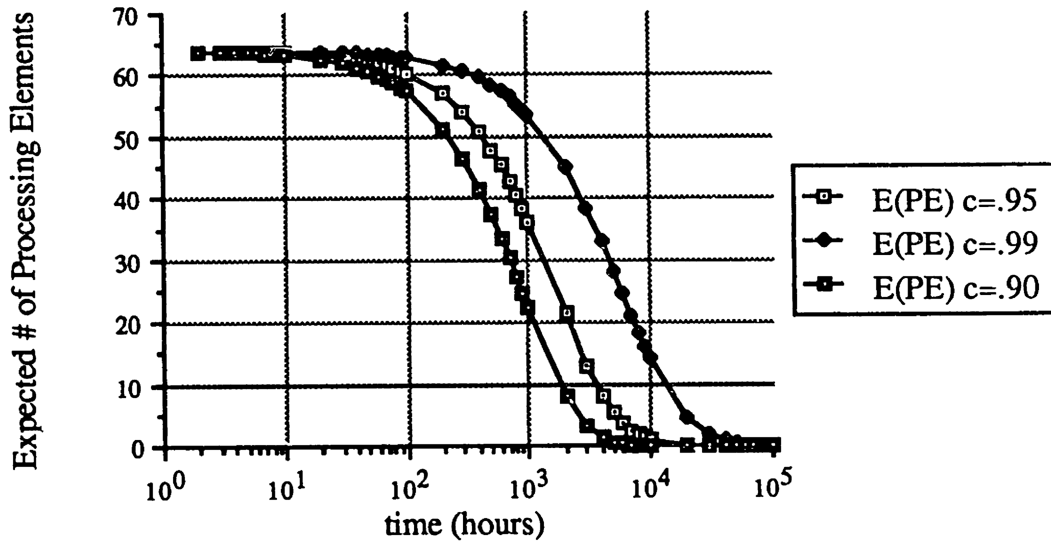


Figure 4.3

Expected Value of Number of PEs vs. Mission Time for 64-PE Simplex Ensemble

Table 4.1 shows the general list of performance parameters appropriately filled with the parameters for the 64-processor hypercube ensemble.

Parameter	Value
# PE	64
#ports	64*6
system loss probability	
10 hours	5 E-3
100 hours	5 E-2
1000 hours	0.39
10000 hours	0.83
E (t)	
PE	
10 hours	63.6
100 hours	60.4
1000 hours	36.2
10000 hours	0.83
mean time to system failure	2,089 hours
intra-group communication	
distance	
diagnosability	
anisotropy	
inter-group interactions	N/A
inter-group communication	
distance	
diagnosability	
anisotropy	
transparency of FT technique to programmer	
Low transparency due to complicated checkpointing and recovery techniques	
efficiency of FT technique	
Low efficiency due to tradeoff between coverage and time spent on diagnostics	

Table 4.1

Performance Parameters for

64-Processor Simplex Ensemble[2]

---

[2] c=0.95.

4.2 Duplex Analysis

The preceding analysis showed that a processing ensemble composed of simplex processing elements was incapable of meeting the reliability requirements imposed by the mission specification. This inadequacy is a result of the low coverage provided by fault detection and isolation methods which rely for their effectiveness upon restrictive and hypothetical enumeration of possible failure modes and the intrinsic inability of such methods to tolerate unanticipated failure modes. As is evident by the strong sensitivity of the analytical results to the coverage parameter, what is clearly needed is a mechanism that provides a higher failure coverage regardless of failure behavior.

The next logical step in the escalation of fault tolerance techniques in an attempt to obtain high coverage is to duplicate the processors, provide them with identical inputs and code, run the duplicated units synchronously, and perform bitwise comparison of the outputs. Disagreement indicates that one of the duplex units has failed. In terms of the framework outlined above, this approach has the potential of increasing the processor fault detection probability, p_{det}, to unity.

Before beginning any analysis of an ensemble composed of duplex processors, the prerequisites for achieving unity coverage via duplication must be elucidated. These requirements are 1) the redundant processors must be synchronized to within a known and bounded skew; 2) the redundant processors must be provided with bitwise identical inputs; and 3) a mechanism must be in place to detect bitwise disagreement between the redundant sites. Each of these prerequisites will be examined in turn. It will be shown that they cannot be met by traditionally designed duplex systems, calling into question the degree of coverage such systems can provide.

1. Synchronization: Synchronization of the redundant sites is necessary for three reasons. First, some *a priori* limit must be imposed upon the temporal behavior of a processor to differentiate between normal behavior of a slow processor and an inert

processor failure. Second, the comparison mechanism must have some means to know when valid outputs have been received in order to compare them. Third, it is theoretically demonstrable that the redundant sites must be synchronized in order for the duplicated sites to be able to arrive at bitwise consensus on the input data. This point will be elaborated upon later.

Synchronization may be achieved in a variety of ways. In a commonly used method each of the redundant channels sends a message, which may be a discrete signal, to the other and awaits the reception of a matching signal from the other channel. Unfortunately, this technique, as is any technique which relies solely upon the two channels to generate a consensus value (i.e., the presence or absence of a synchronization signal), is not resilient to faults of either channel or in the inter-channel communications medium.

This problem was first described in [Gray79] as the problem of two generals who are trying to agree upon a plan of attack. The generals are physically isolated and must rely for communication upon messengers who may get lost or captured, defect, or otherwise fail to deliver the message. In this context, the question is whether the two generals can decide upon a common plan of attack (i.e., whether or not to attack) in the face of unreliable inter-general communication, given that the two generals are initially undecided. This problem is known as the "Two Generals Problem", and is clearly identical to the problem of two channels of a duplex trying to agree upon a common value, such as the existence of a synchronization signal. That it is impossible to achieve such agreement can be shown by the following argument.

Assume that the generals are initially undecided about a plan of action, and begin sending messengers back and forth. Without loss of generality, assume that the generals alternate sending and receiving messengers. Also assume that it is possible to achieve agreement using the unreliable messengers, and that both generals reach agreement at some point in time. These two hypotheses can be shown to be mutually exclusive.

Consider the last messenger to be sent prior to achieving agreement. Since this was indeed the last messenger sent, the sending general must have decided upon a plan prior to sending the message, since he receives no further messages. Therefore the loss of the message could not possibly affect that general's decision. Suppose that the message was in fact lost. Since the sending general has decided upon a plan regardless of the message loss, the receiving general must also decide upon the same plan regardless of the message loss, since it has been hypothesized that a protocol exists which guarantees agreement in the face of message loss. Therefore the last message must have been incapable of influencing the recipient general's decision as to a plan of action. The two generals were in agreement regardless of the loss or delivery of the last message, so assume that the last message was not lost. The two generals were in agreement before the sending of that last message.

Now consider the next to last message sent. The sender of the next to last message must have decided on a plan of action prior to sending the next to last message since the only further message he receives is the ineffectual last message, so the loss of the next to last message cannot affect his decision. Since its loss cannot affect the sender's decision, it cannot affect the recipient's decision because, unlike the recipient, the sender of the next to last message has no way of knowing whether or not the next to last message was delivered and adjusting his decision accordingly. Therefore the two generals were in agreement prior to the transmission of the next to last message.

Applying this argument in this way to the entire sequence of messages sent, it can be seen that the generals must have been in initial agreement before sending any messages at all, contradicting the hypothesis that the generals were initially undecided. Therefore a protocol does not exist which allows two initially undecided generals to achieve agreement upon a plan of attack in the face of unreliable messengers.

This result implies that, in the face of interchannel communications faults, it is impossible for two channels of a duplex processor to reach agreement upon the presence of

a synchronization signal, an input value, or even whether two inputs agree.

A second synchronization technique is to provide each channel with a reliable clocking signal and force the channels to be synchronized with the edges of this fault tolerant clock. Note that this clock signal cannot be derived from the two channels alone because the clocking mechanism would be subject to the Two Generals failure mode. The reliability of this outside source then becomes the upper bound on the reliability of the duplex. If it is designed according to a duplex architecture, it too is open to the Two Generals failure mode, although since a clock source is simpler than a processor its failure rate is likely to be much lower than a processor's.

2. Input Data Consistency: Recall that the purpose of duplicated computation was to achieve near-unity failure detection probability for arbitrary failure modes, thus avoiding the low coverage and concomitant low system reliability resulting from fault tolerance means based on restrictive failure hypotheses. Therefore it is necessary to provide inputs to the two channels in such a way that arbitrary failure behavior occurring during input distribution does not circumvent the high-coverage comparison technique by allowing the two channels to receive differing inputs. To guarantee bitwise identical computation and hence bitwise identical output agreement in the absence of computation errors each channel of the duplex must be given bitwise consistent input data via some input consistency maintenance mechanism or protocol. These protocols have been extensively studied, resulting in the well-known and theoretically demonstrable prerequisites for consensus algorithms which are capable of correctly functioning in the face of arbitrary failure behavior on the part of "f" of the participants in the algorithm. One form of these requirements may be expressed as:

1. There must be at least 3f+1 participants in the algorithm [Pease80].

2. Each participant must be connected to at least 2f+1 other participants through disjoint communication paths [Dolev82].

3. The algorithm must consist of a minimum of f+1 rounds of communication among the participants [Fischer82].

4. The participants must be synchronized to within a known skew of each other [Dolev84].

A system which meets these prerequisites and is capable of executing the appropriate algorithm is called "f-Byzantine Resilient".

The term "participant" in the above set of requirements corresponds to the concept of a *Fault Containment Region*, a crucial term in fault tolerant computer architecture. A fault containment region (FCR) is a region to which faults are contained to the degree that the probabilities that different FCRs suffer faults are statistically independent. If p_1 is the failure probability of FCR_1 and p_2 is the failure probability of FCR_2, the joint probability of the failure of FCRs 1 and 2 is p_1p_2. Independence of failure may be obtained by provision of four fundamental architectural features: physical isolation[1], electrical isolation, independent power, and independent clocking.

Let us examine a duplex system in light of Requirement 1. A system designed to tolerate a single Byzantine fault, i.e., f=1, would require 4 fault containment regions. (See Figure 4.10 for a diagram of a minimal Byzantine Resilient processing site.) Clearly the two (or three, if the comparator module resides in a separate FCR) fault containment regions of a duplex do not meet this requirement, so such a system cannot guarantee bitwise input consistency in the presence of arbitrary failure behavior. However, typical approaches to input consistency maintenance in a duplex will be briefly discussed.

A first attempt at ensuring duplex input consistency might be to provide the data to one channel which then forwards the data to the other. This approach is inadequate because a

[1]Clearly the extent of the physical damage that a FCR can do under failure conditions must be known in order to determine what degree of physical isolation guarantees independent failure probabilities.

faulty forwarding channel would cause both channels to receive consistent but incorrect information, thus allowing the circumvention of the high coverage of the output comparison circuit. Alternatively, one might try to cross-strap the input to both channels. In this case, a replication fault might cause both channels to receive different data. If it were attempted to circumvent this fault mode by having the channels exchange their data and select the common value or possibly a default value if no common value exists, it turns out that it is not possible for both channels to come to agreement upon whether the two input data were in agreement in the first place, as a result of the Two Generals failure mode.

These considerations appear to be minor inconveniences until it is realized that an insufficiently robust input consistency mechanism could allow both channels to receive identical but incorrect input data. This would result in the members of the duplex group achieving bitwise identical agreement upon an erroneous output value and propagation of that output value into the remainder of the ensemble via circumvention of the comparison module. This could result in system loss via uncovered corruption of the computational result. The reliability of the input distribution mechanism in such a case is a lower bound on the probability of system loss.

3. Output Comparison: There are typically two design options for comparing duplex output. In one option the comparator is an integral part of the duplex processing unit, usually performing both input consistency maintenance and concurrent error detection by input and output data comparison (Figure 4.4). Such designs are known as "self-checking pairs". In the event of a disagreement between the channels, the comparator disables its outputs and signals an error condition to the duplex itself as well as the remainder of the system. The comparator module is clearly a single point of failure, but self-checking checkers have been extensively studied and relatively reliable implementations of this simple circuit are possible [Depaula82]. Additionally, the self-checking pair technique allows the use of the significant conceptual simplification that any output received from a

correctly functioning checker is a correct one. However, there are conceptual difficulties in an approach in which the checker is incorporated into the fault containment region of the device it is supposed to check. The duplex channels are typically in a single fault containment region, so it is not possible to guarantee independence of failure modes. Second, as mentioned before, the comparator reliability is an upper bound on the reliability of the duplex, and hence on the ensemble.

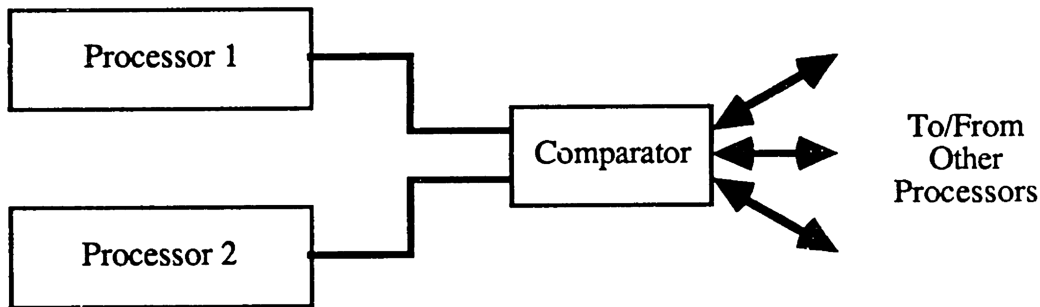


Figure 4.4

Self-Checking Pair Processing Element

In the second design option a duplex transmits two copies of each output to its destination over disjoint communications paths (Figure 4.5). The recipient then compares the messages and detects errors. This approach ostensibly eliminates the reliability bottleneck represented by the output comparator and in addition masks communications errors. A closer inspection of the approach shows that the comparator function has been moved to the neighboring processors, where the Two Generals failure mode still exists. In addition, the distributed nature of the method complicates the fault detection and isolation process. In particular, it could become impossible to differentiate between the failure of a source duplex channel, a faulty transmission line, or the failure of a recipient duplex channel's input consistency mechanism.

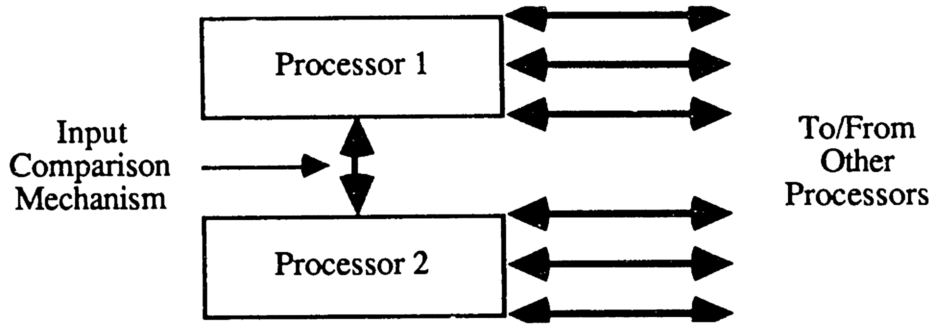


Figure 4.5

Externally Checked Pair

Additional assumptions are now made to facilitate the performance analysis of a processor ensemble consisting of unreconfigurable duplex sites.

1. The system consists of 64 unreconfigurable duplexes (128 processors total).

2. The duplexes are operated as self-checking pairs. This architecture was chosen for analysis because it is the most widely discussed and most common implementation of a duplex processor.

3. The comparator failure rate λ_c is constant with time and parameterized with respect to the number of inter-duplex ports it must support, i.e., $\lambda_c=\lambda_{c0}(1+f_{port}*\#nports)$. From [Depaula82] the value of 10^{-6}/hour will be used for λ_{c0}.

4. Processor and comparator failures are independent.

The Markov model representing the behavior of one duplex is presented in Figure 4.6. State 1 is the state in which there are no failures present in the duplex. This may be because none have yet occurred or because a transient fault was covered and the duplex has subsequently successfully retried an operation or a segment of computation. In State 2, the comparator has suffered a failure or a second channel failure has occurred before a first has been recovered from. Either of these is conservatively assumed to result in failure of the duplex, and in addition, to allow erroneous information to propagate into the remainder of the ensemble and result in eventual system loss. In State 3, the duplex has suffered a covered intermittent or permanent fault and is in the process of safely shutting down. This

transition occurs with rate μ. If a failure of the second channel occurs during this process, it is conservatively assumed that such a near-simultaneous failure mode results in an uncoverable failure and a transition into State 2. In State 4, the duplex has detected a transient fault in a channel and is in the process of successfully performing a rollback operation. Successful completion of this operation results in a transition back to State 1. If, however, a second fault occurs during this process, the duplex enters the System Loss state.

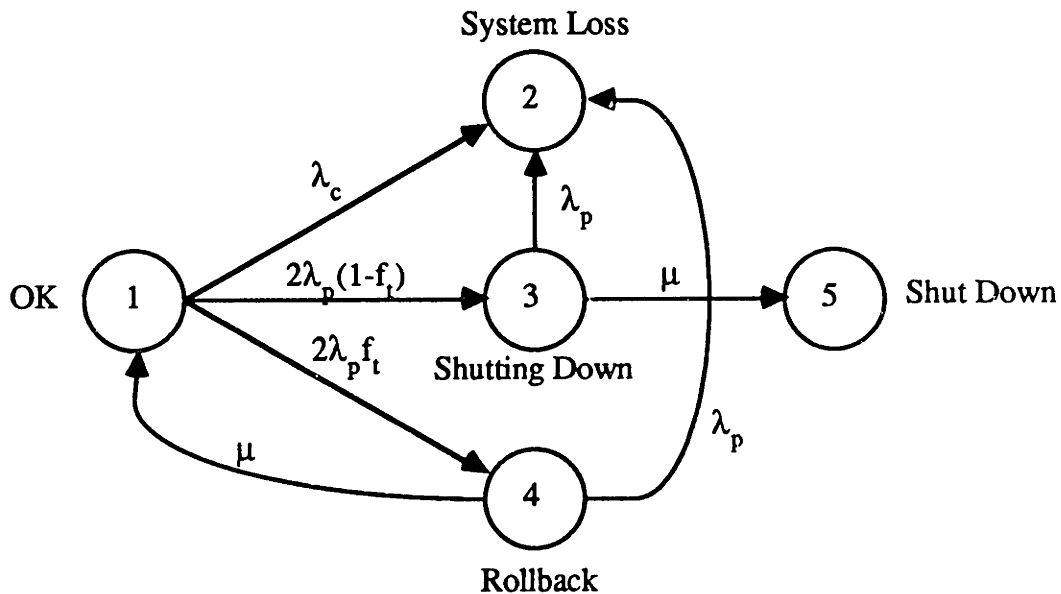


Figure 4.6

Markov Model of Self Checking Pair

Define "system loss due to uncovered failure" as the probability that any duplex is in State 2,

$$p_{SL/E(UF)}(t)=1-(1-p_2(t))^N$$

and "system loss due to attrition" as the probability that all duplexes are in State 5,

$$p_{SL/E(ATT)}=p_5(t)^N.$$

The total system loss probability is the probability of the union of the two mutually exclusive events uncovered failure and attrition,

$p_{SL(TOTAL)}(t) = p_{SL/E(UF)}(t) + p_{SL/E(ATT)}(t)$, and is plotted in Figure 4.7.

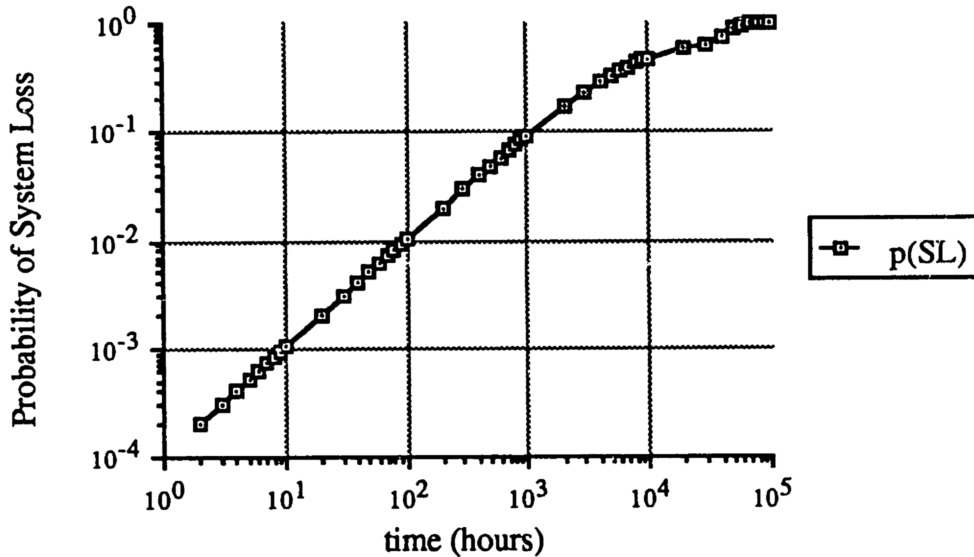


Figure 4.7

Ensemble Loss Probability for 64-PE Duplex Ensemble

The Mean Time to System Failure of the ensemble is calculated as

$$MTTSF = \int_0^\infty R(t)\,dt = \int_0^\infty (1-p_{SL/E(TOTAL)}(t))\,dt\,.$$

It is assumed that duplexes which are undergoing reconfiguration are not counted as useful processors. The expected value of the number of processing elements in the ensemble versus mission time may be expressed as

$$E_{PE}(t) = \sum_{i=1}^{N} i\,C(N,i)\,p_1(t)^i\,p_5(t)^{N-i}\,, \text{ where } C(N,i) = \frac{N!}{(N-i)!\,i!}\,.$$

This quantity is plotted in Figure 4.8.

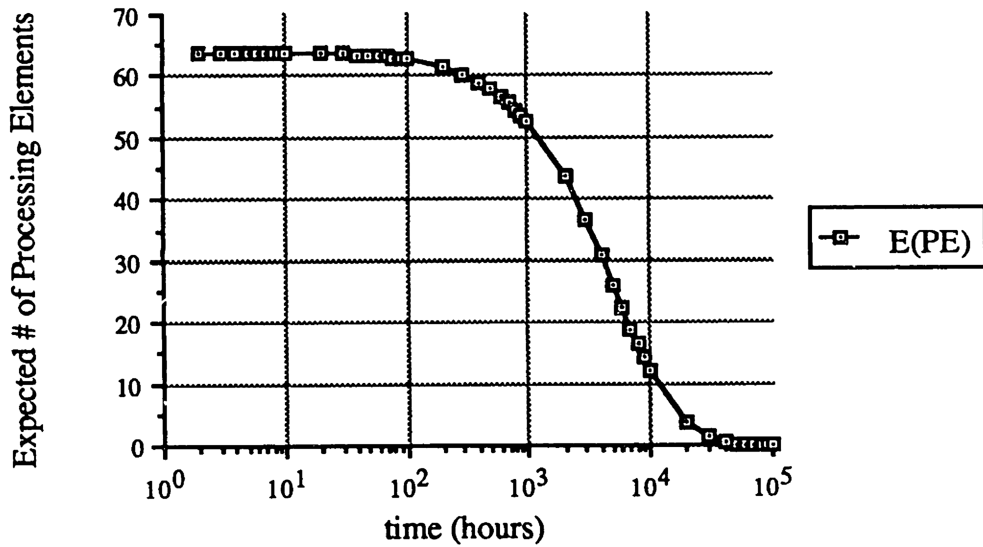


Figure 4.8

Expected Value of Number of Groups vs. Mission Time for 64-Group Duplex Ensemble

Table 4.2 shows the general list of performance parameters appropriately filled with the parameters for the 64-duplex hypercube ensemble.

# PE	128
#ports	128 + 64*6
system loss probability	
10 hours	1 E-3
100 hours	1 E-2
1000 hours	9 E-2
10000 hours	0.48
E (t)	
PE	
10 hours	63.9
100 hours	62.7
1000 hours	52.5
10000 hours	12.3
mean time to system failure	19,064 hours
intra-group communication	
distance	0
diagnosability	comparator
anisotropy	none
inter-group interactions	none
inter-group communication	
distance	lg 64
diagnosability	uses comparator
anisotropy	none
transparency of FT technique to programmer	
Low transparency due to complicated checkpointing and recovery techniques	
efficiency of FT technique	
Low due to necessity of global recovery point maintenance	

Table 4.2

Performance Parameters for

64-Duplex Processor Hypercube

4.3 Fault Masking Analysis

It has been shown that fault tolerance methods which provide fault detection capability but no fault masking capability require generation and management of system-wide coordination of rollback points. Such coordination is necessary to prevent the occurrence of domino-effect rollback, and the execution and communication cost of the global rollback point management algorithms make the duplex architecture look less tractable and efficient than a superficial examination would indicate. Also, the synchronization, input consistency, and output comparison mechanisms of the duplex architecture were shown to be incapable of allowing unity coverage of arbitrary failure behavior and to be possible single points of failure. These considerations lead to the investigation of means to obtain fault masking computation, in which the need for rollback techniques is alleviated, and which have the potential for tolerating arbitrary failure behavior. Processing sites capable of such fault masking operation will be denoted *fault masking groups*, or *FMGs*.

Traditional approaches to obtaining concurrent masking of computation faults utilize 2n+1 "data synchronous" processors, where "n" refers to the number of simultaneous faults it is desired to tolerate. In such systems 2n+1 identical copies of a computation are synchronously executed on 2n+1 independent processing units. Output comparisons provide masking of any n processor faults, regardless of the fault or its manifestation, as well as diagnosis of the faulty channel. This is the classical N-Modular Redundancy (NMR) technique. An important point to realize is that a NMR processor can diagnose its own status, assuming that the redundancy has not been overwhelmed by failures. A duplicated processor cannot decide whether its two channels are in agreement because of the Two Generals failure mode. Therefore the former can gracefully shut itself down by load balancing without relying on recovery point maintenance. In addition, the former can be guaranteed to fail-safe.

A minimal FMG is usually considered to be a Triply Modular Redundant (TMR)

processor. However, the requirements for Byzantine Resilient input consistency maintenance outlined in the previous section hold for a triplicated processor. In particular, the three fault containment regions traditionally associated with the TMR approach do not meet the 3f+1 fault containment region lower bound of Byzantine Resilience theory. For this reason truly Byzantine Resilient processors must provide additional fault containment regions.

This lower bound on the number of fault containment regions, henceforth known as the *cardinality constraint*, is sufficiently central to the theme of this thesis and in fact counterintuitive that it deserves some discussion. Consider a TMR group composed of three fully connected processors (Figure 4.9).

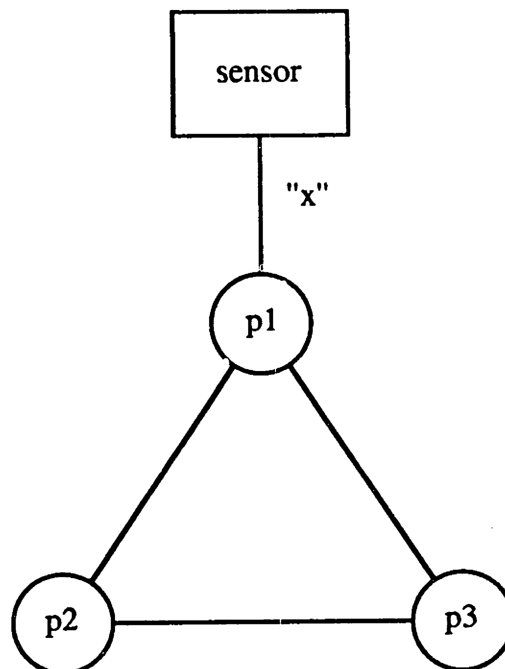


Figure 4.9

Simple TMR Architecture

Suppose that a sensor is connected to p1, and it is desired to perform some computation based on the value read by this sensor. The results of this computation will be voted by the

three processors to detect and mask any errors which may have occurred during the computation. Before this can be done, it is necessary to distribute the datum generated by the sensor to all three processors. During this distribution process the triad must be able to tolerate any failure behavior which may occur. Otherwise if there is a failure mode which this process cannot tolerate, the triad as a whole cannot be considered capable of surviving arbitrary failure behavior, and hence the desired near-unity fault coverage cannot be obtained.

Studying the distribution process in more detail, we can define two requirements that the process must be able to guarantee in the face of arbitrary failure behavior. These are called the *agreement* property and the *validity* property [Pease80]. Agreement means that each non-faulty processor in the triad comes to the same conclusion on the value of x. This value may of course reflect a faulty sensor or a faulty processor hosting a sensor, and secondary sensor FDI mechanisms are assumed to be in place to tolerate such faults. The important point is that the agreement property implies that, even in the presence of arbitrary failure behavior during the input distribution process, the triad as a whole has been delivered bitwise identical inputs and hence subsequent voting can mask and/or detect faults occurring during computation. In the case of the faulty sensor or host processor, each redundant copy of the sensor FDI program will obtain the same value and come to the same conclusion about the state of the sensor. The validity property means that, if the processor which is hosting the sensor is non-faulty, then each other non-faulty processor will obtain that same value after the distribution process is over. This implies that a faulty processor which does not host the sensor cannot confuse the nonfaulty processors into misunderstanding the information sourced by the nonfaulty source processor.

It is straightforward to show that the TMR architecture in Figure 4.9 is incapable of guaranteeing validity and agreement. The approach is to describe two input distribution scenarios, assume that validity and agreement can be satisfied, and derive a contradiction.

In the first scenario, suppose that p1 is faulty. Then all that needs to be guaranteed is agreement, since validity is only important when the information source is nonfaulty. Assume $x \in \{0,1\}$. Let p1 be failed in such a way that it transmits a "0" to p2 and a "1" to p3. Clearly at this point p2 and p3 do not have identical inputs, so they then exchange their values, such that p2 knows that p3 received a "1" and p3 knows that p2 received a "0". It should not be hard to convince oneself that now p2 and p3 know everything about the triad that they ever can, barring further intervention from p1, so that there is no point in any further information exchanges. Denote the set of values received by a participant at the end of the distribution process by the term *value set*, a linearly ordered set having as its ith element the value received from participant i on the second round of the algorithm. Now, p2 has the value set {-,0,1} and p3 has the value set {-,0,1}, where the "-" indicates that the source does not send any values in the second comparison round. Since there is no clear majority, p1 and p2 must arbitrarily choose a default value. Without loss of generality let this default value be "0". Agreement has been achieved.

Now consider another scenario in which p1 is nonfaulty and p2 is faulty. Since p1 is nonfaulty the validity condition must hold in addition to the agreement condition. P1 transmits a "1" to both p2 and p3. Note that p2 and p3 cannot immediately use this value because p2 has no way of knowing what p3 received at this point, and vice versa. Therefore p2 and p3 again exchange their values as before. Suppose p2 is faulty in such a way that it transmits a "0" to p3. Since p2 is faulty we can ignore its conclusion and concentrate on the situation as seen from p3. P3 has the set of values {-,0,1}. Since we assumed that validity holds, and since nonfaulty p1 sent a "1", then p3 must choose a "1". But this scenario looks the same to p3 as the first, and we have decided that p3 chooses a "0" in that case. Therefore p3 must choose a "0" in the present scenario. Since p3 cannot choose both "0" and "1", this contradiction implies that validity and agreement cannot both be satisfied with three participants in the face of participant faults. More formal results

confirm this rigorously [Pease80].

In [Pease80] it is also shown that agreement and validity can be achieved using four participants, if they are fully connected. Figure 4.10 shows such an arrangement of processors.

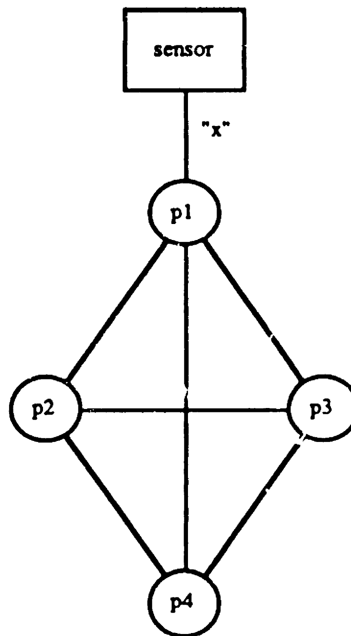


Figure 4.10

Minimal Byzantine Resilient Architecture

We now informally demonstrate how the above architecture can tolerate a failure scenario similar to the one described above. In the first scenario in which p1 is faulty, let p1 transmit a "0", "1", and "0" to p2, p3, and p4, respectively. Thereafter, p2, p3, and p4 exchange the values they received from p1, resulting in each participant having the value set {-,0,1,0}. In this case, as in any case, a clear majority must exist. Here it is the value "0". Each participant decides upon a value of "0" and hence agreement is achieved.

Now, in the second scenario, let p1 transmit a "1" to all participants. During the second exchange, let p2 be faulty and transmit a "0" to p3 and a "1" to p4. P3 and p4 both transmit a "1" to each other because they faithfully relay the value received from p1. Participant p3

thus has the value set {-,0,1,1} and p4 has the value set {-,1,1,1}. Both participants choose the majority value, "1", and both validity and agreement are achieved. Again, this result is formally demonstrated in [Pease80], where it is also shown that to tolerate "f" such faults simultaneously, it is necessary to provide "3f+1" such participants.

Note that not all participants need be processors. Only 2f+1 sites are needed to mask computation faults, so the f additional participants in the distribution protocol can be either other processors which are not members of the FMG, or hardware elements specially designed for participation in the protocols. Also note that the protocols have been assumed to be synchronous, consisting of two "rounds" of communication, a distribution round and a comparison round. In [Fischer82] it was formally shown that to tolerate "f" participant failures then "f+1" such rounds of communication are required. In [Dolev84] it was shown that these rounds must be synchronous, in the sense that the time required for each participant to complete a round must be bounded.

In existing Byzantine Resilient computers, the requisite number of fault containment regions has been provided in one of two ways. The first way is to add fault containment regions whose sole purpose is to participate in the input consistency protocol. These additional fault containment regions, called *interstages*, are potentially much simpler than processing elements, so a net savings in terms of hardware complexity and power consumption is realized. This approach is applied in the CSDL Fault Tolerant Processor (FTP) [Smith83], and is illustrated in Figure 4.11. To allow inclusion into a parallel processing ensemble, the processors are slightly modified from the traditional FTP architecture by the addition of ports to the processors to support interprocessor communication.

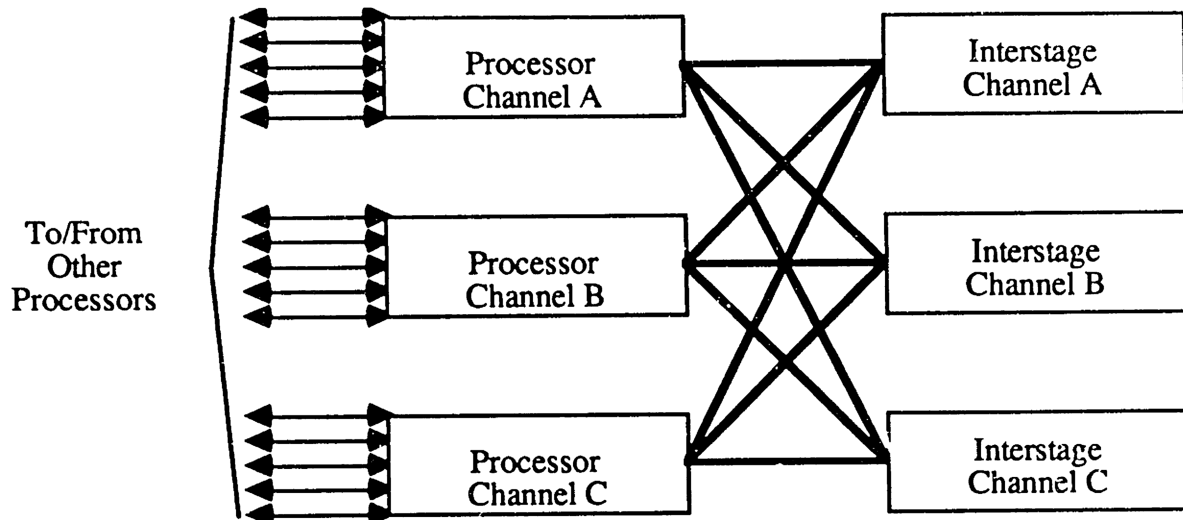


Figure 4.11

CSDL Fault Tolerant Processor Architecture

Another approach is to provide the requisite fault containment regions in the form of extra processors. The SRI Software Implemented Fault Tolerance (SIFT) computer uses this approach [Wensley78], which is illustrated in Figure 4.12, again slightly modified for interprocessor communications. This computer has significant operational differences from the CSDL FTP, but these will not be discussed in this chapter, where the only concern is the effect of the arrangement of hardware resources on the relative performances of parallel systems. Both approaches solve the problem of theoretically adequate input consistency maintenance and fault masking with varying degrees of efficiency.

The operation of a fault-masking processing architecture is roughly as follows. The processor executes normal computation, interspersed with periodic presence tests and voting of the computational results. Any simplex inputs or channel-specific data are distributed using a Byzantine Resilient interactive consistency protocol (called "source congruency" in the FTP architecture). Unless it is latent, a fault in any single channel results in a disagreement at some vote phase or causes the channel to miss a presence test, resulting in the identification of the faulty channel and the subsequent shutdown of the processor. Note that the error indication resulting from the vote, known as the *error*

syndrome, may differ in each channel because a faulty processing site may present a correct result to one channel and an incorrect result to another. Therefore syndrome information must be treated as channel-specific data and distributed using Byzantine Resilient consensus protocols before the FMG can make any decision based upon it. While it may appear inefficient to shut down a 4-processor site after only one failure, it can be shown that if this prudent action is not taken, then the inability of the remaining 3-processor site to tolerate a second Byzantine failure increases the probability of catastrophic system loss by several orders of magnitude.

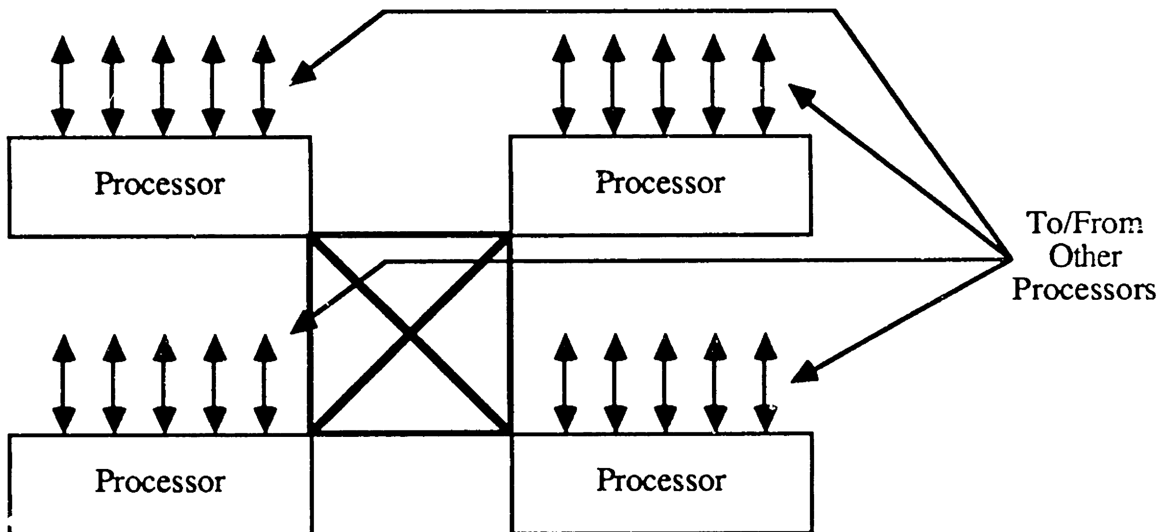


Figure 4.12

SRI Software Implemented Fault Tolerance (SIFT) Computer

4.3.1 Analysis of Quadruply Redundant Fault Masking Architecture

For the purposes of the first calculation of the performance aspects of a fault masking architecture, a baseline architecture will be assumed to consist of 4 processors arranged in a fully connected topology, as in Figure 4.12. The Markov model for a symmetric 4-processor is shown in Figure 4.13.

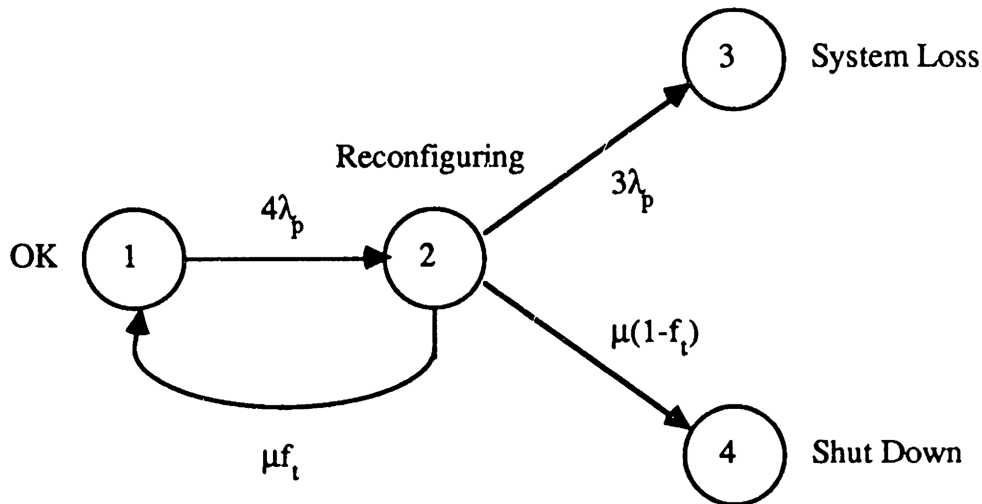


Figure 4.13

Markov Model for Symmetric 4-Processor Fault Masking Architecture

In the Markov model describing the failure behavior of the quad, State 1 is the unfailed state. When the quad is in State 1, it either has never suffered a failure or it has suffered a transient failure and has since recovered from the transient. Note that a fault masking group can always eventually recover from a transient fault if no intervening colluding fault occurs prior to complete recovery. This is because the fault cannot cause contamination of any logical system structure due to the fault masking nature of the design. In State 2, a processor fault has occurred and the quad is in the process of detecting, diagnosing, and recovering from the fault. The time spent in this state is assumed to be distributed according to a Poisson distribution, hence the transition rates out of State 2 are constant and denoted by μ. If the fault is a transient fault, the quad transitions back to State 1. If not, the quad transitions to State 4, in which it shuts itself down safely. If a second processor fault occurs while the quad is in State 2, there is a likelihood that the entire quad will undergo total failure because it is incapable of tolerating two simultaneous processor faults. Erring on the side of conservatism, it is assumed that this will always be the case. Further, it is assumed that such a simultaneous processor fault allows erroneous information to propagate into the remainder of the ensemble, resulting in system loss.

Define "system loss due to simultaneous failures" as the probability that any quad is in State 3,

$P_{SL/E(SF)}(t) = 1-(1-p_3(t))^N$

and "system loss due to attrition" as the probability that all quads are in State 4,

$P_{SL/E(ATT)}(t) = p_4(t)^N$.

The total system loss probability is the probability of occurrence of either of the above mutually exclusive events

$P_{SL/E(TOTAL)}(t) = P_{SL/E(SF)}(t) + P_{SL/E(ATT)}(t)$, and is plotted in Figure 4.14. The shape of this curve exhibits several distinct characteristics which it is informative to study in some detail. The observations so obtained will come in handy in the more complicated calculations to follow.

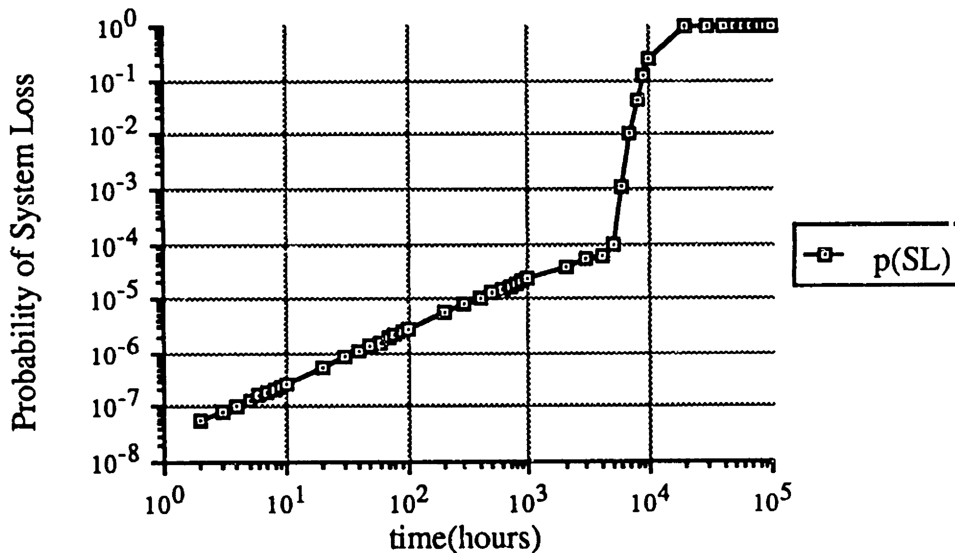


Figure 4.14

Ensemble Loss Probability for 64-PE Quadruplex Ensemble

First, we examine the curve in the short mission time regime, from time zero to a few thousand hours. The curve has unity slope on the log-log plot, indicating that in this regime

system loss probability is linear with respect to time. This can be confirmed by the following analysis. The Markov model to be used to show this is a degenerate form of the full model for the quad, in which it is assumed that the transient failure rate is equal to zero.

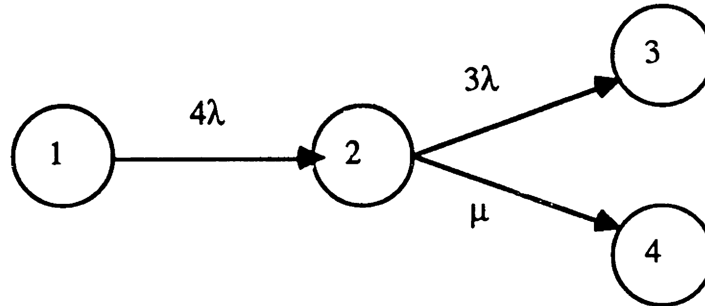


Figure 4.15

Markov Model to Demonstrate Linear Approximation

to System Loss Probability

The probability that a quad represented by the above model suffers loss due to near-simultaneous failures is

$$p_3(t) = \frac{12\lambda^2}{\mu-\lambda}\left[\frac{1}{4\lambda}(1-e^{-4\lambda t}) - \frac{1}{3\lambda+\mu}(1-e^{-(3\lambda+\mu)t})\right].$$

To make the desired point, some approximations need to be made. The processor failure rate is typically much less than the reconfiguration rate, so that $\lambda << \mu$. This allows us to neglect λ in terms involving the sum or difference of λ or μ. Furthermore the terms involving exponentials can be approximated by

$1-e^{-4\lambda t} \approx 4\lambda t$ and $1-e^{-(3\lambda t+\mu)t} \approx 1$.

Making these approximations,

$$p_3(t) \approx \frac{12\lambda^2}{\mu}\left[\frac{4t}{4} - \frac{1}{\mu}\right].$$

Noting that $1/\mu << t$ for $t>O(1 \text{ hour})$ yields

$$p_3(t) \approx \frac{12\lambda^2 t}{\mu}.$$

The probability of ensemble loss due to near-simultaneous quad failures is equal to the probability that any quad in any cluster has suffered near simultaneous failures, or

$$p_{SL/E(SIMF)}(t) = 1-(1-p_3(t))^N$$

where N is the total number of quads in an ensemble. For small values of $Np_3(t)$, this is approximately equal to

$$p_{SL/E(SIMF)}(t) \approx Np_3(t) \approx N12\lambda^2 t/\mu.$$

Figure 4.16 shows this approximation compared to the results of the full Markov model introduced above, and shows very close agreement except where attrition sets in. Here, the result from the full analysis droops because, as the relationship above shows, $p_{SL/E(SIMF)}(t)$ is linearly proportional to N. As attrition occurs, N is reduced by failures, resulting in the droop, whereas the linear approximation assumes that there are always N quads to fail. Therefore the linear approximation can be used as a conservative overapproximation in the short mission time regime.

From this relationship, two important conclusions can be drawn. First, the probability of ensemble loss in the short mission time regime is proportional to the square of the processor failure rate. Therefore minimization of λ is paramount. Second, this probability is inversely proportional to the reconfiguration rate. Although the variation of the probability of system loss with this quantity is not so dramatic as with the former, a second concern clearly is to minimize the amount of time spent in the reconfiguration process. This consideration impacts synchronization decisions and will be discussed in more detail later.

At a mission time of around 5000 hours, the system loss probability curve undergoes a dramatic increase in slope. This is due to the increasing dominance of attrition as the primary failure mode of the ensemble. It is instructive to compare the curve for attrition obtained by a combinatorial calculation to the curve obtained using the full Markov model. Let the reliability of each processor in a quad be denoted by r. Given constant processor

failure rate λ, the reliability of a processor is $r=e^{-\lambda t}$. Since we are conservative and shut the quad down when a single channel fails so as to not be open to an uncoverable Byzantine fault, the probability that a quad shuts down safely is simply one minus the probability that the all four channels of the quad are functional,

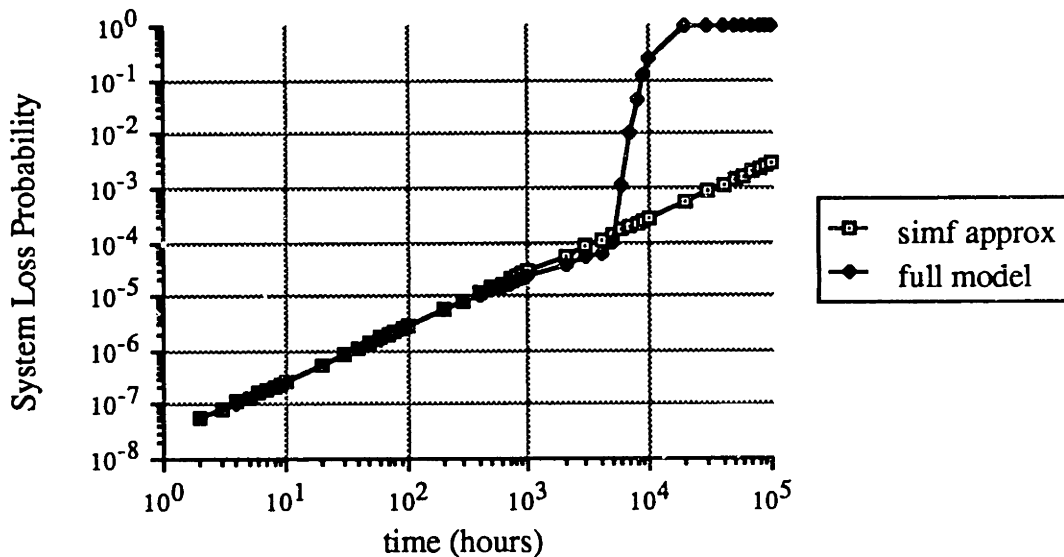


Figure 4.16

Comparison of Linear Approximation to Full Markov Model Result

$$P_{SD/QUAD}(t) = 1 - r^4.$$

The probability that the ensemble suffers attrition is equal to the probability that all quads have safely shut down,

$$P_{SL/E(ATT)}(t) = P_{SD/QUAD}(t)^N = (1 - r^4)^N.$$

This quantity is plotted compared to the full Markov model results in Figure 4.17. Note that there is a wide discrepancy between the values for system loss probability. This is because the combinatorial model does not take into account the predominance of transient failures. In the combinatorial model, it is assumed that hard failures occur at a rate of λ, whereas in actuality they occur at a rate of approximately $(1-f_t)\lambda$. If, in the formulation for

processor reliability r we replace λ with the quantity $(1-f_t)\lambda$ and compare the resulting curve to the result from the Markov model, much closer agreement can be seen (Figure 4.18). This implies that, in future calculations for more complex systems, the probability of system loss in the attrition regime may be calculated using a combinatorial model instead of the much more computationally intensive Markov model, if the appropriate adjustment to the failure rate is made. The error in this calculation will be on the same order of magnitude as the probability that transient failures do not result in a return to the non-faulty state, but instead to a system loss due to simultaneous failure. As can be seen by the relative magnitudes of these probabilities, the relative error is several orders of magnitude below the probability of interest for mission times greater than the time at which the two curves intersect.

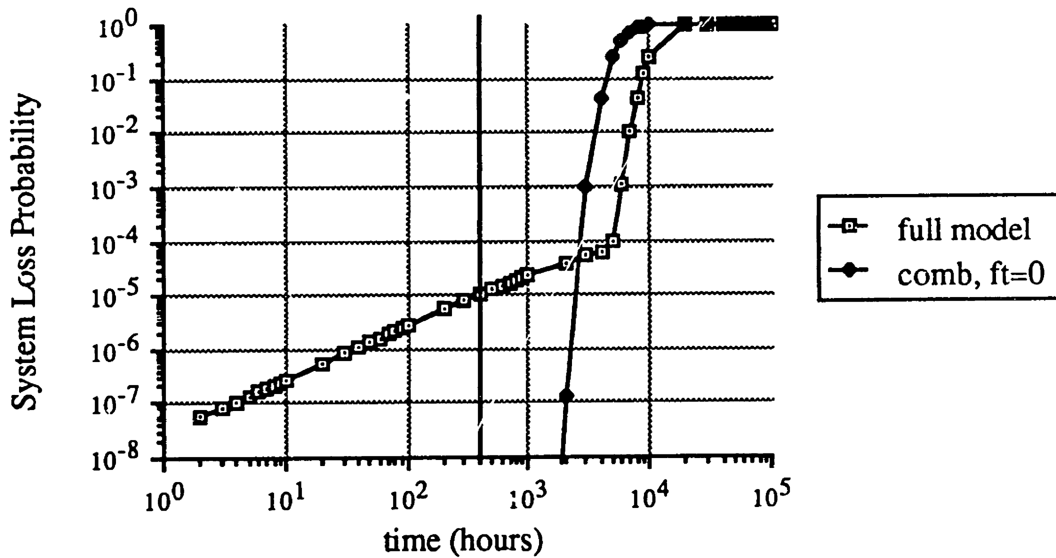


Figure 4.17

Comparison of Unadjusted Combinatorial Model to Full Markov Model Results

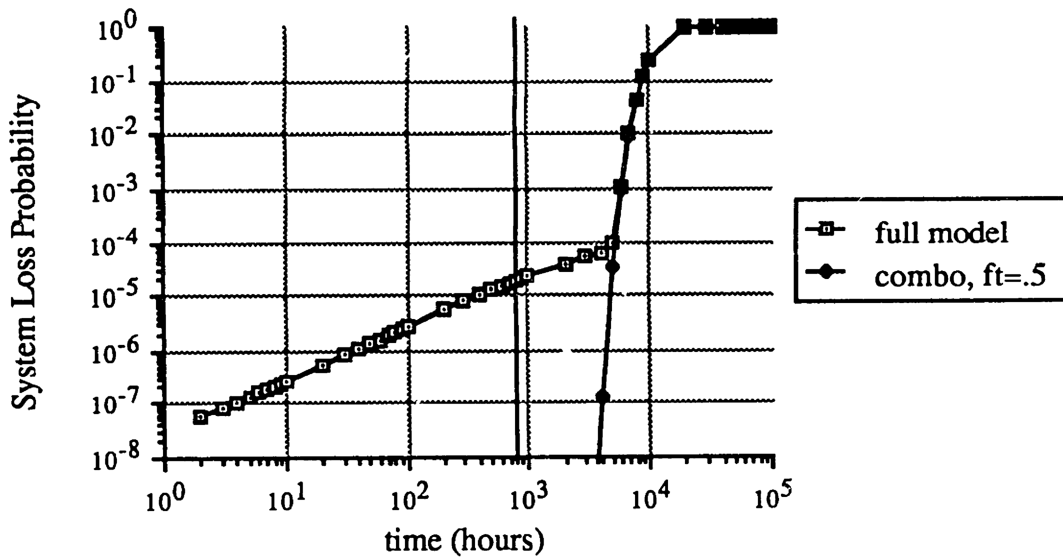


Figure 4.18

Comparison of Adjusted Combinatorial Model to Full Markov Model Results

An approximate composite system loss probability curve may be formed by adding the linear approximation to the probability of system loss due to simultaneous failure and the combinatorial approximation to the probability of system loss due to attrition. Figure 4.19 shows this composite system loss probability curve compared to the system loss probability curve generated by the full Markov model.

The Mean Time To System Failure is given by

$$\text{MTTSF} = \int_0^\infty R(t)\, dt = \int_0^\infty (1 - p_{SL/E(TOTAL)}(t))\, dt .$$

It is assumed that quads which are undergoing reconfiguration are not counted as useful processors. The expected value of the number of processing elements in the ensemble versus mission time may be expressed as

$$E_{PE}(t) = \sum_{i=1}^{N} i\, C(N,i)\, p_1(t)^i\, p_4(t)^{N-i} , \text{ where } C(N,i) = \frac{N!}{(N-i)!\, i!} .$$

This quantity is plotted in Figure 4.20.

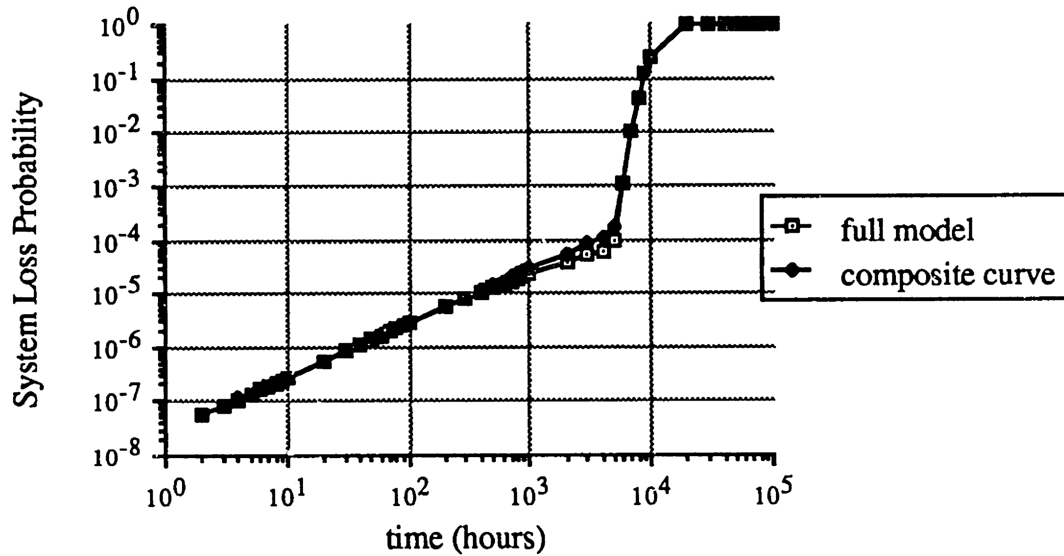


Figure 4.19

Comparison of Composite System Loss Curve to Full Markov Model Results

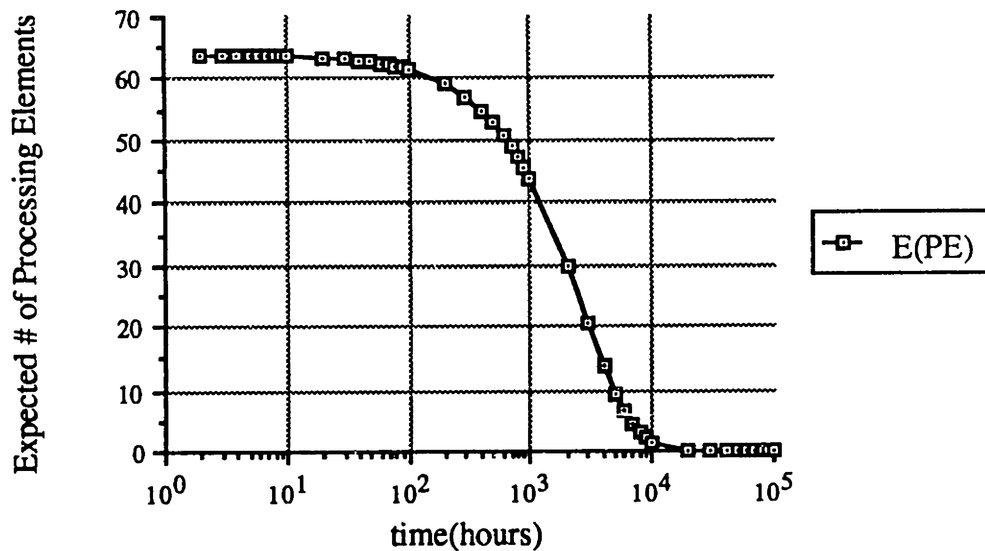


Figure 4.20

Expected Value of Number of FMGs vs. Mission Time for 64-FMG Quadruplex Ensemble

Table 4.3 shows the general list of performance parameters appropriately filled with the parameters for the 64-quad hypercube ensemble.

Parameter	Value
# PE	256
#ports	256*(3+6)
system loss probability	
10 hours	3 E-7
100 hours	3 E-6
1000 hours	2 E-4
10000 hours	0.23
$E_{PE}(t)$	
10 hours	63.8
100 hours	61.6
1000 hours	43.8
10000 hours	1.4
mean time to system failure	9,913 hours
intra-group communication	
distance	1
diagnosability	maximal
anisotropy	none
inter-group interactions	none
inter-group communication	
distance	lg 64
diagnosability	diagnosible links
anisotropy	none
transparency of FT technique to programmer	High transparency if correctly implemented
efficiency of FT technique	Requires quadruplication of processors plus execution of Byzantine Resilient protocols

Table 4.3

Performance Parameters for

64-Quad Processor Hypercube

4.3.2 Analysis of Triply Redundant Fault Masking Architecture

The Markov model of the triply redundant fault masking architecture is shown in Figure 4.21. Before performing the analysis, a simplified version of the redundancy management strategy of the FTP will be outlined. The FTP is composed of six fault containment regions. Three of these regions are processing elements, and the other three are *interstages*,

whose purpose is to participate in input consistency maintenance by replicating input data and distributing it to the processors. There is an interstage for each processing channel for reasons of uniformity and symmetry of design and operation. The triplex FTP can tolerate the failure of any single fault containment region (FCR) be it a processor or interstage. The processor then undergoes reconfiguration which consists of switching out the faulty fault containment region as well as the non-faulty fault containment region associated with the faulty fault containment region. For example, if processor channel A of the FTP shown in Figure 4.11 fails, the triplex will isolate processor channel A from the triplex. In addition, Interstage channel A will also be removed from the triplex. Conversely, if Interstage channel A fails, then processor channel A will also be removed from the triplex. If, during reconfiguration of an interstage or a processor, a second failure of either an interstage or a processor occurs, then triplex loss is assumed to occur. If the reconfiguration procedure is completed prior to a second failure, the triplex is safely shut down.

In the Markov model, State 1 is the unfailed state, in which either no faults have ever occurred or one or more transient processor or interstage faults have occurred and been covered. In State 2, a processor fault has occurred which may be either transient, permanent, or intermittent. Recovery from such a fault occurs with a constant transition rate μ. Of the faults, a fraction f_t of them are transient, in which case the triplex transitions from State 2 back to State 1, and a fraction $1\text{-}f_t$ of them are permanent, in which case the triplex transitions to State 3. If a further fault of a processor or an interstage occurs while the triplex is in State 2, with a transition rate of $2\lambda_p+3\lambda_i$, the triplex undergoes total loss because it cannot tolerate simultaneous FCR failures. In State 3, the triplex has diagnosed its own failure and has safely shut down. Similar to processor faults, interstage faults take the triplex from State 1 to State 4 with transition rate $3\lambda_i$. In State 4 the triplex is reconfiguring, and a second fault of either a processor or interstage, occurring with transition rate $3\lambda_p+2\lambda_i$, results in system failure. If reconfiguration occurs before a second

failure occurs, the triplex enters State 3.

Processor faults occur with a constant rate λ_p. λ_p is parameterized with respect to the number of interprocessor ports, #ports, and intra-processor ports, 3, that the triplex processing element must support as $\lambda_p=\lambda_{p0}(1+f_{port}(\#ports+3))$, where λ_{p0} is the "core" complexity of the processor. A baseline value of 10^{-4}/hour will be used for λ_{p0} [Hopkins78], [Wensley78]. A baseline of 10^{-6} per hour will be used for λ_i [CSDLlore].

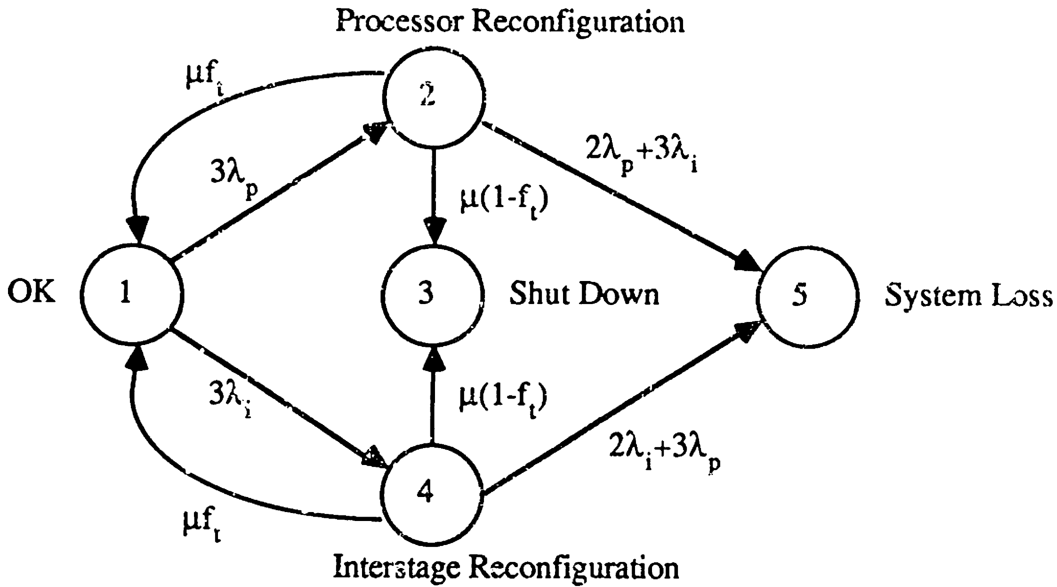


Figure 4.21

Markov Model of Triplex Fault Masking Architecture

Define "system loss due to simultaneous failures" as the probability that any triplex is in State 5,

$$PSL/E(SF)(t) = 1-(1-p_5(t))^N$$

and "system loss due to attrition" as the probability that all triplexes are in State 3,

$$PSL/E(ATT)(t) = p_3(t)^N.$$

The total system loss probability is the probability of occurrence of either of the above two mutually exclusive events

PSL/E(TOTAL)(t) = PSL/E(SF)(t) + PSL/E(ATT)(t), and is plotted in Figure 4.22.

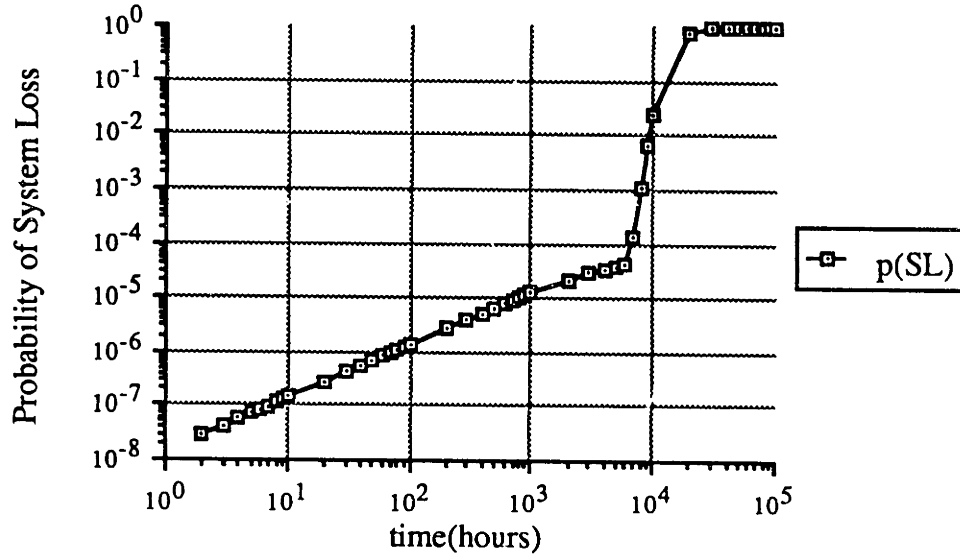


Figure 4.22

Ensemble Loss Probability for 64-PE Triplex Ensemble

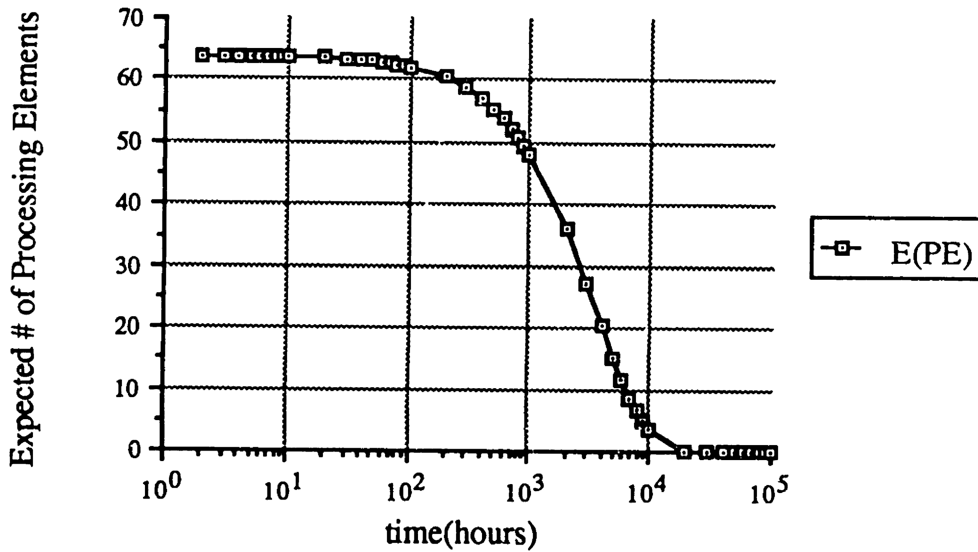


Figure 4.23

Expected Value of Number of FMGs vs. Mission Time for 64-FMG Triplex Ensemble

As usual, it is assumed that triplexes which are undergoing reconfiguration are not

counted as useful processors. The expected value of the number of processing elements in the ensemble versus mission time may be expressed as

$$E_{PE}(t) = \sum_{i=1}^{N} i\, C(N,i)\, p_1(t)^i\, p_3(t)^{N-i}, \text{ where } C(N,i) = \frac{N!}{(N-i)!\, i!}.$$

This quantity is plotted in Figure 4.23.

# PE	192
#ports	192*(3+6) + 192*3
system loss probability	
10 hours	1 E-7
100 hours	1 E-6
1000 hours	1 E-4
10000 hours	2 E-2
$E_{PE}(t)$	
10 hours	63.8
100 hours	62.2
1000 hours	48.1
10000 hours	3.7
mean time to system failure	11,973 hours
intra-group communication	
distance	2
diagnosability	maximal
anisotropy	none
inter-group interactions	none
inter-group communication	
distance	lg 64
diagnosability	diagnosible links
anisotropy	none
transparency of FT technique to programmer	
High transparency	
efficiency of FT technique	
Requires triplication of processors plus execution of Byzantine Resilient protocols, although typically protocols implemented in hardware	

Table 4.4

Performance Parameters for

64-Triplex Processor Hypercube

The Mean Time To System Failure is given by

$$MTTSF = \int_0^\infty R(t)\, dt = \int_0^\infty (1 - p_{SL/E(TOTAL)}(t))\, dt.$$

Table 4.4 shows the general list of performance parameters appropriately filled with the parameters for the 64-triplex hypercube ensemble.

CHAPTER 5
CONNECTIVITY

Previous sections led to the conclusion that the addition of sufficient redundancy greatly increases ensemble reliability. The import of this observation in a parallel system is that the parallel processing elements are available to be used for redundancy *per se*, or as non-redundant processing sites when the increased liability is an acceptable tradeoff for throughput. A parallel system can be used in any of several operating modes, depending on the mission needs at the moment. However, in the previous designs we have not taken full advantage of an important characteristic of parallel systems, namely the superfluity of redundancy and the capability to flexibly group members into new groups as the old groups degrade. Such flexibility of grouping satisfies the dual goals of trading reliability for throughput, mentioned above, and the staving off of attrition by appropriately reconfiguring surviving processors. To understand what conditions must be satisfied to allow such flexible grouping, we must know the requirements upon the way in which members must be physically connected such that the group is a true Fault Masking Group (FMG). It must be known what *contingent connectivity* must be provided between members of different groups such that they have the potential for being grouped together into a FMG at some future time.

A Byzantine Resilient computer must be composed of 3f+1 fault containment regions for the consensus process to be possible in the face of arbitrary failure behavior. Each fault containment region must be capable of participating in the requisite Byzantine Resilient consensus, synchronization, and voting protocols. Most theoretical work has assumed that the participants are arranged in a fully connected network, and in fact all Byzantine Resilient computers constructed to date have utilized such a network. We briefly discuss how such computers provide this connectivity.

5.1 Existing Approaches to Connectivity Provision

In terms of their approaches to solving the connectivity management problem, existing Byzantine Resilient computers fall into one of two categories. The first category includes systems composed of minimally-sized islands of theoretically sufficient connectivity. Examples in this category are the CSDL Fault Tolerant Processor (FTP) [Smith83] and the SRI SIFT (Software Implementation of Fault Tolerance) [Wensley78]. Expansion of systems such as these usually consists of connecting these islands to each other through diagnosable links, as in CSDL Advanced Information Processing System (AIPS) [CSDL84]. The preceding analyses assumed that the ensemble was constructed in this manner. In this approach to system expansion, Byzantine Resilient sites cannot be formed from members of different islands, limiting the configuration options of the system and hence its attrition resilience.

The second category contains systems which provide enough connectivity between their members that a redundant site can be formed from any subset of members. A system which falls in this category is the CSDL Fault Tolerant MultiProcessor (FTMP) [Hopkins78]. This approach works well for a small number of members, but the cost of providing such connectivity becomes excessive as the number of members increases. The two extremes represented by AIPS and the FTMP span a continuum of intermediate approaches in which lies an optimum from the points of view of reliability, efficiency, and cost.

5.2 Embedding Regions of Connectivity in Existing Parallel Topologies

Dolev showed that as long as there are 2f+1 disjoint communication paths from any participant in the protocols to any other, enough correct information can get through to allow a consensus to be reached [Dolev82]. Byzantine Resilient systems need not be fully connected, as long as they are embedded in a network having the required fault containment characteristics and a connectivity of 2f+1, where the *connectivity* of a graph is defined to be

the minimum of the maximum number of disjoint paths between one vertex in the graph and another.

The intuitive explanation for the 2f+1 lower bound, henceforth known as the *connectivity constraint*, is that the graph representing the set of processors can be partitioned by a cut set composed of a number of processors equal to the graph's connectivity. If a majority of these processors can be maliciously faulty, then they can collude and cause one partition of the graph to achieve one consensus value and the other partition(s) to achieve another. Put another way, between any two nonfaulty processors there must exist a majority of disjoint fault-free paths. If it is desired to tolerate "f" simultaneous faults in the system, then there must be 2f+1 disjoint fault-free paths between any two processors, and hence the graph representing the interprocessor topology must have a connectivity of 2f+1.

It is not clear how to implement this theoretically necessary connectivity in a parallel system, and in fact it is absent in many parallel systems. We will therefore initially discuss the possibility of *embedding* sufficient regions of connectivity into existing parallel architectures' topologies to see if they can be modified to support Byzantine Resilient computational sites. We raise four critical issues: the existence of an embedding at all, the efficiency of the embedding, the diagnosability of the embedding, and the capability to reconfigure an embedded system as failures occur.

5.3 Existence of an Embedding

First consider the problem of determining whether it is possible to embed a Byzantine Resilient computational site in a network topology. This problem is the well-known subgraph isomorphism problem, which is known to be NP-complete [Tsai82]. Therefore the question can at best be answered by hypothesizing an embedding and then demonstrating that it meets the connectivity constraint. The following example illustrates this problem.

Consider a parallel system which consists of a number of computational nodes connected in a planar mesh, a common interconnection topology. To form a redundant 1-Byzantine Resilient FMG, 3f+1=4 nodes are needed. This presents no problem, but if one tries to form FMG "p" from the nodes at the corners of a minimal square of the mesh, it proves impossible to provide each member with 2f+1=3 disjoint paths to the other members of p because the paths through the plane must cross (Figure 5.1). The p1 to p4 connection cannot be made without using a node which is already a part of another inter-member path, violating the requirement that the paths be disjoint.

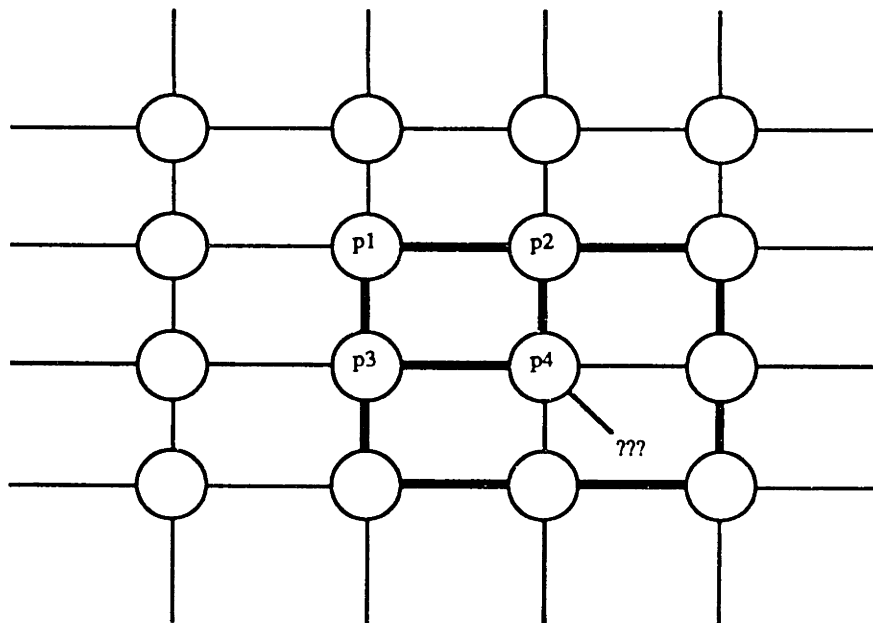


Figure 5.1

Impossibility of Embedding a Byzantine Resilient Group in the Minimal Square of Planar Mesh

It is possible to embed a 1-Byzantine Resilient topology into a planar mesh in other ways. The embedding in Figure 5.2 allows all processors in the mesh to be used as members of a Byzantine Resilient group because the embeddings "tessellate" in the network (Figure 5.3). However, issues such as anisotropy of communication, tessellation of the

sites in the plane, efficiency, reconfiguration, and diagnosability are difficult to address in general, especially under random site failures. This is true of several other common interconnection networks, such as the hypercube, the tree, the Banyan, the Omega, the star, rings, and others. Following sections will discuss these issues.

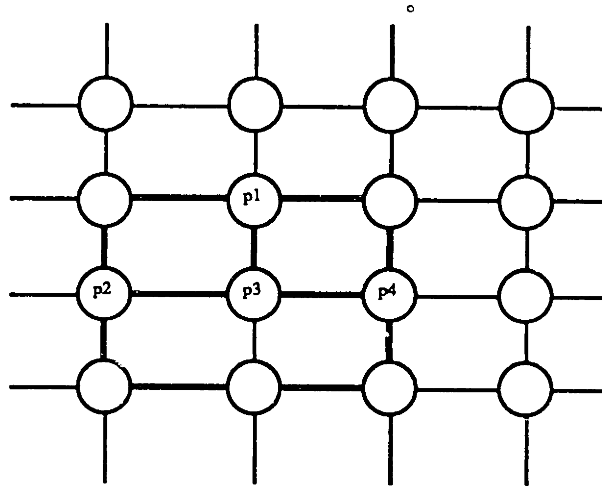


Figure 5.2

One Possible Embedding of Byzantine Resilient Group in Planar Mesh

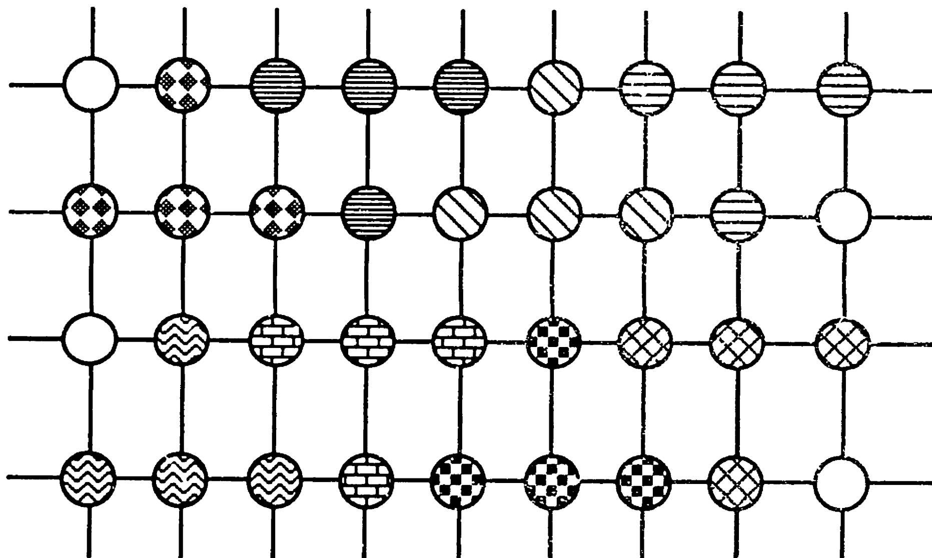


Figure 5.3

Tesselation of Embedded Byzantine Resilient Sites in a Planar Mesh

5.4 Efficiency of the Embedding

In addition to the theoretical sufficiency of the topology, the efficiency of supporting the required rounds of communication must be considered. In the case of the embedding above, each processor is responsible for relaying transactions between members of groups other than the one to which the processor belongs. This can consume significant overhead which would detract from the computational capacity available for execution of the application. If the planar mesh is turned into a toroidal mesh, then the embedding of a 1-Byzantine Resilient computational site into the four corners of a minimal square is possible (Figure 5.4). However, transactions between members of the site must now pass through roughly $\sqrt{N}$ forwarding sites, where N represents the total number of sites in the mesh. The latency required for the messages to transit this path would be enormous in a large ensemble.

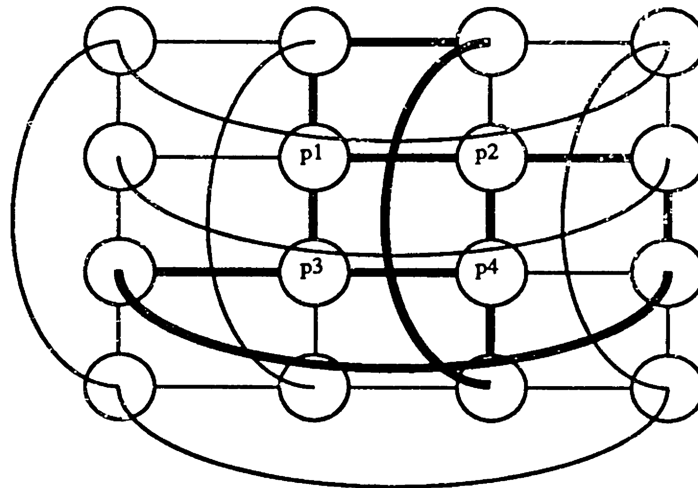


Figure 5.4

One Possible Embedding of Byzantine Resilient Group in a Toroidal Mesh

5.5 Reconfiguration of the Embedding After Failures

Figure 5.5 shows four 1-Byzantine Resilient computational sites embedded in a 16-processor binary hypercube. In such an embedding, reconfiguration under random hardware failures is likely to be difficult, especially in real-time. While it may be possible to find an initial embedding by trial and error, finding an embedding in the random graph corresponding to a system suffering failures is difficult and time-consuming. Moreover, the analysis of the random graphs which arise when the systems they represent undergo random site and link failures is in general intractable from the point of view of Byzantine Resiliency determination, forcing the use of Monte Carlo calculation schemes. Therefore it is unlikely that an ensemble in which Byzantine Resilient groups are embedded will be reconfigurable in real-time, resulting in the loss of this desirable synergism between fault tolerance and parallel systems.

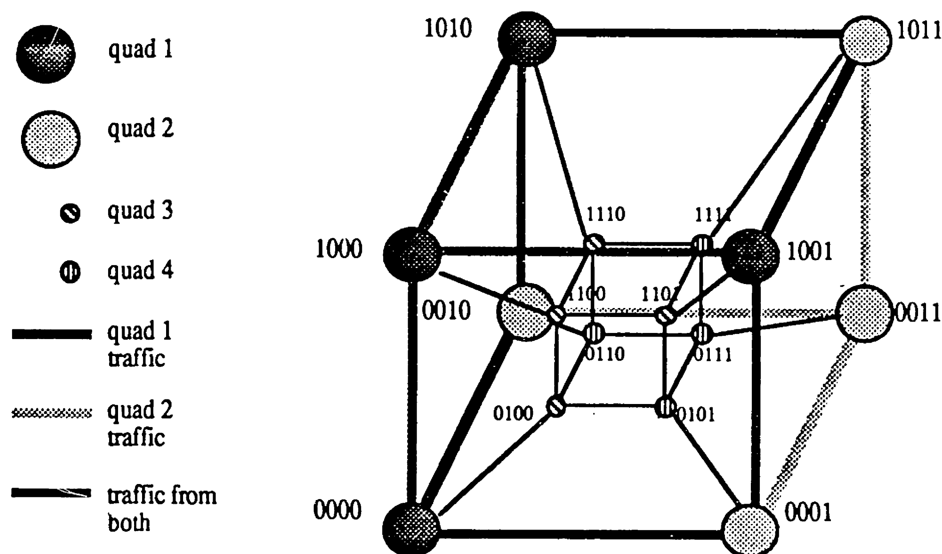


Figure 5.5

Embedding of Four 1-Byzantine Resilient Computing Sites in a Hypercube

5.6 Diagnosability of the Embedded System

Diagnosability of arbitrary failure behavior must be provided if the ensemble is to

survive multiple failures. Otherwise the undiagnosed failures can eventually overwhelm any fault tolerance techniques in use. Even if the theoretically requisite topology could be embedded into an interconnect network, disjoint paths which may of necessity span multiple nodes may cause difficulty in diagnosing arbitrary failure behavior on the part of algorithm participants or forwarding sites. Figure 5.6 shows how the embeddings in the hypercube shown in Figure 5.5 are mapped onto the Byzantine Resilient processing site quad 1. A malicious failure of any processor marked "q2" can seriously disrupt the operation of quad 1. It may even be able to convince quad 1 that the fault is in a member of quad 1. Note that the PMC fault diagnosis model [Preparata67], in which it is assumed that a nonfaulty processor can always correctly diagnose its neighboring processor's state, does not fully capture malicious failure behavior, so the results of this large body of literature are not directly applicable to this problem.

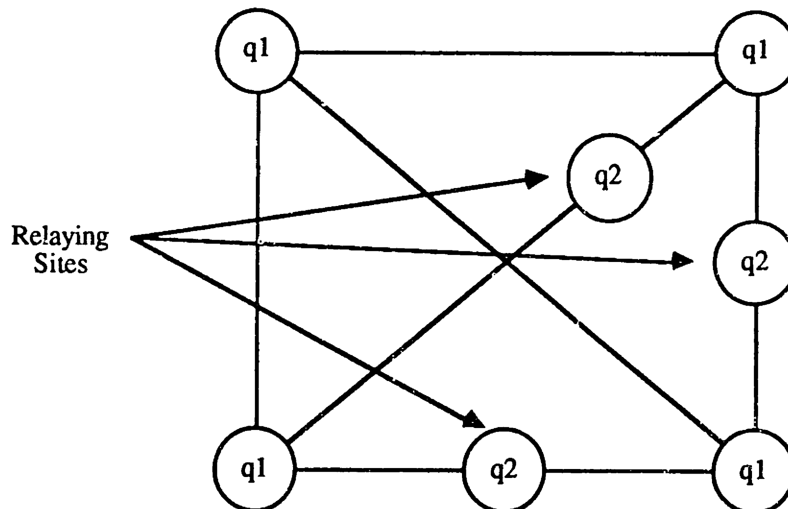


Figure 5.6

Single Byzantine Resilient Processing Site

Embedded in a Hypercube

5.7 Conclusions

The following conclusions summarize this discussion. First, a Byzantine Resilient

parallel system must be designed from the very start to efficiently satisfy a certain minimal topological constraint known as the *connectivity constraint*. Such a fundamental system parameter as interprocessor connectivity would be difficult to alter at a later date. Second, the two currently utilized techniques for providing adequate connectivity in existing Byzantine Resilient systems, namely minimally-sized fully-connected regions and global fully-connected regions, either do not allow the desired degree of reconfigurability or are too expensive for large systems. Third, parallel system interconnection topologies which support Byzantine Resilient embeddings do so with questionable efficacy. Specifically, doubts regarding the existence of the embedding, the efficiency of supporting the protocols in the embedding, the reconfigurability of the embedding, and the diagnosability of the embedding indicate that, with the possible exception of geographically distributed parallel systems, it is not desirable to embed Byzantine Resilient computational sites into networks not specifically designed for that purpose. Therefore we will return to the fully connected processors and study the effects of varying the size of the region of full connectivity.

CHAPTER 6

RELIABILITY ANALYSIS OF CLUSTER-BASED ARCHITECTURES

6.1 Fully-Connected Cluster-Based Description

The fault masking processing element architectures discussed above provided unity coverage of arbitrary hardware faults via bit-for-bit voting because of their capability to satisfy the requirements for exact consensus on input data, member-specific data, and computational outputs in the presence of arbitrary faults, combined with their lack of a single-point failure mode. Systems possessing the former property are referred to as being capable of achieving Byzantine Resilient input consistency.

Consider an architecture constructed according to the principles of the last section (Figure 6.1), of which CSDL Advanced Information Processing System (AIPS) [CSDL84] is an example. In such an architecture, self-contained Byzantine Resilient computational sites are connected to each other by diagnosible links. This architecture provides all the benefits of fault masking enumerated earlier. However, it can be improved upon because, as it stands, it permits no sharing of processors between sites. For example, surviving members of group 1 cannot be combined with surviving members of group 2 to form a new Byzantine Resilient group because there is not adequate *contingent connectivity* between the different groups to allow such freedom of configuration. Similarly, if standby redundancy were used, standby components for group 1 could not be used by group 2. In addition, graded redundancy must be "wired in", in that the maximum redundancy level of any given group must be prespecified since this level could not be increased during operation. Finally, members of a group cannot "help out" members of another by acting as participants in metabolic algorithms such as consensus and synchronization. Therefore the full 3f+1 fault containment regions are required for each group.

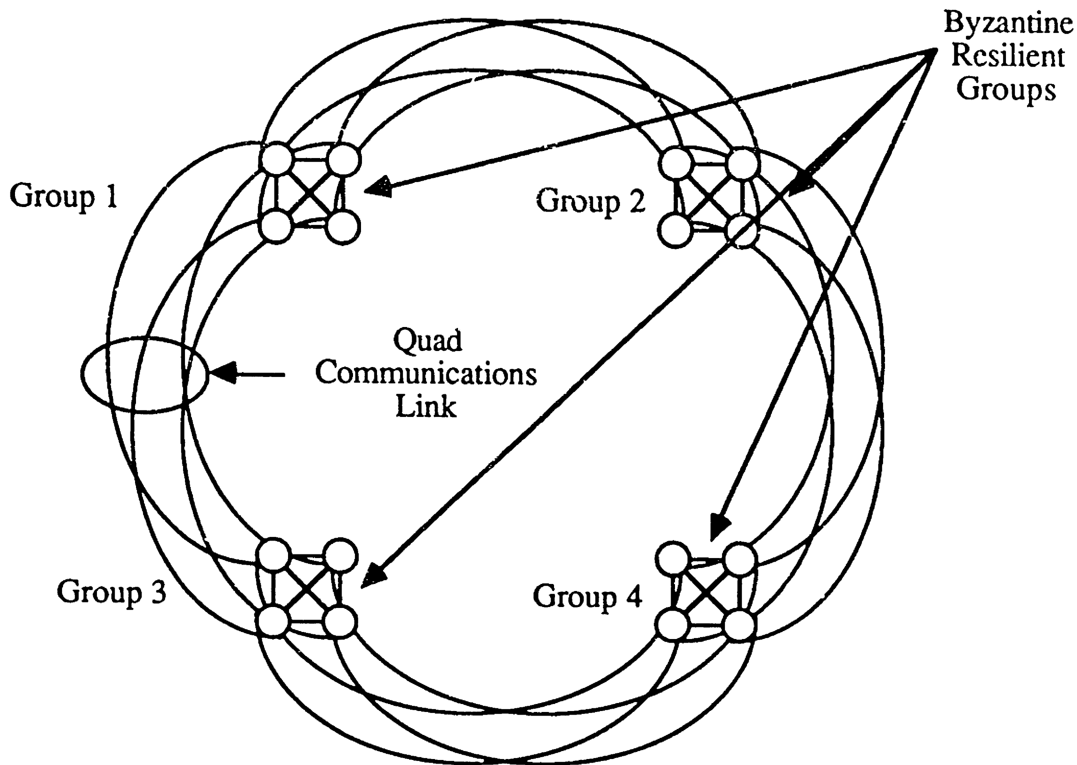


Figure 6.1

Parallel Architecture Composed of Small

Byzantine Resilient Sites Connected by Diagnosible Links

With these drawbacks in mind, the present section investigates schemes of extending Byzantine Resilience to larger aggregates of processors. The approach is to enlarge the regions of contingent connectivity to allow a degree of flexibility in the reconfiguration options available to the system while in operation, thus staving off attrition to some extent. We have argued that embedding such regions of connectivity into existing network architectures have serious drawbacks from the viewpoints of analytical tractability, reconfigurability, diagnosability, efficiency, and other aspects. Therefore we will focus on the use of fully connected regions of processors.

As a first step in this direction, consider enlarging the region of fully connected processors into *clusters* (Figure 6.2). In a cluster, any processor may be combined with any other processors to form a redundant group. Typically, there are enough processors in

a cluster to form several groups at a time. Each processor in the cluster may participate in synchronization and consensus protocols for groups of which it is not a member. We note in passing that the overhead incurred by software execution of such algorithms by the processors is substantial, and it is difficult to provide a transparent implementation of redundancy when the processors must participate in the consistency protocols of groups other than their own [Palumbo86]. Reduction of this overhead and restoration of the transparency will be discussed later when Network Elements are introduced.

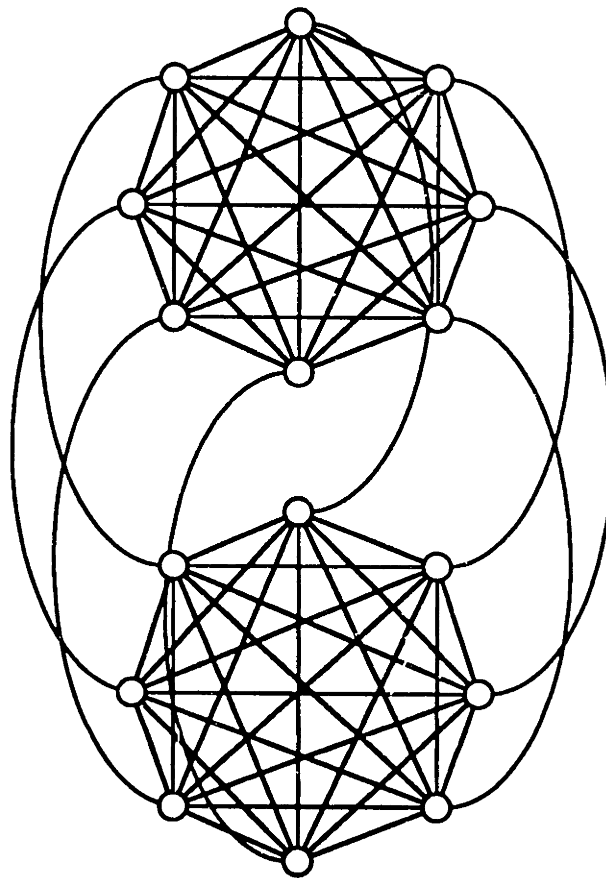


Figure 6.2

Two 8-Processor Clusters

As long as there are at least four non-faulty processors in the cluster, 1-Byzantine Resilient groups can be formed from any three processors, while any fourth processor need

merely participate in the requisite protocols when requested. With $M \geq 4$ processors in a cluster, $\lfloor M/3 \rfloor$ groups can be formed, as opposed to the M/4 groups which can be formed from M processors arranged in unreconfigurable quadruplex groups. Similarly, $\lfloor M/5 \rfloor$ 2-Byzantine Resilient groups can be formed from M processors as long as there are at least 7 non-faulty processors in the cluster, as opposed to the M/7 unreconfigurable 2-Byzantine Resilient groups which can be formed from M processors.

Any three members of the cluster can be grouped together to form a triplex FMG. If there are $M>3$ processors in a cluster at a given time, then $\lfloor M/3 \rfloor$ triplex groups can be formed. The remaining $M-3\lfloor M/3 \rfloor$ processors can be used as spares. These can be shut down under a standby sparing policy, if the physical switching is implemented, resulting in an increase in ensemble longevity. If a triad fails, its members may become available as spares if adequate spares do not exist to restore the triad's redundancy level.

Such reconfiguration capability is practicable only to a point, for two fundamental reasons. After a certain number of failures have transpired and the system has been reconfigured accordingly, the probability of system loss due to attrition is so high anyhow that further reconfiguration capability is of little use [Arlat83]. Second, the complexity associated with the cluster's connectivity increases quadratically with the number of processors in the cluster. Therefore there is an optimal cluster size from the point of view of performability, and the processing elements of the ensemble should be arranged into clusters of this size and the resulting clusters connected via diagnosible links.

The goal of this section, then, is to quantitatively determine the effect on reliability and performance of partitioning the ensemble into a set of fully-connected clusters of given size.

To parameterize the reliability penalty of supporting the intra-cluster communications, it is assumed that the failure rates of the intra-cluster communications ports are reflected in the failure rate of the processors. Therefore let the processor failure rate be increased by $(N_{PE/C}-1)*f_{port}$, where $N_{PE/C}$ is the number of processors in the cluster. To model the

complexity of inter-cluster communications, assume that each processor supports a number of inter-cluster communications ports. The number of such ports and the associated reliability penalty is a function of the inter-cluster connection topology. It will be assumed for the sake of analysis that the clusters are connected in a binary hypercube (Figure 6.3). Therefore there are lg N_C inter-cluster ports, where N_C is the number of clusters in the ensemble. Under these assumptions, the processor failure rate is $\lambda_p=\lambda_{p0}*(1+(N_{PE/C}-1+\lg N_C)*f_{port})$.

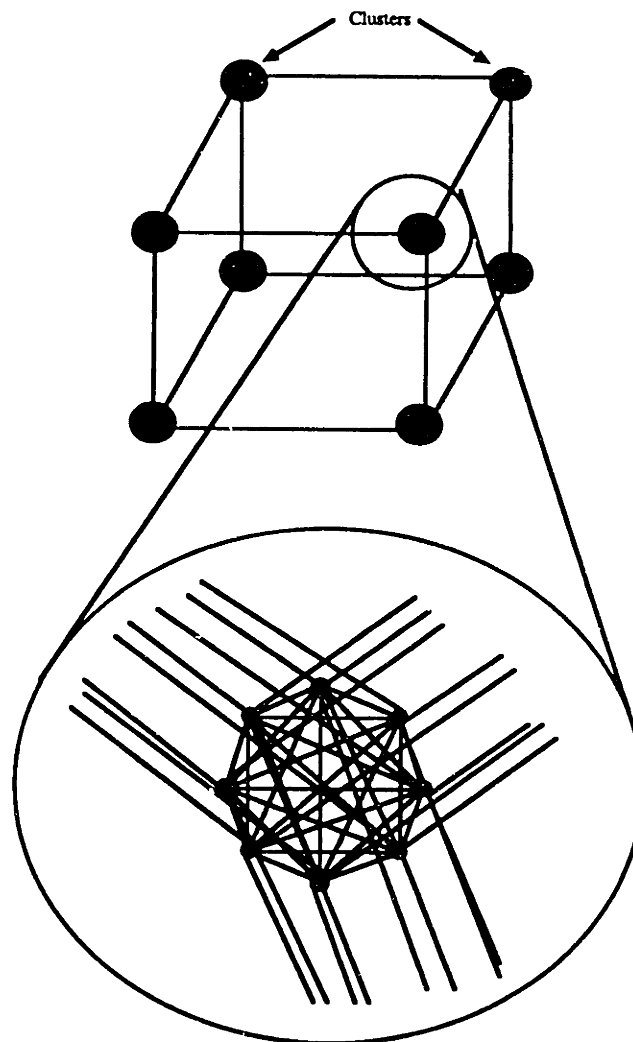


Figure 6.3

8 8-Processor Clusters Connected into a Hypercube

The total number of processors in the ensemble is $N_{PE/C}N_C$. The number of triads in the cluster at mission beginning is $N_{TR/C}=\lfloor N_{PE/C}/3 \rfloor$ and the total number of triads in the ensemble at this time is $N_{TR/E}=N_C\lfloor N_{PE/C}/3 \rfloor$.

The analytical approach will be to formulate a Markov model for each cluster. The solutions for the state probabilities will then be combined to form the usual performance parameters for the ensemble as a whole. These performance measures will be parameterized on the number of processing elements in the cluster.

The Markov model describing the failure behavior of the cluster is given in Figure 6.4 for a cluster having 7 processors (the number 7 was chosen because the model would fit neatly on a single page). The states of the model are arranged in a staircase pattern. As failures occur, the system progresses to states below its current state on the page, and as reconfigurations occur, the system transitions to states to the right of its current state on the page. At the top of the staircase is the unfailed state, in which the cluster either has not yet suffered a processor failure or it has successfully reconfigured after a transient processor failure. In the next state, the processor failure has occurred and the group in which the failure has occurred is in the process of detection, isolation, and reconfiguration. Transitions to this state occur at a rate of $3\lfloor N_{PE/C}/3 \rfloor\lambda_p$. This rate is derived from the assumption that the $N_{PE/C}-3\lfloor N_{PE/C}/3 \rfloor$ spare or unused processors have a failure rate of zero, and that, if there are M processors in the cluster at any given time, then $\lfloor M/3 \rfloor$ triads are in use. If a second fault occurs in any group undergoing reconfiguration, cluster loss due to simultaneous failures occurs. Note that the transition rate from the state in which there are "i" triads undergoing reconfiguration to the system loss state due to simultaneous failures is equal to $3\lambda_p i$, because it is assumed that a triad is using another processor which is a member of another triad to make up the requisite 3f+1 participants in the synchronization and consensus protocols. Thus, if this member of the other triad fails, it is equivalent to a simultaneous failure in the original triad.

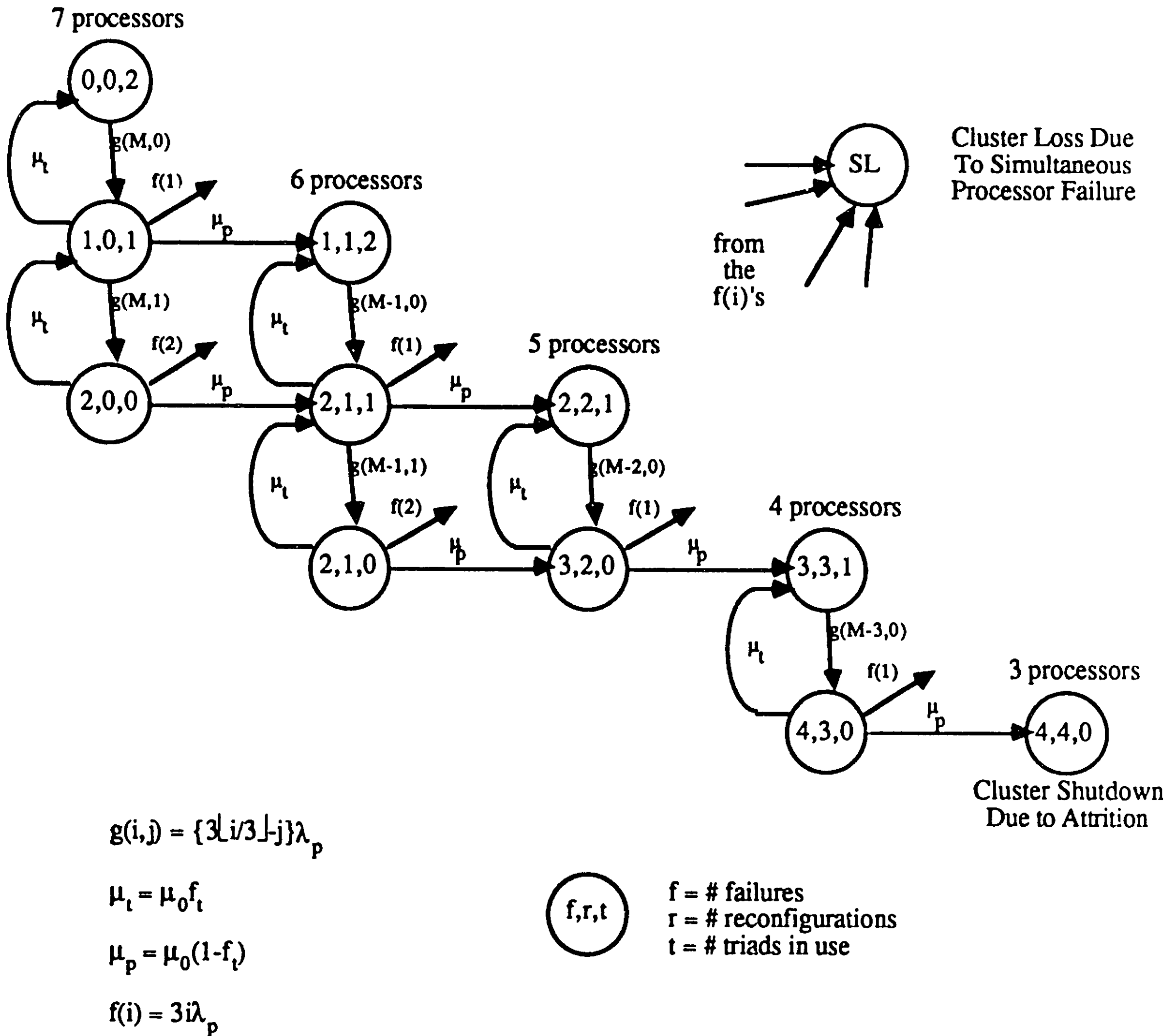


Figure 6.4

Markov Model for 7-Processor Cluster

Reconfiguration transitions occur to one of two states. It is assumed that transitions out of states in which reconfiguration is taking place split according to the proportion of faults that are either transient or permanent/intermittent, based on the assumption that fault latency, diagnosis, and recovery times are the same for both classes of faults. In the case of a transient failure, reconfiguration is back to the previous state; in the case of a permanent or intermittent failure, reconfiguration is to a degraded state in which there is one fewer processor in the cluster, and $\lfloor (N_{PE/C}-1)/3 \rfloor$ groups. From this reconfiguring state the

staircase pattern repeats until enough failures have occurred that there are only 3 processors left, at which point cluster loss due to attrition is declared.

The probability that cluster loss occurs because of simultaneous processor failures is denoted by $p_{SL/C(SIMF)}(t)$. The probability of ensemble loss in terms of probability of cluster loss is equal to one minus the probability that no cluster is in State SL,

$$p_{SL/E(SIMF)}(t) = 1-(1-p_{SL/C(SIMF)}(t))^{N_C} .$$

The probability that the ensemble suffers system loss due to either a simultaneous failure or attrition is the probability of occurrence of either of the mutually exclusive events,

$$p_{SL/E(TOTAL)}=p_{SL/E(ATT)}(t) + p_{SL/E(SIMF)}(t).$$

The mean time to system failure is as usual,

$$MTTSF = \int_0^{\infty} R(t)\, dt = \int_0^{\infty} (1-p_{SL/E(TOTAL)}(t))\, dt .$$

To calculate the expected value of the number of triads in the cluster using this model, it is convenient to number the states in radix-M,M notation. Each state is given a unique numbering I,J, where I represents the number of processor failures which have occurred in that state, and J represents the number of reconfigurations which have occurred in that state. Thus, in State 1,1 one processor failure has occurred and one processor failure has been reconfigured from. Letting $M=N_{PE/C}$ denote the number of processors in a cluster, the number of states in the first column of the model is $\lfloor M/3 \rfloor+1$. These states are numbered from 0,0 to $\lfloor M/3 \rfloor$,0. In column 2 there are $\lfloor (M-1)/3 \rfloor+1$ states, numbered from 1,1 to $\lfloor (M-1)/3 \rfloor$,1. In column 3 there are $\lfloor (M-2)/3 \rfloor+1$ states, numbered from 2,2 to $\lfloor (M-2)/3 \rfloor$,2, et cetera. There are M-2 columns in the model. The first column corresponds to all states in which no failures have been reconfigured, the second column corresponds to all states in which 1 failure has been reconfigured, et cetera. Let $p_{f,r}$ represent the probability of occurrence of the state in which there are f failures and r reconfigurations. Using this numbering scheme, the expected value of the number of triads in the cluster is easily expressed as

$$E_{TR/C}(t) = \sum_{r=0}^{M-3} \sum_{f=r}^{\lfloor \frac{M-r}{3} \rfloor} \lfloor \frac{M-f}{3} \rfloor p_{f,r} .$$

The expected value of the number of triads in the entire ensemble at time t is $N_C E_{TR/C}(t)$.

6.2 Approximate Formulation of the Probability of Ensemble Loss

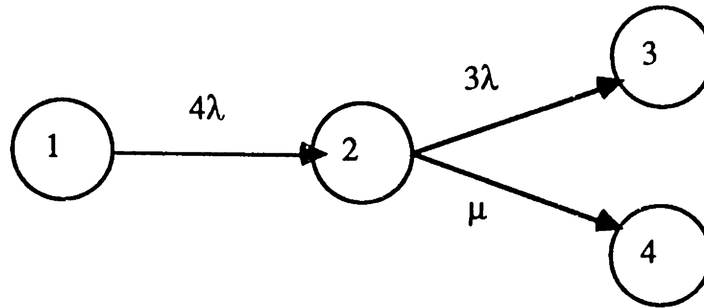
The performance of an architecture constructed according to the above principles will be compared to the others under consideration. Before this can be done, it is necessary to determine a cluster size which provides the architecture with the "best" performance characteristics. This in turn can be greatly facilitated if approximate formulations for system loss probability can be obtained, thus alleviating the computational intensity of the search process.

In an earlier section it was shown that the probability of system loss can be broken up into two main contributors: near-simultaneous failures and attrition. Closed-form approximations to these probabilities were generated and used to help determine reliability optimizations. In the present case, the former approximation is still of use. The latter is not usable because of the implicit sparing mentioned earlier. Therefore to proceed in this area we will revert to the full Markov model.

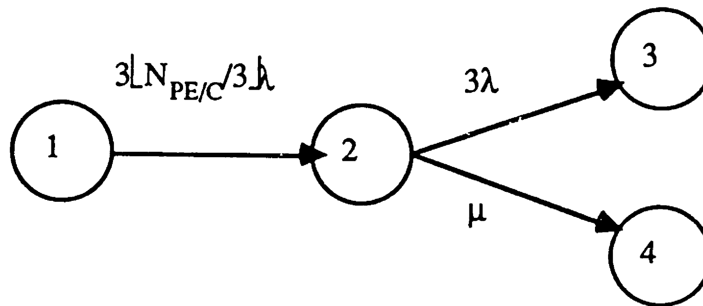
6.3 Simultaneous Failures

In the short time frame, the dominant contribution to system loss probability is due to near-simultaneous failures. This contribution is calculated using the Markov model of Figure 6.5 combined with the analysis of Section 4.3. The main difference between the analysis presented there and the present analysis is that a redundant group consists of three processing elements in the latter as opposed to four in the former. Therefore the constant term is changed from N_C*4*3 to $N_C*3\lfloor N_{PE/C}/3 \rfloor*3$ (see Figure 6.5), to yield

$$p_{SL/E(SIMF)}(t) \approx N_C 9\lfloor N_{PE/C}/3 \rfloor \lambda^2 t/\mu.$$



Linear Overapproximation for a Single Quad



Linear Overapproximation for a Cluster of Processors

Figure 6.5

Comparison Between Linear Overapproximations for Quad and Cluster

The results of this approximation are compared to the results of the full Markov model for a representative cluster size of 8 processors. As can be seen in Figure 6.6, there is close agreement between the two methods in the short term. This provides the expected result that the short-term system loss probability is quadratically proportional to the processor failure rate.

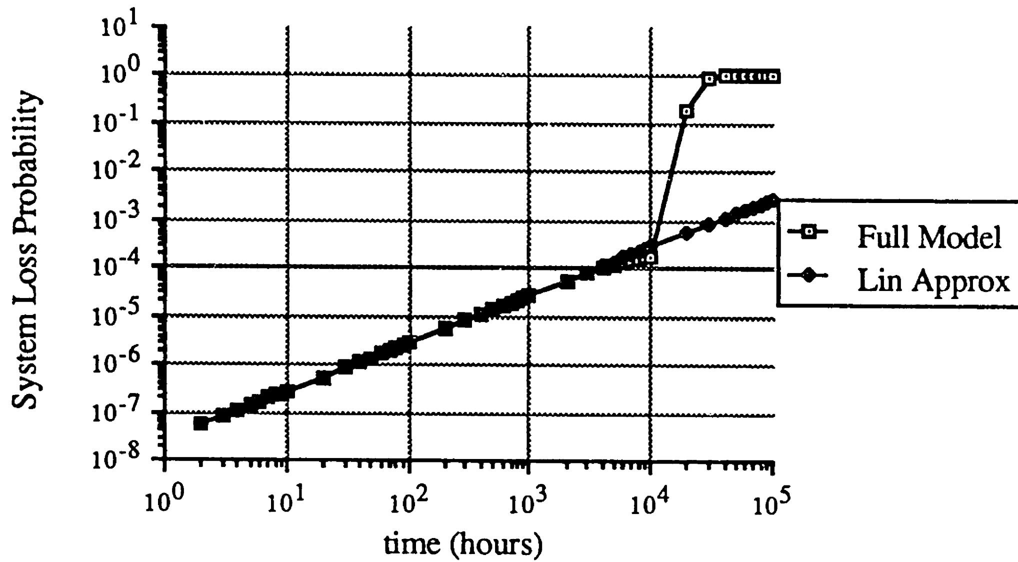


Figure 6.6

Comparison of Linear Approximation and Full

Markov Model Result for 8-Processor Cluster

Several techniques may be used to reduce the failure rate of a processing element. Here we will focus on the effect of varying the topology in which the processing elements are connected upon processor failure rate. The more ports that a processing element must support, the higher its failure rate. The number of ports however scales differently with the inter-cluster topology. For example, in a bus-oriented topology, the number of inter-cluster ports on a processing element is constant and equal to 1. In a hypercube-like intercluster topology, the number of ports on a processor scales logarithmically with the number of clusters. In a fully connected topology in which each cluster is connected to each other, the number of ports per processor scales linearly with the number of clusters in the ensemble.

In the case of the bus-oriented topology (Figure 6.7), the failure rate of the processor can be parameterized as $\lambda_p = \lambda_{p0}(1 + (N_{PE/C}-1)f_{port} + f_{port})$. The first term in this expression is attributed to the intrinsic or "core" complexity of the processor, the second term $(N_{PE/C}-1)f_{port}$ is attributed to the $N_{PE/C}-1$ intra-cluster ports, and the final term is due

to the processor's port to the inter-cluster bus. In this case to minimize λ_p one must minimize $N_{PE/C}$. Therefore the clusters should be made as small as possible. This is the approach taken in the AIPS architecture.

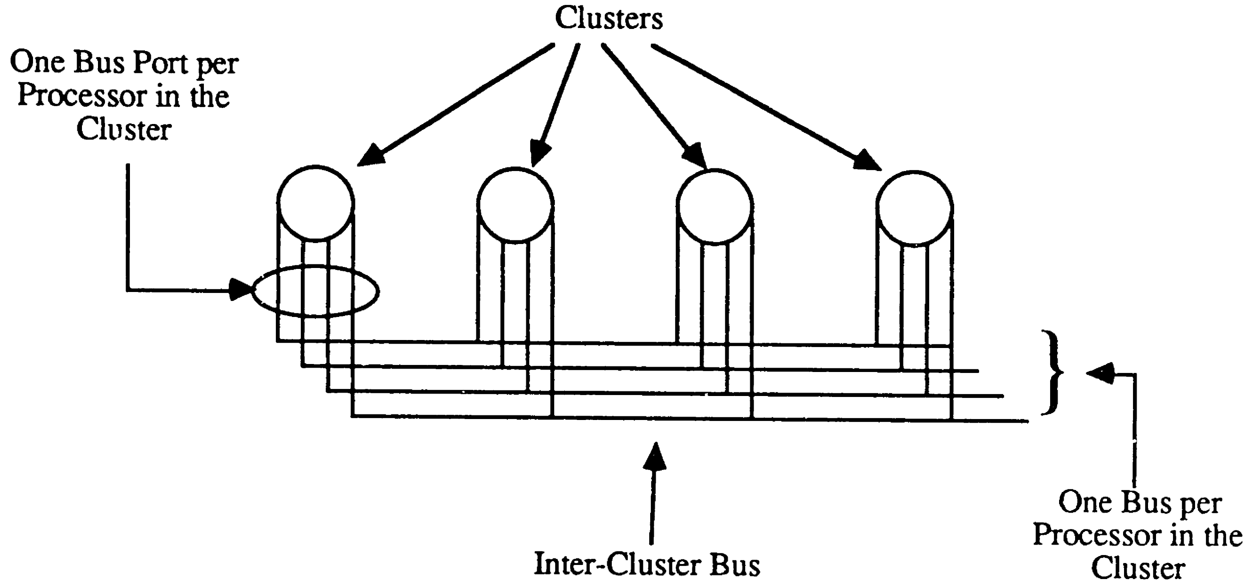


Figure 6.7

Bus-Oriented Inter-Cluster Topology

In the case of the hypercube topology (Figure 6.3), the failure rate of the processor can be expressed as $\lambda_p = \lambda_{p0}(1 + (N_{PE/C}-1)f_{port} + \lg(N_C)f_{port})$, where the last term $\lg(N_C)$ has been added to account for the inter-cluster ports that the processor must possess. The processor failure rate is therefore linearly proportional to the quantity

$$\lambda_p \propto N_{PE/C} + \lg N_C = \frac{N_{PE}}{N_C} + \lg N_C .$$

Differentiating λ_p with respect to N_C and equating to zero,

$$\frac{\partial \lambda_p}{\partial N_C} = \frac{-N_{PE}}{N_C^2} + \frac{1}{\ln 2}\frac{1}{N_C} = 0 ,$$

gives the result that in order to minimize λ_p one must have $N_{PE}/N_C = N_{PE/C} = 1/\ln 2 \approx 1.44$. This is a minimum because

$$\frac{\partial^2 \lambda_p}{\partial N_C^2} = \frac{2N_{PE}}{N_C^3} - \frac{1}{\ln 2\, N_C^2} \;\alpha\; 2N_{PE/C} - 1.44 \;> 0, \text{ since } N_{PE/C} > 3.$$

Since there must be at least 4 processors in each cluster, the cluster size should be set at its minimum value, i.e., that $N_{PE/C}=4$.

In the final case, each processor in a cluster has a dedicated port to its corresponding processor in each other cluster (Figure 6.8). This results in an expression of the processor failure rate of $\lambda_p = \lambda_{p0}(1 + (N_{PE/C}-1)f_{port} + (N_C-1)f_{port})$. In this case the processor failure rate is proportional to the quantity

$$\lambda_p \;\alpha\; \frac{N_{PE}}{N_C} + N_C .$$

Differentiating λ_p with respect to the number of clusters in the ensemble and setting equal to zero,

$$\frac{\partial \lambda_p}{\partial N_C} = \frac{-N_{PE}}{N_C^2} + 1 = 0$$

yields the result that, to minimize the processor's failure rate in the fully connected case one must have $N_C=\sqrt{N_{PE}}$. That this results in a true minimum can be seen by taking the second derivative of the processor failure rate with respect to the number of clusters, i.e.,

$$\frac{\partial^2 \lambda_p}{\partial N_C^2} = \frac{2N_{PE}}{N_C^3} > 0 .$$

6.4 Optimization of Attrition Resilience

The above analysis showed that resilience to near simultaneous failures could be maximized via the reduction of the processor failure rate. This implies that the clusters should have a minimum size, in most cases. However, with reduction of the cluster size, the resilience of the ensemble to attrition is reduced because there is not adequate contingent connectivity between clusters to allow the formation of Byzantine Resilient processing sites from members of different clusters as failures reduce the number of processors in the clusters. Therefore it is desirable to increase the cluster size to some extent to allow such

sharing of resources. This can be done only at the expense of increasing the processor failure rate, so a design tradeoff must be performed on the relative merits of short-term reliability and attrition resiliency. This tradeoff was performed using the results of the full Markov model presented in Figure 6.4.

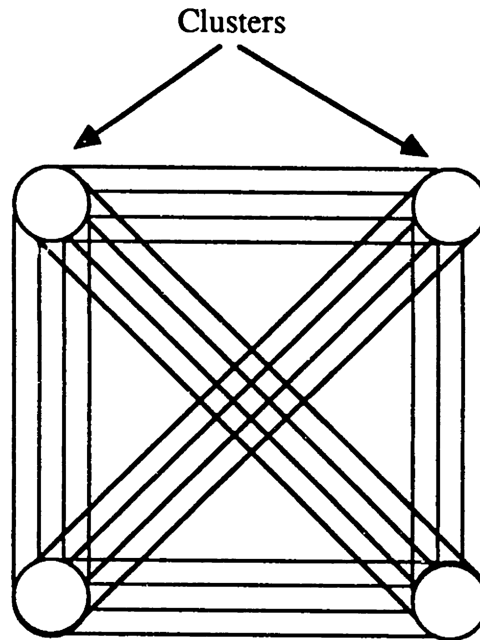


Figure 6.8

Fully Connected Clusters

6.5 Analytical Results and Conclusions

Calculations of system loss probability and expected value of the number of processing elements in the ensemble were performed using the full Markov model for several cluster sizes. The results are plotted in Figures 6.9 and 6.10. The following conclusions may be drawn.

In the short mission time frame, the configuration which minimizes λ_p results in the highest reliability. In this time frame, $p_{SL/E}(t)$ is quadratically proportional to λ_p. Of all the inter-cluster interconnection topologies except the fully connected case, λ_p was monotonic with respect to cluster size, implying that smaller clusters have a higher short-term

reliability. In the fully connected inter-cluster topology, λ_p was minimized when the number of clusters was equal to the square root of the number of processing elements, $N_C=\sqrt{N_{PE}}$. For purposes of uniformity of comparison of this architectural approach to the others under consideration, the hypercube inter-cluster topology will be used.

In the long mission time frame, there is a tradeoff between the processor complexity and the ensemble reliability. Smaller clusters, although more reliable in the short term, have a worse attrition resiliency than slightly larger clusters. As the cluster size is increased beyond 8 processors per cluster, increases in attrition resiliency are less dramatic. The higher complexity involved in supporting the necessary interprocessor ports increases the failure rate of the processor quadratically with respect to the cluster size. This corroborates the previous assertion that the benefits of reconfigurability can be diminished by the complexity associated with providing it.

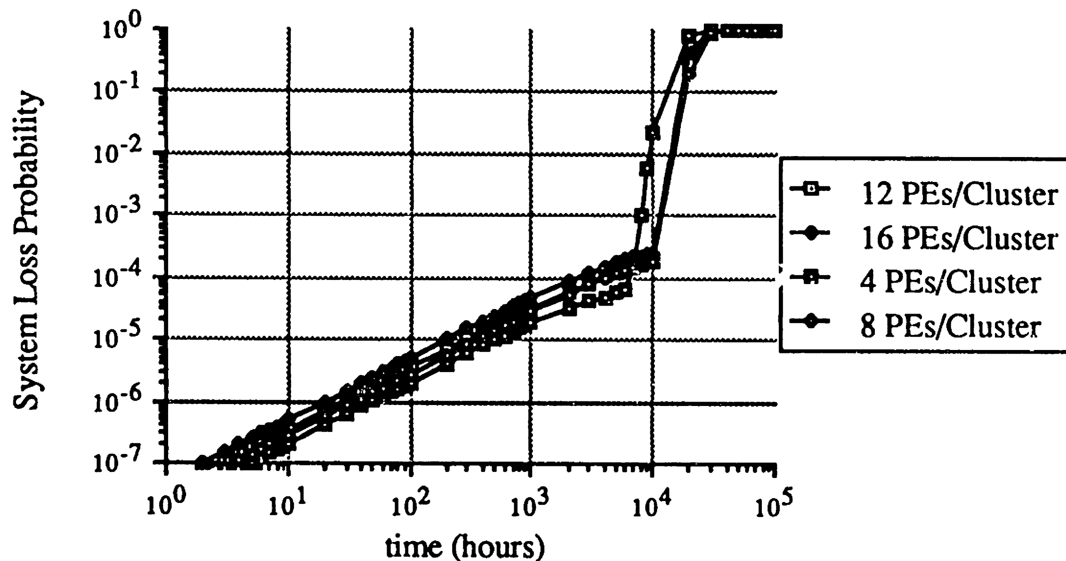


Figure 6.9

Ensemble Loss Probability for 64-FMG Fully Connected

Cluster-Based Ensemble

From examination of the curves of system loss probability and expected value of the number of processing elements, a cluster size of 8 will be chosen for comparison with the other approaches under consideration.

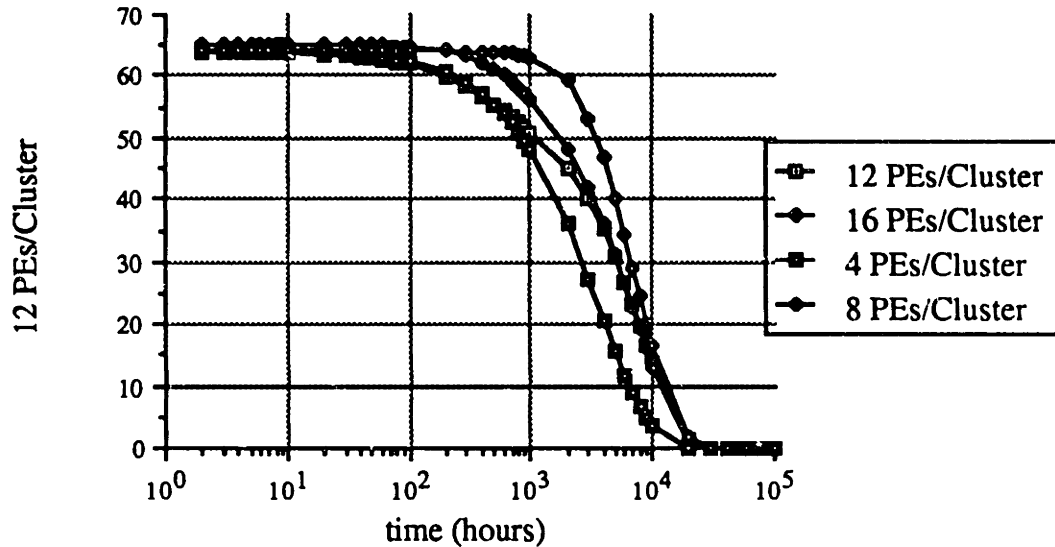


Figure 6.10

Expected Value of Number of FMGs for 64-FMG Fully Connected Cluster-Based Ensemble

The generalized performance parameters for the ensemble composed of 32 8-processor clusters connected in a hypercube are presented in Table 6.1.

# PE	256
#ports	32*8*(7+6)
system loss probability	
10 hours	3 E-7
100 hours	3 E-6
1000 hours	3 E-5
10000 hours	2 E-4
E (t)	
PE	
10 hours	64
100 hours	64
1000 hours	63
10000 hours	17
mean time to system failure	19,106 hours
intra-group communication	
distance	1
diagnosability	maximal
anisotropy	none
inter-group interactions	each group uses 1 member from another group as co-participant in protocols
inter-group communication	
distance	1 (intra-cluster) 1 + lg 4 (extra-cluster)
diagnosability	good (intra-cluster)
anisotropy	some, intra-cluster vs. extra-cluster
transparency of FT technique to programmer	
High transparency if implemented correctly	
efficiency of FT technique	
Requires processor triplication plus processor execution of consensus, voting, synchronization	

Table 6.1

Performance Parameters for 64 FMG Fully Connected Cluster-Based Ensemble[1]

---

[1] 32 clusters, 8 PEs per cluster.

Chapter 7
Reliability Analysis of Network Element-Based Architectures

7.1 Network Element-Based Cluster Analysis

In the fully connected cluster-based architecture introduced in the previous section each processor in the cluster participated in the synchronization, voting, and consistent ordering protocols which must be executed by any Byzantine Resilient system. One difficulty associated with this approach is that the use of Processing Elements to execute such protocols in even a small multiprocessor (e.g., SIFT) has been shown to result in excessive utilization of the Processing Elements and a comcomitant low availability of the Processing Elements' throughput for the execution of the applications programs. In [Palumbo86], it is stated that "Three functions of the SIFT operating system can benefit directly from hardware support. They are the implementation of the synchronization algorithm, the vote algorithm, and interactive consistency." The operating system overhead of the implementation discussed in [Palumbo86] consumed about 60% of the processors' raw throughput. In the MAFT (Multicomputer Architecture for Fault-Tolerance [Walter85]) computer these functions were performed by dedicated programmable processors called Operations Controllers. One Operations Controller serviced the needs of each processor in a MAFT. It was estimated in [Walter85] that the execution of these functions by the Operations Controller was two orders of magnitude too slow for a usable system. They have since embarked on a more hardware-intensive implementation of the Operations Controller. It can be expected that substantially more overhead would be involved in executing the protocols for the reconfigurable cluster depicted in Figure 6.2, which is significantly larger than the 5-processor SIFT or the 5 (or so) processor MAFT. Hardware implementations of the protocols have better performance, with the actual processor overhead devoted to fault tolerance-specific functions in the neighborhood of 5% [Smith83] to 30% [Hopkins78].

In the field of parallel processing a similar phenomenon has been observed by some users of the Intel iPSC, a binary hypercube-based multicomputer. In the iPSC, routing of the inter-processor messages is obtained by interrupting a relaying processor upon receipt of a message on an input link and forcing the processor to execute the routing algorithm and write the message to the appropriate output link. This scheme has been observed to consume substantial processor capacity and result in high message latency times. Special routing chips have been developed in part to correct this deficiency [Dally86].

An additional consideration in the case of Byzantine Resilient Processing is the high cost of complete inter-processor connectivity. It was shown in an earlier section that, from the point of view of ensemble reliability, reconfigurability via contingent connectivity has diminishing returns, especially when the reliability penalty of supporting such reconfigurability is taken into account. Analogous to this line of reasoning it is reasonable to ask whether the degree of inter-processor connectivity, of which the completely connected case is the extreme, exhibits a similar characteristic. It is not clear that N processors must be connected in a N-fold point to point network where the use of a relatively modest region of theoretically adequate connectivity which is shared by a large aggregate of processors might present a cost-effective tradeoff.

Finally, a hallmark of Byzantine behavior is the phenomenon of lying inconsistently, that is, making one recipient of a message believe one thing and another recipient believe another. If a Byzantine Resilient arrangement of highly reliable elements was interposed between the less reliable processors such that the processors were physically incapable of lying inconsistently, the likelihood of inconsistent Byzantine processor faults could be eliminated. Not only would the system reliability be improved via reduction of the degree of connectivity and its associated complexity, but a Byzantine Resilient group could be formed from 2f+1 processors instead of the canonical 3f+1, as long as the number of interposed elements exceeded 3f+1.

These considerations lead to the investigation of the use of specialized hardware

components to execute these frequently occurring, performance-critical, fault tolerance-specific functions. This approach is common in computer architecture. Its general outline is to identify frequently executed performance-critical operations which are suitable for a hardware-intensive implementation, perform a tradeoff analysis, and make the appropriate design decision. Classical examples include floating point arithmetic operations and virtual to physical address translations, operations characterized by high frequency, strong impact on system performance, and amenability to hardware implementation. The above observations point to a similar implementation of the synchronization, consistency, voting, and consistent ordering functions in a Byzantine Resilient cluster of Processing Elements.

Consequently, this section explores the reliability and performance ramifications of the use of *Network Elements* (NE) to construct a fully connected, Byzantine Resilient "hard core" for the cluster. The function of the Network Element is to act as host to one or more subscriber processors, for which the Network Element aggregate performs Byzantine Resilient synchronization, consensus, voting, and consistent ordering functions by proxy. In effect, the purpose of the Network Element aggregate is to "broker" Byzantine Resilient consistency for the Processing Element subscribers, leaving the latter to carry on with their proper task of computation. In addition, the Network Elements mask inconsistent processor behavior from the remainder of the cluster. Discussion of the detailed operation of Network Elements will be deferred until a later chapter.

The overall approach of this section is to generate an architecture based on the above observations. A composite reliability model of this architecture will be defined and its approximate applicability to the architecture will be demonstrated. Then, this reliability model will be used to determine an optimal arrangement of the resources for a given ensemble throughput, as measured by probability of ensemble loss. Specifically, the optimal number of Network Elements in a cluster, Processing Elements per Network Element, and clusters in an ensemble will be calculated. In the process, it will be shown that the use of Network Elements can result in an enhancement in system reliability.

Figure 7.1 shows one possible arrangement of Network Elements and Processing Elements into a 16-processor, 4-Network Element *cluster*. The salient features of such a cluster can be characterized by two parameters: the number of Processing Elements per Network Element, denoted $N_{PE/NE}$, and the number of Network Elements per cluster, denoted $N_{NE/C}$. The number of processors in the cluster $N_{PE/C}$ is $N_{PE/C}=N_{PE/NE}N_{NE/C}$, and the total number of Processing Elements in an ensemble composed of N_C clusters is $N_{PE/C}N_C$.

A Network Element and its associated Processing Elements comprise a primary fault set or fault containment region, in the sense that, given adequate Network Elements and Processing Elements in the cluster to form Byzantine Resilient computational sites, the cluster can tolerate Byzantine failures of a given number of the Network Elements and Processing Elements. These primary fault containment regions are depicted in Figure 7.1. Since the Network Elements and Processing Elements are in the *same* primary fault set, a Network Element failure can affect proper operation of a subscriber Processing Element, although the converse is not necessarily the case. Secondary fault tolerance techniques which are not Byzantine Resilient may be employed for intra-fault set tolerance of likely failure modes of the Network Element and Processing Element, but the use of these techniques will not be modeled. These boundaries are indicated in Figure 7.1 as "secondary fault containment regions". Since each Network Element and the associated Processing Elements form a fault set, it is necessary when forming Byzantine Resilient computational sites to select a member from each fault set, i.e., each channel of a redundant processing group must subscribe to a different Network Element. This lack of configuration flexibility is the primary price paid for the use of a small, commonly shared region of connectivity supported by Network Element consistency brokers. Figure 7.2 shows one possible configuration of the cluster shown in Figure 7.1. If a member of quad Q1 fails, as in Figure 7.3, then simplexes S5, 6, or 7 may be reassigned to Q1 to allow it to regain its degree of redundancy.

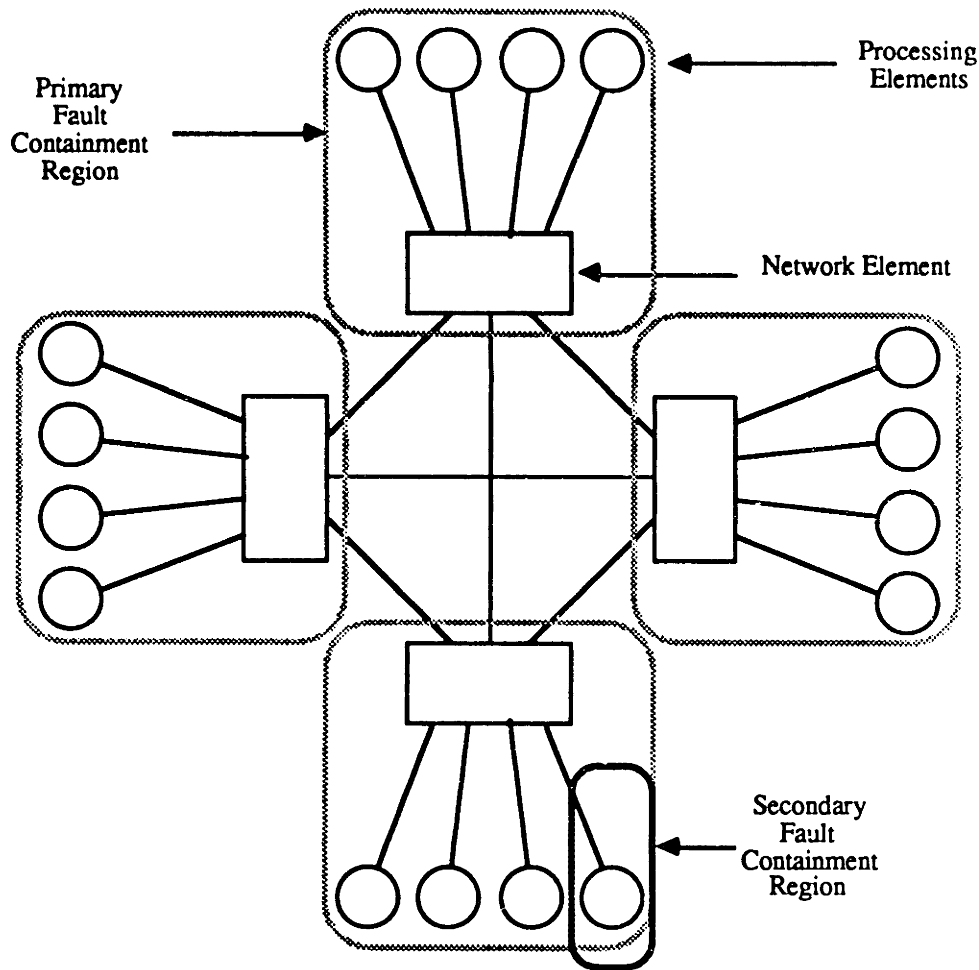
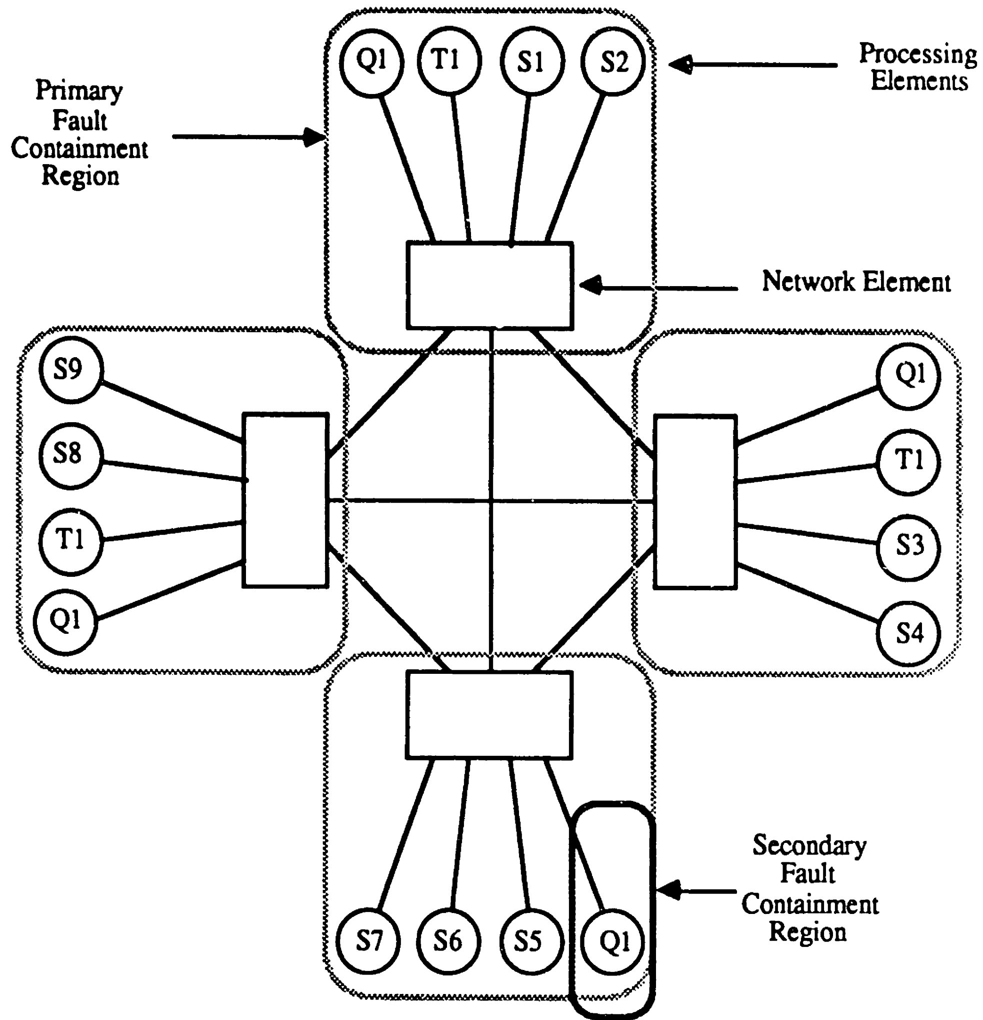


Figure 7.1

16-Processor Cluster Using 4 Network Elements

The Network Elements in the cluster are fully connected to each other via point to point communications links, which also serve as physical fault isolation barriers. Also connected to each Network Element is a port for each subscriber Processing Element. For simplicity it is assumed that the inter-Network Element ports are identical to the Network Element-Processing Element ports. The failure rate of the Network Element can be expressed as $\lambda_{NE}=\lambda_{NE0}(1+f_{port}(N_{NE/C}-1+N_{PE/NE}))$, and the Processing Element failure rate as $\lambda_p=\lambda_{p0}(1+f_{port})$. Note that λ_p is now independent of the size of the cluster.



Q1 Quad 1
T1 Triad 1
S1-S9 Simplexes 1-9

Figure 7.2

Possible 16-Processor Cluster Configuration

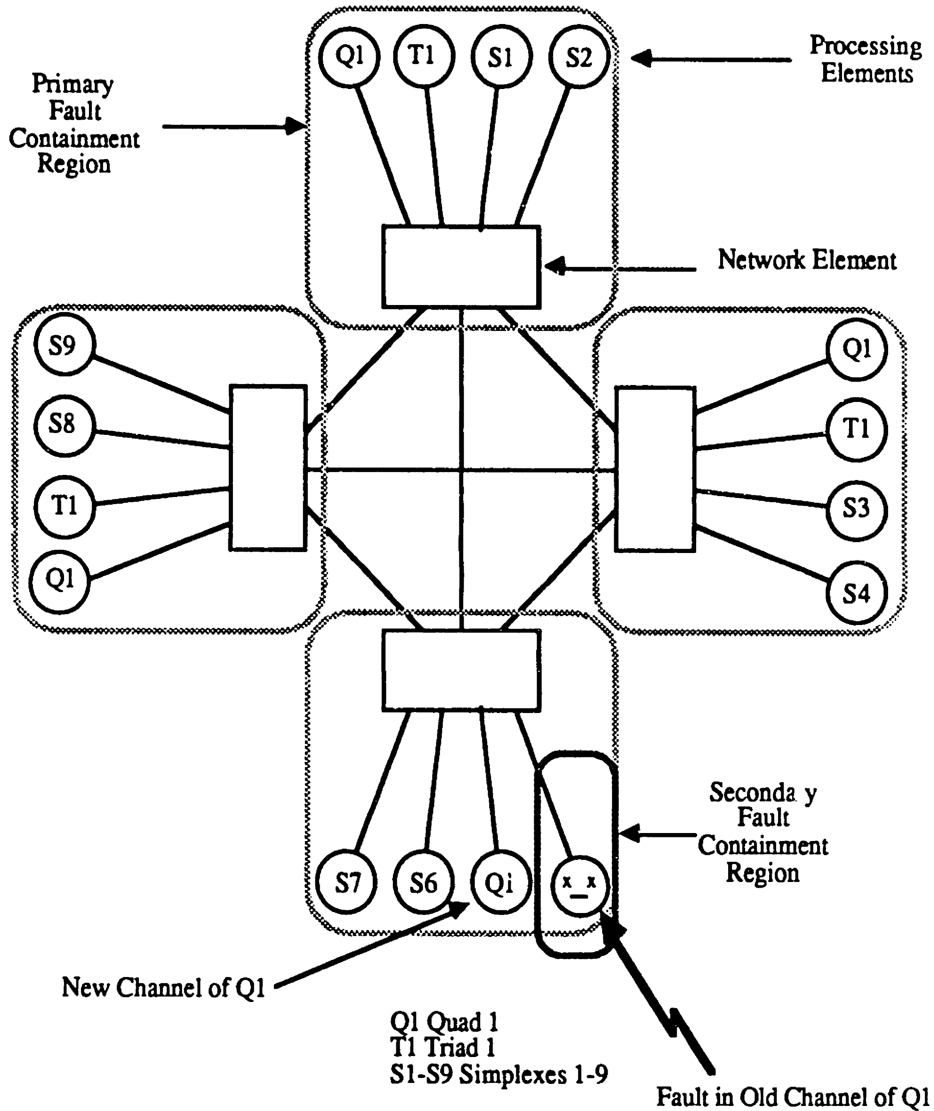


Figure 7.3

Reconfiguration of a Network Element-Based Cluster

Further advantages of this approach of consistency brokering and its operational details will be discussed in the subsequent sections on synchronization, consensus, and total ordering. However, it is germane at this point to briefly address how to construct an entire ensemble out of clusters of a given size. Because of limited Network Element execution speed and communications link bandwidth, it is unlikely that all of the processors of an ensemble can be efficiently supported by a single cluster. An ensemble will probably be formed from a set of clusters, the size of each of which is optimized for some performance

parameter. A later section will show a calculation of this optimal size. Given clusters of such an optimal size, it is necessary to connect them together. Of the several possible ways to physically implement this connection, the use of specially designated *I/O Elements* (IOE) will be discussed here. The reason for this approach is that the Network Elements are likely to be highly optimized for the efficient execution of the consensus, synchronization, and total ordering algorithms. This is their primary purpose. The techniques suitable for intra-cluster communication are not likely to be similar to those employed for inter-cluster communication, which may use encoding, temporal redundancy, and other means to tolerate the bursty fault modes associated with long communications lines. Furthermore, routing, blocking, and deblocking functions may be employed for inter-cluster communications, which may in turn depend on the inter-cluster topology. Since it is undesirable to compromise the throughput of the Network Element aggregate by forcing it to perform these tasks, a specialized component has been selected. For uniformity, it is assumed that one IOE subscribes to each Network Element and its main function is inter-cluster communication. These IOEs are grouped into a redundant IOE Group (IOEG), which operates a synchronous inter-cluster message processing algorithm which will be detailed in a future chapter on the operation of the proposed architecture.

Figure 7.4 shows how two 16-Processing Element clusters may be connected together using IOEs. Further operational details will be presented in a subsequent section on system expansion. With the above as an architectural overview, we will proceed with the reliability calculations for the Network Element-based ensemble.

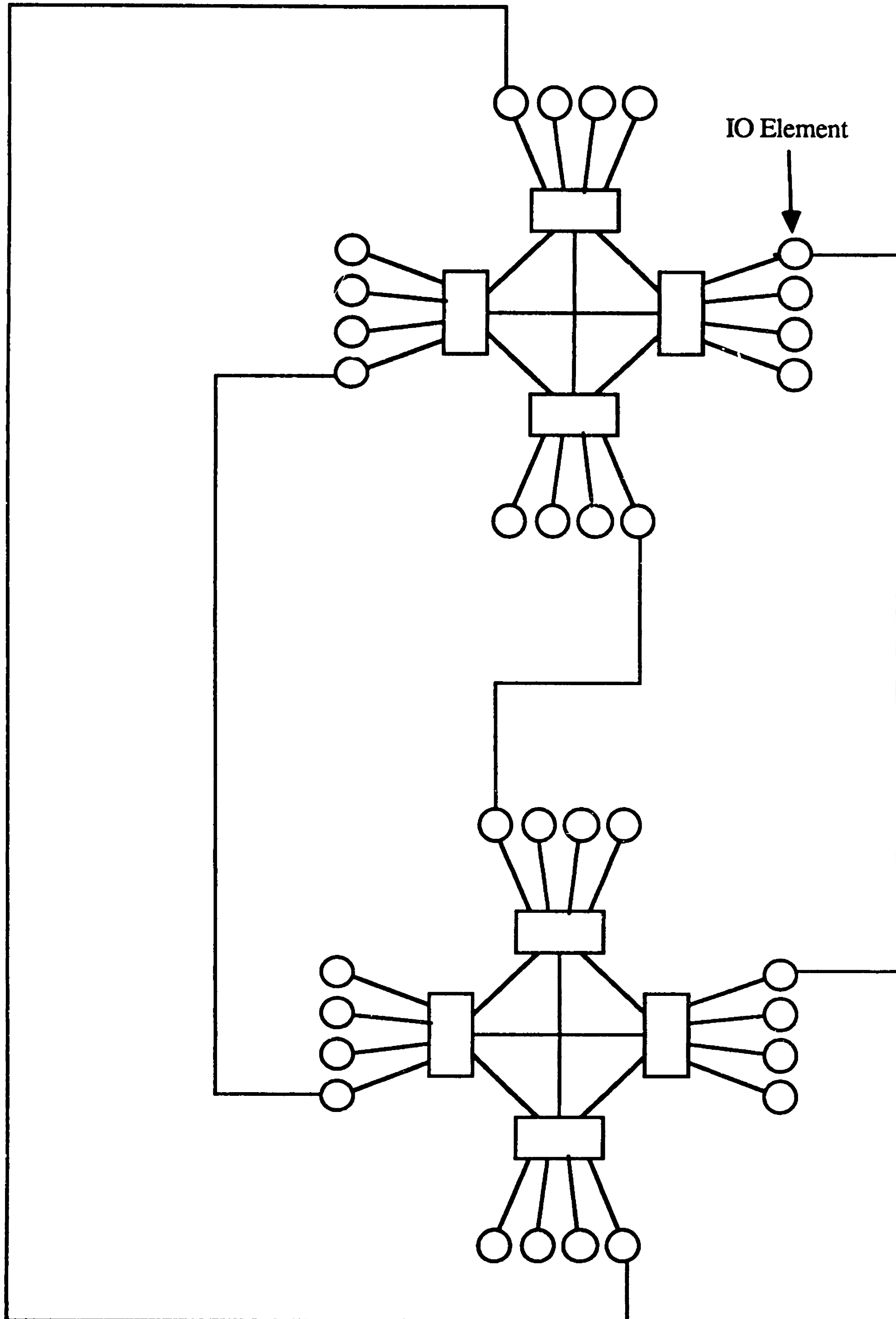


Figure 7.4

Inter-Cluster Connection Using IO Elements

7.2 Assumptions and Approach

As the first step in obtaining the reliability of an ensemble, formulations for the reliability of a cluster will be obtained. These will be composite formulations consisting of the probability of system loss due to simultaneous Network Element failures, simultaneous Processing Element failures, attrition due to Network Element and Processing Element failures, and cluster isolation due to IOE failures. Earlier calculations showed that two distinct failure modes dominate system loss probability in the short and long-term. System loss due to near-simultaneous failures dominates in the short term and attrition dominates in the long term. In the Network Element-based clusters of the present section, the new failure mode of cluster isolation due to IOE failures will be shown to be of importance in the long term. The probability of ensemble loss due to these effects will be computed separately, using conservative assumptions, and the resulting quantities combined to produce a composite formulation of ensemble loss probability. Then, the expected number of triads in a cluster will be computed. The resulting formulations will be used to determine appropriate coordinates in the design space.

The assumption will be made that all processors in the ensemble are grouped into triply redundant FMGs. Given a certain number of operational processors in a cluster, it will be assumed that a maximal number of triads possible will be formed, subject to the configuration constraint that no two members of an FMG subscribe to the same Network Element. Given *a* Network Elements and *b* Processing Elements per Network Element, this number of triads is $\lfloor ab/3 \rfloor$

7.3 Network Element Aggregate Markov Model

The Markov model for a cluster's Network Element aggregate is depicted in Figure 7.5. The model has $2(N_{NE/C}-3)+2$ states. In State 1, the aggregate has suffered no permanent failures. Transitions out of State 1 to State 2 occur at a rate of $N_{NE/C}\lambda_{NE}$, where $\lambda_{NE}=\lambda_{NE0}(1+(N_{NE/C}-1+N_{PE/NE})f_{port})$, and $N_{NE/C}$ is the number of Network

Elements in the cluster. The value of 10^{-5}/hour will be used for λ_{NE0}. This is an order of magnitude less than the failure rate of a processor, and an order of magnitude greater than the failure rate of a duplex comparator or an FTP interstage. This value will be estimated in Appendix B.

In State 2, the cluster is in the process of detecting and isolating the first Network Element failure. This function is performed by a fault masking group in the cluster and is described in Section 10.7. While in State 2, further Network Element failures can occur, with a rate of $(N_{NE/C}-1)\lambda_{NE}$, into State $2(N_{NE/C}-3)+2$, the system loss state due to near-simultaneous Network Element failures. In addition, from State 2 reconfigurations from transient failures occur back to State 1 with rate μf_t, and reconfigurations from permanent failures occur forward to State 3 with rate $\mu(1-f_t)$. Since fault masking groups are responsible for reconfiguration of both PEs and NEs, it is reasonable to assume that μ is the same for PEs and NEs. In State 3, one Network Element failure has occurred and been tolerated, and the situation is the same as in State 1, with one fewer Network Element in the cluster. The progression of states in the model continues until the redundancy of Network Elements has been exhausted in State $2(N_{NE/C}-3)+1$, where only 3 Network Elements are left and the cluster has been safely shut down.

7.4 Probability of Ensemble Loss Due to Simultaneous Failures

The probability of ensemble loss due to simultaneous failures by time t is equal to the probability that any cluster has suffered a near-simultaneous failure of a triad or of the Network Element aggregate by time t. These two events will be calculated first on a cluster basis and combined to yield the probability that the ensemble fails due to simultaneous failures. The probability of cluster loss due to simultaneous Network Element failures by time t is simply the probability that the aggregate is in State $2(N_{NE/C}-3)+2$ of the Markov model shown in Figure 7.5. Denote the probability of occupancy of this state at time t by $P_{SL/C(NE\ SIMF)}(t)$.

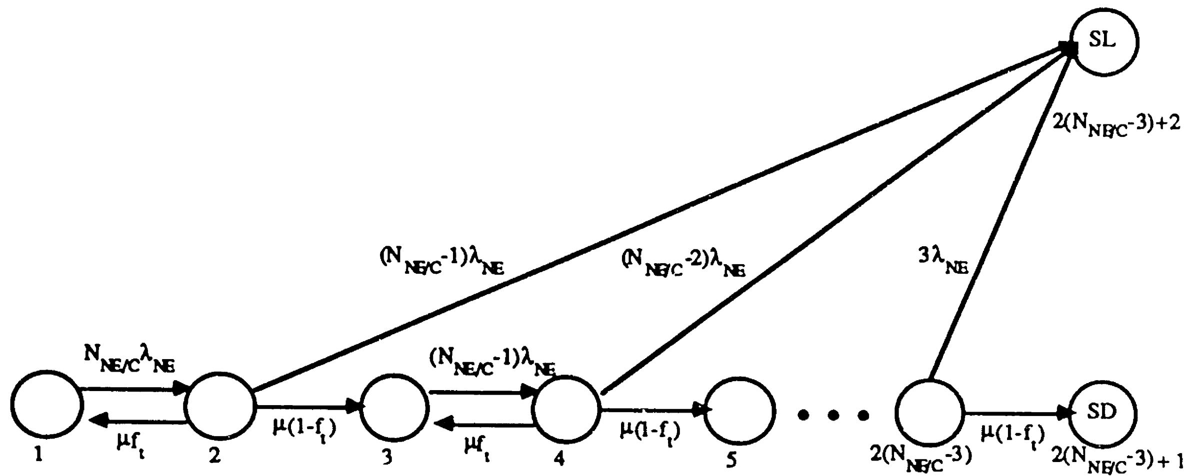


Figure 7.5

Markov Model of Network Element Aggregate

Calculation of the probability of cluster loss due to simultaneous processor failures is not so straightforward because the number of triads in the system at a given time is a function of the number of nonfaulty Network Elements in the cluster. Therefore the number of triads cannot be expressed as a Markov model which is independent of the Network Element Markov model. To obtain an exact solution for the desired probability, a full Markov model for the entire cluster could be constructed. The deficiency of this approach is that an extremely large number of states is required to express all the possible combinations of failed Network Elements and Processing Elements. A classical approach to this combinatorial explosion is to truncate the model at a given number of simultaneous failures. However, this solution does not address the real problem, which is that the complexity of the model is due to the combinatorics of the number of possible failure modes and failure trajectories of the Network Element/Processing Element mixture, not the total number of failures in existence at any given time.

Due to the complexity of the interacting Network Element and Processing Element

failure modes, upper and lower bounds on the probability of cluster failure due to simultaneous processor failure will be calculated.

To calculate the upper bound, a formulation will be used which overestimates the number of triads in the cluster at any given time by assuming that the cluster is completely reconfigurable, i.e., that any processor in a cluster can be grouped with any other two in the same cluster to form a triad. Given the constraints upon such grouping in the actual cluster, this technique can be seen to overestimate the number of triads in the cluster.

To calculate the lower bound, it will be assumed that the triads are not reconfigurable, in the sense that when a triad suffers a covered permanent failure it is shut down and its constituent components are unavailable for further use by any other triad. This approach will underestimate the probability of system loss due to simultaneous failures because it will consistently underestimate the number of triads in the system at any given time.

Finally, the usual linear approximation for system loss due to simultaneous failures will be calculated and shown to be sufficiently accurate. This approximation will in fact be used in the optimization process.

7.5 Overestimation of Simultaneous Processor Failure Probability

To perform the overestimation of $p_{SL/C(SIMF)}(t)$, denoted $p_{SL/C(SIMF/OE)}(t)$, a model will be used which is similar to that used in Section 6.1 for the cluster composed of an aggregate of completely connected processors in which a triad could be formed from any three processors using any other processor in the cluster as the fourth participant in the consensus and synchronization protocols. The model is slightly modified to account for the fact that, in the cluster incorporating Network Elements as consistency brokers, only three processors are needed to form a triad, as opposed to the four needed in the fully connected cluster. This allows the redefinition of the system loss due to attrition to be the state in which there are fewer than three processors in the cluster, instead of fewer than four processors in the cluster as in the earlier design. In addition, the transition rate from a state

in which there are "i" triads under reconfiguration to the system loss due to near-simultaneous failure state is $2\lambda i$ instead of $3\lambda i$ because in the former case such failure occurs only when another processor of the triad fails as opposed to when another processor in the quad fails in the latter case. This model is presented in Figure 7.6.

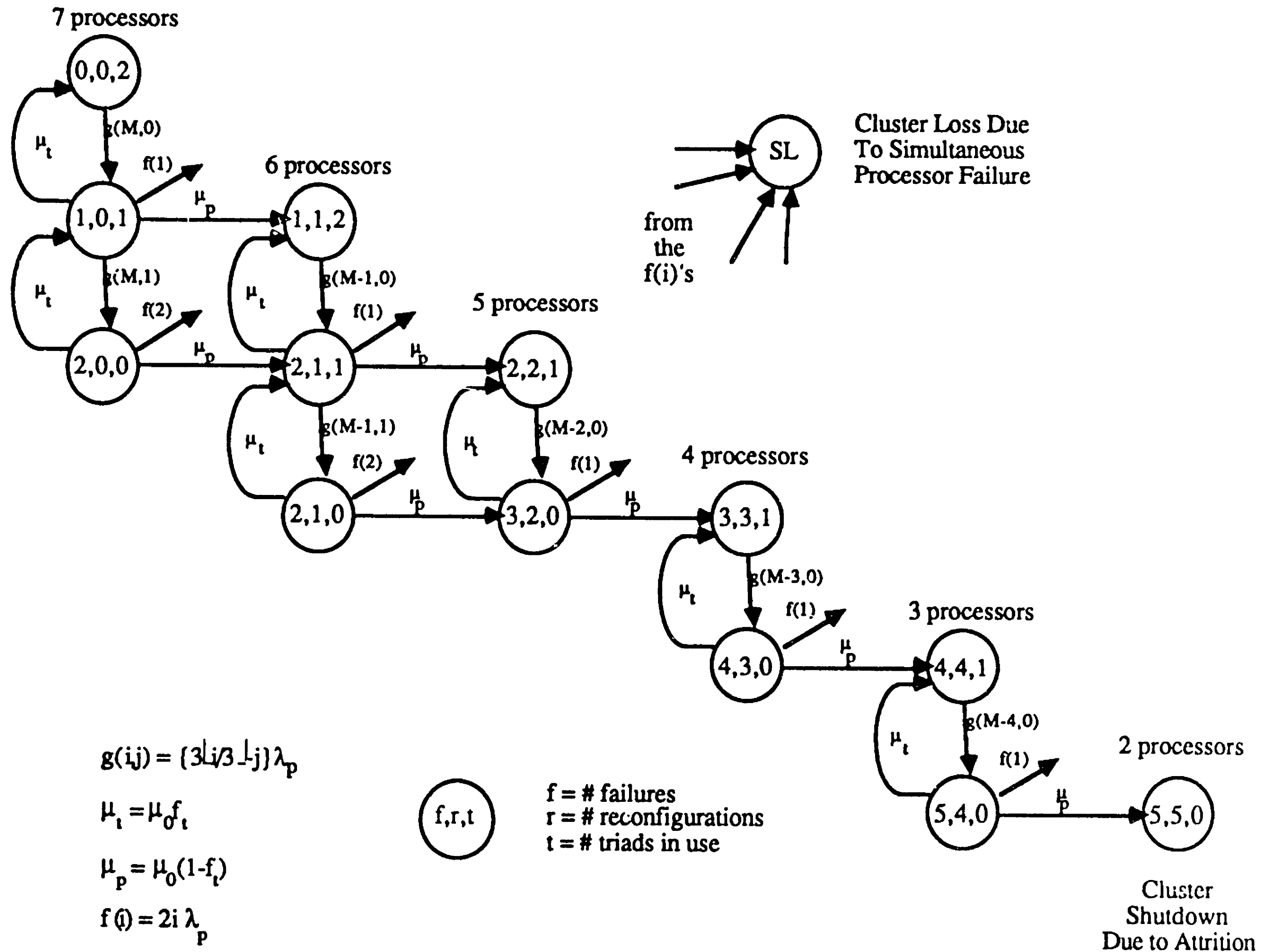


Figure 7.6

Overestimation Model for a Cluster of 7 Processors

In the above model, let the probability that cluster loss occurs because of simultaneous processor failures be denoted by $p_{SL/C(PE\ SIMF)}(t)$. The overestimate for the probability of

cluster loss due to simultaneous failures is the probability that the Network Element aggregate suffers simultaneous failures plus the probability that any triad suffers a simultaneous failure. Since the two events are not mutually exclusive but are independent,

$$p_{SL/C(SIMF/OE)}(t) = p_{SL/C(PE\ SIMF)}(t) + p_{SL/C(NE\ SIMF)}(t) - p_{SL/C(PE\ SIMF)}(t)\ p_{SL/C(NE\ SIMF)}(t).$$

The overestimate of the probability of ensemble loss due to simultaneous processor failures is equal to one minus the probability that no cluster has suffered near-simultaneous failures,

$$p_{SL/E(SIMF/OE)}(t) = 1-(1-p_{SL/C(SIMF/OE)}(t))^{N_C}.$$

7.6 Underestimation of Simultaneous Processor Failure Probability

The underestimation Markov model of an unreconfigurable triad is shown in Figure 7.7. In State 1 of this model, the triad has either not suffered a failure or has recovered from a prior transient failure. In State 2, the triad is in the process of recovering from a failure. In State 3, a second failure has occurred during reconfiguration resulting in triad and hence system loss. Finally, in State 4 the triad has shut down by successfully covering a permanent failure. The underestimate of the probability of cluster loss due to simultaneous failures in a triad or in the Network Element aggregate is

$$p_{SL/C(SIMF/UE)}(t) = 1-(1-p_3(t))^{N_{TR/C}} + p_{SL/C(NE\ SIMF)}(t) - (1-(1-p_3(t))^{N_{TR/C}})p_{SL/C(NE\ SIMF)}(t),$$

where $N_{TR/C}$ is the number of triads in the cluster and is equal to $\lfloor N_{NE/C} N_{PE/NE}/3 \rfloor$. The underestimate of the probability of ensemble loss due to simultaneous failures is the probability that no cluster has suffered a near-simultaneous failure,

$$p_{SL/E(SIMF/UE)}(t) = 1-(1-p_{SL/C(SIMF/UE)}(t))^{N_C},$$

where N_C is the number of clusters in the ensemble.

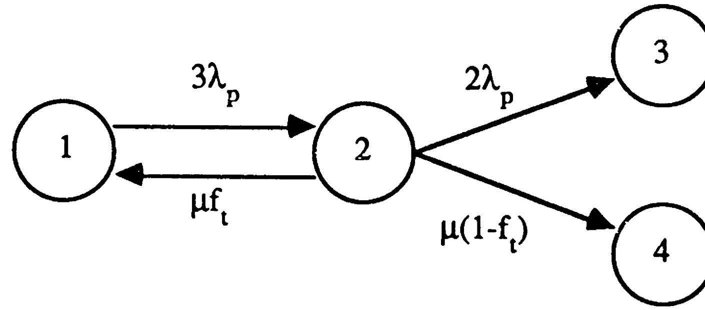


Figure 7.7

Markov Model of Underestimate Model

7.7 Linear Approximation to Overestimation

The third approximation technique uses the fact that the primary contribution to the short-term system loss probability is the probability that system loss occurs due to a near-simultaneous failure occurring after the first processor failure. The approximate value for the probability of near-simultaneous processor failure resulting in ensemble loss is

$$p_{SL/E(PE\ SIMF/SIMPLE)}(t)=N_{TR}6\lambda_p{}^2t/\mu.$$

By a simple modification of this result the approximate formulation for near-simultaneous Network Element failure is

$$p_{SL/E(NE\ SIMF/SIMPLE)}(t)=N_C N_{NE/C}(N_{NE/C}-1)\lambda_{NE}{}^2t/\mu.$$

The probability that the ensemble suffers a near-simultaneous failure in either a triad or in the Network Element aggregate is

$$p_{SL/E(SIMF/SIMPLE)}(t) = N_{TR}6\lambda_p{}^2t/\mu + N_C N_{NE/C}(N_{NE/C}-1)\lambda_{NE}{}^2t/\mu - N_{TR}6\lambda_p{}^2t/\mu\, N_C N_{NE/C}(N_{NE/C}-1)\lambda_{NE}{}^2t/\mu,$$

where N_{TR} is the number of triads in the ensemble and is equal to $N_C \lfloor N_{NE/C} N_{PE/NE}/3 \rfloor$.

This formulation is an overapproximation which does not take attrition into account (and therefore does not exhibit droop near the attrition regime). At the same time, examination of Figure 7.8 shows that in the short-term the approximation, denoted "simple" in the legend of Figure 7.8, is accurate.

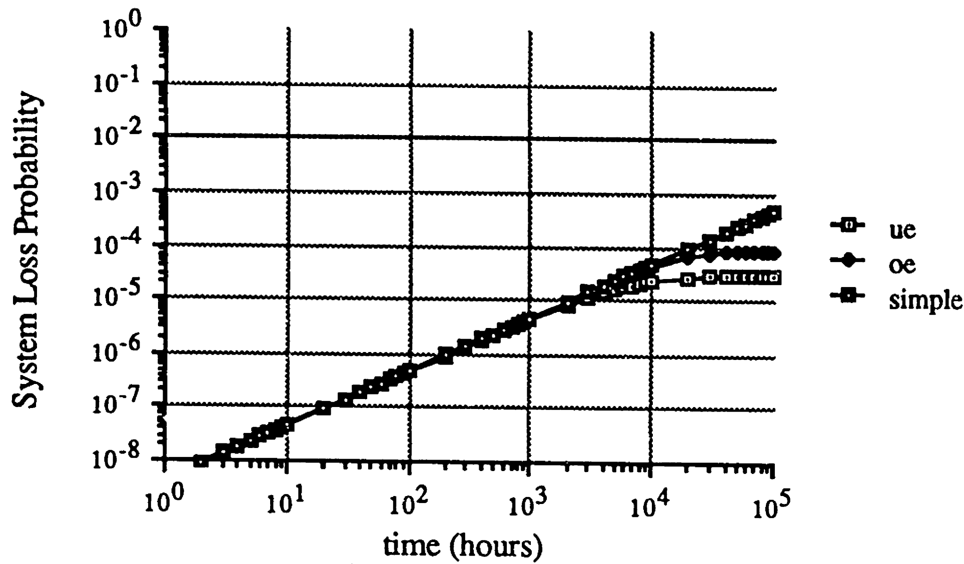


Figure 7.8

Comparison of Approximation Methods for Ensemble Loss Probability

For Network Element-Based Clusters

7.8 Probability of Ensemble Loss Due to Attrition

A combinatorial calculation of the probability of ensemble loss due to attrition will now be performed. This calculation will be done replacing λ_p by $\lambda_p(1-f_t)$ in the formulation for processor reliability and λ_{NE} by $\lambda_{NE}(1-f_t)$ in the formulation of Network Element reliability, in accordance with the previous observation on the accuracy of this technique. The probability that the cluster suffers loss due to Network Element or Processing Element attrition is the probability that there are insufficient Network Elements to form a Byzantine Resilient core regardless of the number of non-faulty Processing Elements, plus the probability that there are insufficient Processing Elements to form a Byzantine Resilient triplex FMG, given that there are in fact enough Network Elements. Mathematically,

$P_{SL/C(ATT)}(t)$ = prob(0, 1, 2, or 3 nonfaulty Network Elements at time t)

$$+\sum_{i=4}^{N_{NE/C}} \text{prob(Processing Element attrition | i nonfaulty Network Elements)}\cdot$$

$$\text{prob(i nonfaulty Network Elements at time t)}\ .$$

Computing the first term in the above summation,

prob(Processing Element attrition | i nonfaulty Network Elements)

= prob(0 Processing Elements in cluster | i nonfaulty Network Elements)

+ prob(only 1 Network Element has any nonfaulty processors i i nonfaulty Network Elements)

+ prob(only 2 Network Elements have any nonfaulty processors | i nonfaulty Network Elements)

$= (1-r_{PE}(t))^{iN_{PE/NE}}$

$+ i\,(1-(1-r_{PE}(t))^{N_{PE/NE}})\,(1-r_{PE}(t))^{((i-1)N_{PE/NE})}$

$+ C(i,2)\,(1-(1-r_{PE}(t))^{N_{PE/NE}})^2\,(1-r_{PE}(t))^{((i-2)N_{PE/NE})}$.

The probability that there are exactly i operational Network Elements in a cluster is $C(N_{NE/C}, i)\, r_{NE}(t)^i\, (1-r_{NE}(t))^{N_{NE/C}-i}$, where $r_{NE}(t) = e^{-\lambda_{NE}(1-f_t)t}$.

Therefore the total probability that a cluster suffers attrition is equal to the probability that there are fewer than 4 Network Elements in the cluster plus the probability that processor attrition occurs given that there are more than 4 Network Elements in the cluster:

$$p_{SL/C(ATT)}(t) = \sum_{i=0}^{3} C(N_{NE/C}, i)\, r_{NE}(t)^i\, (1-r_{NE}(t))^{N_{NE/C}-i}$$

$$+ \sum_{i=4}^{N_{NE/C}} C(N_{NE/C}, i)\, r_{NE}(t)^i\, (1-r_{NE}(t))^{N_{NE/C}-i}$$

$$\{\ (1-r_{PE}(t))^{iN_{PE/NE}} +$$

$$i\,(1-(1-r_{PE}(t))^{N_{PE/NE}})\,(1-r_{PE}(t))^{((i-1)N_{PE/NE})} +$$

$$C(i,2)\,(1-(1-r_{PE}(t))^{N_{PE/NE}})^2\,(1-r_{PE}(t))^{((i-2)N_{PE/NE})}\ \}\ ,$$

where $r_{PE}(t) = e^{-\lambda_p(1-f_t)t}$.

The total probability of attrition for the ensemble is the probability that all clusters have suffered attrition. Since the cluster attrition probabilities are independent, $p_{SL/E(ATT)}(t)=p_{SL/C(ATT)}(t)^{N_C}$.

7.9 Probability of Ensemble Loss due to Cluster Isolation

In the previous architectures there was a high degree of inter-group or inter-cluster connectivity. For example, in the unreconfigurable FMGs there were three (or four, in the case of the quad FMG) paths from each FMG to its neighbors in the hypercube. In the fully-connected cluster-based ensemble, each processor in the cluster had a dedicated connection to its equivalent processor in the neighboring cluster. Because of the rich connectivity in these architectures the likelihood of attrition due to insufficient Processing Elements is greater than the probability that ensemble loss occurs due to inadequate inter-cluster connectivity.

This is not the case in the Network Element-based clusters of the present section, because in providing regions of shared connectivity we have robbed the ensemble of its high degree of inter-cluster connectivity by using IOEs to perform inter-cluster communication as opposed to providing each processor with this capability. Therefore there is a significantly higher likelihood that, even though there are enough processors to form one or more FMGs, these FMGs cannot communicate across cluster boundaries because of IOE loss.

Define the situation which occurs when no cluster can communicate with any other as *ensemble loss due to cluster isolation*. In a given application it is likely that ensemble loss can be defined to occur when a particular subset of clusters cannot communicate with some other subset. However, because of the application-dependent nature of this quantity we will compute the probability that all clusters are incommunicado.

We will compute the probability that all clusters are isolated because of failures of the IOEs and their associated Network Elements using a generalization of a formulation presented in [Gjermundsen86]. A cluster may be isolated due to one of two causes. First, the cluster may be isolated because it does not possess enough IOE/NE pairs to form a redundant path to any of its neighboring clusters. We denote this event as "cluster isolation due to local attrition". Second, the cluster may possess adequate IOE/NE pairs to form such

a redundant path to one or more of its neighbors, but a failure pattern involving the cluster under consideration and its neighbors may result in inadequate overlap between functional IOE/NE combinations residing on the cluster and its neighbors, resulting in the inability to form a redundant communications link between the cluster and any of its neighbors. We will denote this failure mode as "cluster isolation due to unfortunate combinations".

Define "ensemble failure due to cluster isolation" to be the isolation of all clusters from each other. Some of these clusters will be isolated due to local attrition and some due to unfortunate combinations. We desire to enumerate these cluster isolation modes. Assume that ensemble failure due to cluster isolation has occurred, and let j denote the number of clusters which are isolated due to local attrition. Then N_C-j clusters are isolated due to unfortunate combinations. The probability that the ensemble is failed because all clusters are isolated is the sum of the probabilities of all possible combinations of j local attritions and N_C -j isolations due to unfortunate combinations, for all possible values of j,

$$p_{SL/E(CI)} = \sum_{j=0}^{N_C-1} C(N_C,j)\, p(u \mid (N_C-j-1)u, ja)\, p(u \mid (N_C-j-2)u, ja) \ldots p(u \mid 0u, ja) \cdot$$
$$p(a \mid (j-1)a) p(a \mid (j-2)a) \cdots p(a \mid 0a),$$

where p(u | ku, ja) is the conditional probability that a given cluster is isolated due to unfortunate combinations of IOE failures, given that k other clusters have been isolated due to unfortunate IOE failure combinations and j other clusters have been isolated due to local attritions. The quantity p(a | ka) is the conditional probability that a cluster is isolated due to local attrition, given that k other clusters are isolated due to local attrition. Note that the upper limit on the summation is N_C-1, since if N_C-1 clusters are isolated, then the last cluster is isolated also.

To proceed with the calculation, first note that the probability that a given cluster is isolated due to local attrition is independent of the state of any other cluster...local attrition is by definition a local phenomenon. In particular, local attrition is independent of the number of clusters isolated due to unfortunate combinations or the number of other clusters

isolated due to local attrition, so the probability that j clusters are isolated due to local attrition is $p(a)^j$, where p(a) is the probability that a single cluster is isolated due to local attrition. This is the probability that the cluster in question has fewer than two IOE/NE combinations surviving,

$$p(a) = u(t)^{N_{NE/C}} + N_{NE/C}r(t)u(t)^{N_{NE/C}-1}$$

where $r(t) \equiv r_{IOE}(t)r_{NE}(t)$ and $u(t) \equiv 1-r(t)$. The probability that j clusters are isolated due to local attrition is then

$$p(ja) = p(a)^j = \{u(t)^{N_{NE/C}} + N_{NE/C}r(t)u(t)^{N_{NE/C}-1}\}^j.$$

Now we calculate p(u | ku, ja), the conditional probability that an otherwise functional cluster is isolated due to unfortunate combinations, given k other isolations due to unfortunate combinations and j other isolations due to local attrition. Because of the pre-existence of k+j isolations, the combinatorial enumeration of the number of failures required to cause isolation of the cluster in question cannot be performed assuming that we have the complete complement of $N_{NE/C}$ IOEs to choose from. Upper and lower limits on the number of nonfaulty IOEs possessed by this cluster must be determined, given that there are k+j pre-existing isolated clusters.

First observe that the number of IOEs surviving in such a cluster must be between 2 and $N_{NE/C}$ -1. There must be at least two IOEs in a cluster which is isolated due to unfortunate combinations because if there were fewer than two the cluster would be isolated because of local attrition, by definition. There must be no more than $N_{NE/C}$ -1 because more than this number of IOEs in an unisolated cluster would imply that one of the k clusters isolated due to unfortunate combinations would have fewer than two IOEs because there is a cluster in the ensemble having all its IOEs. Since this would in turn imply that one of these k isolated clusters is isolated due to attrition, i.e., it has fewer than two surviving Network Elements in it, and since it is assumed that the k clusters are isolated due to unfortunate combinations as opposed to attrition, the upper bound on the cluster under consideration follows.

A second pertinent observation is that, if one of the k previously isolated neighboring clusters has q nonfaulty IOEs, then any unisolated cluster can possess at most $N_{NE/C}$-(q-1) nonfaulty IOEs. If this were not the case and the unisolated cluster were allowed to possess more than $N_{NE/C}$-(q-1) nonfaulty IOEs, then at least two nonfaulty IOEs in both the previously isolated cluster having q nonfaulty IOEs and the unisolated cluster having more than $N_{NE/C}$-(q-1) nonfaulty IOEs would overlap, and it would be impossible to isolate the former due to unfortunate failure combinations. Since it is assumed in the enumeration that the former is isolated due to unfortunate combinations, this upper bound on nonfaulty IOEs in any unisolated cluster must hold.

The probability that a cluster suffers isolation due to unfortunate failure combinations is equal to the probability that the cluster has enough IOEs, denoted i, to communicate with the remainder of the ensemble, but that an unfortunate combination of failures has occurred in both the cluster under consideration and the $N_{neighbors}$ neighboring clusters such that the former does not have a redundant path to any of them. This may be expressed mathematically as

$$p(u \mid N_{neighbors}) = \sum_{i=2}^{N_{NE/C}-1} C(N_{NE/C},i)r(t)^{i}u(t)^{N_{NE/C}-i}\{u(t)^{i} + ir(t)u(t)^{i-1}\}^{N_{neighbors}},$$

where the number of neighbors depends upon the topology of the inter-cluster network and the number of clusters remaining connected to the network. However, in light of the second observation above, this formulation is not completely correct because, if any neighboring clusters are themselves isolated due to unfortunate combinations, then an upper bound *lower* than $N_{NE/C}$-1 exists on the number of IOEs in the cluster. Let q denote the maximum number of IOEs surviving in a neighboring cluster which is assumed to be isolated due to unfortunate combinations. We will compute p(k,q), the probability that, given that k clusters are isolated due to unfortunate combinations, the largest number of nonfaulty IOEs possessed by any of them is q. This is the summation over all the isolated

clusters of the probability that j of them have q surviving IOEs times the probability that the remaining k-j of them have fewer than q surviving IOEs, times the appropriate combinatorial coefficient,

$$p(k,q) = \sum_{j=1}^{k} C(k,j)\ \{C(N_{NE/C},q)r(t)^q u(t)^{N_{NE/C}-q}\}^j \left\{ \sum_{i=2}^{q-1} C(N_{NE/C},i)r(t)^i u(t)^{N_{NE/C}-i} \right\}^{k-j}.$$

Now, given that we know that the number of nonfaulty IOEs in an unisolated cluster is $N_{NE/C}$-(q-1), and the minimum number of nonfaulty IOEs in an unisolated cluster is 2, the probability that a cluster is isolated due to unfortunate combinations can be correctly expressed as

$$p(u \mid N_{neighbors}) = \sum_{q=2}^{N_{NE/C}-1} p(k,q) \sum_{i=2}^{N'} C(N',i)r(t)^i u(t)^{N'-i} \{u(t)^i + ir(t)u(t)^{i-1}\}^{N_{neighbors}},$$

where $N' \equiv N_{NE/C}-(q-1)$.

We must now obtain $N_{neighbors}$ in terms of k, the number of unfortunate isolations, and j, the number of local attritions. For convenience we will assume that the clusters are connected in a fully-connected topology. This simplifies the analysis enormously and it can be shown that little difference exists between cluster isolation probability in the fully-connected and hypercube intercluster topologies, for the small number of clusters which will be considered. Thus, under no failure conditions, each cluster has N_C-1 neighbors. When k+j clusters are isolated due to whatever cause, the number of neighbors is reduced to N_C-1-(k+j). Therefore the probability that a cluster is isolated due to unfortunate combinations, given that there are already k unfortunate combinations and j local attritions, is

$$p(u \mid ku, ja) = \sum_{q=2}^{N_{NE/C}-1} p(k,q) \sum_{i=2}^{N'} C(N',i)r(t)^i u(t)^{N'-i} \{u(t)^i + ir(t)u(t)^{i-1}\}^{N_C-1-(k+j)}.$$

Combining these results gives the expression for the probability that ensemble loss occurs due to cluster isolation,

$$p_{SL/E(CI)} = \sum_{j=0}^{N_C} C(N_C,j) \left[\prod_{k=0}^{N_C} \left(\sum_{q=2}^{N_{NE/C}-1} \right. \right.$$

$$\left. \sum_{m=1}^{k} C(k,m) \{C(N_{NE/C},q) r(t)^q u(t)^{N_{NE/C}-q}\}^m \left\{ \sum_{n=2}^{q-1} C(N_{NE/C},n) r(t)^n u(t)^{N_{NE/C}-n} \right\}^{k-m} \right) \cdot$$

$$\left. \sum_{i=2}^{N'} C(N',i) r(t)^i u(t)^{N'-i} \{u(t)^i + i r(t) u(t)^{i-1}\}^{N_C-1-(k+j)} \right]$$

$$\{u(t)^{N_{NE/C}} + N_{NE/C} r(t) u(t)^{N_{NE/C}-1}\}^j$$

7.10 Total Probability of Ensemble Loss

The total probability of ensemble loss is the probability that any of the above four failure mode occurs, i.e., near-simultaneous Network Element failures, near-simultaneous Processing Element failures, attrition of the Processing Elements and Network Elements in all clusters, and isolation of all clusters from each other. These failure modes are not all mutually exclusive, nor are they all independent. Therefore the associated probabilities must be combined according to the general form for the probability of the union of n events,

$$
\begin{aligned}
P(A_1+A_2+\cdots+A_n) = &\sum_{i=1}^{n} P(A_i) \\
&- \sum_{i,j=1,i\neq j}^{n} P(A_iA_j) \\
&+ \sum_{i,j,k=1,i\neq j\neq k}^{n} P(A_iA_jA_k) \\
&\cdots \\
&+ (-1)^{n-1}P(A_1A_2\cdots A_n)\,.
\end{aligned}
$$

To apply this rule to the problem at hand, first make the following definitions for the sake of brevity:

$P(N) \equiv p_{SL/E(NE\ SIMF/SIMPLE)}(t)$,

$P(P) \equiv p_{SL/E(PE\ SIMF/SIMPLE)}(t)$,

$P(A) \equiv p_{SL/E(ATT)}(t)$,

$P(I) \equiv p_{SL/E(CI)}(t)$.

Let the joint probabilities be denoted similarly, e.g.,

$P(NIP) \equiv p_{SL/E(NE\ SIMF/SIMPLE) and SL/E(CI) and SL/E(PE\ SIMF/SIMPLE)}(t)$

The total probability of ensemble loss, denoted P(T), is then

$$
\begin{aligned}
P(T) = &P(N) + P(P) + P(A) + P(I) \\
&- P(NP) - P(NA) - P(NI) - P(PA) - P(PI) - P(AI) \\
&+ P(NPA) + P(PAI) + P(AIN) + P(INP) \\
&- P(NPAI).
\end{aligned}
$$

Triad simultaneous failure and triad attrition are mutually exclusive, so P(PA)=0. Similarly, triad simultaneous failure and cluster isolation are disjoint, so P(PI)=0. Finally, Network Element simultaneous failure and cluster isolation are disjoint, so P(NI)=0. Furthermore, any joint probability involving the intersection of these pairs of events is also zero. Appropriately simplifying the above expression yields

$$P(T) = P(N) + P(P) + P(A) + P(I) - P(NP) - P(NA) - P(AI).$$

Examination of the remaining joint probabilities reveals that Network Element simultaneous failure and triad simultaneous failures are independent, so the term $P(NP)$ can be replaced by the quantity $P(N)P(P)$.

The term $P(NA)$ can be expressed as $P(A|N)P(N)$. It is difficult to formulate $P(A|N)$, so an approximation will be sought. The conditional probability $P(A|N)$ is the probability that the ensemble suffers attrition, given that any cluster in the ensemble has suffered near-simultaneous Network Element failures. Compare this conditional probability to $P(A)$, the probability that the ensemble suffers attrition regardless of the occurrence of near-simultaneous Network Element failures. It can be argued that $P(A|N)>P(A)$. The conditional probability is predicated on the fact that Network Element failures are in existence, whereas in the case of the latter probability no such assumption can be made about any pre-existing failures. Since there are pre-existing failures in the case of the conditional probability, the likelihood that attrition occurs must be higher than the case in which there is no knowledge of pre-existing failures. Since $P(A|N)>P(A)$, replacement of the subtrahend $P(AN)=P(A|N)P(N)$ by the smaller quantity $P(AN)\approx P(A)P(N)$ results in an upper bound on $P(T)$. Therefore this approximation will be made.

Considering the final joint probability $P(AI)$, we again express $P(AI)$ as $P(I|A)P(A)$ and argue that $P(I|A)>P(I)$. If ensemble attrition is assumed to have occurred, then there is a non-zero probability that this attrition occurred because of Network Element failures (refer to the first four terms in the expression for cluster attrition). This non-zero likelihood of Network Element failures increases the likelihood of cluster isolation since the probability of cluster isolation depends upon the number of existing Network Elements. Therefore $P(I|A)>P(I)$, and the resulting replacement of $P(I|A)P(A)$ by $P(I)P(A)$ in the subtrahend again results in a conservative upper bound on $P(T)$.

With these observations and approximations in place, we have the final expression for

total ensemble loss probability

$$P(T) = P(N) + P(P) + P(A) + P(I)$$
$$- P(N)P(P) - P(N)P(A) - P(A)P(I).$$

Decoding this notation yields the full expression for total ensemble loss probability,

$$p_{SL/E(TOTAL)}(t)=p_{SL/E(NE\ SIMF/SIMPLE)}(t) + p_{SL/E(PE\ SIMF/SIMPLE)}(t)$$
$$+ p_{SL/E(ATT)}(t) + p_{SL/E(CI)}(t)$$
$$- p_{SL/E(NE\ SIMF/SIMPLE)}(t)p_{SL/E(PE\ SIMF/SIMPLE)}(t)$$
$$- p_{SL/E(NE\ SIMF/SIMPLE)}(t)p_{SL/E(ATT)}(t)$$
$$- p_{SL/E(ATT)}(t)p_{SL/E(CI)}(t).$$

7.11 Relative Magnitudes of Failure Modes

It is relevant at this point to investigate the relative magnitudes of the probability of the above failure modes for a NE-based ensemble of a "characteristic" size. In anticipation of subsequent results we will choose an ensemble composed of 4 clusters, each of which possesses 6 Network Elements. Subscribing to each Network Element are 8 Processing Elements and 1 IO Element. The 4 clusters are fully-connected. Figure 7.9 shows the constituents of ensemble failure probability versus mission time for this ensemble.

The following observations can be made. First, as expected, near-simultaneous Processing Element failure dominates system loss probability in the short term. Recall that the probability of this failure mode is not a function of any architectural parameter, so the magnitude of the probability of this failure mode is the same for all forms of the NE-based ensemble. The second largest contributor to ensemble loss probability in the short term is near-simultaneous Network Element failures. It can be seen that, because of the reduced number of them in the ensemble and their relatively low complexity, this probability is substantially below that of Processing Element failure. Therefore, with respect to near-simultaneous Network Element failure probability, a wide latitude exists with which decisions about cluster size may be made.

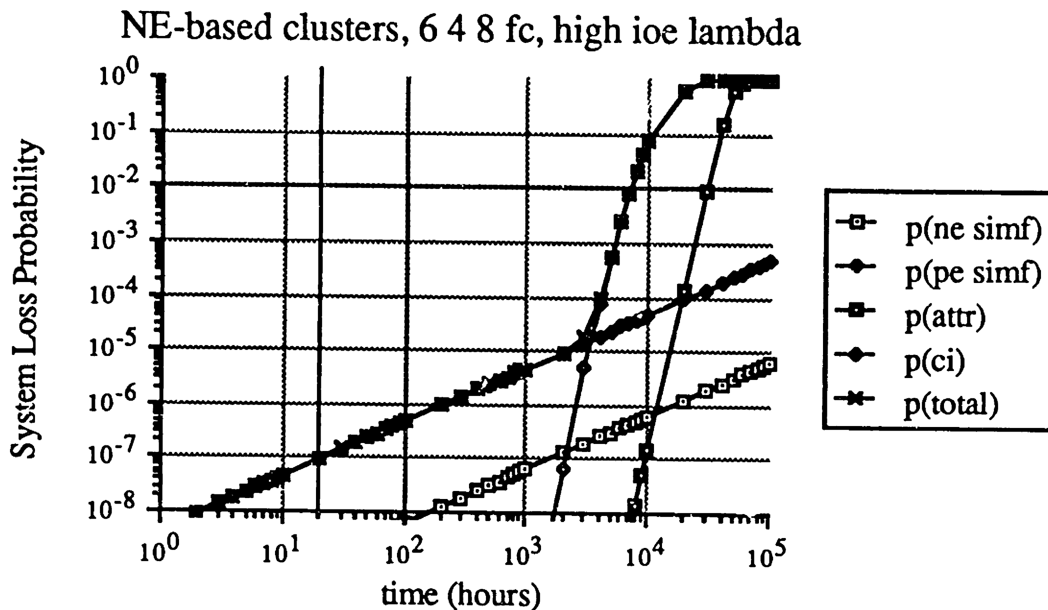


Figure 7.9

Constituents of Ensemble Loss Probability for Network Element-Based Cluster

$\lambda_{IOE0} = 10^{-4}$/hour

Moving to the long term, cluster isolation probability is seen to dominate. It begins domination at around 2000 hours and quickly becomes so severe that it is obvious that, as it stands, the Network Element-based clusters have inadequate long-term reliability. Finally, Processing Element/Network Element attrition becomes a factor at around 10,000 hours, but can be seen to never viably compete with cluster isolation probability.

Because of the high probability of cluster isolation, it is of interest to investigate means of reducing the likelihood of this failure mode. This may be done in one or more of several ways. First, one may implement an IOE which has its own internal redundancy. However, since the problem occurs in the attrition regime, it is clear that unless standby redundancy is employed, with the redundant components in a low-failure rate operational mode (e.g., powered down), the use of such additional redundancy would not improve IOE reliability. Assuming that such a power-down mechanism is implemented, the use of standby

redundancy may therefore be used to advantage. However, because of the analytical difficulties of such standby systems (see [Depaula84]), and because it will be shown below that another feasible solution to this problem exists, we will proceed to another architectural means to improve IOE reliability.

The long-term reliability of the IOE may also be increased by a reduction of its failure rate. Figure 7.10 shows the strong sensitivity of ensemble loss probability, as represented by MTTSF, to IOE failure rate. From this one may conclude that it is advantageous to strive to reduce λ_{IOE0}.

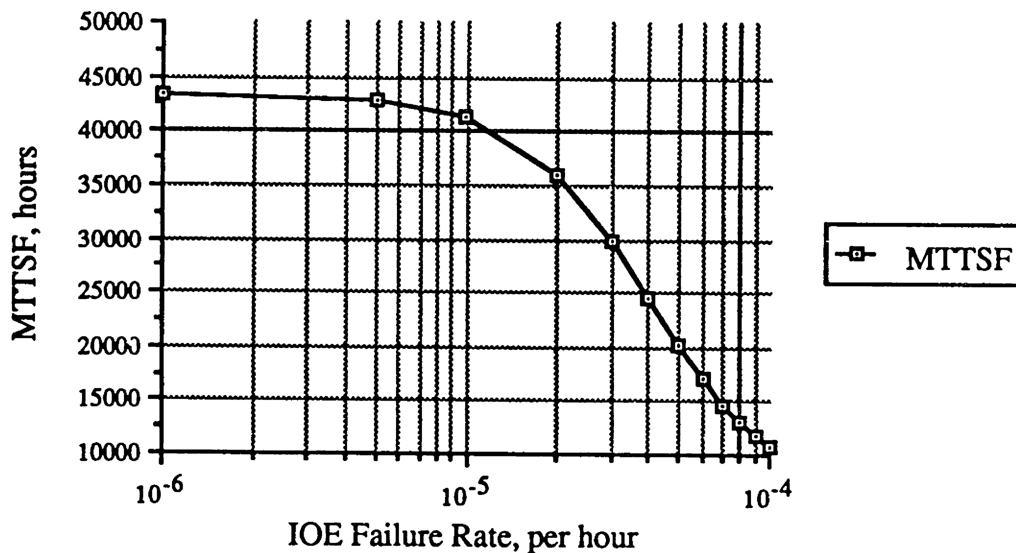


Figure 7.10

Sensitivity of MTTSF to λ_{IOE0}

In addition to a judicious choice of low-failure rate components, another obvious approach is to simplify the IOE. This reduction in complexity can result in a reduction in the component's failure rate. To this end it is possible to offload many of the functions of the IOE onto one or more of the Processing Elements of the cluster. Such functions include message formatting, blocking and deblocking, message routing, sophisticated error

checking, and acknowledgement/retry protocol execution. Thus the architecture of the IOE can be simplified to consist of a component consisting of an address decoder, control logic, and a bank of UARTs. Figure 7.11 shows the architecture of such a simplified IOE. Using device counts for the components comprising the IOE, the core IOE failure rate can be shown to be under 1/10th that of a processor. Figure 7.12 shows the ensemble loss probability curve calculated using this value (10^{-5}/hour) for λ_{IOE0}. Cluster isolation probability is extremely sensitive to IOE failure rate, to the extent that at this new value cluster isolation probability no longer dominates attrition probability. With this design optimization in mind, further calculations will therefore assume that IOEs have a core failure rate that is 1/10th that of a Processing Element.

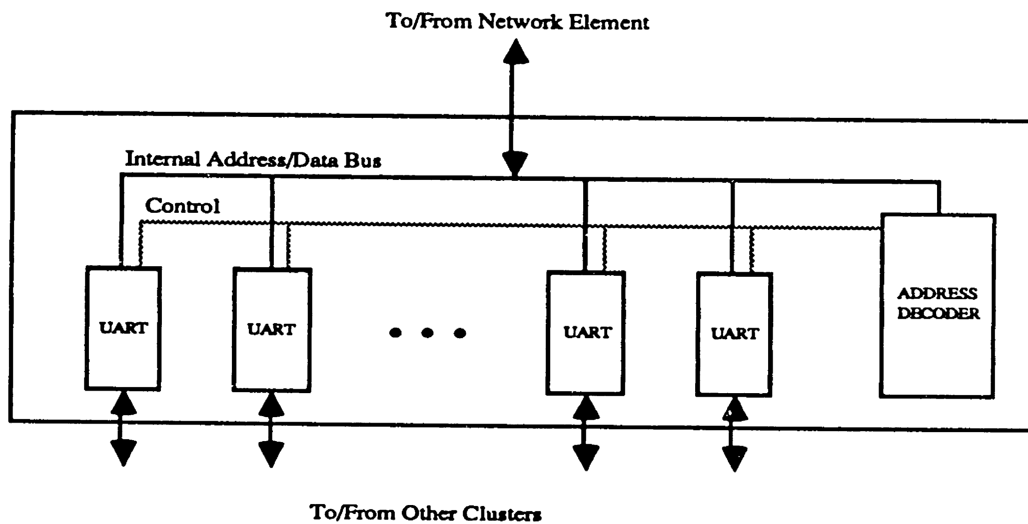


Figure 7.11

Architecture of Simplified IO Element

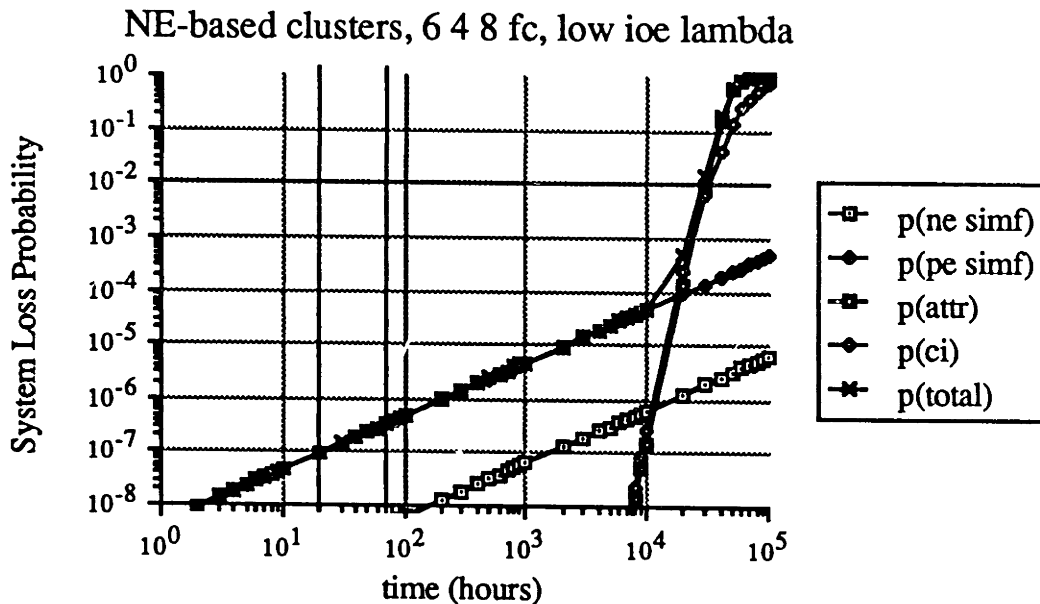


Figure 7.12

Constituents of Ensemble Loss Probability for Network Element-Based Cluster

$\lambda_{IOE0} = 10^{-5}$/hour

7.12 Expected Value of Number of Triads in Ensemble

As a first step in the combinatorial calculation of the expected value of the number of triads in the ensemble, the expected value of the number of triads in a cluster will be calculated. This calculation will be made assuming that Network Element failures and Processing Element failures are independent, i.e., that the failure of a Processing Element does not influence the failure behavior of its associated non-faulty Network Element. This is strictly true only if each subscriber Processing Element has a dedicated link to its Network Element, and if the Network Element has a means of disabling that link. This will be assumed to be the case in this idealized cluster architecture. The expected value of the number of triads in the cluster at time t is given by the summation over all possible numbers of functional Network Elements of the expected value of the number of triads in the cluster at time t given "i" functional Network Elements times the probability that there are in fact "i" functional Network Elements in the cluster at time t,

$$E_{TR/C}(t) = \sum_{i=4}^{N_{NE/C}} E_{TR/C|i\ NEs}(t)\ \mathrm{prob}(i \text{ NEs at time } t)\ .$$

The expected number of triads in the cluster at time t given that i Network Elements are nonfaulty, $E_{TR/C|i\ NEs}(t)$, is equal to the number of nonfaulty Network Elements in the cluster times the expected value of the number of Processing Elements per Network Element at time t, divided by three,

$$E_{TR/C|i\ NEs}(t) = i\ E_{PE/NE}(t)/3,$$

where

$$E_{PE/NE}(t) = \sum_{j=1}^{N_{PE/NE}} j\ C(N_{PE/NE},j)\ r_{PE}(t)^j (1-r_{PE}(t))^{N_{PE/NE}-j}\ .$$

$N_{PE/NE}$ is the number of Processing Elements per Network Element under no failure conditions, and $r_{PE}(t)$, the processor reliability at time t, is given by

$$r_{PE}(t) = e^{-\lambda_p(1-f_t)t}\ .$$

The probability that there are i non-faulty Network Elements in the cluster is as before

$$\mathrm{prob}\ (\ i \text{ Network Elements at time } t) = C(N_{NE/C},\ i)\ r_{NE}(t)^i\ (1-r_{NE}(t))^{N_{NE/C}-i}\ .$$

Therefore

$$E_{TR/C}(t) = \sum_{i=4}^{N_{NE/C}} i \sum_{j=1}^{N_{PE/NE}} j\ C(N_{PE/NE},j)\ r_{PE}(t)^j\ (1-r_{PE}(t))^{N_{PE/NE}-j}/3$$

$$*\ C(N_{NE/C},\ i)\ r_{NE}(t)^i\ (1-r_{NE}(t))^{N_{NE/C}-i}\ .$$

The calculation of the expected value of the number of triads in the ensemble must take into account the likelihood that a cluster possessing viable triads may be isolated from the remainder of the ensemble. Therefore the expected value of the number of triads in the ensemble is

$$E_{TR/E}(t) = \sum_{c=2}^{N_C} C(N_C,c)\ \Pr(c \text{ clusters not isolated at time } t)\ c\ E_{TR/C}(t)\ .$$

The lower limit in the sum is 2 because 2 clusters must be unisolated for the ensemble

to be declared operational, by definition.

The probability that exactly c clusters are not isolated at time t is the probability that N_C-c clusters are isolated by time t, so

$$E_{TR/E}(t) = \sum_{c=2}^{N_C} C(N_C,c)\ \Pr(N_C-c \text{ clusters isolated at time t})\ c\ E_{TR/C}(t)\ .$$

The probability that N_C-c clusters are isolated at time t is given by a variation of the probability that all clusters are isolated by time t,

$\Pr(N_C-c$ clusters isolated by time t)

$$= \sum_{j=0}^{N_C-c-1} C(N_C-c,j)\ p(u \mid (N_C-c-j)u, ja)\ p(u \mid (N_C-c-j-1)u, ja) \ldots p(u \mid 0u, ja)\ p(ja)\ ,$$

Combining these formulations yields the expectation of the number of triads in the ensemble at time t,

$$E_{TR/E}(t) = \sum_{c=2}^{N_C} C(N_C,c)\ c\ E_{TR/C}(t) \cdot$$

$$\left\{ \sum_{j=0}^{N_C-c-1} C(N_C-c,j)\ p(u \mid (N_C-c-j)u, ja)\ p(u \mid (N_C-c-j-1)u, ja) \ldots p(u \mid 0u, ja)\ p(ja) \right\}.$$

7.13 Mean Time to System Failure

The Mean Time to System Failure of the ensemble is calculated as

$$MTTSF = \int_0^{\infty} R(t)\,dt = \int_0^{\infty} (1-p_{SL/E(TOTAL)}(t))\,dt\ .$$

$P_{SL/E(TOTAL)}(t)$ is the quantity derived in Section 7.10.

7.14 Architectural Optimization

In this section we will determine the architectural parameters which result in an architecture having minimal system loss probability in both the short and long term. The design parameters which will be varied to yield this optimum are (a) the number of clusters in the ensemble, N_C, (b) the number of Network Elements in a cluster, $N_{NE/C}$, and (c) the number of Processing Elements per Network Element, $N_{PE/NE}$. The number of FMGs in the ensemble will be assumed to be fixed.

As a first step in this process, it is necessary to reiterate the relationships among the above parameters and show their influence on other fundamental system parameters.

We have taken as a specified quantity the number of processing sites available at the beginning of a mission. In the fault masking architecture currently under investigation, each processing site must be triply redundant. Denote the number of triads required to meet this specification by N_{TR}. In the baseline architecture N_{TR}=64. The number of triads in the ensemble N_{TR} is equal to the number of triads in a cluster times the number of clusters, $N_{TR}=N_C N_{TR/C}$. The number of triads in a cluster is equal to the number of Network Elements per cluster times the number of Processing Elements per Network Element, divided by three and truncated to the lowest integer, so $N_{TR}=N_C\lfloor N_{NE/C}N_{PE/NE}/3\rfloor$.
The relationships between the design parameters and the component failure rates are as follows. The processor failure rate λ_{PE} is equal to $\lambda_{PE} = \lambda_{P0}(1+f_{port})$, where the core failure rate is increased by a factor of f_{port} to account for the complexity of the port from the Processing Element to the Network Element. Note that the complexity due to inter-cluster communications ports is confined to the IOE. The Network Element failure rate λ_{NE} is a function of the number of intra-cluster connections to the $N_{NE/C}$-1 other Network Elements and the number of connections to the $N_{PE/NE}$ subscriber Processing Elements, such that

$$\lambda_{NE} = \lambda_{NE0}(1+f_{port}(N_{NE/C}-1+N_{PE/NE})).$$

The IOE core failure rate λ_{IOE0} is assumed to be $1/10^{th}$ that of a processor, and the total IOE failure rate similarly parameterized on the number of inter-cluster ports that the IOE

must support, as

$\lambda_{IOE} = \lambda_{IOE0}(1 + f_{port}(1 + N_{Neighbors}))$. Again, $f_{port} = 0.1$.

7.15 Optimization of Simultaneous Failure Resilience

It was shown earlier that the probability of near-simultaneous failure for a Processing Element triad could be expressed by the conservative approximation

$p_{SL/TR(PE\ SIMF/SIMPLE)}(t) \approx 6\lambda_{PE}^2 t/\mu$.

For an ensemble composed of N_{TR} such triads, the probability that any triad in an ensemble has entered such a state is

$$p_{SL/E(PE\ SIMF/SIMPLE)}(t) = 1 - (1 - p_{SL/TR(PE\ SIMF/SIMPLE)}(t))^{N_{TR}}$$
$$\approx N_{TR} p_{SL/TR(PE\ SIMF/SIMPLE)}(t)$$
$$= N_{TR} 6\lambda_{PE}^2 t/\mu.$$

No quantity in this formulation depends on any of the architectural parameters which will be varied. Therefore the probability of ensemble failure due to near-simultaneous triad failures is not a function of these parameters, and hence this failure mode need not be considered in the optimization process. Also note that λ_{PE} is as low as it can possibly be and still allow the processor to be connected to anything by a communications port.

The probability that the cluster's Network Element aggregate suffers a near-simultaneous failure can be conservatively approximated in a manner similar to that for triad simultaneous failure. The resulting approximation for the probability that the ensemble suffers loss due to near-simultaneous Network Element failures is

$$p_{SL/E(NE\ SIMF/SIMPLE)}(t) \approx N_C N_{NE/C}(N_{NE/C} - 1)\lambda_{NE}^2 t/\mu.$$

Recall that

$$\lambda_{NE} = \lambda_{NE0}(1 + f_{port}(N_{NE/C} - 1 + N_{PE/NE})).$$

Then

$$p_{SL/E(NE\ SIMF/SIMPLE)}(t) \approx N_C N_{NE/C}(N_{NE/C} - 1)[\lambda_{NE0}(1 + f_{port}(N_{NE/C} - 1 + N_{PE/NE}))]^2 t/\mu.$$

For brevity define $p=p_{SL/E(NE\ SIMF/SIMPLE)}(t)$, $f=f_{port}$, $a=N_C$, $b=N_{NE/C}$, and $c=N_{PE/NE}$. Also divide the above expression by $\lambda_{NE0}{}^2 t/\mu$. Then

$$p = ab(b\text{-}1)(1+f[b\text{-}1+c])^2 .$$

By the constraint on the number of triads in the ensemble, we know that abc=constant. Substituting constant/bc for a yields

$p = (b\text{-}1)/c(1+f[b\text{-}1+c])^2$, where p has been redefined to be p/constant.

This relationship shows that, regardless of the values of f or c, p is monotonic with respect to b, implying that from the point of view of simultaneous Network Element failure resilience it is desirable to minimize the number of Network Elements per cluster. From the requirement of Byzantine Resilience on the cluster, the number of Network Elements must be at least 4. Therefore tentatively set b_{opt}=4 in the sequel.

To find the optimal number of Processing Elements per Network Element, differentiate p with respect to c and set equal to zero. This procedure yields

$$c_{opt}=1/f+b\ \text{-}1 .$$

The intuitive explanation for the leading $1/f$ term is that the cheaper, i.e., more reliable, the ports to the subscriber processors, the more processors should be added to the Network Element because is is cheaper to add a port to an existing Network Element than to add another Network Element. For the baseline parameter of f=0.1 and for the optimal value of b_{opt}=4, we have that from the point of view of minimal simultaneous NE failure probability one would take c_{opt}=13. From an implementation point of view it may not be desirable to support so many subscriber processors because of the problems of contention for the Network Element resource (a phenomenon modeled in Chapter 11). Moreover, it will be shown to be undesirable from the point of view of attrition resiliency to put so many eggs in one basket. Therefore it is instructive to examine the variation of p with c (Figure 7.13). From this curve, it can be seen that the minimum is very shallow, and that most of the benefits of the optimum value can be obtained with fewer than optimal processors per Network Element. This fact will be used to advantage when attrition resiliency is

discussed.

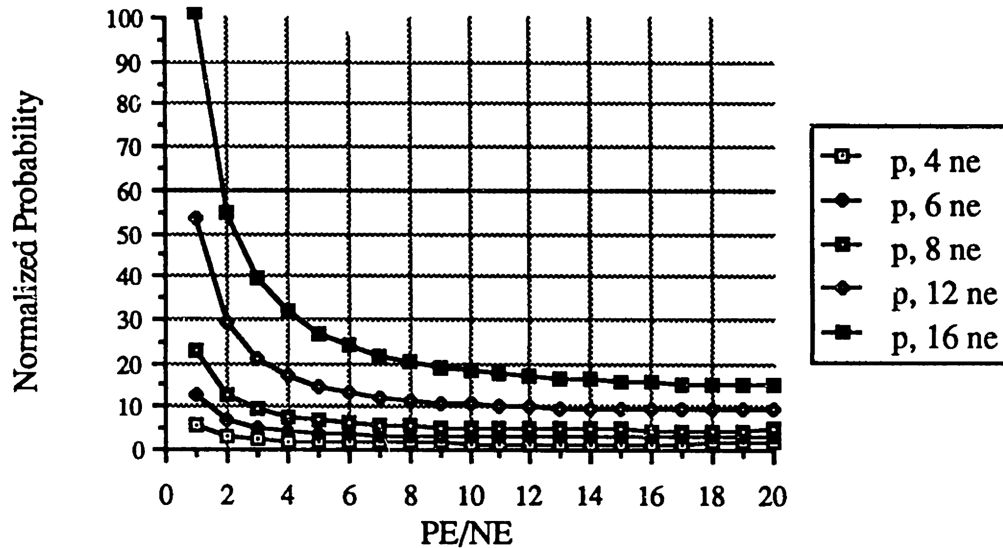


Figure 7.13

Sensitivity of Normalized Near-Simultaneous

NE Failure Probability to $N_{PE/NE}$

Using the optimal values of 4 Network Elements per cluster and 13 Processing Elements per Network Element, we obtain $\lfloor 4*13/3 \rfloor = 17$ triads per cluster. Therefore we will need $\lceil 64/17 \rceil = 4$ clusters to have 64 triads at mission beginning.

7.16 Optimization of Attrition Resilience

In the previous paragraphs it was shown that the parameters $N_C=4$, $N_{NE/C}=4$, $N_{PE/NE}=13$ yielded an architecture which was minimal with respect to near-coincident failure probability. We now wish to calculate the attrition resilience of this architecture. Attrition resilience depends upon the number of spares which are available. The optimal architecture had 17 triads per cluster, but because of the 4*13=52 processors actually resident in the cluster, there were spare processors in the cluster because 52 does not divide

evenly by three. Also, there were 4 clusters required to make up the required 64 triads since each cluster had 17 triads. Assuming equal distribution of the spare processors across the clusters, each cluster has 4 spare PEs. In order to dissociate the issue of initial spares from the effects that we wish to model in the attrition study, namely the effects of reconfigurability and its concomitant complexity on attrition resilience, we will choose architectures which have a number of processors per cluster which is exactly divisible by three, so that there are no spares at mission beginning. The closest such architecture to the optimum derived above has the parameters $N_C=4$, $N_{NE/C}=4$, and $N_{PE/NE}=12$. These baseline parameters will be varied to determine their impact on attrition resilience.

Most ultra-reliable systems are designed to be highly reliable for short mission times. This implies massive redundancy, the usual effect of which is to actually reduce the Mean Time To System Failure [Siewiorek82] because, after a point, the likelihood that the requisite majority of components is failed is higher than the likelihood that the fewer components comprising a non-redundant implementation have failed. There are simply more components to fail in a redundant system. Therefore MTTSF has not been considered a meaningful performance parameter for high-reliability systems. However, in this case, we specifically wish to study attrition resilience, a failure mode which is important only in the long term. It is a unique characteristic of parallel systems that a single system can potentially meet both the short-term ultra-high reliability requirements and the longevity requirements. In this case, the MTTSF does indeed become a meaningful performance parameter alongside short-term reliability, and will therefore be used to quantify attrition resilience. It should be stressed that for mission times shorter than the MTTSF alternate longevity parameters may be used. We use MTTSF because it is general.

Table 7.1 shows the combinations of design parameters which yield exactly 64 triads with no spares, alongside the values of MTTSF derived using the above formulation. These data are plotted in Figure 7.14. The full curves showing system loss probability versus time for all these cases are included in Appendix A.

$N_{NE/C}$	$N_{PE/NE}$	N_C	MTTSF (K Hours)	Dominant Failure Mode
4	3	16	32	attrition
4	6	8	34	attrition
4	12	4	27	attrition
6	4	8	41	attrition
6	8	4	41	attrition/cluster isolation
6	16	2	24	cluster isolation
8	3	8	42	attrition
8	6	4	45	attirition/cluster isolation
8	12	2	33	cluster isolation
8	24	1	35	attrition
12	2	8	42	attrition
12	4	4	46	attrition
12	8	2	40	cluster isolation
12	16	1	48	attrition
16	3	4	46	attrition
16	6	2	42	cluster isolation
16	12	1	50	attrition

Table 7.1

MTTSF Versus Architectural Design Parameters[1]

The curves in Figure 7.14 show that a large jump in attrition/isolation resilience occurs when we go from the baseline of 4 Network Elements per cluster to 6 Network Elements per Cluster. In both cases, the ideal number of Processing Elements per Network Element is between 6 and 8. Referring back to Figure 7.13 shows that increasing the number of Network Elements per cluster from 4 to 6 results in an increase in normalized simultaneous failure probability from 2 to 3.5 for $N_{PE/NE}=6$ and from 1.6 to 3.3 for $N_{PE/NE}=8$. A general judgement regarding these tradeoffs is impossible to make, so we will with a degree of arbitration select the architectural parameters to be $N_{NE/C}=6$, $N_{PE/NE}=8$, and $N_C=4$.

[1] $\lambda_{IOFO}=10^{-5}$/hour.

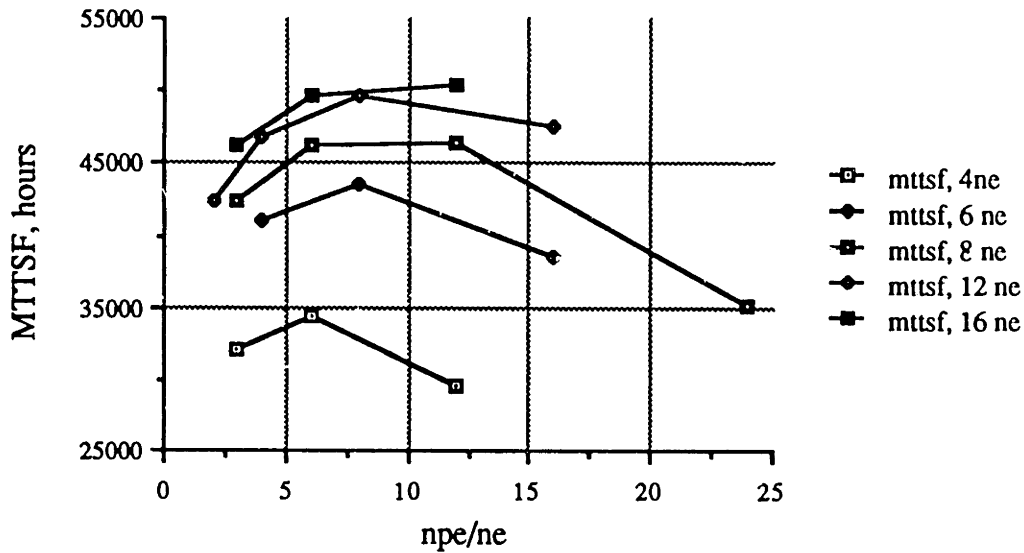


Figure 7.14

MTTSF Versus Architectural Design Parameters

7.17 Results for the Optimized Architecture

The system loss probability curve for the optimized architecture is plotted in Figure 7.15. The expected value of the number of Processing Elements for this architecture is plotted in Figure 7.16.

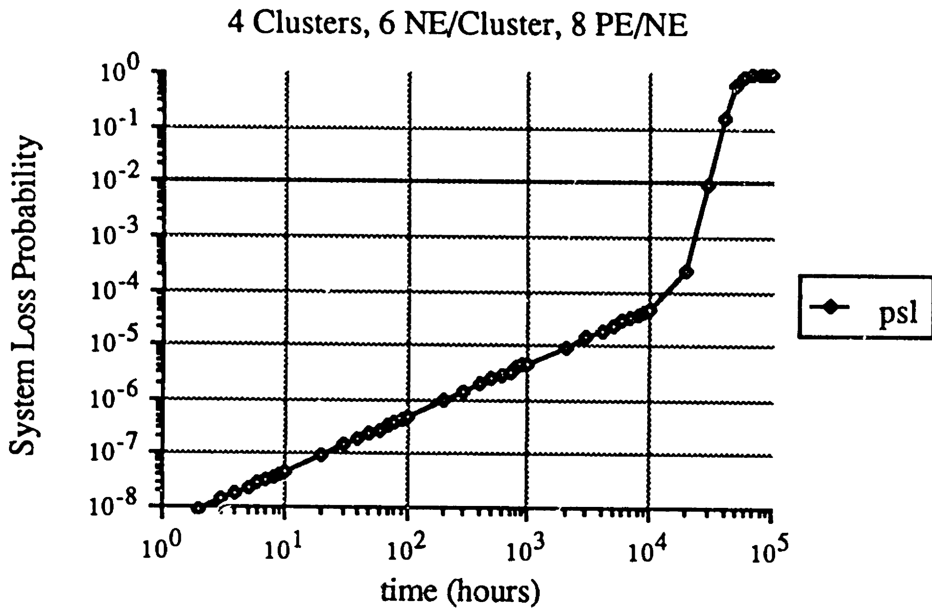


Figure 7.15

Ensemble Loss Probability for 64-FMG Network Element-Based Ensemble

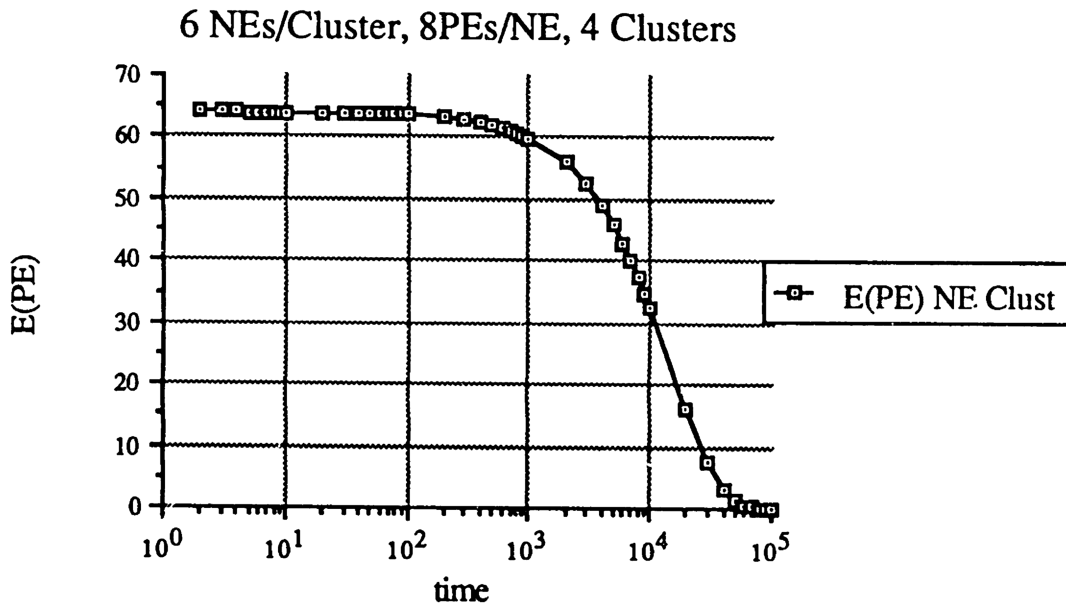


Figure 7.16

Expected Value of Number of FMGs for 64-FMG Network Element-Based Ensemble

Finally, the table of performance parameters for the optimal architecture derived above is presented in Table 7.2.

Parameter	Value
# PE	192 (24 Network Elements)
#ports	192 + 24*(5+8)
system loss probability	
10 hours	5 E-8
100 hours	5 E-7
1000 hours	5 E-6
10000 hours	5 E-5
$E_{PE}(t)$	
10 hours	64
100 hours	64
1000 hours	60
10000 hours	33
mean time to system failure	43,568 hours
intra-group communication	
distance	3
diagnosability	possible interference of NE and PE failures
anisotropy	none
inter-group interactions	none
inter-group communication	
distance	3 (intra-cluster) 3 + lg 4 (extra-cluster)
diagnosability	good (intra-cluster)
anisotropy	some, intra-cluster vs. extra-cluster
transparency of FT technique to programmer	
High transparency if implemented correctly	
efficiency of FT technique	
Requires processor triplication	

Table 7.2

Performance Parameters for 64 FMG

Network Element-Based Ensemble

CHAPTER 8

RELIABILITY ANALYSIS CONCLUSIONS

8.1 Summary of Quantitative Analysis

The quantitative analyses are summarized in Figures 8.1 through 8.4 for the following architectures:

64 simplex processing elements;

64 duplex processing Groups;

64 SIFT-like quadruplicated Fault Masking Groups;

64 FTP-like triplicated Fault Masking Groups;

32 8-processor fully-connected clusters;

4 Network Element-based clusters each having 6 Network Elements and 8 processors per Network Element.

Figure 8.1 shows the system loss probabilities for these cases versus mission time. Figure 8.2 shows the expected value of the number of processing sites versus time. Figure 8.3 shows the slope of the short-term system loss probability curve, $p_{SL/E(TOTAL)}(t)/t$, for each architecture. This quantity can be considered to be the short-term failure rate of the ensemble since in the short term system loss probability, $p_{SL/E(TOTAL)}(t)$, is linear with time. Figure 8.4 shows the MTTSF for each architecture. The conclusions which can be drawn from these analyses are summarized below.

8.2 Conclusions From Simplex Analysis

The foremost conclusion that may be drawn from the simplex analysis is that the ensemble composed of simplex processing elements does not meet the reliability requirements of the mission specification. This is a direct result of the low coverage of fault detection techniques based on restrictive hypothetical models of failure behavior, as can be seen from the sensitivity of the results to the coverage parameter. Therefore more

sophisticated methods of fault detection must be employed in the application of interest.

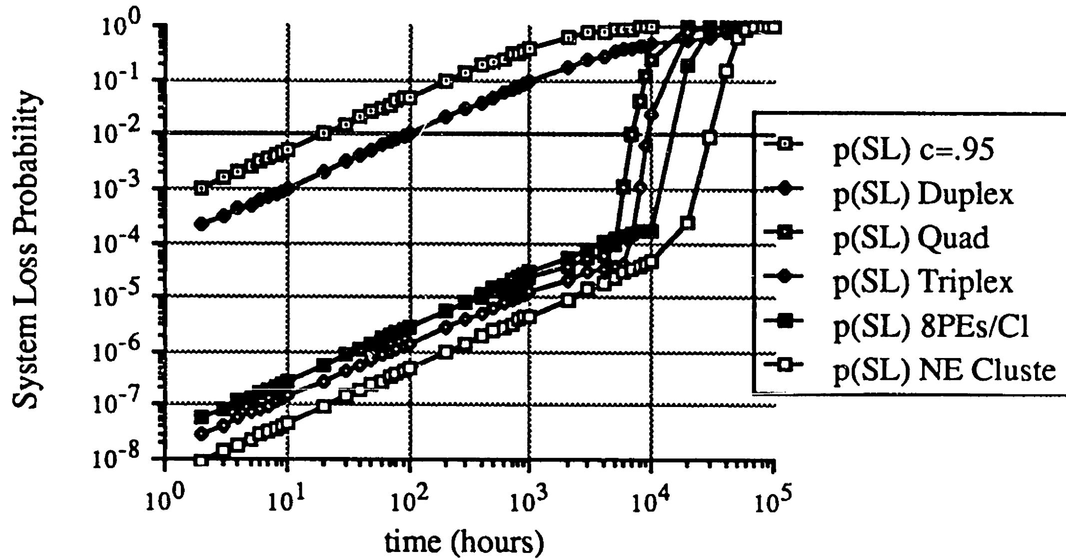


Figure 8.1

Comparison of Ensemble Loss Probabilities vs. Mission Time

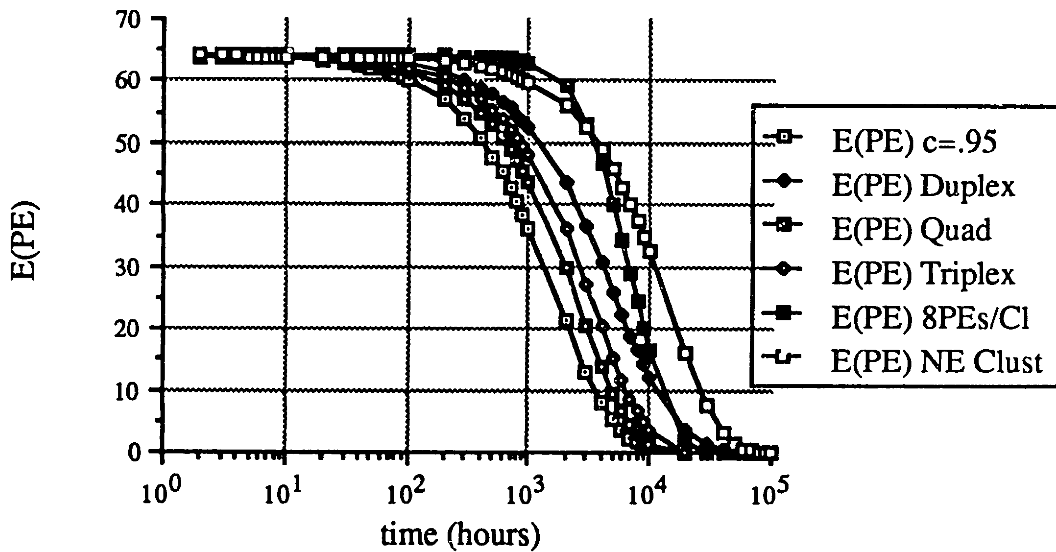


Figure 8.2

Comparison of Expected Number of Processing Groups vs. Mission Time

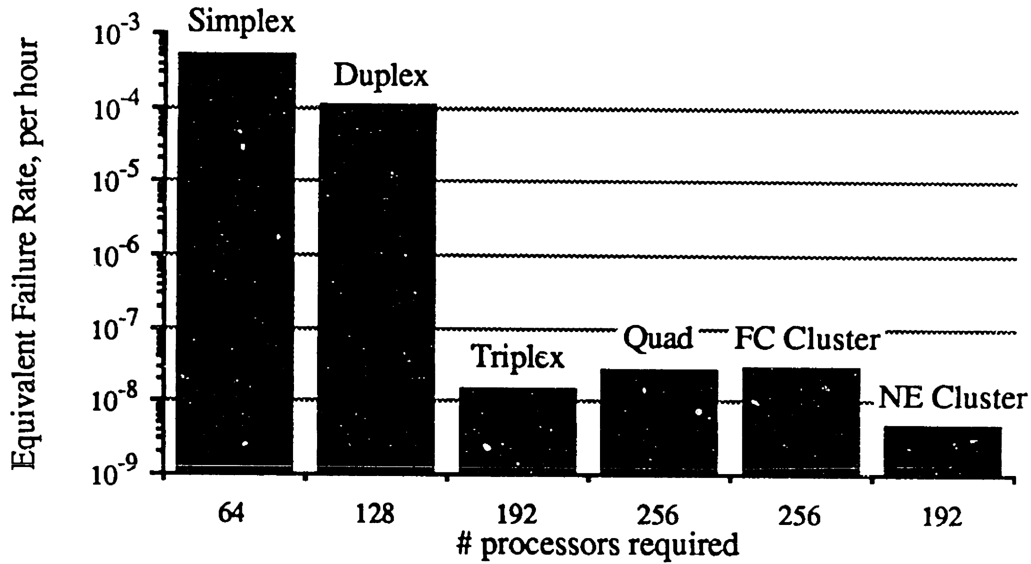


Figure 8.3

Comparison of Ensemble Short-Term Failure Rate

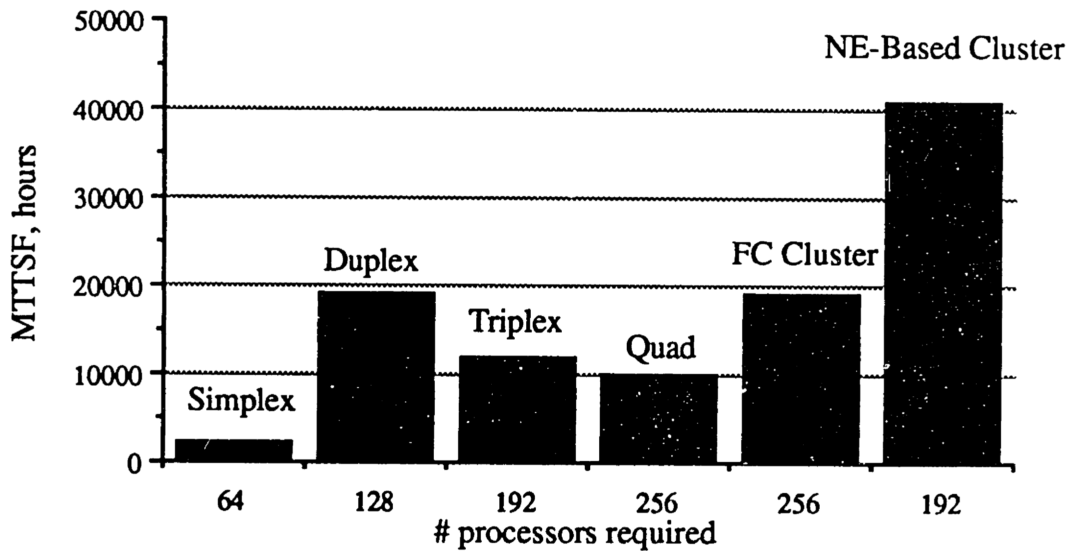


Figure 8.4

Comparison of Mean Time to System Failure

8.3 Conclusions from Analysis of Duplex Ensemble

From an examination of the requirements for consistent distribution of inputs to the duplex channels, it has been shown that the input consistency requirements upon which

the duplex paradigm depends for high-coverage fault detection cannot be met using duplicated components. Specifically, a traditional duplex architecture does not meet the cardinality constraint for Byzantine Resilience. Therefore it cannot tolerate arbitrary failure modes during input data distribution or output comparison.

Second, it was shown that, in the self-checking pair architecture, the output comparison mechanism was a single point of failure and that its failure rate lower-bounded that of the duplex processing site.

Third, the overhead of global management of rollback points can result in a significant reduction in throughput of the duplex units.

> "In a distributed environment, the dynamic (automatic) maintenance of recovery lines for large task forces that exhibit a high degree of intertask communication requires complex protocols and a potentially large amount of overhead [Hosseini83], [Tamir84]. Alternatively the responsibility for establishing recovery lines can be placed within the applications themselves, by requiring the tasks of a task force to explicitly interact in the setting of recovery points. Between these two extremes lies the notion of restricting the programming model available to the application designer, in a manner that allows easy management of the recovery line by the system [Anderson81]. Whichever method is utilized, the problem is compounded by the fact that the procedures for establishing and maintaining recovery lines must themselves be made robust with respect to failure." [Kuhl86]

Therefore, to obtain a given amount of useful processing power using duplexes incorporating checkpointing and rollback, the actual number of duplex processing units must be increased to compensate for the throughput lost in executing the fault tolerance overhead. If this overhead is monotonic in the size of the ensemble then a point will be reached at which the use of triplex FMGs is more efficient. Consequently it is possible that, counterintuitively, a fault masking paradigm incorporating triplication of the processing elements may result in a more efficient utilization of the processing elements and hence

actually *lower* total hardware requirements than a duplex paradigm. A naive analysis of the necessary circumstances for this to be the case reveals that if the net overhead of rollback point insertion and actual rollback exceeds 33% of a duplex ensemble's throughput, then the use of fault-masking triplex processing elements yields the requisite throughput at a lower cost in hardware. Hard figures for this overhead are not available for a parallel system, but it is not uncommon for these functions to consume well over 25% of the throughput of a duplex uniprocessor. In addition to operational efficiency one must consider the difficulty with which one generates and validates the duplex code compared to the same effort for a fault-masking architecture.

Finally, when the fault recovery technique involves retry, it is possible that the system may miss an important deadline because of time spent in the rollback process. This failure mode is known as "dynamic failure" and is further discussed in [Krishna83].

8.4 Conclusions From the Fault Masking Analysis

The use of an ensemble composed of FTP-like or SIFT-like Fault Masking Groups (FMGs) results in a significant increase in ensemble reliability over both the simplex ensemble and the duplex ensemble, in both the short-term and long-term. For these ensembles, the expected value of processing elements was uniformly greater than for the simplex ensemble, but uniformly lower than for the duplex ensemble. In the FMGs the processing elements have more communications ports to support, resulting in increased processing element complexity and a resultant higher rate of occurrence of covered FMG failures. Therefore attrition occurs at a rate which is higher than that of the duplex-based ensemble.

The MTTSF of the FMGs was uniformly larger than that of the simplex-based approach but smaller than that of the duplex-based ensemble. This unusual former result is due to the dominance of the probability of simultaneous or uncovered failures in the formulation of ensemble loss probability. Usually, a redundant system has a lower MTTSF

than a simplex system. This is because after a point in the mission is reached at which the component reliability is below 0.50, the probability that the f+1 components required for the redundant system to mask faults is less than the probability that the single component of the nonredundant system is still operational. In such a formulation, system loss due to covered failure is given the same importance as system loss due to uncovered failure, in the sense that total system loss probability is the sum of the uncovered failure probability and the covered failure probability. In the ensembles under consideration in this thesis, system loss is defined to occur when *any* processing site enters a state of uncovered failure or *all* processing sites suffer covered failures;

$p_{SL/E(TOTAL)} = 1-(1-p_{SL/E(UF)})^N + p_{SL/E(ATT)}^N$, or approximately,

$p_{SL/E(TOTAL)} \approx Np_{SL/E(UF)} + p_{SL/E(ATT)}^N$.

From this it can be seen that the two failure modes are unequally "weighted", with the superior coverage of the fault-masking designs emphasized in this formulation while their inferior attrition resilience is attenuated. This is an interesting case of the inapplicability of a classical rule of thumb to a parallel system.

A useful side-effect of the analysis was the demonstration that, for FMGs, near-coincident failures of the components composing a given FMG dominated system loss probability in the short term, while attrition of these FMGs dominated this quantity in the long term. The former was further shown to be closely overapproximated by a linear function of time whose coefficient had a quadratic relationship between processor failure rate and probability of system loss due to near-coincident failures. The latter was shown to be closely approximated by a combinatorial formulation if the component failure rate is replaced by the rate of occurrence of permanent and intermittent failures. These conclusions simplified the subsequent analyses.

8.5 Conclusions from the Fully Connected Cluster Analysis

In an attempt to increase the longevity of the ensemble, the processing elements were

arranged into fully connected clusters such that any processing element could be grouped with any other to form an FMG. It was found that this did indeed increase ensemble longevity, as indicated by probability of ensemble attrition and expected number of processing elements, at the cost of an increase in complexity of the processing elements. This complexity increase in turn resulted in an increase in the short-term probability of near-coincident failures, which was shown to be proportional to the square of processor failure rate. It was found that a significant increase in longevity occurred in increasing the cluster size from 4 (the minimal size of a cluster and identical to the SIFT-like architecture modeled above) to 8, whereas no further significant increase in longevity was noted in increasing cluster size to greater than 8. To keep near-coincident failure probability at a minimum while retaining some benefits of reconfigurability a cluster size of 8 was chosen for comparison with the competing approaches.

8.6 Conclusions from the Network Element-Based Cluster Analysis

The interposition of Network Elements between sets of subscribing processing elements as regions of Byzantine Resilient consistency resulted in an increase in ensemble reliability over the entire ensemble lifetime. This is a result of the relatively low failure rate of the Network Element coupled with the reduction of the failure rate of the processing element to its lowest possible value. The constraint that different members of a given FMG subscribe to different Network Elements within a cluster did not appear to significantly reduce the benefits of reconfigurability.

It was also found that the probability of ensemble loss is sensitive to the failure rate of the IOEs connecting the clusters, to the extent that if the IOE failure rate is the same as a processor failure rate, the probability that all clusters are isolated from each other is significantly greater than the probability that no FMGs can be formed due to attrition. Reduction of the IOE failure rate by offloading its sophisticated functionality onto one or more processing elements residing on the same network element as the IO element was

seen to be effective in adequately reducing IOE failure rate, such that cluster isolation probability no longer dominated the ensemble loss probability as a failure mode.

The expected value of the number of processing sites for this architecture fell below that of the fully-connected cluster early in the mission, only to cross it at a point and remain above the latter thereafter. An explanation for this is that the 8-processing element cluster to which the comparison was made contained two spare processors at mission beginning, compared to zero spares at mission beginning for the Network Element-based cluster under consideration. Furthermore, the Markov modelling technique used in the fully-connected case was capable of modelling the assumption that unused processors had a failure rate of zero, as opposed to the combinatorial modelling technique used in the case of the Network Element-based cluster which modelled that all processors, unused or not, failed at their normal failure rate. Thus in the fully-connected case an overestimate of system performance was obtained, and in the NE-based case an underestimate was obtained. This is not serious in this comparative analysis, especially as it can be seen that, on the whole, the overestimate of performance is worse than the underestimate.

It was found that there exists an optimal number of Network Elements per cluster and processing elements per Network Element. Simple formulations were derived to allow the determination of these parameters. For the baseline architecture, these were determined to be four clusters composed of six Network Elements, each of which possessed eight subscriber processing elements.

8.7 General Conclusion

The above analyses indicate that the use of Network Elements to construct clusters of processors results in an improvement in short-term reliability, attrition resilience, and throughput as expressed by the expected value of the number of processing sites. The short-term and long-term reliability goals described in Chapters 1 and 2 can indeed be met by an ensemble composed of Network Element-based clusters. The thesis will now turn to

the implementation of such an ensemble and calculations of the efficiency of that implementation.

Chapter 9.
Consistency Maintenance

9.1 Introduction

Previous chapters showed that for the ensemble to be adequately reliable for the application of interest a fault tolerance scheme must be used which tolerates arbitrary failure behavior of a subset of the ensemble's components. This requirement led to the use of Byzantine Resilient Fault Masking Groups (FMGs) as the computational elements of the ensemble. It was also shown that by the interposition of a Byzantine Resilient aggregate of Network Elements (NEs) between the members of the FMGs, the complexity and failure rate of the ensemble could be reduced, the short term reliability improved, and the attrition resilience increased.

The second part of this thesis investigates mechanisms whereby the ensemble is able to achieve Byzantine Resilient operation. The key concept which underlies the capability of supporting such operation is that of *consistency*. Using Byzantine Resilient guarantees of certain consistency abstractions, one is able to construct a useful communications concept called the Byzantine Resilient Virtual Circuit abstraction.

The concept of consistency is best introduced by summarizing the state machine approach used to obtain ultra-high ensemble reliability [von Neumann56]. Redundant state machines are grouped into Fault Masking Groups (FMGs) which execute bitwise identical computations on bitwise identical inputs. Voting of their outputs provides near-unity coverage of arbitrary failure behavior on the part of f FCRs of the FMG. A discrepancy in the vote indicates that a faulty copy of the execution exists, assuming that the redundancy of the FMG has not been overwhelmed by faults.

For the voting process to have any effectiveness certain assumptions about the input distribution process must hold. Specifically, it must be guaranteed that the non-faulty members of an FMG are provided with bitwise identical inputs upon which to base their

computations. If there is some possibility of ambiguity in the order in which different non-faulty FMG members receive inputs, it must also be guaranteed that the non-faulty members all agree on the order in which to process the inputs. Finally, it must be guaranteed that corresponding actions of the non-faulty members occur within a finite and known amount of time of each other. This last condition is not only trivially obvious in the case of the arrival of the members at a diagnosis point, thus allowing the differentiation of a halted processor from a non-faulty one, but it is a rigorously demonstrable precondition on the ability to achieve the first guarantee on bitwise identical inputs. The above three guarantees must be made in the presence of faults occurring during the input distribution and synchronization processes.

9.2 Consistency Abstraction

The use of abstractions greatly reduces the cognitive overhead involved in reasoning about the behavior of complex systems. Abstractions suppress irrelevant details about how a given feature is achieved and simplify complex behavior into a few easily comprehensible characteristics, thereby allowing the user of the abstraction to concentrate on the application-specific problem at hand. Much research is focused on the determination of appropriate primitives for programming of parallel computers (e.g., fetch and add, rendezvous, etc.), but no generally accepted consensus presently exists. [Schneider86] presents two useful abstractions for fault tolerance in distributed systems. They are the *agreement abstraction* and the *order abstraction*. To paraphrase [Schneider86],

Agreement: Every non-faulty member of an FMG receives an identical copy of every input.

Order: Inputs are processed in the same order by every non-faulty member of an FMG.

Abstractions such as these are intended to be the substrata for the primitives of the chosen programming model, vastly simplifying their development by obviating consideration of faulty component behavior in the implementation of that model. In this spirit this section

will extend this set of abstractions by adding the *time abstraction*. These three abstractions together will be denoted the *Consistency Abstraction*. Byzantine Resilient guarantees of these abstractions will be sought, and a higher level abstraction called the Byzantine Resilient Virtual Circuit will be introduced.

9.2.1 Consistency in Data

Two data abstractions are useful, the input data consistency abstraction and the output data consistency abstraction. The input data consistency abstraction is a statement that the data presented to the inputs of all non-faulty members of an FMG are bitwise identical. Figure 9.1 is a representation of this abstraction. Note that the input data consistency abstraction does not imply that all members of an FMG receive the value x sent by the source S, but that they all receive z, a value which may or may not be equal to x but will certainly be the same for each member. Faults occurring in the distribution of the single input datum x may cause this datum to be corrupted to some value z, but the input data consistency abstraction guarantees that all non-faulty members of an FMG receive the same, albeit an erroneous, value.

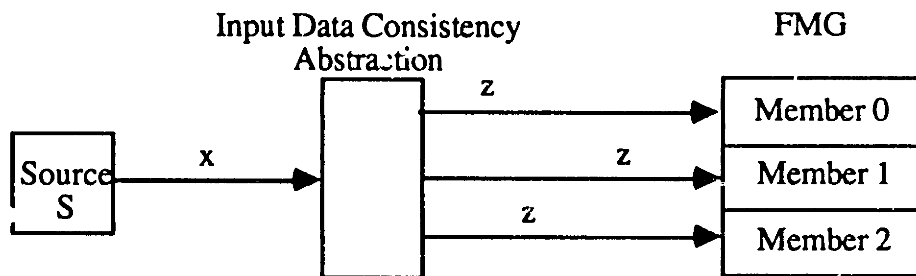


Figure 9.1

Input Data Consistency Abstraction

The input data consistency abstraction is useful but does not go far enough in abstracting reliable inter-FMG communication. Therefore define the output data consistency

abstraction as the guarantee to both the sender and a recipient of an inter-FMG message that non-faulty members of a recipient FMG receive identical and correct copies of the (assumed common) datum sent by the non-faulty members of a sending FMG. This is clearly a simple extension of the input data consistency abstraction. If the members of the sending FMG have member-specific data to send, then the input data consistency abstraction will hold for the transmission from each member.

The input data consistency and the output data consistency abstractions together imply that a higher level abstraction holds, that of guaranteed delivery of what was sent.

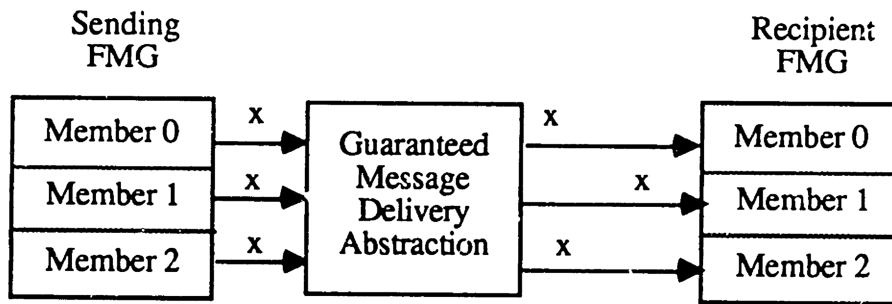


Figure 9.2

Guaranteed Message Delivery Abstraction

Inputs may be classified according to the redundancy level of the source. The source may be a simplex input, e.g., one emanating from a single channel of an FMG, or the source may be a redundant input, e.g., one value which is consistent in the case of non-faulty members, emanating from all members of an FMG. In the case of the former, a f+1-round interactive consistency protocol must be executed to guarantee the input and output data abstractions. In the case of the latter, a simple one-round broadcast/vote protocol suffices for the guarantee to hold.

The performance of data consistency maintenance techniques may be measured by two criteria. The first is the operational efficiency of the implementation. Because a demonstrable lower bound exists on the amount of communication required to achieve

interactive consistency, it is important to support this requisite communication with a reasonable degree of efficiency. Historically, hardware-intensive implementations have been more efficient than software-intensive implementations, but a tradeoff to be considered is the necessity of construction of fault tolerance-specific hardware in the former and the constraints such hardware may impose. The second, more important, criterion is the conceptual transparency of the implementation. As an example, the time required for the communications and voting involved in consistency maintenance cannot be reduced to zero in any physical implementation. Therefore some mechanism must ensure that processors do not read the result of the agreement protocol until the execution of the protocol is complete. In the SIFT computer, the applications programmer must personally determine that the data consuming task does not get scheduled so soon after the data producing task that the agreement protocol has not been completed and an incorrect result is read. As a second example of an inadequately transparent abstraction implementation, the use of a fourth processor to achieve consistency in a basic triplex FMG, as intimated in Chapter 6's discussion of fully-connected cluster-based architectures, requires elaborate scheduling of consistency exchanges in all processors, and must be driven by the applications programs. These are examples of inadequate conceptual transparency of the data consistency abstraction.

9.2.2 Consistency in Ordering

The consistent input ordering abstraction is a guarantee that the order in which input data are processed by all non-faulty members of an FMG is identical. This guarantee is important in parallel systems which are by their nature highly asynchronous. In this thesis we will deal with ordering of inter-FMG messages in the ensemble. Messages arriving at different FMG members at very close to the same time must be processed in the same order in each FMG, otherwise divergence of the computations under execution in the members will occur and chaos will ensue.

Figure 9.3 depicts the consistent input ordering abstraction Again, note that even though all members receive the same partial ordering of inputs "z x y", this order is not necessarily the same as the input ordering "x y z" sent by the source S, because faults in the distribution of inputs can occur, changing this ordering. However, the ordering will be the same at each member of the FMG.

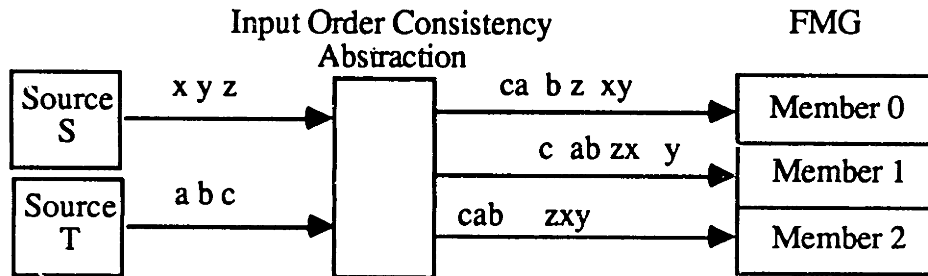


Figure 9.3

Input Ordering Consistency Abstraction

Analogous to the output data consistency abstraction, define the output ordering consistency abstraction as a guarantee that the non-faulty members of a recipient FMG receive the transmissions of the non-faulty members of a sending FMG (which are assumed to be transmitted in the same order in each non-faulty member of the sending FMG) in the order sent.

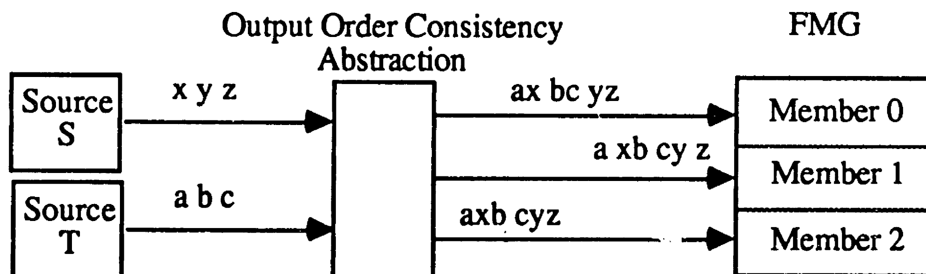


Figure 9.4

Output Ordering Consistency Abstraction

Provision of order consistency is isomorphic to provision of data consistency, since the redundant members of the recipient FMG could always perform member-specific data consistency protocols until some ordering agreement condition is met. The performance parameters of order consistency maintenance methods are similar to those of data consistency maintenance, being composed of operational efficiency and conceptual transparency.

9.2.3 Consistency in Time

The consistency in time abstraction is a statement that the times of occurrence of certain unambiguously-defined actions performed by the members of an FMG differ by at most a finite and known amount, denoted "δ" in Figure 9.5, which portrays the sending of a message as one of these actions.

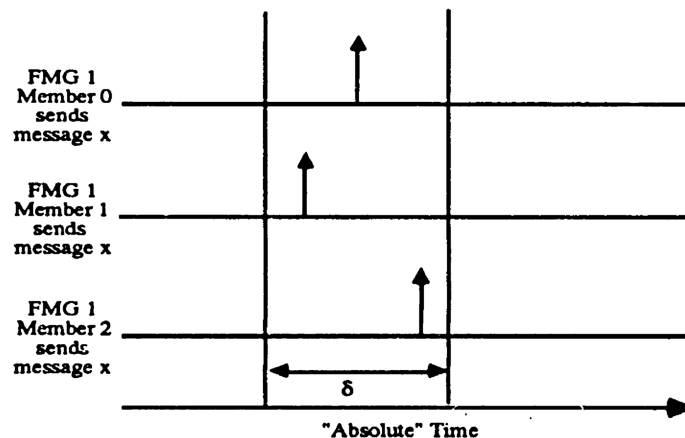


Figure 9.5

Consistency in Time Abstraction

To study the problem of consistency in time it will be useful to first construct a framework for thinking about the problem. Define a *frame* to be the occurrence of a specified number of *events*. An event is defined to be any countable and unambiguously definable activity of the components making up the redundant group. In all synchronous

fault tolerant computers, the frame is by definition the fundamental unit of synchrony. Different implementations merely differ in how the frame is defined and how long it lasts. A frame in this discussion corresponds to the *segment instantiation* in [Friend86].

The number of events per frame need not be a constant from frame to frame, although it has been in all Byzantine Resilient computers constructed or proposed to date. Events may be generated by either hardware activity, such as the expiration of a countdown timer, or software activity, such as operating system traps. Examples of events which could be used to define a frame include a given number of instructions, processor clock cycles, operating system traps, or iterations of an iterative algorithm. The above definitions of frame and event capture the means used to obtain event timing, the periodicity, and the period of the frame.

Let f_{rk} denote the time of the beginning of frame k on processor r, as measured by an atomic clock which is at rest with respect to all members of the redundant group. Assume that the members of the group are sufficiently proximate that the time delay due to propagation of electrical signals between them is negligible. Then the frames executed by the redundant members of a group are said to be *synchronous* if $|f_{pi}-f_{qi}|<\delta$, $\forall$ p,q,i, where δ is some constant (Figure 9.6).

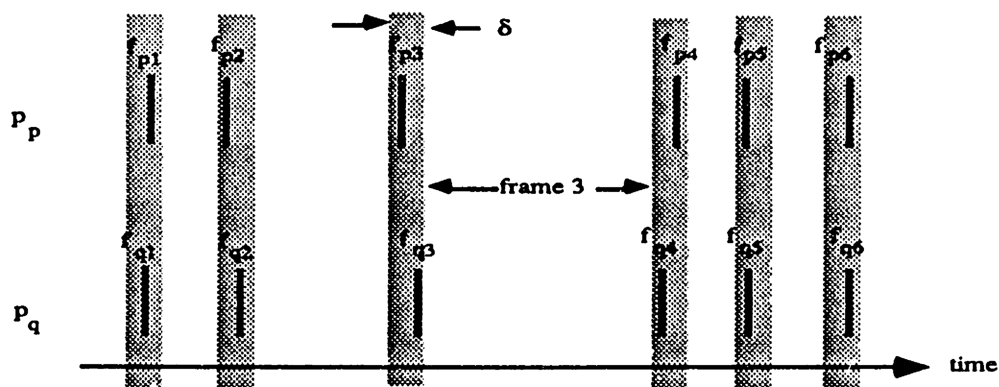


Figure 9.6

Framewise Synchrony

So far no mention has been made of the actions performed by the members of the group. The group members may execute synchronous frames without executing the same set of actions within those frames. The SIFT does exactly this. To express the concept of group members which in some sense execute the same actions within a frame, define *functional congruence.* The members of a redundant group are functionally congruent if they, in an unambiguously definable sense, perform the same function F. An assertion of functional congruence does not necessarily imply that redundant sites perform the same *actions*, only that at some level of abstraction they may be regarded as having performed the same *function.* For example, members of a group may perform the same arithmetic calculation, but one member may have a faulty memory which needs error correction on every memory fetch. This member has not performed exactly the same set of actions as its counterparts, but, assuming the memory error is correctable, is has performed the same function.

Combining the two notions of synchronous frames and functional congruence, define *framewise functional congruence* as the situation which obtains when all members of a redundant group *g* perform the same function F_i in the same frame i. Explicitly, p and q are functionally congruent if $F_{pi}=F_{qi}$, $\forall$ p,q $\in$ *g*, $\forall$ i.

Existing Byzantine Resilient computers differ more in the way in which they implement synchronous framewise functional congruence than in the fact that they implement it. These implementation schemes have a strong impact on the performance and utility of the resultant computers.

Performance Measures for Synchronization Schemes

Granularity of Action: One performance measure for synchronization schemes is the granularity of action that they permit. Granularity in this context is defined to be the minimum time difference achievable between an action performed by one member of a redundant group and another member of that same group, and is a fundamental limit on the temporal coordination obtainable by the members of a group. Systems having a fine

granularity of action have the property that the redundant sites can execute actions that differ temporally by a small upper bound. Such systems are necessary where rapid coordinated response to events must be provided, as in control applications. In systems having a coarse granularity of action, the difference in time between actions taken by different members of a group can be larger. Systems of this sort, such as Online Transaction Processing Systems, are useful in executing tasks that have no hard real time timing constraints, but in which fault tolerance is needed nevertheless.

Synchronization Frequency: A related classification basis is the frequency with which the synchronization step takes place. In any synchronization scheme, given a rate of open loop divergence of the clock speeds of members of a group, granularity is monotonic with the time between synchronizations. Systems that are synchronized relatively frequently have a fine granularity of action, while those which execute synchronization steps infrequently tend to have a coarser granularity of action. Synchronization frequency and granularity of action are impacted by the efficiency of the synchronization protocol in that, with efficient protocols, more frequent synchronization and hence a finer granularity of action can be supported.

Post-Synchronization Skew: Another way to grade synchronization schemes is by the theoretical lower bound on the inter-site skew that the sites can achieve after execution of the synchronization step. This clearly relates to the granularity issue, but in this context it is viewed separately as an intrinsic property of the algorithm. Synchronization methods based on an interactive convergence-based estimate of the pre-synchronization skew reduce the skew between sites to a constant fraction of the skew existing before the execution of the synchronizing step. Interactive convergence algorithms are less complicated and hence more efficient algorithms, but the resultant post-synchronization skew is greater than that of other schemes, thus requiring more frequent execution for a specified post-synchronization skew. Synchronization schemes based on interactive consistency-based estimation of the pre-synchronization skew reduce the inter-site skew to a bound which is independent of the

initial skew between nonfaulty sites, and dependent only upon the properties of the synchronization mechanism itself. These algorithms are more complicated in execution, but need to be executed less often for a given desired skew because the resultant post-synchronization skew can be reduced to the given value in one execution of the synchronizing step as opposed to repeated executions of the step in the case of the former.

There is an important quantitative connection between the performance of the synchronization scheme and the overall reliability of the redundant group. For concreteness, it will be assumed that the group of interest is a simple minimal Byzantine Resilient processor group (Figure 9.7), which was introduced in Chapter 4.

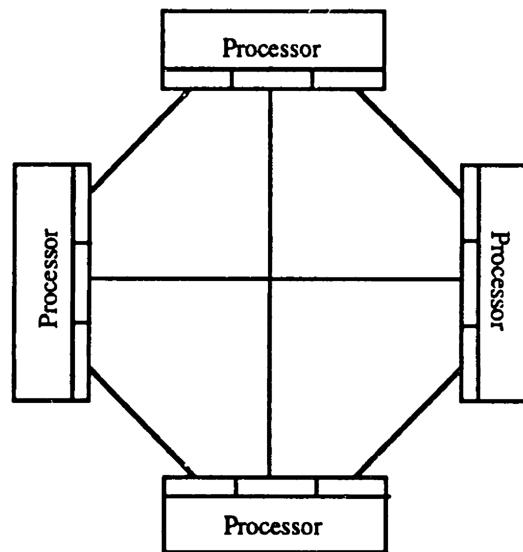


Figure 9.7

Minimal Byzantine Resilient Processing Group

The failure behavior of the group is modeled by the Markov process depicted in Figure 9.8. For simplicity it is assumed that the transient failure probability f_t is equal to zero.

The probability that the group suffers catastrophic loss due to a second processor failure while the group is in the process of shutting down from the first is

$$p_3(t) = \frac{12\lambda^2}{\mu-\lambda}\left[\frac{1}{4\lambda}(1-e^{-4\lambda t}) - \frac{1}{3\lambda+\mu}(1-e^{-(3\lambda+\mu)t})\right].$$

Typically λ is of the order of 10^{-4} failures per hour and μ is on the order of 10^3 to 10^4 reconfigurations per hour. Therefore $\lambda/\mu<<1$. Using this fact the above expression can be approximated by

$$p_3(t) \approx \frac{12\lambda^2 t}{\mu} \text{ for } \lambda t<<1.$$

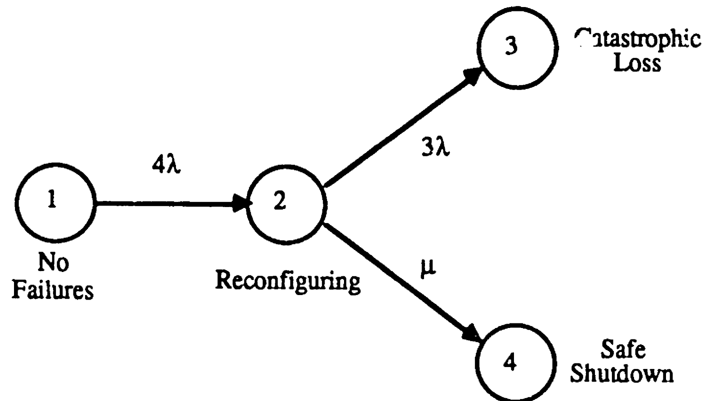


Figure 9.8

Markov Model for Symmetric 4-Processor Fault Masking Architecture

For short mission times the probability of group loss due to near-simultaneous failures is linearly proportional to $1/\mu$, the mean fault latency time plus time spent in reconfiguration. This time cannot be shorter than the time between comparison of outputs plus the time required to resolve any such comparisons, which in turn cannot be shorter than the skew between channels. The "tightness" of synchronization therefore has a direct impact on reliability. For synchronization schemes in which the synchronization points are far apart one would expect the probability of near-simultaneous failure to be higher than that of an identical tightly synchronous system in which the inter-synchronization time interval plus inter-member skew, and hence fault latency and reconfiguration time, is

short[1].

Finally, synchronization schemes may be parameterized based on the constraints that they impose on the redundant state machines to be synchronized.

Homogeneity Constraints: Certain synchronization methodologies impose constraints on the degree of disparity allowed among the redundant state machines. Certain synchronization mechanisms require that the actions of the redundant sites be precisely and deterministically related to the passage of time as measured by a hardware clock, such that each member of a replicated site must execute the same number of instructions in the same number of clock cycles. This constraint is called the *clock determinism* constraint. The clock determinism constraint disallows a wide class of otherwise acceptable behavior on the part of the redundant sites. For example, if each site has error-correcting memory and it is desired to synchronize these channels under the assumption of clock deterministic behavior, each channel must wait for the worst case error correcting time on each memory access because a given channel does not know when another channel might encounter a memory error and have to correct it. If each channel did not wait this worst case time, then the error correcting process, which could lengthen the number of clock cycles taken by one processor to execute a memory fetch, would cause the processors to lose synchronization. As a second and more far-reaching example, the assumption of clock deterministic processor behavior excludes the possibility of the use of hardware and software design diversity in a single FMG as a technique for the tolerance of generic or correlated faults.

Programming Constraints: In some implementations the synchronization scheme may constrain the applications programmer. Some schemes, such as hardware implemented lockstep schemes, are totally transparent to the programmer. Others assume that the tasks to

---

[1] Arguments to the contrary state that loosely synchronized members are less likely to suffer from correlated faults such as single event upsets. It is arguable that the reliability of the group is increased simply because the different members are not executing the same instruction at the time the common-mode fault occurs.

be executed on the system are iterative in nature, with a predetermined amount of computation to be done per iteration. The constraint imposed by this assumption is that the inter-synchronization interval is the same as the iteration interval. This may be the case in many real time control systems, but is clearly a major programming constraint when no natural iteration interval exists, and when the workload per imposed iteration interval may vary.

9.2.4 Definition of Consistency Maintenance

The Consistency Abstraction may be summarized as the provision of input and output data consistency, input and output order consistency, and consistency in time.

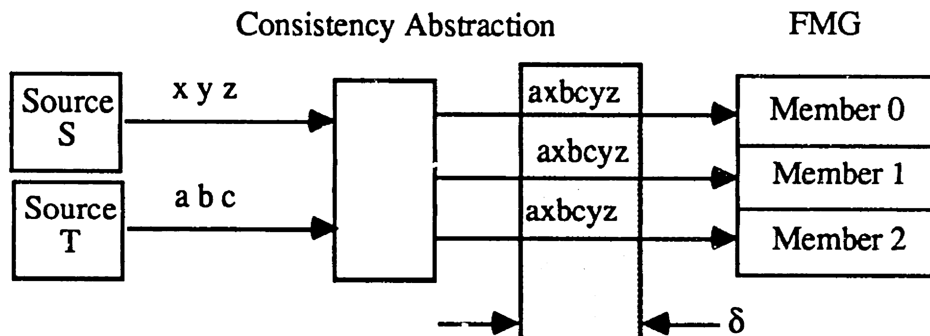


Figure 9.9

Consistency Abstraction

With the above as background it is possible to succinctly state the concept of *Consistency Maintenance*. Consistency Maintenance is defined to be the guaranteed provision of the Consistency Abstraction to the programmer of the ensemble, in the presence of arbitrary failure behavior on the part of a subset of the components of the ensemble. The preconditions for Consistency Maintenance will now be discussed.

9.3 Requirements for Consistency Maintenance

Input consistency maintenance mechanism or protocols have been extensively studied, resulting in the well-known and theoretically demonstrable prerequisites for consensus algorithms which are capable of correctly functioning in the face of arbitrary failure behavior on the part of "f" of the participants in the algorithm. These requirements, appropriate for processors which do not affix unforgeable digital signatures to the messages they send, may be summarized as:

1. There must be at least 3f+1 participants in the algorithm [Pease80].

2. Each participant must be connected to at least 2f+1 other participants through disjoint communication paths [Dolev82]. Requirement 1, the cardinality constraint, and Requirement 2, the connectivity constraint, have already been discussed.

3. The algorithm must consist of a minimum of f+1 rounds of communication among the participants [Fischer82]. Requirement 3 is a lower bound on the amount of communication which must transpire for the participants to agree on consistent inputs. It implies that the mechanism used to guarantee consensus must support a lower bound on information traffic, and the only way to make this traffic less painful is to provide an efficient implementation of the mechanism.

4. The participants must be synchronized to within a known skew of each other [Dolev84]. Regarding Requirement 4, in an additional result [Dolev84] demonstrated that for clock synchronization to be possible 3f+1 participants in the synchronization algorithm are required. The connectivity requirements for the problem of clock synchronization using unauthenticated messages are not known. Synchronization is "the hub of the wheel" [Twain85] in Byzantine Resilient computer systems and it is responsible for many of the characteristics of the resulting computers. Numerous papers have been written in this area, but in the opinion of the author, they have not addressed the fundamental problems that synchronization causes in fault tolerant computer systems nor proposed solutions that seem readily extendible to a parallel heterogeneous distributed system.

5. To mask f simultaneous faults by voting of the output of redundant executions of the computation, 2f+1 executions (i.e., 2f+1 processors) are required to guarantee that a majority of non-faulty executions exist. This observation allows certain implementations of the above requirements to utilize less hardware than if 3f+1 processors were used.

A system which meets these prerequisites and is capable of executing the appropriate algorithm is called "f-Byzantine Resilient".

9.4 Existing Approaches to Consistency Maintenance

This section discusses the implementations of consistency maintenance in the SRI SIFT computer and the CSDL AIPS FTP. Ths SRI SIFT is a research flight control multiprocessor developed by SRI and described in [Wensley78]. Intra-FMG communication is via dual-ported memories. Each processor possesses a dual-ported memory exclusively for the reception of messages transmitted by another processor in the FMG. Multiple SIFT systems have not been constructed.

The CSDL AIPS FTP is a two-CPU version of the triplex Fault Tolerant Processor processor developed by CSDL and described in [CSDL84]. Intra-FMG communications are performed using a dedicated "exchange network" which performs Byzantine Resilient data consistency maintenance algorithms in hardware. Members of the FMG communicate with the exchange network through memory-mapped ports. The AIPS FTP is designed to share a redundant multiplexed bus with other simplex, duplex, triplex, and quadruplex groups.

Because of its preeminence as an unresolved issue in Byzantine Resilient parallel computing, existing approaches to maintenance of consistency in time, also known as synchronization, will be discussed first.

9.4.1 Consistency in Time

9.4.1.1 SIFT Synchronization Methodology

Figure 9.10 shows the architectural details of the SIFT which are relevant to the discussion on synchronization. In the SIFT synchronization scheme, each processor of the FMG has a local clock which is used to generate events. A frame is defined by the occurrence of a given number of ticks of this hardware clock. A frame is approximately 100 ms long, composed of 33 subframes each of which lasts 3.2 msec. Each 1.6 msec the local clock interrupts its processor, at which time the local executive resident on each processor decides whether to schedule a new task. One of these tasks is the synchronization task. The synchronization task must be scheduled at the same tick of the hardware clock on each processor. Thus, if this task is scheduled when the clock reads 90 on one processor, it must be scheduled on all other processors when their clocks read 90.

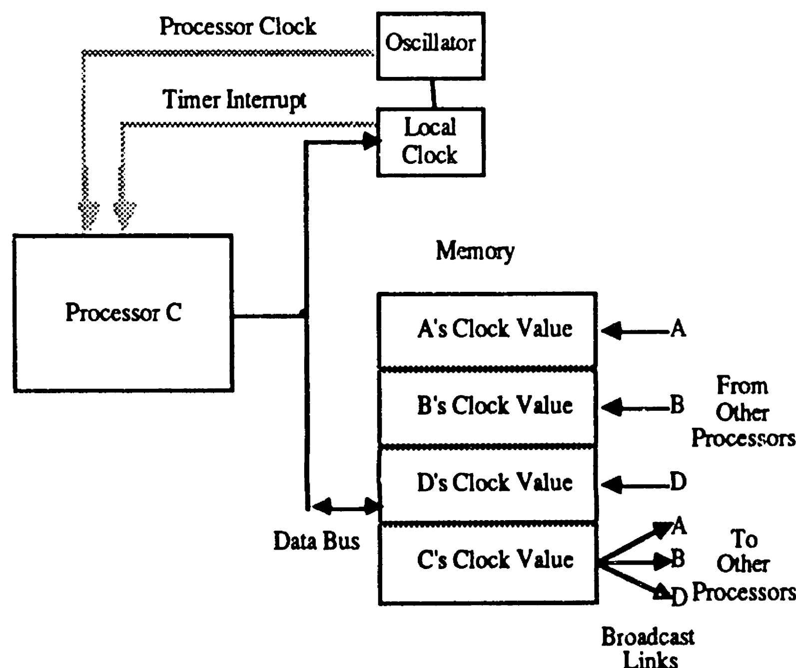


Figure 9.10

Synchronization-Specific SIFT Architectural Details

One may without loss of generality define the end of a frame to be the initiation of this synchronization task and the beginning of a frame to be the termination of the synchronization task. The applications programmer must guarantee that exactly the same number of hardware clock cycles have occurred in each processor during a given frame, usually by ensuring that the worst-case execution time of a task does not exceed one subframe. If each processor's hardware clock reads one common value at the termination of a previous synchronization task, then at the beginning of a subsequent synchronization task invocation each processor's hardware clock reading must also be identical. In the synchronization task each processor reads the value of each other processor's hardware clock, compares it to its own local clock's reading at the time of the observation, and computes a new value for its own clock based on the mean deviation observed between its own clock and those of the other clocks in the group. The synchronization task then updates its processor's local clock with the new value and begins the next subframe at a common agreed-upon value of local clock time. Note that each processor begins the subframe following a synchronization subframe with identical local clock values; the local clocks have merely been set forward or back via the above adjustment algorithm.

The synchronization task begins by running a set of time windows. A window is allocated to a specific processor to broadcast the value of its local clock time. The remaining processors listen for the broadcast from this processor during this window by repeatedly reading the location in its memory corresponding to the sending processor's broadcast port. When a recipient processor detects that the broadcast has occurred, denoted by a change in the value read from that memory location, the recipient processor immediately records this value, reads its own clock value, and calculates the difference between the two. This difference will later be used to correct the recipient processor's local clock value. Following this, the processor marks time until the window for the next processor is ready to begin. This procedure is repeated for each processor in the SIFT. If a faulty processor declines to

send its clock value, the erstwhile recipient processor substitutes a default value for the nonexistent clock value.

At the end of the set of windows, each processor has a value for the deviation of its local clock from the local clock of each other processor at the time that it received the latter. Using this set of values, each processor discards all values which differ from its own by δ or more and calculates the mean of the remaining values. Note that this correction value is obtained via a one-round interactive convergence algorithm.

Each processor now reduces the skew between its local clock and the local clock of the other processors by adjusting the amount of time that the processor waits to begin the next subframe by the mean difference computed above. For example, suppose that at the beginning of the synchronization subframe all processor's clocks read the value 90. Since all processors take exactly the same number of clock cycles to complete the windowing and the adjustment calculation, at the end of the calculation phase each processor will again have identical readings on their hardware clocks, say 95. Now, assume that the subframe following the synchronization task is scheduled to begin at local clock time 100. A processor requiring no clock adjustment would schedule this task 5 ticks after the termination of the synchronization task. If the processor detected that it was fast by a mean of 2 ticks, it would adjust its clock by this value to read 93. Thus it would wait 7 ticks before the value 100 occurred on its local clock and it scheduled the next task. Similarly, a processor that was slow by 2 ticks would set its clock forward by 2 ticks to read 97 at the end of the synchronization task, causing it to reach the scheduling of the next task in 3 ticks.

Note four important points. First, the adjustment in the local clock values has effected a reduction in the differential between the times at which any two processors schedule the task following the synchronization task. Synchronization has been achieved. Second, the first task following the synchronization task, i.e., the first task in the next frame, is started on every processor with an identical reading on its local clock. Thus this first task can with

impunity reset its local clock to some mutually agreed-upon common value, such as zero. Third, no statement has been made about any actions taken by the members of a group between invocations of the synchronization task. All that is required is that all processors arrive at the synchronization point at the same reading of their local clock. This is easily achieved using a hardware timer. Thus, the SIFT synchronization scheme provides framewise synchrony but does not per se provide (or require) functional framewise congruence. Finally, it is clear that no meaningful activity may take place during the adjustment period indicated by the crosshatched area in Figure 9.11 since the duration of this period can change from frame to frame and will not in general be the same on all processors on a given frame.

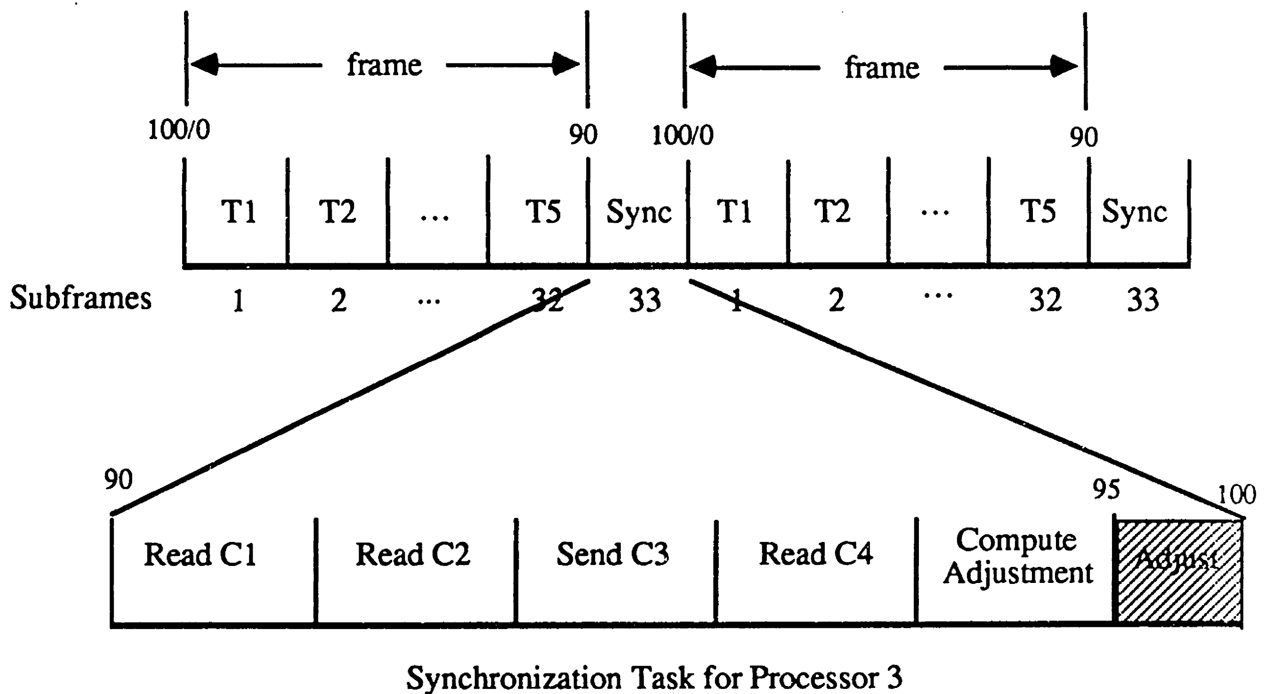


Figure 9.11

SIFT Synchronization Scheme

The software-intensive aspect of this synchronization method is a source of operational

inefficiency, increased post-synchronization skew, and increased granularity of action. First, an entire subframe must be allocated strictly for the synchronization task. Second, because the recipient processor must tight-loop waiting on the reception of the clock update from the other group members, a clock read error equal to the time required to perform one such loop can be induced. Third, the relatively long time between synchronization steps, which is necessary because of the high overhead involved in software execution of the synchronization activity, results in a coarser granularity of action than other methods. These problems spill over into the data consistency maintenance scheme. For example, it is necessary for the programmer to ensure that the FMG members wait until the maximum temporal uncertainty in completion of the consistency protocols expires before any member can access the results of the protocol. This uncertainty is increased by the effects enumerated above.

One benefit of this scheme is that it provides only framewise synchrony; functionally congruent design diversity is possible because of the low degree of interaction between the synchronization mechanism and the function of the redundant state machines. Another is that redundant tasks need not be run in corresponding subframes on the redundant processors. This has been argued to reduce the probability of failure due to correlated transient faults.

9.4.1.2 AIPS FTP Synchronization Methodology

When viewed in terms of the framework outlined above, the hardware-intensive synchronization scheme of the CSDL FTP is strikingly similar to the software-intensive SIFT technique. Unlike SIFT, the clocking mechanism in the FTP may be classified as "extrinsic", i.e., not integral to the members of the processing group. A FTP Fault Tolerant Clock (henceforth known as a FTC) can exist independently of any processors. This distinction is of passing significance in that the discussion to follow will initially focus on the synchronization of the redundant channels of a phase locked hardware clock circuit.

Figure 9.12 shows the architectural details of the FTP which are relevant to the present discussion.

In the FTP, an event is the occurrence of a tick of a hardware clock, just as in the SIFT. A frame is again defined by the occurrence of a prespecified number of these ticks. At roughly the beginning of the frame, a hardware circuit measures the difference between the time of occurrence of the beginning of the local frame and the median time of occurence of the beginning of the frames of the other "channels" of the FTC. The channels indicate to each other that they have reached the beginning of a frame by asserting a "FTC" signal. This standard nomenclature for this signal should not be confused with the appellation for the entire Fault Tolerant Clock (FTC) circuit. Based on the perceived difference between these two values, the hardware circuit inserts an appropriate number of "wait state" cycles of the local clock during the frame, causing the differential in time between the occurrence of the ends of the frames to be reduced.

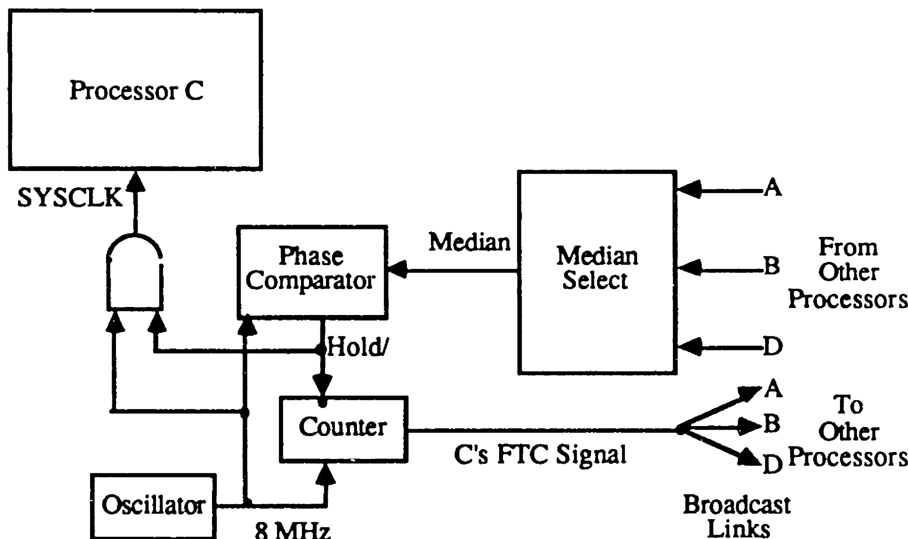


Figure 9.12

Architectural Details of FTP Synchronization Technique

As in the SIFT, the hardware clock used to generate events runs at the frequency of the processor clock (8 MHz in the FTP implementation discussed in [Gauthier86]). The frame

size in the FTP, however, is only 40 clock periods. This works out to a frame length of 5 μsec, so it can be seen that the frame size is considerably smaller than the 100 msec SIFT frame size. Thus the FTP is capable of a much finer granularity of action than the SIFT. This high frequency of synchronization is possible only because the FTP uses a hardware implementation of the synchronization algorithm.

The frames are divided into two phases. In the first phase, which lasts 19 8-MHz clock cycles, each channel asserts FTC. In the second phase, which lasts 21 8-MHz cycles, each channel deasserts FTC. Recipient channels detect that a given channel is entering a new frame when FTC is seen to transition from a deasserted to an asserted level.

Figure 9.13 depicts three possible frames that a channel may execute. At the rising edge of the local signal denoted FTC, circuitry resident in each channel of the clock system detects whether that channel is ahead ("Self Ahead"), in phase ("Self Normal"), or behind ("Self Behind") the "median" rising edge of the other channels, where the median rising edge is defined to be the second-to-occur rising edge received from the other channels of the clock. If the circuitry detects that its channel is in phase ("Self Normal") with the median signal, it holds up the counter which determines the end of the frame by one clock cycle (125 nsec in [Gauthier86]). If the circuitry detects that its channel is behind ("Self Behind") the median signal, it allows the counter to count out the remainder of the frame unimpeded. If the circuitry detects that its channel is ahead ("Self Ahead") of the median signal, it holds up the counter by *two* clock cycles (250 nsec), thus retarding the completion of the frame in that channel. Note that each clock can only adjust itself.

This technique lengthens or shortens the frame by arresting the frame counter by the appropriate number of clock cycles. However, the number of clock pulses in the frame must be kept constant because these pulses are fed to the processor as the SYSCLK (processor clock nomenclature for the 68000 family of microprocessors) pulses. Therefore it is necessary for the hold circuitry to mask from the processor those pulses upon which the counter is delayed. This pulse masking is indicated by the dashed pulses in Figure

9.13.[2] If the redundant processors begin each frame in identical states, if they are provided with one frame's worth of SYSCLK pulses (exactly 40), and if they are clock deterministic, then each processor in the FMG will execute the same set of instructions or parts thereof in that frame. Furthermore, at the beginning of the subsequent frame, each processor will be provided by its channel of the fault tolerant clock with a SYSCLK pulse which differs temporally from the corresponding pulse in the other channels by an amount which has been reduced via the local adjustment of the frame length.

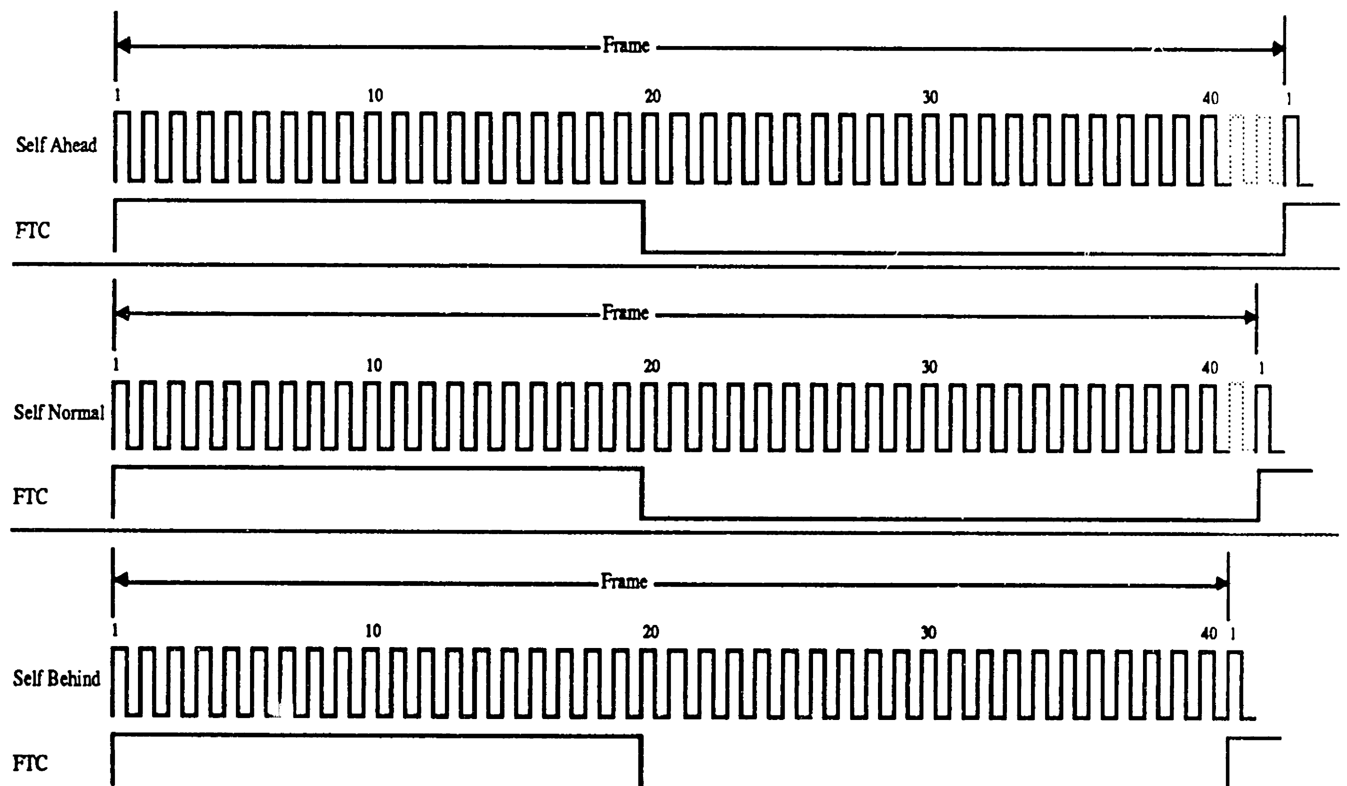


Figure 9.13

Adjustment of Frame Length in FTP Synchronization Scheme

[2]In the actual implementation discussed in [Gauthier86], when the phase-locking state machine removes two pulses from the frame, it removes one pulse at the falling edge of the FTC signal and another at the end of the frame instead of two adjacent pulses at the end of the frame. This is because the 68000 family of microprocessors possesses internal dynamic elements which cannot tolerate the absence of a clocking signal for as long as 250 ns. This detail will be ignored in the above discussion.

The median signal used for comparison of the local clock to the others is obtained via a hardware implementation of an interactive consistency algorithm. Because of this the skew with which the different channels of the fault tolerant clock begin each frame is a small constant which is independent of the inter-channel skew existing before the synchronization adjustment. This is probably of little significance because the high frequency of synchronization in this approach allows very little skew to build up between synchronizations anyhow.

The primary benefits of this synchronization technique are threefold. First, it is totally transparent to the programmer. Software may be written as if for a non-redundant processor, having a significant effect on programmer productivity. Second, the fine granularity of action allowed by the frequent synchronizations allows rapid coordinated response to events. Third, synchronization overhead consumes an average of roughly $1/40^{th}$ of the thoroughput of the FMG, resulting in an efficient implementation.

This technique requires the FMG members to execute exactly the same instructions or parts thereof in a given number of clock cycles, i.e., to be clock deterministic. The implications of this constraint were discussed earlier and were seen to impose a homogeneity constraint. Where there exists a global fault tolerant clocking mechanism, the reliability of the clock generating mechanism upper bounds the system reliability. This may not be a serious problem because of the simplicity of the FTC compared to the other FMG components. Finally, the fine granularity of action is a double-edged sword in that a FMG synchronized according to this technique has difficulty dealing with redundant events whose occurrence possesses a greater skew than the FMG. In effect, the FMG so synchronized imposes its granularity of action upon any devices with which it must interface, requiring the design of "synchronizers" (for an example, see the VFI design in [Lala86b]).

9.4.2 Consistency in Data

9.4.2.1 SIFT Data Consistency Maintenance Methodology

To provide each member of a SIFT FMG with bitwise congruent values of simplex or member-specific inputs the SIFT executes an interactive consistency consensus algorithm as follows. At the time designated in the subframe for output of the value, the datum to be distributed is broadcast to all members of the FMG. Recipients receive the datum in the dual-ported memory segment corresponding to the broadcasting processor. Then each member rebroadcasts the datum that it received to the other members, and the members perform a software vote of the result. This two-round interactive consistency protocol guarantees that each member of the SIFT receives identical copies of the input value.

When all members of a FMG have a value which is the same in all non-faulty members, a simple one-round broadcast of the datum by the source processors followed by a vote is sufficient to guarantee that recipients receive a consistent value.

As mentioned above, this datum is not available to another task running on the SIFT until sometime in the next subframe because of uncertainty in the time at which all processors have completed the second rebroadcast. Thus the applications programmer must ensure that the data buffer corresponding to the consistent value is not read until after this maximum temporal uncertainty has expired.

Investigators have found that software voting consumes a significant amount of the processor's throughput. For example, a five-way vote of one datum consumes about 413 µsec in the absence of vote errors [Palumbo86]. More time is required if the voting process detects errors[3]. This time is increased by the necessity for the voting algorithm to

[3] In early studies on the SIFT failure to allow for the maximum time required for the voting task to execute in the presence of a erroneous datum resulted in an apparent single point of failure when all vote tasks running on non-faulty processors missed their scheduled synchronization points.

associatively search the "Vote Table" for values that needed to be voted in the present subframe, because it is unknown a priori which values need to be voted. Finally, the manual insertion of the requisite wait for the voting process to complete and the need to designate values in the input buffers that need to be voted represents significant cognitive overhead for the applications programmer.

9.4.2.2 AIPS FTP Data Consistency Maintenance Methodology

When a member of an FTP sources an input, it places the value into a memory-mapped register which feeds an "exchange network". Voting logic allows all channels of the exchange network to verify the type and source of exchange desired by the FMG. The exchange network is a hardware implementation of the interactive consistency algorithm discussed earlier. It is clocked with the leading and falling edges of the FTC. Thus, values placed in the network during a frame must be available at most two frames later (one frame to place the value in the network and one frame to perform the exchange, a total of 10 μsec). Processors attempting to read the result of the exchange before the exchange is complete are "held off" by insertion of the appropriate number of wait states by a hardware circuit. The processors are released on an FTC edge following the completion of the exchange.

When all channels of a FTP have a common value to be voted for validation of that value and diagnosis of possible computation faults, the same network is used and it still takes the same amount of time, but the mode of the network is modified such that it votes values emanating from all three channels of the FTP and provides the consistent results, along with the error syndrome, to the recipient channels.

This technique has the dual advantages of conceptual transparency and high efficiency. Data distribution and validation is simply the writing to and reading from a memory-mapped location, with the processor being automatically held up until the exchange is over. The hardware implementation requires on the order of microseconds.

A disadvantage of this approach to data consistency maintenance is that it cannot process data emanating from diverse redundant sites, which may not be bitwise identical in the case of non-faulty operation, but instead may have differences caused by truncation or noncommutativity of arithmetic operations.

9.4.3 Consistency in Ordering

9.4.3.1 SIFT Order Consistency Maintenance Methodology

In SIFT consistency in ordering is performed by the applications programmer who must manually construct the task schedule for the subframes. The programmer knows the order in which tasks are scheduled and how long it takes to perform each data distribution and validation operation. Therefore the programmer knows at what point in time the data are ready. Task scheduling is statically fixed a priori, so the order in which the buffers containing the consistent data are read is fixed and known on all processors, thus guaranteeing consistent input and output ordering. Typically, a static scheduling sequence must be manually constructed for all possible combinations of processor failures. The conceptual overhead involved in this process is clearly monumental for any but the most simple task loads, while the possibility of dynamically varying the task load is out of the question.

9.4.3.2 AIPS FTP Order Consistency Maintenance Methodology

The AIPS FTP is a dual-processor FTP. It consists of two triplex FMGs, a Computational Processor (CP) and an I/O Processor (IOP) (Figure 9.14). These processors share the relatively expensive exchange network, FTC, and other resources in much the same way as the processors subscribing to a cluster share the connectivity and other features provided by the Network Element aggregate. Communication between the CP and IOP FMG can occur through the shared memory, and either FMG may use the

network for distribution of simplex source data, validation of data and detecting computational faults via voting, and writing to a shared memory Both FMGs cannot use the network at the same time, so some scheme must be implemented whereby the "messages" sent by the CP and IOP are consistently ordered as they emerge from the exchange network.

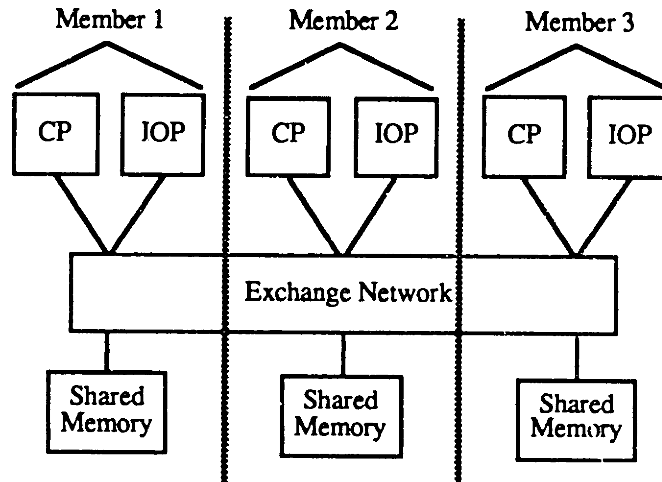


Figure 9.14

Simplified AIPS FTP Architecture

The method chosen for this architecture uses the FTC signal to determine times at which requests for the resource will be honored. A CP or IOP may request the use of the exchange network at any point in the frame. However, the resource is multiplexed such that if the CP and IOP constantly request the network, each receives it on alternate FTC cycles. On the appropriate edge of the FTC, the request pattern is latched and the requesting FMG notified that it has access to the network, which it retains until the data exchange is completed. Thus, access to the exchange network is fairly multiplexed by the FTC and a consistent ordering of messages into and exiting from the network is obtained.

9.5 Byzantine Resilient Virtual Circuit Abstraction

Using the above definitions of data consistency, order consistency, and time consistency one is able to construct a version of the *Virtual Circuit* abstraction [Tanenbaum81] applicable to an ensemble composed of Byzantine Resilient Fault Masking Groups. This abstraction may be expanded to the *Byzantine Resilient Virtual Circuit* (BRVC) abstraction which may be regarded as a communications channel which provides the following guarantees (Figure 9.15):

BRVC1: Messages sent by one FMG to another are delivered uncorrupted. Specifically, messages sent by non-faulty members of a source FMG are correctly delivered to the non-faulty members of the recipient FMG.

BRVC2: Messages sent by one FMG to another are delivered in the order sent. Specifically, non-faulty members of the recipient FMG receive messages in the order sent by the non-faulty members of the source FMG[4].

BRVC3: All non-faulty members of a recipient FMG receive messages in identical order.

BRVC4: The absolute times of arrival of corresponding messages at the members of a recipient FMG differ by a known upper bound.

The BRVC abstraction is proving useful in the development of a programming model for the CSDL FTPP [Troxel87].

9.6 Desired Features of a BRVC Implementation in a Parallel System

The implementation to be described in the subsequent chapter provides the Byzantine Resilient Virtual Circuit abstraction to the ensemble's programmer. A list of desirable features of this BRVC implementation for a parallel processor is given below.

1. The BRVC implementation must be demonstrably resilient to malicious failures of

[4]Under the reasonable assumption that non-faulty members of an FMG send messages in the same order.

the fault sets of the computing mechanism, the communications mechanism, and the clocking mechanism.

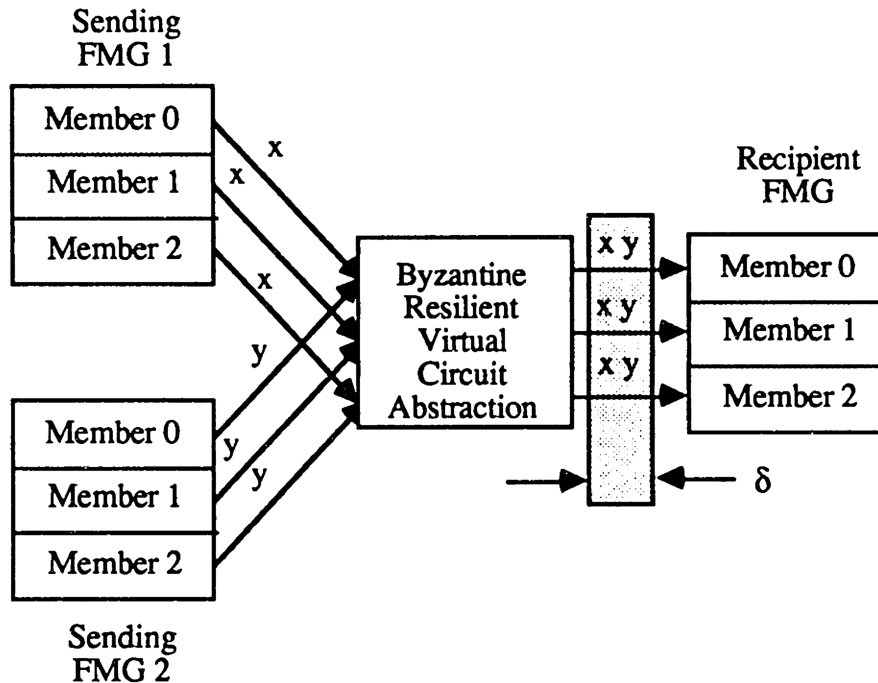


Figure 9.15

Byzantine Resilient Virtual Circuit Abstraction

2. The BRVC implementation must exhibit a high degree of conceptual transparency to the applications programmer.

3. The BRVC implementation should not excessively degrade computational throughput.

4. The BRVC implementation must be capable of being distributed over space to provide fault set boundary maintenence and to be consonant with the precepts of a distributed system.

5. The BRVC abstraction implementation must be shareable by many redundant parallel sites. This is because of the high cost of providing a BRVC abstraction for each FMG. Put

another way, it is desirable that there be a low ratio of fault tolerance-specific hardware to computational hardware. This sharing has the additional goal of allowing dynamic reconfiguration of the members of the FMGs.

6. The BRVC implementation must not overly constrain the functional characteristics of the redundant sites in an area of synchrony. In particular, clock nondeterministic and heterogeneous redundant sites must be integrable within an area of synchrony.

The next chapter will propose an implementation of BRVC which meets these needs.

Chapter 10
Proposed Approach to Consistency Maintenance

10.1 Cluster Architecture and Operation

This chapter proposes an implementation of the Byzantine Resilient Virtual Circuit (BRVC) abstraction which is in accordance with the desirable attributes outlined in Chapter 9 and the earlier observations on the reliability benefits of shared connectivity. In this implementation, Processing Elements of the ensemble subscribe to a Byzantine Resilient core of Network Elements which in turn provides BRVC to its subscribers. A proof-of-concept architecture is being constructed at CSDL according to these principles. This architecture is designated the Fault Tolerant Parallel Processor or FTPP. To demonstrate how the Network Element aggregate provides BRVC to its subscribers, we will first focus on intra-cluster operation.

10.1.1 Organization of the Cluster

For clarity of presentation, a specific configuration of the cluster will be discussed. Consider a 1-Byzantine Resilient cluster having 4 Network Elements and 4 Processing Elements per Network Element. This configuration is 1-Byzantine Resilient with respect to Network Element faults by virtue of the provision of 4 fully interconnected primary fault containment regions. Fault containment region boundaries are assumed to be maintained via strict compliance with the four requirements for fault containment: electrical isolation, physical isolation, independent power, and independent clocking. The Network Elements possess adequate functionality to achieve mutually synchronous operation and perform interactive consistency and voted message exchanges, both on their own behalf and on the behalf of processors subscribing to the Network Element aggregate. Details of this functionality will be presented shortly.

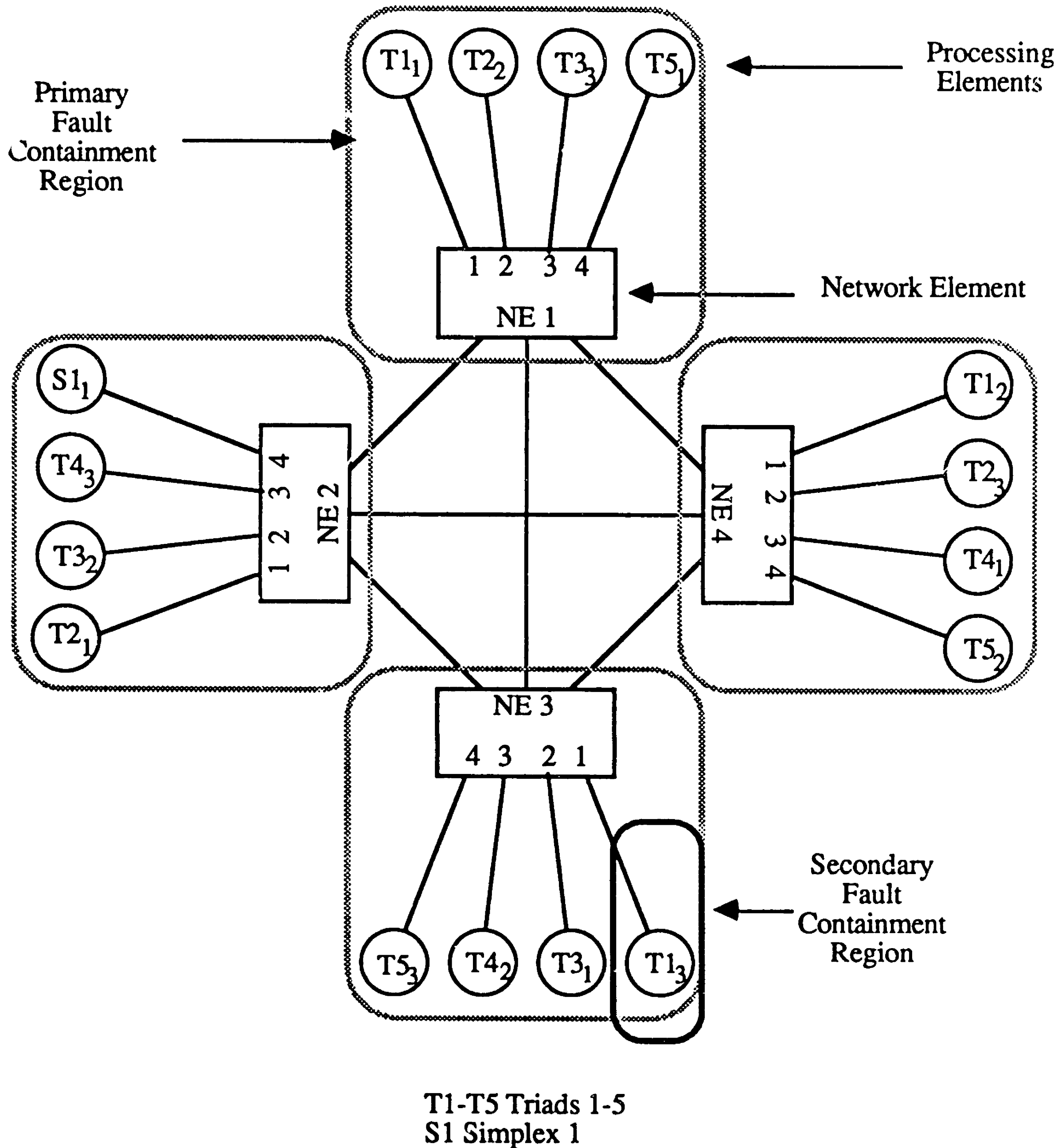


Figure 10.1

FTPP Cluster

This cluster will be assumed to be initially configured into six *virtual groups*. A virtual group is a possibly redundant processing site which is fault masking if it possesses 2f+1 members, fault detecting if it has f+1 members, and simplex otherwise. The programmer

views the virtual group as a single processing site, and need not be aware of the level of redundancy possessed by any virtual group[1]. In the cluster shown in Figure 10.1 are 5 triads and 1 simplex processor. Details of how this configuration is obtained will be presented in section 10.7.1. The 5 triads in this configuration are denoted T1 through T5. Each is a triply redundant FMG. The f-Byzantine Resilience of 2f+1 processing sites is assured because the single port from any member to the remainder of the ensemble makes two-faced behavior on the part of FMG members impossible. A non-faulty Network Element to which such a liar subscribes forwards the input received from the liar faithfully and consistently.

A virtual group possesses n members, where n is the level of processor redundancy of the virtual group. Member numbers range from one to n, inclusive, and are denoted by subscripts in Figure 10.1. Each member of a virtual group must be chosen from a different primary fault containment region. The one simplex virtual group in this example is denoted S1. By convention the member number of the sole member in a simplex group is one.

10.1.2 Definitions

Several terms will be defined before proceeding with the detailed discussion of the operation of the cluster. Define VID (for Virtual ID) to be the ephemeral identifier of a virtual group. For example, T1 is the VID of triad 1. Depending on their redundancy level, virtual groups are made up of one or more members, each of which is designated by a *Member Number*. For example, T4 has three members, $T4_1$, $T4_2$, and $T4_3$, having member numbers 1, 2, and 3. There is no relation between member number and any other number, with the exception that the member number must be unique within a group.

Processing Elements interface to Network Elements through FIFOs (First-In First-Out buffers). Define FID (for FIFO ID) to be an ordered pair (i,j), where the first number of the

[1] Whether this abstraction can be maintained in a heterogeneous mixed redundancy ensemble is an open question.

pair i is the number of the Network Element to which the Processing Element subscribes and the second number of the pair j is the number of the Network Element's port upon which the Processing Element resides. For example, the FID of simplex S1 is (2,4). FIDs are unique ensemble-wide. For multiple clusters, one can redefine FID to be an ordered triple with the first entry being the cluster number. For the time being we will assume that there is only one cluster in the ensemble and use the ordered pair notation for FID.

FID	VID	Member
(1,1)	T1	1
(1,2)	T2	2
(1,3)	T3	3
(1,4)	T5	1
(2,1)	T2	1
(2,2)	T3	2
(2,3)	T4	3
(2,4)	S1	1
(3,1)	T1	3
(3,2)	T3	1
(3,3)	T4	2
(3,4)	T5	3
(4,1)	T1	2
(4,2)	T2	3
(4,3)	T4	1
(4,4)	T5	2

Figure 10.2

Configuration Table

Reconfiguration is effected by changing the mapping from FIDs to VIDs. For example, if the member of T4 corresponding to FID (2,3) were to fail a possible reconfiguration option is to change the VID of S1 to T4 to restore T4's redundancy. Alternatively the remaining members of T4 could be disbanded to form simplexes S2 and S3. A data structure called the *Configuration Table* (CT) resident in the Processing Elements and in the

Network Elements contains this mapping information. Reconfigurations such as the one described above are effected by a change in the CT. A later section will show how to update the CTs such that reconfigurations occur in an orderly manner. Figure 10.2 shows the Configuration Table for the cluster in Figure 10.1.

10.1.3 Fault Containment Region Architecture

A Processing Element sends messages into the Network Element aggregate and receives messages from the Network Element aggregate through a pair of FIFOs which buffer the loosely synchronous world of the Processing Elements from the tightly synchronous world of the Network Elements[2]. Each Processing Element possesses a memory-mapped interface to its own dedicated FIFO pair. In addition to the memory-mapped "Out FIFO" and "In FIFO", a Processing Element possesses another smaller outgoing "Control FIFO" into which the Processing Element writes control information such as the exchange class and the source member of the exchange, if pertinent. A small array of control/status registers (CSRs) indicates to the Processing Element the status of the Network Element, such as whether the Out FIFO is full, the In FIFO has a message available, and other information. The message format consists of the Source VID (including the member number if the message is a Class 2 exchange), the Destination VID, and the contents of the message to be sent. Figure 10.3 depicts the architecture of a primary fault containment region, showing four subscriber Processing Elements, their FIFOs, and other components.

---

[2] In the proof-of-concept FTPP under development at CSDL, messages are composed of an arbitrary number of "packets", which are 32 bytes in length. In the subsequent discussion the term message and packet will be synonymous.

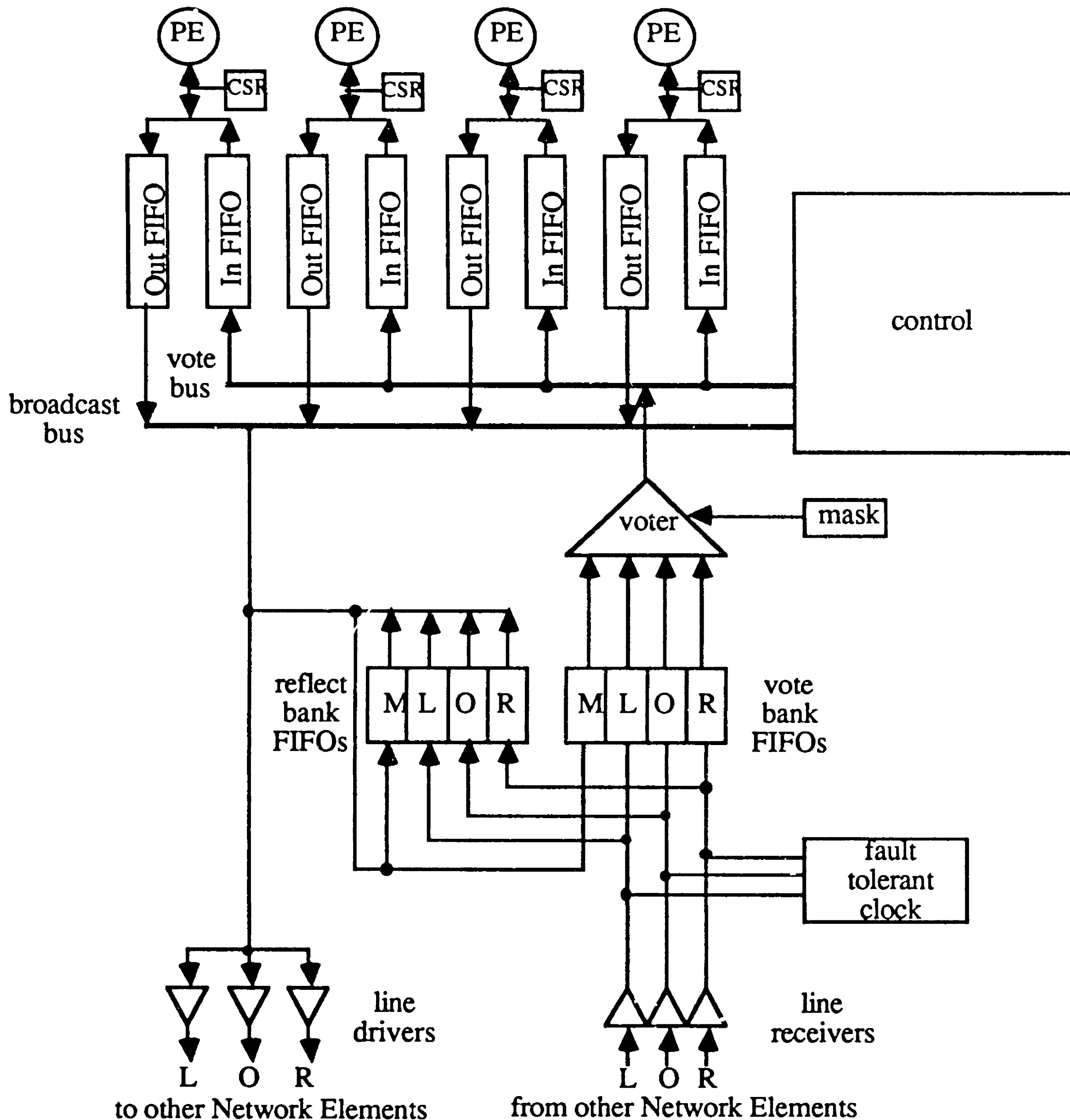


Figure 10.3

Fault Containment Region Architecture

In addition to the In and Out FIFOs there are two smaller sets of FIFOs called the Reflect Bank and the Vote Bank. In many cases the Network Element aggregate must execute a two-round interactive consistency protocol; the first round consists of a broadcast

of the message from the sourcing Network Element to all other Network Elements, and the second round consists of another broadcast of the message by all Network Elements to each other, followed by a vote. The Reflect FIFOs are the destinations of the messages transmitted in the first round of such protocols and the source of the messages broadcast in the second round. In the second round, the messages are routed into the Vote bank FIFOs, from which they are fed into the voter at the end of the communal broadcast. In an extension of the traditional method of labelling members of an FTP, these two FIFO banks are labeled according to the relative position of the neighboring Network Elements to which they are connected. The FIFOs labelled "L" are connected to a Network Element's left neighbor, those labelled "R" to the right neighbor, those labelled "O" to the opposite neighbor, and "M" connected to "myself". FIFO banks are labelled identically in all Network Elements, such that one Network Element's left neighbor will be some other's right.

To send a message, the Processing Element executes the following sequence of actions.

1. The Processing Element first examines the control/status register to determine whether the Out FIFO can accept a packet.

2. If the Out FIFO is full, the Processing Element must wait until there is enough room to hold the outgoing packet. If the Out FIFO is not full, then the Processing Element writes the packet to the Out FIFO.

3. After writing the message the Processing Element writes the exchange class and source member number to the Control FIFO. This act notifies the Network Element that a message transmission has been requested by the Processing Element corresponding to the FID of the Control FIFO. From the point of view of the Processing Element, the message has now been sent into the Network Element core for delivery and the Processing Element can carry on with other actions.

To receive a message, the Processing Element executes the following sequence of actions.

1. The Processing Element determines if a packet may be read from the In FIFO by reading the control/status register. The Network Element maintains a counter which is incremented by the arrival of a byte of data from the voter[3]. When this counter is equal to the number of bytes in a packet, a signal is asserted in the control/status register which indicates that a packet is available[4].

2. If the control/status register indicates that a packet is available in the In FIFO, the Processing Element reads the packet from the In FIFO. The counter in the Network Element will be automatically decremented for every byte read by the Processing Element.

Any ordering or interleaving of sending and receiving messages of parts thereof is allowable. A Processing Element may send part of a packet, then attempt to read a packet, then continue sending the original packet. This feature comes in very handy in solving a deadlock problem unearthed by G. Nagle, which is as follows. The Network Element core implements a form of "flow control" to prevent messages destined for FIFOs which are legitimately full (e.g., corresponding to an FMG which is executing a computationally intensive task and therefore has not emptied its In FIFO for some time) from being lost. Failure to provide such flow control would invalidate the guaranteed inter-FMG message delivery abstraction since messages sent by a non-faulty FMG to a non-faulty FMG would be lost. Suppose an FMG has a full In FIFO and it wishes to send a message to itself. The flow control mechanism of the Network Element core will retard the transmission of the message until the destination In FIFOs of the recipient FMG are capable of accommodating a packet. If the destination FMG cannot unload its In FIFO before sending the message (or after finding out that the message cannot be sent because of a flow control problem), the message will be delayed indefinitely by the flow control mechanism. The capability to

---

[3]In the actual FTPP implementation the counter is incremented on reception of every four bytes, or, equivalently, every longword.

[4]More or less. In the FTPP, the Processing Element reads a longword counter which indicates the number of longwords in the In FIFO. The Processing Element may optionally read the In FIFO regardless of the existence of a full packet in the FIFO.

interleave the reading and transmission of partial messages allows the FMG to empty its In FIFO at a high rate regardless of the task's schedule of sending and reading messages, and thereby avoid this deadlock situation. In the Proof-of-Concept FTPP, real-time clock interrupts cause each FMG member to empty the In FIFO at a high rate. These interrupts are "functionally invisible", since they are not allowed to affect the decisions made by any process until a scoop[5] is performed. Their only effect is to decrease the apparent execution rate of the members of the FMG.

No sequence numbers or timestamps are required on the messages. The requirement that FMG members send messages in the identical order and the guarantee that the Network Element core delivers messages in the order sent is adequate for the construction of all distributed protocols that we have needed to date (bootstrapping, FMG formation, reconfiguration, FDI, load balancing, deadlock recovery, and inter/intra-FMG communication).

10.2 Functional Synchronization

This section uses the synchronization taxonomy developed earlier to introduce *Functional Synchronization*, a proposed method of obtaining synchronization of processors composing a virtual group.

In the functional synchronization scheme, events are equated to the occurrence of unambiguously defined actions performed by the members of a virtual group during the course of their normal task execution. For example, sending a message could be defined to constitute an event, as could reading an incoming message buffer or rescheduling a process[6]. A frame is defined as usual by the occurrence of a specified number of these events. When the requisite number of events has transpired, the FMG executes a

[5] To be defined later.

[6] It is necessary to be careful here in the case of nondeterministic hardware. See [Friend86].

synchronizing act, which defines the end of the old frame and the beginning of the new. The purpose of functional synchronization is simply to reduce the skew existing between FMGs after a synchronizing act to a much smaller value than that existing before the synchronizing act.

As an example of this idea, Figure 10.4 shows the use of reading a message from an input buffer to define events, with one event constituting each frame. However, in functional synchronization, the number of events per frame need not be constant, and can be a function of the length of the current frame (i.e., the time since the last synchronizing act) or any other operational parameter. In addition, the frames of different FMGs have no temporal relationship to each other. They may overlap, be of different lengths and periodicities, and exhibit a high degree of disparity.

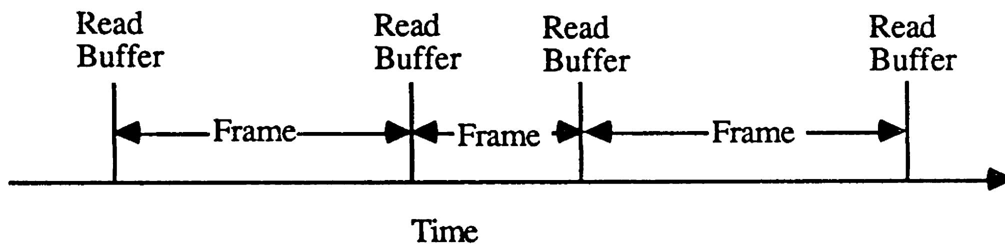


Figure 10.4

Frame Definition by Reading Input Buffer

Three requirements are placed upon members of an FMG participating in functional synchronization. First, the members of an FMG must exhibit a differential execution rate which can be upper-bounded. A certain degree of nondeterminism and heterogeneity is allowed among the members of an FMG, but the maximum difference between the execution rates of the fastest and slowest members of an FMG must be known. Second, the length of time between synchronizing acts must also be upper-bounded. An FMG cannot continue forever without performing a synchronizing act. The assertion that an FMG meets the first two requirements implies that it is possible to upper-bound the time

differential with which different FMG members arrive at a synchronizing act. Denote this maximum differential or "skew" by σ. Members that do not arrive at a synchronizing act within σ time units of a non-faulty member can be assumed to be faulty. The third requirement is that the members of the FMG must perform corresponding synchronizing acts in an identical order. An additional requirement, not necessary to obtain functional synchronization but necessary to obtain high coverage fault detection and masking via voting, is that a specifiable subset of messages emanating from the members of an FMG must have the characteristic that bitwise agreement on message content implies non-faulty behavior, and bitwise disagreement implies otherwise.

Assume that the members of an FMG satisfy the conditions enumerated above. Each member is further assumed to be executing a functionally congruent instantiation of a process which can be unambiguously partitioned into segments. The partitioning may be either manual (the programmer uses a "perform synchronizing act" primitive) or automatic (the compiler or operating system inserts synchronizing acts upon message transmissions, Remote Procedure Call invocations, reading of an input buffer, etc.). Let traversal of a segment boundary constitute an event. Upon the occurrence of such an event, the copy of the operating system resident on each member of the FMG decides whether the occurrence of this event is to constitute a frame boundary. As mentioned above, this decision may be a function of the time since the last synchronizing act, the mission phase, or other operational parameters. The only critical requirement is that all copies of the operating system come to an identical conclusion about the status of the event. If the event is determined not to trigger synchronization, the operating system returns control to the process, or, alternatively, to another process awaiting scheduling. Again, all that matters is that these decisions are the same in all members of the FMG. See [Friend86] for a comprehensive discussion of these options.

If the operating system instantiations determine that the event constitutes a synchronizing act, they trigger synchronization of the FMG hosting the segment

instantiations by transmitting a message to themselves into the Network Element core and subsequently awaiting the reception of that message. It will be shown that the difference in time between the delivery of copies of a given message at all non-faulty Network Elements is very small. Therefore the time differential with which the copies of the operating system perceive the arrival of the message of interest is very small. Assuming that the operating systems are awaiting this message, they will be synchronized upon its reception. In addition, if the operating system instantiations immediately return control to the process instantiations which triggered the synchronizing act, then the process instantiations are also synchronized.

The Network Element core is a tightly synchronous, clock deterministic aggregate which executes Byzantine Resilient clocking and consensus algorithms in dedicated hardware and firmware. A subsequent section will discuss how the Network Element core obtains tight synchrony. Regarding the present discussion, the Network Element core achieves near-simultaneous delivery of the identical copies of a given message by delaying the transmission and delivery of the message until the *valid message condition* is met by the members of the FMG. Satisfaction of the valid message condition depends on the redundancy level of the virtual group which sourced the message. For a Fault Masking Group, the valid message condition is fulfilled when all members of a group have requested transmission of a message, or a majority of the members of a group have requested a message transmission and the maximum allowable skew between the transmission of messages by non-faulty members, σ, has expired. For a duplex Fault Detecting Group, the valid message condition is met when either both members of the duplex have requested a message transmission or one member has requested transmission and a suitable timeout has expired. Finally, for a simplex group, a message is transmitted as soon as the sole member of the group makes the request.

The valid message condition is synchronously evaluated for all virtual groups by all of the Network Elements in a cluster, using message transmission request patterns that are

obtained via a Byzantine Resilient interactive consistency protocol and which are therefore identical at all non-faulty Network Elements. Therefore all Network Elements come to identical decisions about which groups pass the valid message condition. Also by virtue of the tight synchrony of the Network Element core, the message transmission and delivery which may[7] denote the completion of an FMG's synchronizing act is performed at very close to the same time in all Network Elements. Thus, FMG members awaiting the completion of this transmission perceive the completion at very close to the same point in time, and continue with their execution with a temporal skew which has been reduced by the act of waiting for the completion of the synchronizing act. Figure 10.5 illustrates the fundamental notions of functional synchronization.

A wide class of synchronizing acts may be defined. As a first example of a synchronizing act, consider the act of the FMG waiting for the arrival of the next message to be delivered. Assume that all members have processed the same set of n messages such that they are all waiting for the arrival of the $n+1^{st}$ message. By virtue of Guarantee 4 of BRVC, members receive their copy of this message within a bounded skew of each other. Therefore waiting for the arrival of the next message synchronizes the members. Alternatively, waiting for the arrival of a *specific* message, such as a control or command message, also constitutes a synchronizing act.

Using the BRVC abstraction and the idea of functional synchronization, an FMG can perform a useful act called a *scoop*. An FMG may be triggered to perform a scoop by a resident process' request to update its input message buffer. It is critical that the message buffers of all FMG members possess identical contents to prevent divergence of any computation resulting from decisions based on those contents. Scooping uses the BRVC abstraction and functional synchronization to obtain a consistently ordered and identical set of messages at each FMG member.

[7]The Network Element core has no knowledge of the purpose of any message, synchronizing or otherwise. All are treated according to the valid message condition.

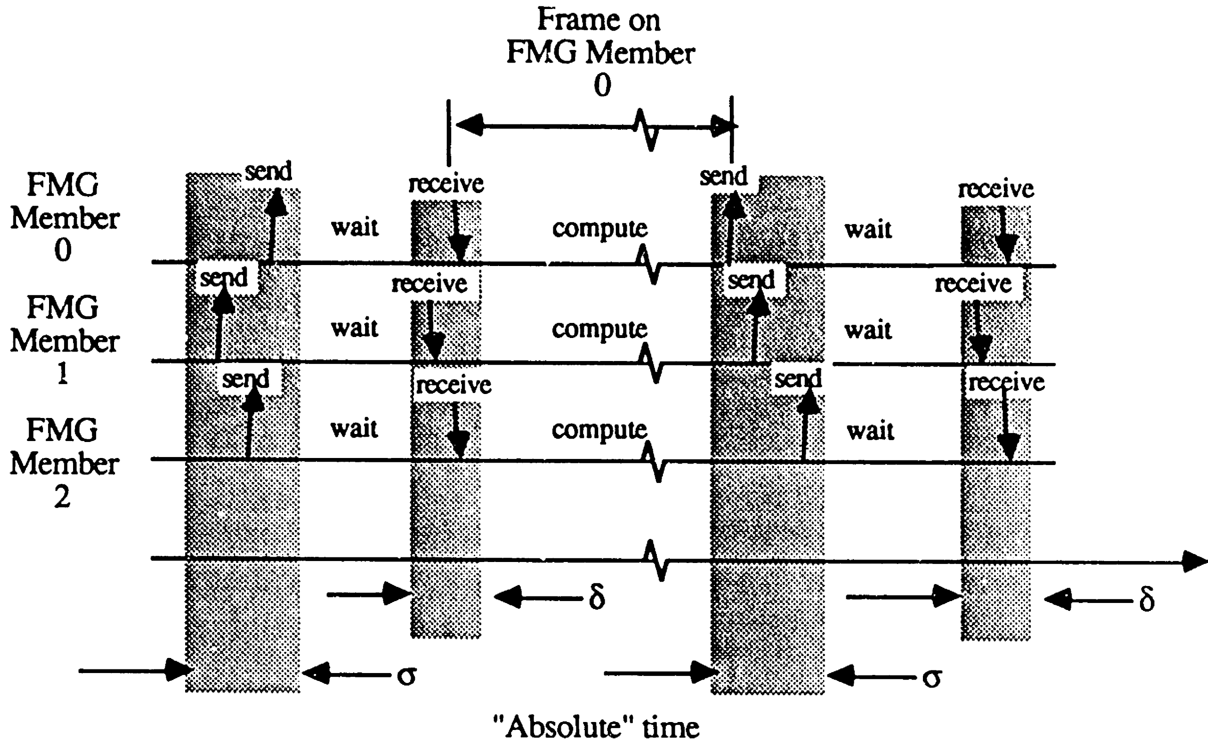


Figure 10.5

Functional Synchronization

An FMG performing a scoop takes advantage of BRVC by simply sending a message to itself and awaiting its reception. By guarantee 3, all members receive identically ordered identical copies of each message. By guarantee 4, the absolute times of arrival of each message, and in particular the scoop message, at the different FMG members differ by a known upper bound. Using these guarantees the recipient FMG is assured that each member of the FMG has received an identical set of identically ordered messages before delivery of its own scoop message. Therefore any decision made based on messages received prior to a scoop reception will be congruent in all members. Upon delivery of the scoop shown in Figure 10.6, each FMG possesses identical input message buffers {A, B, C}. In addition, the FMG has executed a synchronizing act. In an actual implementation,

scoops would probably be imbedded as an option in a "read input message buffer" primitive.

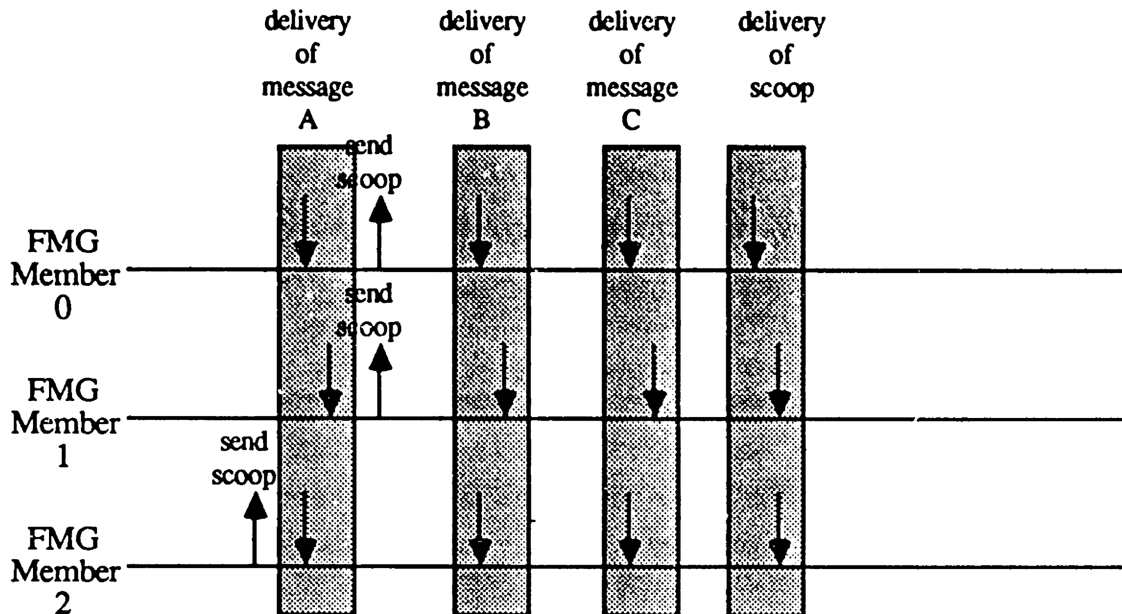


Figure 10.6

Scooping

Functional synchronization sidesteps the homogeneity constraints inherent in the FTP scheme described previously. FMG members need only obey the constraints listed above. The first benefit of this constraint alleviation is that the components constituting an FMG need not be minutely inspected for any nondeterministic behavior which might throw members of an FMG out of synchronization. It is unnecessary to custom-fabricate the many components composing the ensemble to ensure that they meet the clock-determinism constraint, resulting in a significant reduction in the difficulty and cost of fabricating an ensemble which utilizes functional synchronization[8]. In addition, the ensemble's computational components may be chosen according to desired features instead of their capability to meet the nondeterminism constraint. The second benefit arising from removal

[8]Our experience in the CSDL FTPP project thus far supports this point.

of the monogeneity constraint is that it is possible to implement hardware and software design diversity in an integrated and uniform fashion in an attempt to overcome generic faults.

Functional synchronization allows synchronization according to the degree of intra-FMG interaction. Synchronization occurs only when necessary, such as when inter-FMG interactions occur naturally in the course of ensemble operation or when an FMG desires to exchange internal information. The fact that the synchronization/communications mechanism is used only when needed increases the likelihood that it may be shared by multiple subscribers and hence obtain efficient utilization of the core of connectivity, cardinality, and functionality. For example, in a functionally heterogeneous ensemble where some FMGs execute computationally intensive tasks while others execute I/O intensive tasks, the former do not execute any overhead-consuming synchronization activity until they are finished with their work, while the latter who require more of the synchronization/communications mechanism are able to obtain it with greater ease because the former consume little bandwidth. This utilization of the synchronization mechanism only upon need increases the number of FMGs which can share the synchronization mechanism, resulting in a reduction in the ratio of the amount of fault tolerance-specific hardware to computational hardware. A subsequent section will compute the degree to which this sharing can be supported without throughput degradation via excessive bandwidth consumption.

The use of the same hardware for intra and inter-FMG transmissions, in addition to representing efficient sharing of that hardware for multiple purposes, imposes uniformity of treatment upon the messages transmitted throughout the ensemble; intra-FMG exchanges are treated the same way as inter-FMG exchanges. This feature is consonant with the principle of the "law of least astonishment" in the language design community, which has been summarized as "If something is done in one way in one place, it ought to be done the same way everywhere" [Wulf81].

No *a priori* relationship is required between the frames of different FMGs. This allows a total decoupling of the operational characteristics of different FMGs, while nevertheless allowing them to be integrated into a single physical and logical entity in a conceptually coherent manner.

Functional synchronization is conceptually transparent to the programmer, who has the option of never knowing that certain acts like reading input message buffers are events or synchronizing acts. In a distributed algorithm, the sending and receiving of messages is likely to be frequent enough to obtain ample opportunity for synchronizing acts. This is of course dependent on the application. Recall that in SIFT "synchronizing acts" were scheduled every 100 msec, while in the simple parallelization of the A* search an iteration time of 1 sec/iteration was noted.

Among the disadvantages of functional synchronization, the inter-synchronization interval is not known or even constant in general. Therefore specification of the timeout σ is impossible without reference to specific software running on specific hardware. Note however that the structured nature of specific applications code may make it possible to make such guarantees in particular instances. Again, refer to the SIFT and the A* examples. Moreover, validation of the resulting system is exacerbated because it is difficult to "factor" the validation of the synchronization mechanism from the applications code.

10.3 Exchange Classes

Functional synchronization requires that FMGs send and receive messages to be synchronized. To elaborate on the types of messages that an FMG may send and receive, two[9] *Classes* of message exchanges can be defined based on the level of redundancy of the message source. In the first, a message source is sufficiently redundant (i.e., emanating from an FMG) that all is required is the transmission of all copies of the message through disjoint communications paths to its destination, followed by a vote of the received copies by the recipient. This guarantees that in the face of up to f faults, f+1 correct messages will get through and a correct voted copy will be obtained at all recipients. In the second exchange class, the message source is not sufficiently redundant to allow voting to guarantee that all recipients receive correct or even identical copies. An example of such a message is member-specific syndrome information possessed by each FMG member. In this case, an f+1 round Byzantine Resilient interactive consistency exchange must be performed to guarantee the validity and agreement conditions in the presence of up to f faults.

10.3.1 Class 1 Exchange

A *Class 1 exchange* is used to transmit a message which is the same in all non-faulty members of the source FMG. Only FMGs can source Class 1 exchanges. Such an exchange merely needs to be voted prior to delivery to the recipient FMG, which may be the same FMG as the source, another FMG, any another virtual group, or all virtual groups. Since there are 2f+1 information sources, a majority vote ensures that all destinations receive identical results in the presence of any combination of up to f Processing Element and Network Element faults.

[9] In the FTPP implementation there are actually sixteen exchange classes. They are all basically variations on the two described above. Space limitations do not permit discussion of all sixteen.

A Class 1 exchange is a one round exchange requiring one message to be sent by all Network Elements on behalf of the FMG requesting the exchange. As an example of such an exchange, consider the case in which T1 desires to send a Class 1 message "x" to T2. Phase one of this exchange occurs when the members of the FMG write the message to the Network Element and request transmission (Figure 10.7). All non-faulty members of the source FMG perform this act within σ time units of each other.

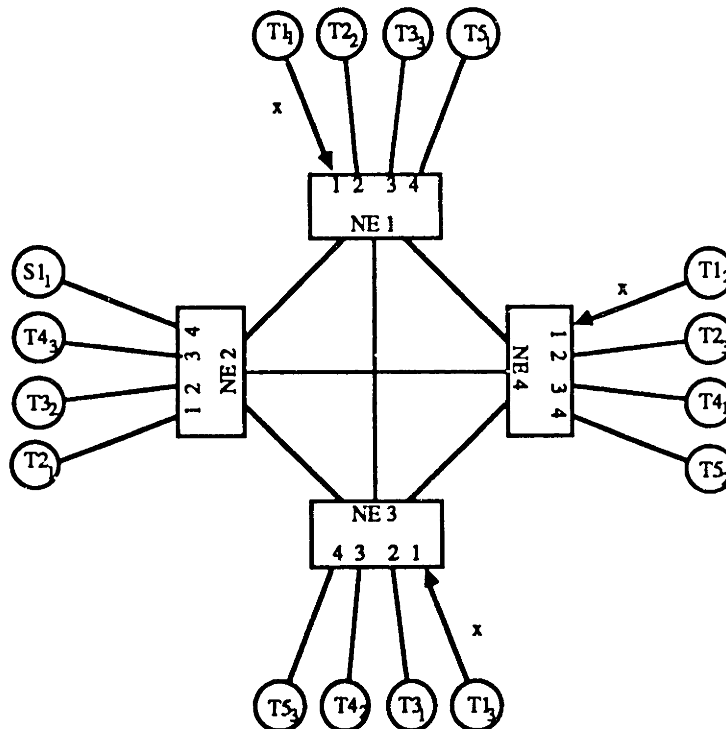


Figure 10.7

Phase 1 of a Class 1 Exchange

As an aid in envisioning the operation of the Network Element aggregate, a notation will be adopted which shows the state of all FIFOs on all Network Elements. Examination of this diagram helps understand the movement of information in the aggregate. There are four sets of FIFOs in a Network Element: the Out FIFOs, the In FIFOs, the Vote Bank FIFOs, and the Reflect Bank FIFOs. Let the contents of a FIFO be denoted by the set

{a,b,...,z}, where a, b,...,z are messages. Messages may be considered to enter the FIFO from the right-hand end of the braces and be shifted out of the FIFO from the left-hand end. After phase one of the class 1 exchange from T1 to T2, the FIFO array appears as shown in Figure 10.8.

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{x}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE2:	**port 1**	**port 2**	**port 3**	**port 4**
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE3:	**port 1**	**port 2**	**port 3**	**port 4**
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{x}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE4:	**port 1**	**port 2**	**port 3**	**port 4**
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{x}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

Figure 10.8

FIFO Map Following Phase 1 of Class 1 Exchange

Assuming that the valid message condition has been met by T1 and that all members of T1 agree that a Class 1 exchange to T2 is desired, phase two of the exchange is executed in which Network Elements 1, 4, and 3 broadcast the three copies of the message out of the outgoing FIFOs corresponding to T1 members 1, 2, and 3. Network Element 2 broadcasts

an "empty" packet. This phase is shown in Figure 10.9.

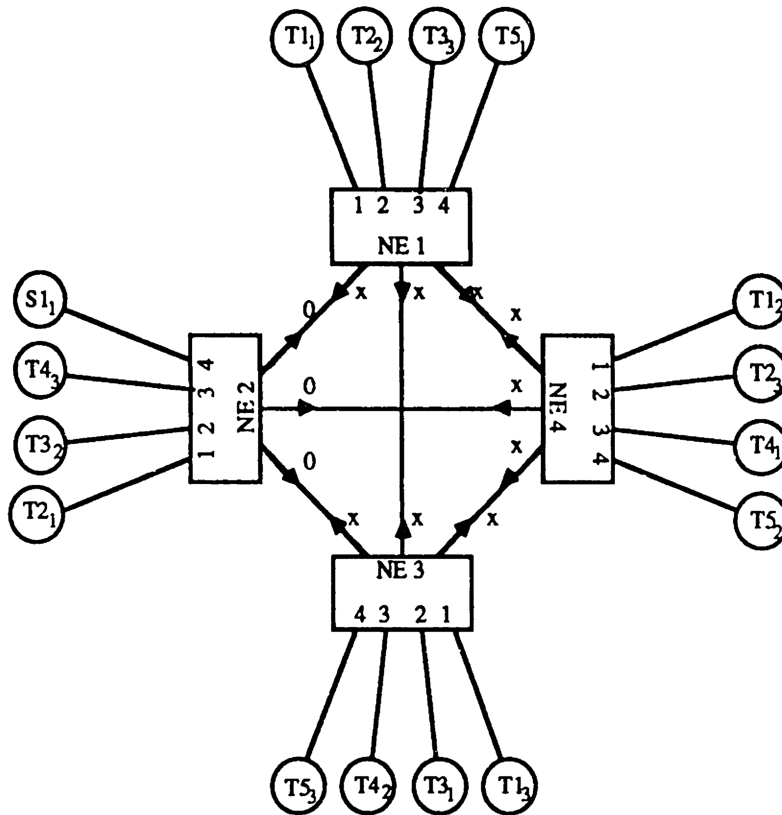


Figure 10.9

Phase 2 of a Class 1 Exchange

The FIFO map resulting after phase 2 is shown in Figure 10.10.

The mutually synchronous broadcast is followed by phase 3 in which the packet is voted by all Network Elements, using an appropriate vote mask to prevent the voting of the empty packet emanating from Network Element 2. This phase is shown in Figure 10.11.

The voted copy of the message is shifted into the FIFOs of the destination group T2 during the delivery phase of the exchange. A given Network Element determines if a member of the recipient group is resident on it by an inspection of its local Configuration Table. In this example, FIFOs (2,1), (1,2), and (4,2) would receive x_{voted}. T2 may read this message

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{x}	{0}	{x}	{x}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE2:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{x}	{x}	{x}	{0}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{0}	{x}	{x}	{x}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{x}	{x}	{0}	{x}
Reflect Bank FIFOs:	{}	{}	{}	{}

Figure 10.10

FIFO Map Following Phase 2 of Class 1 Exchange

according to its own message transmission and reading schedule. It is likely that the message will get scooped in when T2 updates its message buffer. If a member of the destination group does not reside on a particular Network Element, the voted message is discarded. This situation obtains on Network Element 3, which hosts no member of T2.

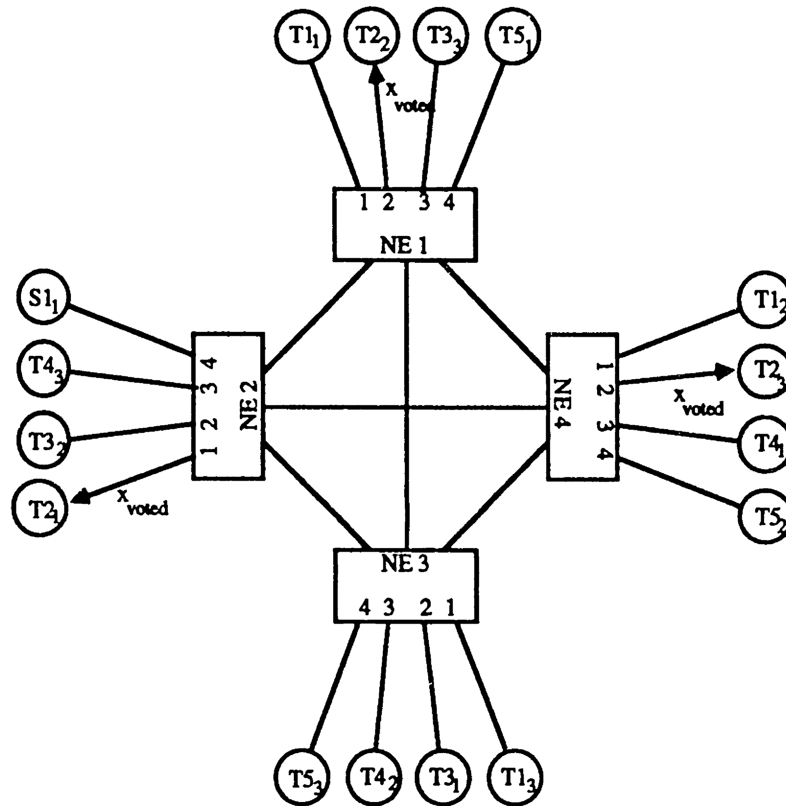


Figure 10.11

Phase 3 of a Class 1 Exchange

The resultant FIFO map is shown in Figure 10.12.

Fault masking is performed by the voting circuit according to a *mask* which indicates to the Network Element which Vote FIFOs to enable. The mask in use depends on which Network Elements are sending votable messages and which Network Element is doing the voting, for, as inspection of the FIFO Map in Figure 10.12 shows, different Vote FIFOs contain different messages, depending on a Network Element's relationship to the message sources. Network Elements determine the appropriate vote mask by the use of the Configuration Table, which has an internally stored vote mask for each virtual group.

The vote *syndrome* or error information is appended to the packet and delivered to the destination FIFO(s) after the packet. Network Elements take no action upon detection of a vote disagreement; any action is up to the group(s) which receive the message. Syndromes

are not necessarily identical at all voting sites. Inter-Network Element link faults or malicious Network Element faults may cause one Network Element to perceive faults while the others perceive none. Therefore syndrome information must be transmitted by the Processing Elements using Class 2 exchanges, which will be discussed shortly.

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	$\{x_{voted}\}$	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE2:	port 1	port 2	port 3	port 4
In FIFOs:	$\{x_{voted}\}$	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	$\{x_{voted}\}$	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

Figure 10.12

FIFO Map Following Phase 3 of Class 1 Exchange

Vote disagreements may be the result of Network Element faults or Processing Element faults. In this instance, the failure modes overlap. However, either failure mode is completely masked assuming the cluster's redundancy has not been saturated. Further failure diagnosis can differentiate between the two failure modes when, for example, another FMG having a member resident on the suspected Network Element performs a

Class 1 transmission. In addition, other information such as Network Element *presence* syndrome, which indicates whether a given Network Element participated in the last frame of the Network Element synchronization cycle, provides additional evidence to help differentiate between Network Element and Processing Element faults. Network Element presence syndrome as well as transmission link parity are appended to the packet prior to forwarding to the destination FIFO(s).

10.3.2 Class 2 Exchange

A *Class 2 exchange* is performed when a message emanates from a single source. This source can be either a single member of an FMG, in which case all members of an FMG must agree on the fact that it is to be a Class 2 exchange and on which member is to be the source, or a simplex group such as S1. Consider the case in which S1 transmits a message "x" to triad T2, shown in Figure 10.13. Since this message emanates from a single source, it must pass through a two-round interactive consistency protocol to ensure that all non-faulty recipients residing on non-faulty Network Elements receive identical copies. The exchange begins with phase one when S1 writes its message into its outgoing FIFO and requests transmission, after which the FIFO map appears as shown in Figure 10.14.

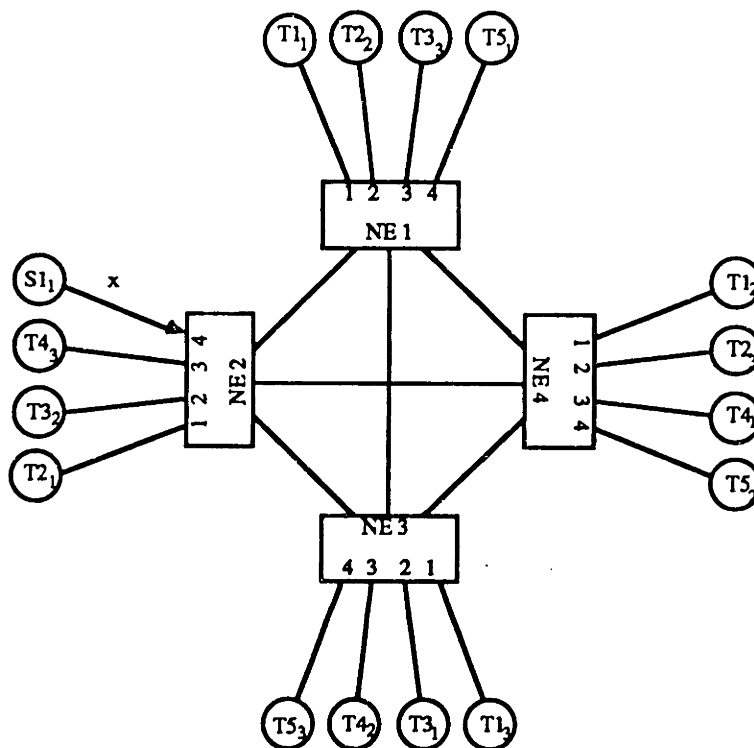


Figure 10.13

Phase 1 of Class 2 Exchange

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE2:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{x}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}
NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

Figure 10.14

FIFO Map Following Phase 1 of Class 2 Exchange

The Network Element core recognizes the request and, after performing the trivial test for fulfillment of the valid message condition for a simplex source, broadcasts the message in phase 2. In this case, Network Element 2, which hosts S1, broadcasts "x" and the other Network Elements broadcast "empty" packets. These Network Element actions are mutually synchronous.

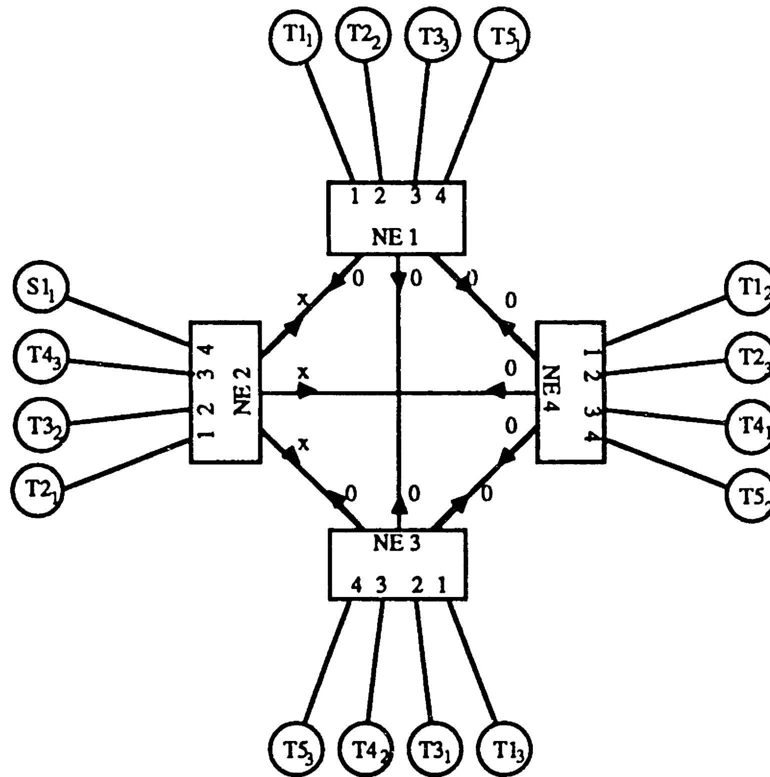


Figure 10.15

Phase 2 of Class 2 Exchange

After completion of the initial broadcast of the message "x", the FIFO map is as shown in Figure 10.16. Note that the notation has been changed slightly from that of the Class 1 exchange. In this example, x_i indicates that value that Network Element "i" received from the simplex source on the first round of the exchange.

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{0}	$\{x_1\}$	{0}	{0}
NE2:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{0}	{0}	{0}	$\{x_2\}$
NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	$\{x_3\}$	{0}	{0}	{0}
NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{0}	{0}	$\{x_4\}$	{0}

Figure 10.16

FIFO Map Following Phase 2 of Class 2 Exchange

In phase three, the Network Elements synchronously perform a "reflect" of the value they received in phase two of the exchange. This third phase is depicted in Figure 10.17. Upon completion of the reflect phase, each Network Element has four copies of the message "x" in their Vote FIFOs (Figure 10.18). Note that the vote syndrome generated in this case indicates the health of the Network Elements, since a vote has no meaning for a simplex or member-specific message source.

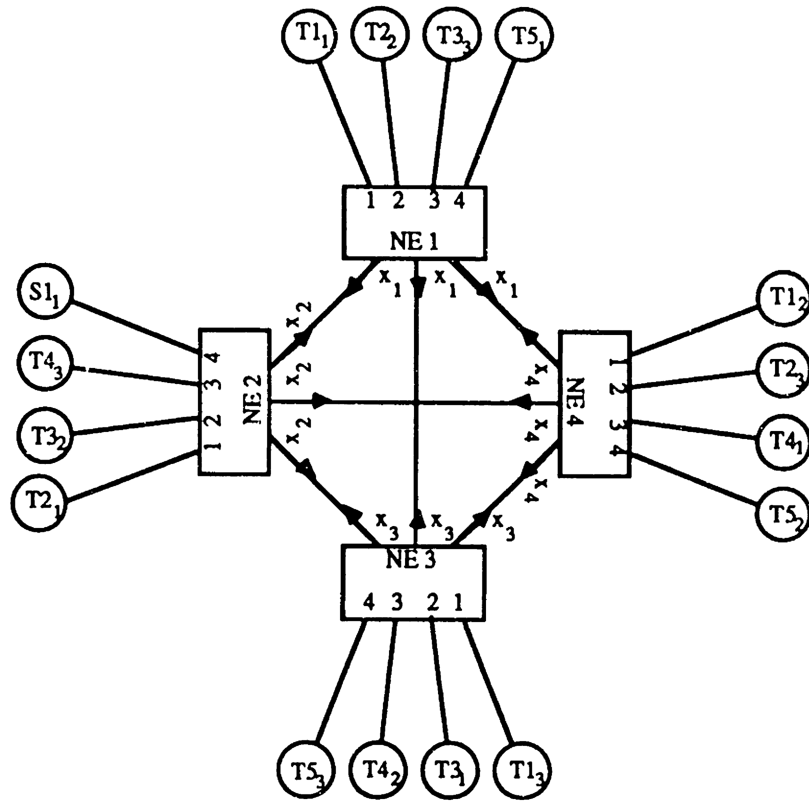


Figure 10.17

Phase 3 of Class 2 Exchange

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	$\{x_4\}$	$\{x_2\}$	$\{x_3\}$	$\{x_1\}$
Reflect Bank FIFOs:	{}	{}	{}	{}
NE2:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	$\{x_1\}$	$\{x_3\}$	$\{x_4\}$	$\{x_2\}$
Reflect Bank FIFOs:	{}	{}	{}	{}
NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	$\{x_2\}$	$\{x_4\}$	$\{x_1\}$	$\{x_3\}$
Reflect Bank FIFOs:	{}	{}	{}	{}
NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	$\{x_3\}$	$\{x_1\}$	$\{x_2\}$	$\{x_4\}$
Reflect Bank FIFOs:	{}	{}	{}	{}

Figure 10.18

FIFO Map Following Phase 3 of Class 2 Exchange

The Network Elements vote the copies in phase four. The vote mask depends on which Network Element sourced the message (the reflection from the message originator, in this case x_1, is never voted...see [Pease80].) and which Network Element is performing the voting (see Figure 10.18, Vote Bank FIFOs). Following the vote, the Network Elements hosting members of T2 deliver the message $x_{consistent}$ to the In FIFOs corresponding to the recipients. They are then ready for transmission of the next message.

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	$\{x_{consistent}\}$	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

NE2:	port 1	port 2	port 3	port 4
In FIFOs:	$\{x_{consistent}\}$	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	$\{x_{consistent}\}$	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{}	{}	{}	{}

Figure 10.19
FIFO Map Following Phase 4 of Class 2 Exchange

10.4 Operation of Network Element Core

10.4.1 Network Element Core Cycle

This section describes the manner in which the Network Element core provides its services to the subscribing Processing Elements. These services may be summarized as (a) delivery of a consistent copy of each message to each destination; (b) delivery of corresponding messages at very close to the same time; and (c) delivery of messages in the same order on all Network Elements.

Operation of the Network Elements is cyclic. A basic cycle, depicted in Figure 10.20, is divided into two or more frames, where frames are defined in the sense of the synchronization taxonomy introduced above. Before discussing how synchronous frames are achieved, each frame of this cycle will be discussed in detail.

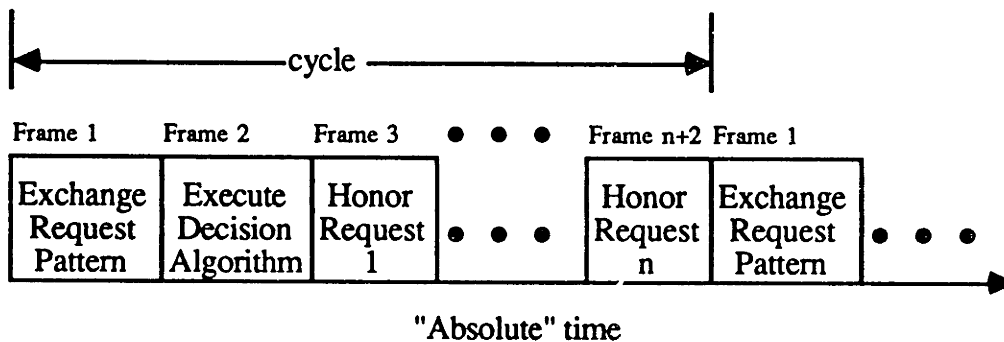


Figure 10.20

Basic Network Element Aggregate Cycle

The purpose of the first frame is to provide each Network Element with an identical copy of the cluster-wide message transmission *request pattern*. The *exchange request* corresponding to a FIFO ID consists of the first byte in the Out FIFO, which contains the destination Virtual ID of the message, the first nibble of the Control FIFO, which contains the exchange class and member number of the message source if appropriate, and a nibble

denoting the state of the In and Out FIFOs (Figure 10.21).

7		0
B	Destination VID	
Class	FIFO Status	

← first byte of Out FIFO
B: Broadcast to all groups

first nibble of Control FIFO (→ Class)
nibble from FIFO CSR (→ FIFO Status)

FIFO CSR nibble format: | TF | TE | RF | RE |

TF: Transmit FIFO full
TE: Transmit FIFO empty
RF: Reflect FIFO full
RE: Reflect FIFO empty

Figure 10.21

Exchange Request Format

This information completely defines the message from the point of view of the Network Element, and in conjunction with the exchange requests emanating from the FIDs corresponding to the other members of the virtual group, is necessary and sufficient to allow the Network Element aggregate to arrive at a decision regarding enabling transmission of that message. Notice that the exchange request does not include the VID of the message source. This is determined by the Network Elements during their message enabling calculations, using their Configuration Table. If this were not done it would be possible for a faulty Processing Element to masquerade as a member of a virtual group of which it is not a member.

Define the *local exchange request pattern* (LERP) to be the aggregation of the exchange requests for all Processing Elements subscribing to a given Network Element. There is one LERP for each Network Element. Let $LERP_i$ denote the LERP for Network Element i. Figure 10.22 shows the LERP for Network Element 1 in the cluster under discussion.

FID:		
(1,1)	B	Destination VID
	Class	FIFO Status
(1,2)	B	Destination VID
	Class	FIFO Status
(1,3)	B	Destination VID
	Class	FIFO Status
(1,4)	B	Destination VID
	Class	FIFO Status

Figure 10.22

Local Exchange Request Pattern for Network Element 1 ($LERP_1$)

Finally, define the *system exchange request pattern* (SERP) to be the aggregation of all local exchange request patterns. There is one SERP for each cluster. The SERP changes from cycle to cycle as message requests are registered and honored.

At the end of the first frame, each Network Element is to be cognizant of message exchange requests emanating from the Processing Elements subscribing to all Network Elements in the cluster, including itself; i.e., each Network Element must be provided with a consistent SERP. Since the request pattern is different on each Network Element, it is necessary to arrive at a Byzantine Resilient consensus on the aggregate request pattern on each Network Element by performing a two-round interactive consistency protocol. To this end, each Network Element begins frame one by broadcasting a packet containing its LERP. All Network Elements broadcast their LERP packets synchronously during the first phase of the first frame. At the end of the first frame, each Network Element has received a LERP from each other Network Element and stored it in the reflect FIFO connected to the source of the individual LERP packet. See Figure 10.24 for the FIFO map obtained after the LERP broadcast.

FID: (1,1)	B	Destination VID
	Class	FIFO Status
(1,2)	B	Destination VID
	Class	FIFO Status
(1,3)	B	Destination VID
	Class	FIFO Status
(1,4)	B	Destination VID
	Class	FIFO Status
FID: (2,1)	B	Destination VID
	Class	FIFO Status
(2,2)	B	Destination VID
	Class	FIFO Status
(2,3)	B	Destination VID
	Class	FIFO Status
(2,4)	B	Destination VID
	Class	FIFO Status
FID: (3,1)	B	Destination VID
	Class	FIFO Status
(3,2)	B	Destination VID
	Class	FIFO Status
(3,3)	B	Destination VID
	Class	FIFO Status
(3,4)	B	Destination VID
	Class	FIFO Status
FID: (4,1)	B	Destination VID
	Class	FIFO Status
(4,2)	B	Destination VID
	Class	FIFO Status
(4,3)	B	Destination VID
	Class	FIFO Status
(4,4)	B	Destination VID
	Class	FIFO Status

Figure 10.23

System Exchange Request Pattern (SERP)

NE1:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{$LERP_4$}	{$LERP_2$}	{$LERP_3$}	{$LERP_1$}
NE2:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{$LERP_1$}	{$LERP_3$}	{$LERP_4$}	{$LERP_2$}
NE3:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{$LERP_2$}	{$LERP_4$}	{$LERP_1$}	{$LERP_3$}
NE4:	port 1	port 2	port 3	port 4
In FIFOs:	{}	{}	{}	{}
Out FIFOs:	{}	{}	{}	{}
	left	right	opposite	mine
Vote Bank FIFOs:	{}	{}	{}	{}
Reflect Bank FIFOs:	{$LERP_3$}	{$LERP_1$}	{$LERP_2$}	{$LERP_4$}

Figure 10.24

FIFO Map Obtaining After LERP Broadcast

In the second phase of frame one, the Network Elements concatenate the LERP packets into a SERP packet and synchronously broadcast the resulting SERP packet. Note that each Network Element activates its Reflect FIFOs in a sequence that is dependent on which Network Element it is. If the format of the SERP is $\{LERP_1, LERP_2, LERP_3, LERP_4\}$, then the Network Elements must sequence their Reflect FIFOs in the following orders:

Network Element 1: {M, R, O, L} $\Rightarrow SERP_1=\{LERP_1,LERP_2,LERP_3,LERP_4\}$;

Network Element 2: {L, M, R, O} $\Rightarrow SERP_2=\{LERP_1,LERP_2,LERP_3,LERP_4\}$;

Network Element 3: {O, L, M, R} $\Rightarrow SERP_3=\{LERP_1,LERP_2,LERP_3,LERP_4\}$;

Network Element 4: {R, O, L, M} $\Rightarrow$ $SERP_4$={$LERP_1$,$LERP_2$,$LERP_3$,$LERP_4$}, where $SERP_i$ is defined to mean the SERP packet reflected by Network Element i on the reflect phase of the SERP exchange.

At the end of this second phase, each Network Element possesses four copies of a SERP in its vote FIFO bank which may be voted in the third phase of frame one to obtain a SERP that is consistent on all Network Elements in the face of Byzantine Network Element faults. Note that if a faulty Processing Element transmits a garbage packet into its Out FIFO and/or its Control FIFO, that garbage is perceived consistently by all non-faulty Network Elements at the end of frame 1. Thus, frame one terminates with each Network Element possessing an identical SERP[10]. From the output of the voter, the SERP is fed into a set of control logic called the *scoreboard*, which executes the second frame of the cycle.

In frame two, the scoreboard of each Network Element operates on its copy of the consistent SERP to decide which if any message(s) to transmit in subsequent frames of the cycle. Each Network Element arrives at the identical decision because all execute the same hardwired decision algorithm on identical inputs (i.e., the consistent SERP). Define an *enabled message* as one which passes the *valid message condition* and the *destination free* condition.

Intuitively, the valid message condition is satisfied when the Network Element has legitimate reason to believe that a non-faulty member of an FMG desires to transmit a message, and in addition all non-faulty members have requested such a transmission. The Configuration Table indicates which Out FIFOs are considered capable of sourcing valid messages; processors previously determined to be faulty may have their Out FIFOs "turned off" via CT updates, and such processors will not be allowed to source messages. An FMG consists of 2f+1 or more members, with the maximum skew between arrival at a synchronization act, in this case a message transmission, being upper-bounded by σ time

[10]In the CSDL FTPP, the SERP exchange takes about 5 μsec.

units. The arrival of the message transmission request from the f+1st member of this FMG heralds the arrival of the first message request which can be guaranteed to emanate from a non-faulty member. Upon perception of this event, the Network Element aggregate must wait another σ time units to be assured that all non-faulty members have requested a transmission, or for the remaining f FMG members to request the transmission. Upon the fulfillment of either of these two conditions, the valid message condition is said to be satisfied. If the source virtual group is not an FMG, then a weaker valid message condition is used. If the group is a duplex, then the valid message condition is met when either both members of the group request a transmission, or one member has requested a transmission and a time interval of σ has expired since the initial request[11]. If the group is simplex the valid message condition is trivially met when the member requests the transmission[12].

In addition to transmitting only those messages which have legitimately emanated from recognized members of a virtual group, the Network Element core must ensure that the In FIFOs of the destination group have adequate space to accept the message before enabling its transmission[13]. The In FIFOs of non-faulty members of a virtual group may legitimately fill up because the group may be executing a computationally intensive segment and therefore neglect reading their In FIFOs for some time. It is critical that messages addressed to such a group not get lost to avoid violation of the Byzantine Resilient Virtual Circuit abstraction. Therefore the decision logic that the Network Elements execute must include the destination free condition. Destination free is calculated much like the valid message condition, and intuitively implies that the Network Element aggregate has reason

[11] This is admittedly not airtight. In the case of a non-fault masking group, it is possible in the worst case to determine with a probability of unity that a fault has occurred in one of the members, but not which member is faulty. Therefore a faulty duplex member could spuriously send a message, trigger a timeout, and have the message sent. Lack of corroboration of that message by the non-faulty member would indicate that a fault had occurred, but this is the best that can be done in the worst case.

[12] In the case of a simplex source, secondary fault tolerance techniques must be used to detect a fault. These techniques of course have a coverage of less than unity.

[13] This problem was initially pointed out by G. Troxel.

to believe that there is a legitimate reason for an In FIFO to be full. A Network Element's LERP includes a "full" bit for each In FIFO subscribing to the Network Element. If this bit is set, it indicates that the corresponding In FIFO cannot accept any more packets. The full bits corresponding to the In FIFOs of a destination virtual group are examined by the Network Element in the decision algorithm. If a majority of the full bits of the In FIFOs of an FMG are set, any message destined for that FMG is indefinitely delayed until the full condition is alleviated. If a minority of the recipients indicate a fullness problem, messages are delayed until a time interval of σ has transpired since the first indication of a congestion problem or the full condition no longer exists, at which time the full condition is overidden and the message is transmitted. In this manner the failure of up to f FMG members cannot delay message transmission indefinitely, but bounded-skew arrival of non-faulty destination members at a full condition is tolerated. If no destination FMG indicates a full In FIFO, the message is enabled. Similar rules apply for a duplex destination. If one member indicates a full In FIFO, the message is delayed for a time interval σ. If the fullness condition is neither cleared nor corroborated by the other member by this time, the message is enabled. If both members indicate fullness, the message is delayed until the condition is alleviated by one or both members emptying their In FIFOs. Finally, if neither member indicates fullness, the message is enabled. For a simplex destination, a full In FIFO indication is allowed to delay a message transmission for a bounded time period. After this period has expired, the message is enabled. As usual, a no fullness indication allows the message to be enabled immediately.

If a message transmission request passes both the valid message condition and the destination free condition, then it is transmitted, voted and delivered in subsequent frames of the Network Element core cycle, according to its exchange class.

In the subsequent and final frames of the Network Element core cycle, the Network Element aggregate sends all enabled messages one at a time. Any fair message transmission prioritization will do since all enabled messages will get sent before the Network Element

aggregate re-reads the request pattern. A particularly simple prioritization scheme is to send messages in the inverse order of the redundancy level of the source virtual group, such that quad-sourced messages get sent first, triplex-sourced messages second, etc. Any mixture and interleaving of Class 1 and Class 2 messages may be sent during a cycle. As intimated earlier and elucidated below, the Network Element core synchronously broadcasts, reflects in the case of Class 2 messages, votes, and delivers a single message per frame. Completion of the message delivery occurs on all Network Elements at very close to the same time.

10.4.2 Achieving Frame Synchrony

This section illustrates the method used by the Network Elements to achieve the framewise synchronization alluded to above. Based on the above discussion, it may be assumed that at the beginning of the frame, the exchange class, the source FIFO(s) (1 for Class 2, at least 2f+1 for Class 1), the destination FIFOs, and the packet size (i.e., frame length) are known. Mutual agreement exists on all these parameters at all Network Elements in the cluster by virtue of the identical decision algorithm executed on the consistent SERP. This algorithm was executed by a hardwired finite state controller called the scoreboard. At the end of the decision phase, the above parameters are known for each enabled message. At the beginning of a message transmission frame, the parameters are input to another finite state controller which is responsible for the sequencing of the components during the frame. The basic operation of a Network Element during the frame is to transmit the appropriate packet in byte-serial fashion until the entire packet has been transmitted. Simultaneously, all other Network Elements transmit the appropriate packet. Not all Network Elements necessarily transmit packets possessing the same contents. Network Elements hosting a packet source will transmit the data packet; those not hosting a packet source will transmit a null packet. Only packets corresponding to a single message from a single virtual group are transmitted in a single frame.

When a Network Element transmits a byte, it also transmits a clock signal which notifies the recipient Network Element that a byte has been sent and is available on the recipient Network Element's input port[14]. The recipient Network Element uses this clock signal to shift the byte into the appropriate FIFO (either the Vote bank or the Reflect bank, depending on the exchange class and phase). When transmission of the packet is over, the sending Network Element stops sending clock pulses for three to five of its local clock cycles. Recipients detect this cessation of clock pulses using a "missing pulse detector" circuit and surmise that the sending Network Element has reached the end of its transmission. The detection of the pulse train cessation arms a circuit that detects the startup of the next train of clock pulses marking the beginning of the transmission of the next packet from that Network Element. Detection of this pulse train's startup following a period of quiescence indicates to the recipient Network Element that the sending Network Element has begun a new frame.

From a Network Element's comparison of the time at which it began the transmission of a given frame with the observed times at which the other Network Elements began transmitting the corresponding frame's packets, a Network Element can calculate its skew relative to the other Network Elements, adjust the time at which it begins the next frame, and thereby reduce the inter-Network Element skew. Two subtleties are important. First, because of physical separation and delays through logic elements the time that a given Network Element detects the beginning of another's transmission may be significantly delayed from the actual time the transmission began[15]. If this fact is not accounted for, all Network Elements would constantly surmise that they are running fast because of their delayed perceptions, attempt to catch up, and fail to remain synchronized. However, if this

---

[14]In the FTPP under construction at CSDL, one byte is shifted out of each Network Element every 125 nsec resulting in an inter-Network Element bandwidth of approximately 64 Mbits/sec.

[15]In the CSDL FTPP, 50 feet of cable plus miscellaneous logic element delays amount to ≈105 nsec of propagation delay.

propagation time is known to within plus or minus 1/2 cycle of the data transmission clock [Shin86], then a recipient Network Element can delay the moment of comparison by that propagation delay and correctly deduce its relative skew. This capability is extremely important in light of the desire to physically disperse the elements of the distributed system at will.

The second subtlety also deals with the selection of the signal used to calculate a Network Element's relative skew. This selection must be Byzantine Resilient. [Krishna85] showed that the usual practice of the use of the median edge arrival as the comparison signal is inadequate if it is desired to tolerate more than one simultaneous clock fault. Define the tick sequence to be the ordered sequence of the times of occurrence of the indications that the Network Elements begin a frame, as perceived by a given Network Element. Thus, one possible tick sequence is $t_j=\{x_2, x_3, x_4, x_1\}$, indicating that from the point of view of the recipient Network Element number j, Network Element 2 began its frame first, followed by Network Element 3, followed by Network Element 4, followed by Network Element 1. Not all Network Elements have identical tick sequences because malicious faults can to some extent reorder the sequences at different Network Elements. Let i denote the position of Network Element j's frame commencement in j's tick sequence t_j. If j=1 in the above example, i would be equal to 4. In [Krishna85], it is rigorously shown that if it is desired to tolerate $m \geq 1$ simultaneous clock faults from among $N \geq 3m+1$ fault containment regions, then the $2m^{th}$ edge must be chosen as the comparison edge by Network Element j if $i<N-m$, and the $m+1^{st}$ edge must be chosen if $i \geq N-m$, where i is defined as above. In the cluster under discussion, it is desired to tolerate only a single (m=1) clock fault, so the above equations imply that the $2m=m+1^{st}=2$ or second-to-occur frame commencement is adequate for comparison purposes. It must simply be borne in mind that if the cluster is to be expanded to tolerate more than one simultaneous Network Element fault, a more complicated comparison edge selection criteria must be used.

The recipient Network Element compares the time of occurrence of the second-to-occur

frame commencement with the suitably delayed time of commencement of its own frame, and from this deviation computes an adjustment, which it will make to its own clock by inserting missing pulses at the end of its frame. The Network Element inserts three missing pulses if it perceives that it is behind the comparison signal, four missing pulses if it perceives that it is in synchrony with the comparison signal, and five missing pulses if it perceives that it is ahead of the comparison signal. Three, four, and five missing pulses are inserted instead of zero, one, or two, respectively, because the Network Elements require a certain amount of dead time between transmission frames to perform bookkeeping and allow pipelines to drain. The insertion of the appropriate number of missing pulses reduces the temporal skew between the commencement of the subsequent frames in exactly the same manner as the Fault Tolerant Clock circuit of the Fault Tolerant Processor. Figure 10.25 shows the adjustments in each of the three cases.

The Network Element's Fault Tolerant Clock circuit generates a *presence syndrome* as an indication of which if any other Network Elements started their frame outside an error window defined by the occurrence of the comparison edge. This syndrome is made available to the circuitry which appends the vote syndrome to the delivered packet[16]. A *clock mask* may be written to a given Network Element instructing it to disregard the transmissions emanating from selected Network Element(s) identified as faulty by a failure detection and isolation task running on one of the cluster's FMGs.

If the frame requires a vote of a received set of packets, each Network Element activates its voter and feeds it with the contents of the vote FIFOs, applying the appropriate vote mask for the Network Element, exchange class, and packet source(s). Upon completion of the vote, the Network Element routes the voter output to the destination FIFOs based on CT-based mapping from the destination VID(s) to the destination FID(s).

If the frame is a reflect frame, the broadcast packet was shifted into the Reflect FIFO on

[16]In the CSDL FTPP, the inter-Network Element links use parity coding as a secondary fault detection technique. Other more extensive secondary methods are possible.

the previous broadcast frame. Thus, in this frame the Network Elements will activate the Reflect FIFO which received the broadcast and re-broadcast the message. This action is followed by the vote phase as indicated above. Because of the use of separate transmit and vote busses, transmission and vote phases can be overlapped, allowing a degree of pipelining in the communication process.

At the end of the frame, the Network Elements each read the length of the next frame, wait the number of cycles required to reduce their apparent skew, and begin the next transmission. If no further messages are enabled, the Network Aggregate begins frame 1 of a new cycle.

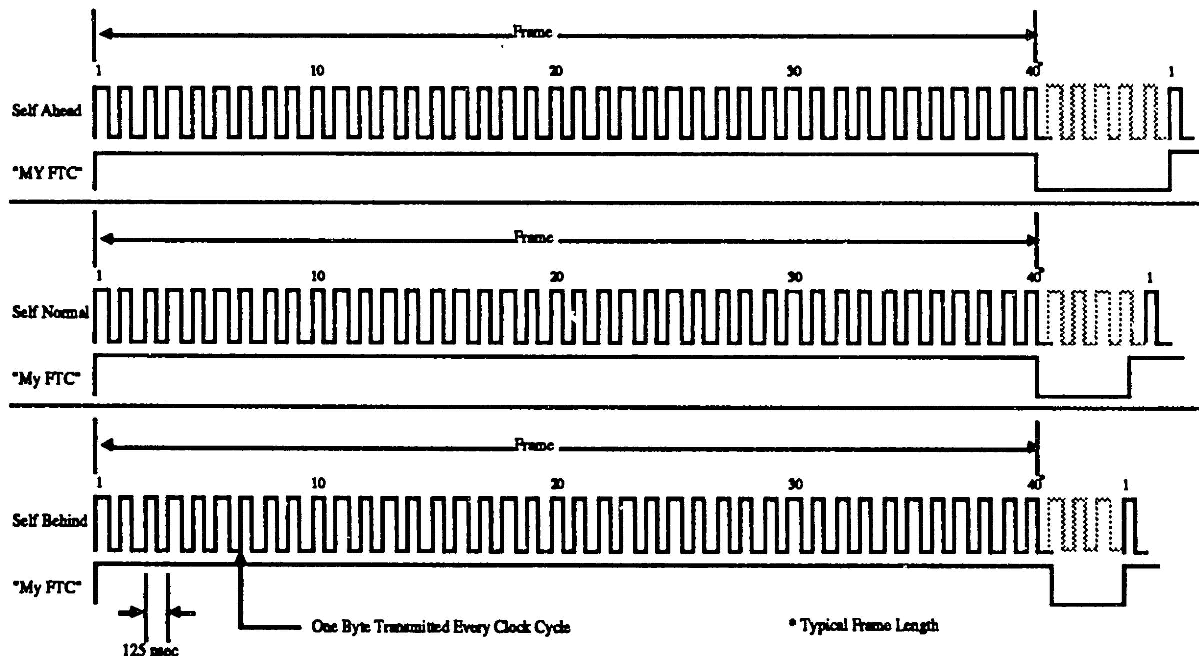


Figure 10.25

Network Element Synchronization Algorithm

10.5 Provision of BRVC Abstraction

From the discussion of the operation of the Network Element core and the a priori assumptions on the behavior of the Processing Elements, it is straightforward to summarize how intra-cluster BRVC is provided.

The NE aggregate executes tightly synchronous frames using an f-Byzantine Resilient hardware synchronization method. Two or more sequential frames compose a cycle. At the beginning of each cycle, the aggregate obtains a f-Byzantine Resilient consensus on the request pattern generated by the subscribing Processing Elements. This is true regardless of Processing Element behavior. Using identical Configuration Tables, the Network Elements synchronously perform a decision algorithm on this consensus to decide which if any message requests emanating from the Processing Elements are to be honored. All non-faulty Network Elements arrive at identical decisions. In each subsequent frame, the Network Elements shift out messages from the FIFOs indicated by the source VID of the message to be transmitted in that frame. They perform either a one-round vote protocol in the case of an FMG comprising 2f+1 members having ostensibly identical messages in each channel, or a f+1-round interactive consistency protocol in the case of a message which is member-specific or does not emanate from all members of an FMG. In the presence of any combination of up to f Network Element or Processing Element faults each destination receives an identical message at the termination of each frame. Since the Network Elements execute identical actions per frame, the messages get delivered by all Network Elements in identical orderings. Since the Network Elements are tightly synchronized via the f-Byzantine Resilient Fault Tolerant Clock scheme, all Network Element actions occur within a very small time of each other. In particular, message deliveries occur very close in time to each other.

FMG members execute functional synchronization by defining a certain class of actions to be synchronizing acts. Non-faulty FMG members send messages into the aggregate in the same order and arrive at synchronizing acts with a maximum skew of σ time units of each other. They achieve synchronization by sending messages to themselves and awaiting the delivery of those messages. Because of the tight synchrony of the NE aggregate, reception of a given message serves to synchronize the FMG. A given exchange will not occur unless a majority of the 2f+1 members of the sourcing FMG submit identical

requests. In the case of Class 1 exchanges, the existence of 2f+1 message sources guarantees that a majority of non-faulty members will send correct messages and correct voted copies of those messages will be delivered. In the case of Class 2 messages, all destinations will receive consistent copies of the message by virtue of the Byzantine Resilient interactive consistency protocol executed by the NE aggregate on behalf of the source. If the source is nonfaulty, all non-faulty destinations will receive the message sent by the source. If the source is faulty, all non-faulty destinations will receive consistent copies of the erroneous message.

10.6 Cluster Expansion

Clusters are expanded by the addition of an IO Element (IOE) to each Network Element and connecting corresponding IOEs together in the desired inter-cluster topology. Since there are 3f+1 Network Elements between any two clusters there will be 3f+1 inter-IOE links. Until the fault masking capability of these inter-cluster links is saturated by the occurrence of IOE or NE faults, faults occurring during inter-cluster transmissions can be masked and the faulty link identified. Figure 10.26 shows three clusters connected in a linear topology, using 4-wide redundant inter-cluster links.

10.6.1 Inter-Cluster Message Transmission

To illustrate inter-cluster communication we will describe the case when an FMG in one cluster transmits a message to an FMG in another cluster[17]. It will be assumed that the message is the inter-cluster equivalent of a Class 1 exchange, in which all non-faulty source FMG members transmit identical messages. Class 2 exchanges are similarly handled. Assume that the VIDs and FIDs have been augmented such that they correctly reflect the cluster upon which a group resides. By inspection of the destination VID, the source FMG's message transmission software determines that the destination FMG does not reside on the current cluster but on another. It therefore re-addresses the message to the VID of the IOE resident on the cluster of the sending FMG. The true destination VID is embedded in the message by the sending software. The Network Element core as usual honors this message when all members of the source FMG request an identical exchange or a majority request such an exchange and a time interval of σ has expired since the f+1st request. All members of the IOE group receive their voted copy of the message at very close to the same time, and "validate" any messages that they receive by performing scoops at a specified rate. All messages received prior to a given scoop are guaranteed to have been

[17] This example does not demonstrate the most efficient mechanization of inter-cluster communication, but instead is designed to illustrate the general idea.

delivered in the same order at all members of the IOE.

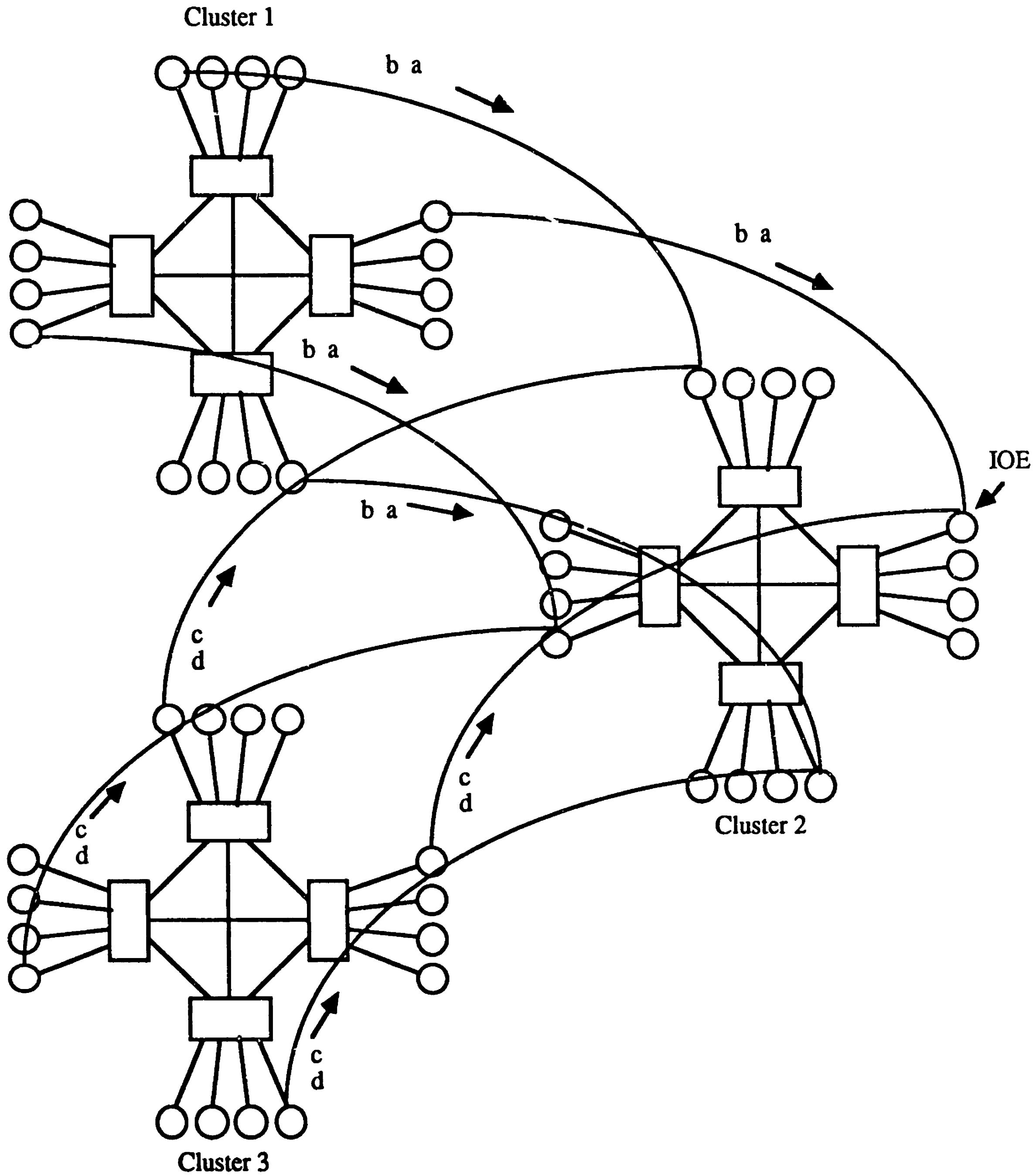


Figure 10.26

Three Clusters

The IOE group busies itself by performing scoops at a prespecified rate. When the IOE

group determines that it has received a message carrying a message destined for another cluster, all IOE members synchronously transmit that message over the inter-cluster links to their corresponding IOE in the cluster on which the destination FMG resides. Note that non-faulty IOE members transmit corresponding messages within time σ of each other. If the message must go through an intervening cluster to reach its destination, the sourcing IOE calculates the routing information and transmits the message to the correct intermediate cluster.

A given scoop may yield zero, one, or several valid transmission requests. The IOE honors these requests in the order received. Note that this guarantees that inter-cluster transmissions emanating from a given FMG are transmitted to the adjacent cluster in the order sent. After performing the requisite transmission(s), the IOE continues with its periodic scooping until the next outgoing inter-cluster message arrives.

10.6.2 Inter-Cluster Message Reception

Reception of inter-cluster messages is not as simple as their transmission because different members of a recipient IOE may perceive transmissions emanating from different source IOEs in different orders. Figure 10.27 shows the situation which obtains when IOEs 1 and 2 in a recipient cluster receive the messages, a, b, c, and d from adjacent clusters. If these messages arrive at close to the same time, then the IOEs may perceive their arrival in different orders.

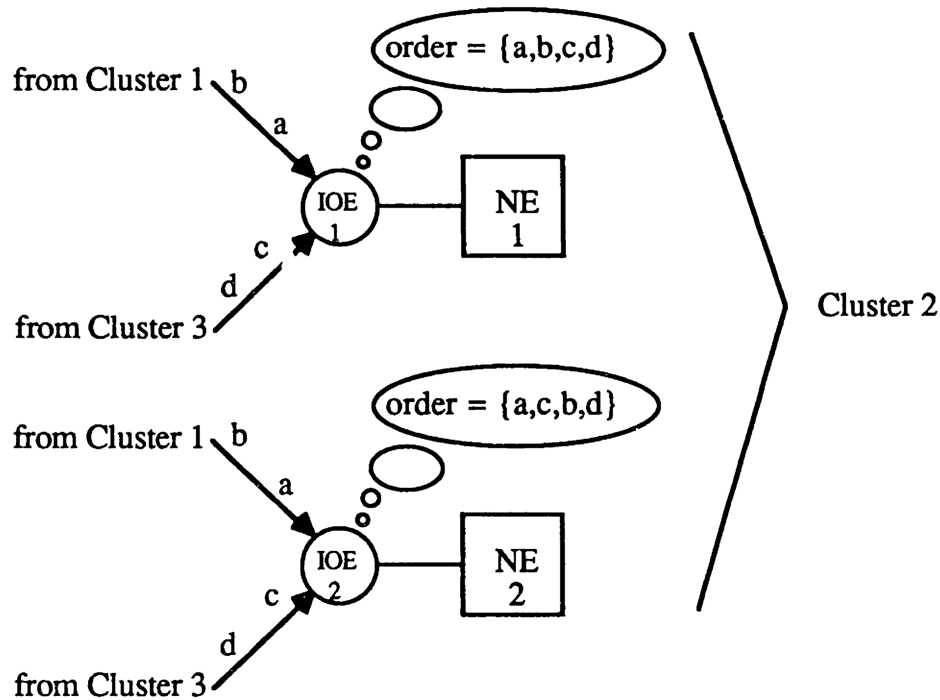


Figure 10.27

Inter-Cluster Message Order Ambiguity

In the above example IOE 1 in cluster 2 has decided that the arrival order is {a,b,c,d}, and IOE 2 has decided that the arrival order is {a,c,b,d}. Notice that messages emanating from the *same* cluster are perceived in identical order at different IOE members since they were sent in the same order by the relaying IOE, and that this was the order in which the messages were sent by the source FMG. However, the different IOEs cannot perform a Class 1 exchange to the destination FMG using the first message in their message queues because this first message can legitimately differ in different IOEs.

The solution to this problem is to use the scoops that the IOEs must perform to obtain a consistent set of outgoing messages to obtain a consistent set of incoming messages as well. Specifically, the members of the IOE group take turns sourcing a Class 2 exchange containing the directory of their incoming inter-cluster message buffer. Class 2 exchanges may be scoops as well as Class 1 exchanges, since the only purpose of a scoop is to serve as a placeholder demarcating group-wide consistent message sets from possibly

inconsistent message sets. After each cycle of Class 2 exchanges, i.e., after each IOE member has had a chance to distribute its incoming message buffer directory, the IOE as a group now has a consistent copy of each member's incoming buffers. Therefore the IOE as a group can decide whether a valid incoming inter-cluster message has been received. The IOEs make this decision based on the simple majority rule: if all IOE members have received a given message, say "a", the message is enabled. If the majority of the IOEs have received a given message, they initiate a timeout. When it expires they enable the message. The value of this timeout is equal to the maximum amount of time which may transpire between the reception of a message from any two non-faulty members of a transmitting IOE.

To guarantee that messages are delivered in the order sent the messages emanating from a given cluster must be delivered in the order in which the majority of the incoming links have transmitted the message. Let b_{ij}, i,j=1,4, denote the contents of the message buffer of IOE member i containing messages from cluster j. At the end of the reception of the messages in the figure above, $b_{11}=\{a,b\}$ and $b_{21}=\{a,b\}$. Not shown are the message buffers for IOEs 3 and 4, b_{31} and b_{41}. Under non-faulty conditions, all message buffers would be identical, i.e., $b_{i1}=\{a,b\}$, i=1,4. Faulty transmissions however may cause at most f of these message buffers to possess incorrect contents. In particular, if a Byzantine fault were to occur in transmitting IOE 1 and cause message "a" to not be transmitted from cluster 1 by IOE member 1, then the message buffers would appear as

$$b_{11}=\{b\},\ b_{21}=\{a,b\},\ b_{31}=\{a,b\},\ \text{and}\ b_{41}=\{a,b\}.$$

To maintain BRVC the recipient IOEs must maintain the original transmission ordering {a,b} by delivering messages in the order in which they were transmitted in the majority of the cases. Therefore they must not deliver the completed transmission b_{voted} before the timed-out transmission of a_{voted}, even though b_{voted} is ready before a_{voted} times out.

When the IOE group has determined that a valid message has arrived on the incoming inter-cluster links upon completion of the cycle of Class 2 exchanges, it has two cases to

consider. If the message is destined for an FMG residing on the recipient IOE's cluster, the IOE may perform a Class 1 exchange to the destination FMG, using the correct VID as retrieved from the internals of the inter-cluster message. If the message is to be forwarded to another cluster, the IOE performs a Class 1 exchange to itself, thus masking any faults occurring in the transmission of the message from the source cluster and regenerating the message's redundancy level. Subsequently, the IOE transmits the voted message to the cluster which is closer to the destination cluster. All IOE members of course perform the same sequence of actions, with corresponding actions occurring within time σ of each other. At the destination FMG the BRVC guarantee holds because of the properties of the intra-cluster protocols discussed earlier.

10.7 Failure Detection, Identification, and Reconfiguration

10.7.1 Distributed Failure Detection and Identification

Each member of a group receives diagnostic information appended to each packet that it receives. This information is the vote syndrome, the Network Element presence syndrome, and, in the case of the CSDL FTPP, the inter-Network Element link parity syndrome. The first two quantities arise from the primary means used to obtain fault tolerance, i.e., the replicated state machine approach, which provides fault masking and detection with unity probability. The last quantity is representative of secondary low-coverage fault tolerance techniques which can beneficially augment the primary technique. Examples of such techniques which would be implemented in a production system are error detecting and correcting codes, watchdog timers, "local" sparing such as spare bit planes in large memories, and a host of others [Siewiorek82], [Anderson81].

The interpretation of the vote syndrome depends on the class of the exchange experienced by the packet to which it is appended. If the exchange was a Class 1 exchange, then a non-zero vote syndrome indicates that either a source FMG member has transmitted a faulty message or the broadcasting Network Element has transmitted a faulty message.

The perception of the two failure modes overlaps. If the exchange was a Class 2 message, then the vote syndrome solely indicates Network Element health since voting has no meaning for a simplex-sourced message.

Examination of the other syndromes helps disambiguate the Class 1 vote syndrome. For example, if the Network Element which transmitted an erroneous voted input does not set off its presence bit or its parity bit, then one has evidence that the vote error resulted from a faulty source Processing Element. Further corroboration of this would be the observation that other Processing Element sources residing on the implicated Network Element do not give rise to errors. Alternatively, if the presence bit and/or parity bit substantiate suspicion of a given Network Element, or if a Class 2 vote indicates a similar error, then the fault may be isolated to the Network Element. In any event, it must be borne in mind that the primary fault containment region comprises a Network Element *plus* its subscribing Processing Elements. In the worst case, fault isolation can do no better than to indicate which two fault isolation regions must contain the fault (Consider an inter-NE link fault). However, the above secondary indicators will be useful in a majority of cases.

Aside from the Network Element error information which group members receive at a high rate, Processing Element error information is received by a recipient when the source group transmits a Class 1 exchange. Since groups transmit messages only to those groups with which the applications or operating system functions request interactions, no group is likely to have at any time a complete or up-to-date view of the error status of all Processing Elements in the cluster. Such a comprehensive view is likely to be impossible in any event. The reconfiguration strategy outlined below recognizes this fact, and therefore allows any recipient group possessing an adequately high degree of redundancy to act as a reconfiguration agent upon perception of a fault.

10.7.2 Distributed Reconfiguration

Distributed reconfiguration relies on the consistent total ordering property of the

Network Element aggregate, which holds in the presence of up to f simultaneous Network Element faults. The need for reconfiguration is triggered when a recipient FMG receives one or more packets containing an unacceptably high number of error indications. We assume for the time being that only FMGs may attempt reconfiguration, and further that the executive software resident on an FMG is cognizant of the fact that it is running on an FMG. Before starting the reconfiguration process, the FMG attempts to become the cluster's sole *Reconfiguration Authority* or *RA* by executing a distributed contention algorithm. Since it is likely that the error threshold is exceeded at very close to the same time in several FMGs, mutually exclusive RA-ship is mandatory to ensure a logically consistent reconfiguration policy.

An FMG desiring to become RA broadcasts a *Request_RA-ship* message indicating this desire. Broadcasts are indicated by an asserted high-order bit in the destination VID of the message. Once enabled by the Network Element aggregate, broadcasts are delivered by the aggregate to all subscribing Out FIFOs. Since the aggregate performs the same actions in each frame of the cycle, a sequence of broadcasts is delivered in the same order to all destination FIFOs. All groups receive any sequence of RA-ship requests in the same order whether they are looking for them or not, and the first such request to be delivered notifies all recipient groups that (1) a reconfiguration phase is in progress, and (2) the sender of the first message is the new RA. Meanwhile, any FMG which has sent a Request_RA-ship message awaits the arrival of the first such message. The lucky sender of this message is the new RA. Any subsequent receptions of Request_RA-ship messages by any group are ignored, until the reconfiguration phase is completed. Upon completion of reconfiguration, the RA broadcasts a *Relinquish_RA-ship* message. Reception of this message by a group indicates that the reconfiguration phase is over and any subsequent Request_RA-ship messages may again be honored.

An established RA achieves reconfiguration by broadcasting Configuration Table updates. Configuration Table updates are a subclass of Class 1 exchanges addressed to the

Network Element which simply change the mapping from FID to VID. In addition, Out FIFOs can be commanded to be disabled or enabled as desired. In the FTPP implementation, a single CT update can only modify the FIDs corresponding to one VID, but the RA may issue an indefinitely long sequence of such updates to perform any desired reconfiguration. In addition to updating its copy of the Configuration Table, a Network Element upon reception of a CT update exchange delivers the update packet to all subscribing FIFOs. Virtual groups, which execute normal operation during the reconfiguration phase, scoop in these CT update packets as they read their input message buffers. The input message processing software resident on each group has the capability to recognize CT update packets.

When a CT update is received by a group and its software determines that the cluster is indeed in a reconfiguration phase, i.e., that a RA is defined and is the same as the sender of the CT update[18], this software examines the FIDs embedded in the packet to determine if its hosting Processing Element is one of the Processing Elements affected. If not, the packet is used to update its local copy of the CT. Note that all CT updates are consistently serialized at all virtual groups. If the hosting processor's FID is found in the packet, after performing the local CT update the message processing software invokes local reconfiguration software which transmits a reconfiguration acknowledgement to the RA. Depending on the characteristics of the newly formed virtual group of which this processor is now a member, the reconfiguration software may attempt to achieve functional synchronization with its other members, await an additional command from the RA, or perform some other function.

For each Processing Element that the RA has attempted to reconfigure, the RA awaits reception of a reconfiguration acknowledgement or the expiration of a suitable timeout. When all such situations have been resolved to its satisfaction, the RA broadcasts a

[18]This is not rigorously necessary but serves as an extra check on the validity of the reconfiguration command.

Relinquish_RA-ship message and continues normal operation.

10.7.3 Distributed Bootstrap and Reintegration[19]

It is desirable to bootstrap the cluster in a manner which is independent of the exact hardware complement of the cluster. Bound up in the bootstrap procedure is the procedure for reintegrating repaired or spare Processing Elements and Network Elements. It is desirable that the two procedures utilize identical mechanisms for the sake of efficiency and conceptual uniformity. Moreover, at bootstrap time no assumption can be made about the order in which the various components of the cluster become active. Therefore the bootstrap procedure must accommodate such arbitrary activation ordering and thereby become indistinguishable from the reintegration procedure. Not all bootstrap procedures described below are Byzantine Resilient because there are times at which there is an insufficient complement of hardware for Byzantine Resiliency. FMGs do not spring out of nowhere but are nucleated from pre-existing simplexes. Therefore in the early stages simplex groups are given powers that they will not possess after an FMG is formed. Once a minimum level of redundant operation is achieved, Byzantine Resilience is guaranteed and simplexes no longer have these powers.

We first describe the Processing Element bootstrap procedure. Upon power-up or reset, the Processing Element performs a self-test regimen. In the event of a persistent perceived failure of this test, the Processing Element disables itself and continues no further. It is possible that self-testing may miss faults, but since bootstrap is not time-critical it is possible to increase coverage somewhat by testing more exhaustively than possible in real-time operation. If the Processing Element incorrectly or otherwise determines that it has passed its self-test, it reads the NE_Ready field in the Network

[19]The concepts discussed in this section resulted from the coordinated efforts of S. Adams, S. Friend, R. Heyda, H. Laning, and the author. The procedure described below is an abbreviated version of that generated by these individuals.

Element's CSR array to determine whether the Network Element is active. If not, the Processing Element loops back to self-test and retries the Network Element at a later time. If the Network Element indicates that it is active, the Processing Element determines the Network Element's operational phase from the NE_Status CSR field, which can indicate that the Network Element is waiting to be tested, being tested, attempting to synchronize with the other Network Elements of the cluster, or in synchrony with the other Network Elements of the cluster.

If the Network Element is waiting to be tested, the Processing Element performs an atomic test-and-set on a field of the Network Element's CSR in an attempt to become the sole Processing Element allowed to test the Network Element. If the Processing Element succeeds in becoming the "debug master", the Processing Element puts the Network Element through a series of tests of its internal data paths and control logic. Following successful completion of these tests, the Processing Element sends the Network Element an "exit debug" command, upon which the Network Element attempts synchronization and changes state to "attempting synchronization". We will discuss this phase of the Network Element's operation later. The Processing Element then waits for the Network Element's state to change to "in synchrony". When the Network Element has achieved synchrony with the other Network Elements of the aggregate, it indicates "in synchrony" in its CSR array, at which time the Processing Element attempts to install itself in the Configuration Table as a simplex virtual group, with its VID equal to its FID. If the Processing Element loses the test-and-set and does not become the debug master, or if the Network Element status indicates to the Processing Element that the Network Element is being tested or attempting synchronization, the Processing Element awaits the "in synchrony" indication upon which it attempts to install itself in the CT as a simplex with its VID equal to its FID.

Upon Network Element power-up or reset, the Network Element initializes its CSRs and FIFOs, sets its NE_Ready to "Ready", and sets its NE_Status CSR to indicate that it is awaiting testing. From this point there are several possible scenarios, all with the same

outcome. The Network Element will wait for the arrival of a Processing Element debug master for a limited amount of time. If none arrives after the expiration of this time the Network Element attempts synchronization with the other Network Elements of the cluster. If on the other hand a Processing Element becomes debug master and sends the Network Element into debug mode, the Network Element responds to commands written into a debug register by the Processing Element, executing the requested debug activity. The Network Element will remain in debug mode for a limited amount of time regardless of the Processing Element's actions, after which it will exit debug mode and attempt synchronization. Alternatively, if the Processing Element writes an erroneous debug command, the Network Element will immediately enter initial synchronization activity. The above timeouts and error responses are to prevent a faulty processor from monopolizing a non-faulty Network Element.

Once out of debug mode, a Network Element ignores any further debug commands. It can be made to enter debug mode only via a valid Class 1 exchange addressed to the Network Element from an FMG commanding it to do so. Upon achieving synchronization, the Network Element begins SERP exchanges and begins transmitting messages from Processing Elements.

No a priori knowledge is available about how many Processing Elements are in existence at boot time, so no static Configuration Table can be defined. As mentioned earlier, Processing Elements come on line as simplexes by installing themselves into the CT with their VID equal to their FID, so the CT is dynamically created according to the Processing Element complement. Note that these CT updates must be Class 2 exchanges since the message source is a simplex. When a Processing Element gets installed in the CT, its In FIFO becomes capable of receiving messages and thus, because CT updates are broadcast, a given Processing Element can witness the arrival of all other Processing Elements. In particular, by observation of the Processing Element arrival sequence Processing Elements can determine when enough Processing Elements exist in different

fault containment regions to allow the formation of an FMG. By repeatedly broadcasting their perceived arrival sequences, the Processing Elements can come to an interactively consistent consensus on which three Processing Elements residing in different fault containment regions came on line first. Once this consensus is reached, the three Processing Elements concerned perform Class 2 updates of their CT entries, this time setting their VIDs to be equal to that of an FMG, say T0. The other Processing Elements perform any other normal initialization, idle, or proceed with other predefined actions, but do not participate further in the nucleation process.

Once the three Processing Elements are installed in the CT as an FMG, they perform a Class 1 exchange to themselves to get synchronized, followed by minor bookkeeping. The most important step of the procedure must now be taken. Recall that the simplexes have been performing CT updates to install themselves. This is acceptable at bootstrap time when Byzantine Resilience is not guaranteed, but after the formation of an FMG and during normal operation this capability cannot be allowed lest a faulty simplex totally corrupt the CTs of all Network Elements. To prevent this occurrence, the Network Element aggregate implements a *Higher Life Form Rule*: simplex-sourced Class 2 commands will be obeyed by the Network Element aggregate until a Class 1 command from a higher life form has been processed by the Network Element aggregate. At this point further Class 2 Network Element commands will be ignored. Thus, the next step that the newly formed FMG must perform is to trigger the HLF rule by issuing a Class 1 CT update simply confirming its own formation. Subsequently, the new FMG declares itself the RA and begins issuing CT updates to the other pre-existing simplexes to implement some initial configuration policy.

There still may be simplexes coming on line after the imposition of the HLF rule and these must be integrated into the cluster. Recall that all CT updates are broadcast, including "failed" CT updates such as those issued by post-HLF simplexes. Thus the new FMG must, in addition to forming groups from simplexes already in existence, scoop in the failed CT updates from latecomers and install these latecomers in the CT according to the

configuration scheme. Meanwhile, these latecoming Processing Elements also receive their failed CT updates, perceive that the HLF rule is in effect by examination of status information appended to all packets by the Network Element aggregate, and await a valid CT update from the FMG. The Reconfiguration Authority FMG completes its configuration tasks, awaits the usual acknowledgements, and issues a Relinquish_RA-ship broadcast.

If there are several FMGs but no RA in existence at the time that a latecomer or repairee arrives, all FMGs scoop the failed CT update and eventually contend for RA-ship as described above. Exactly one FMG will become RA and send a valid CT update to the waiting Processing Element. After receiving the acknowledgement from the newcomer, the FMG relinquishes RA-ship as usual.

Chapter 11
Performance Analysis

11.1 Goal and Approach

The goal of this chapter is to quantify the performance of the proposed implementation of Byzantine Resilient Virtual Circuit in terms of the time required for one Fault Masking Group to send a message to another. Specifically, we will compute the mean elapsed time between when a message has been written to the Out FIFO of a source FMG and when the message is ready to be read from the In FIFO(s) of the destination FMG(s), as a function of architectural parameters discussed in earlier chapters and a few new ones introduced in this chapter. Two types of messages may be transmitted by an FMG, intra-cluster messages and inter-cluster messages. Intra-cluster messages may be sent to another FMG in the same cluster as the sender, to all FMGs in the sender's cluster, or to the sending FMG itself. The latter action constitutes a scoop, and awaiting the reception of such a message represents a synchronizing act in the functional synchronization scheme discussed earlier. Thus, the results of this analysis may be used to compare the performance characteristics of functional synchronization to other fault tolerant computer synchronization methods. Inter-cluster messages are forwarded to FMGs in other clusters by the I/O Element Groups (IOEGs) in the sending, forwarding, and destination clusters. They do not represent synchronizing acts in the functional synchronization paradigm but do serve as a measure of the efficiency with which remote FMGs may communicate.

It will be approximated that the FMGs transmit Class 1 messages exclusively. It is expected that the messages dominating normal operation will be the result of normal computations and thus be the same in all non-faulty FMG members, as opposed to messages arising from member-specific data such as simplex inputs or syndrome data. Therefore the latter will be neglected. It is also expected that intra-cluster messages will outnumber inter-cluster messages by a significant margin because of the locality of

reference common in well-partitioned applications. This degree of locality of reference will be used as a parameter in computing the average delay experienced by inter-FMG messages.

Mean message transmission delay data are obtained using a discrete-event simulation of the basic cycle of a single cluster. The primary input parameters to the simulation are the number of FMGs per cluster, the inter-message transmission period of the FMG, the frequency with which the IOEGs perform scoops, and the physical parameters of the cluster's hardware such as intra-cluster bandwidth, message processing time, and inter-cluster bandwidth. A stochastic equilibrium approach is used to determine the traffic densities of inter-cluster messages, which are then used to determine the IOEG loading and hence the inter-cluster message delay. Finally, the mean inter-FMG message transmission delay is computed as the weighted sum of the intra and inter-cluster delay, based on the locality of reference parameter.

11.2 Detailed Description of Basic Cycle

As discussed earlier, the basic Network Element aggregate cycle is composed of several phases. This basic cycle is depicted in Figure 11.1.

In the cycle's first phase, the SERP exchange provides all Network Elements with a consistent view of the message request pattern. For the FTPP under construction at CSDL, the SERP exchange time is

$$\Delta t_{SERP} = 250 \text{ nsec} * \text{the number of PEs per NE} + 250 \text{ nsec} * \text{the number of PEs per cluster}$$

$$= 250 \text{ nsec} * N_{PE/NE} + 250 \text{ nsec} * N_{PE/NE} * N_{NE/CL},$$

varying with the architectural parameters such as the size of the cluster and the distribution of PEs among the Network Elements of the cluster. The first term of Δt_{SERP} is due to the initial broadcast of the LERP packet, and the second term is due to the reflection of the composite SERP packet. Voting and delivery of the SERP packet are overlapped to reduce the time required for the SERP exchange. The SERP exchange time includes the time

required to determine if there is a valid message transmission, which is done online as the SERP packet is output from the voter. If the Network Element aggregate determines in this phase that there is no possibility of an enabled message, another SERP phase is executed immediately. If the aggregate determines otherwise, a further decision phase called the "think" phase is scheduled.

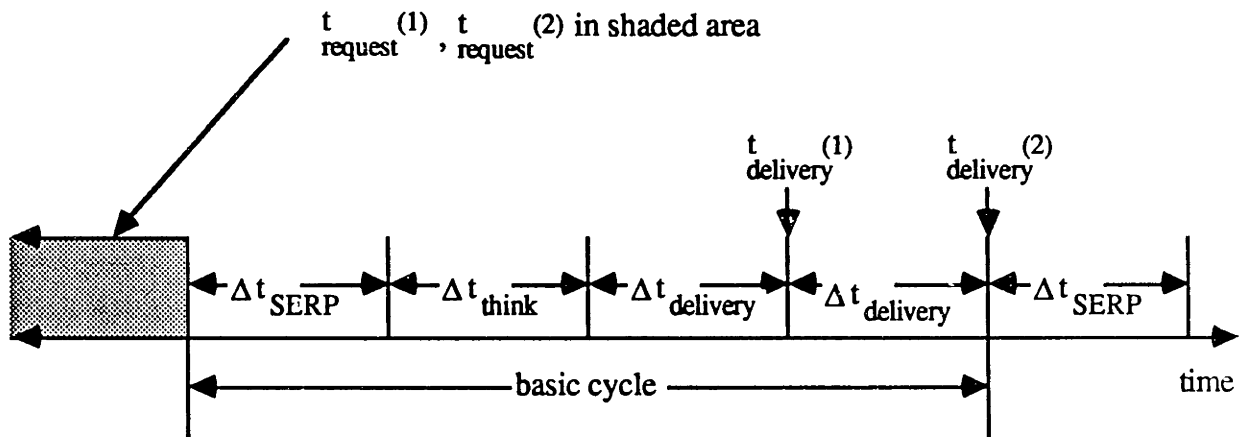


Figure 11.1

Basic Cycle of Cluster

In the think phase of the cycle, each Network Element executes a decision algorithm whose purpose is to determine which messages are to be transmitted on subsequent cycles. This "think time" takes

Δt_{think}=375 nsec * $N_{FE/NE}$,

and transpires only if there is a message to be delivered.

In the think phase, the Network Element aggregate has determined the appropriate number of delivery phases to perform. Since it is assumed that Class 1 messages will dominate operation, the time required to perform each such delivery phase in the FTPP is

$\Delta t_{delivery}$=32 bytes/packet *125 nsec/byte = 4 μsec.

As in the SERP phase, voting and delivery of the message are overlapped for throughput

enhancement. All requests received prior to a given SERP exchange, i.e, in the shaded region in Figure 11.1, are delivered before the next SERP exchange. Thus if there were "i" enabled messages at the beginning of a SERP exchange occurring at t_{SERP}, the jth message to be delivered, $1 \leq j \leq i$, would be delivered at time

$$t_{delivery}(j) = t_{SERP} + \Delta t_{SERP} + \Delta t_{think} + j\Delta t_{delivery},$$

and the next SERP exchange would begin at time

$$t_{SERP} \leftarrow t_{SERP} + \Delta t_{SERP} + \Delta t_{think} + i\Delta t_{delivery},$$

To model the use of functional synchronization, the FMGs are assumed to execute a cyclic behavior in which they compute for a specific amount of time, transmit a message to themselves, await its reception, and resume computation. This message could also be a broadcast to all other FMGs in the cluster since all messages are output from the voters of all Network Elements in the aggregate. In addition to being a measure of the intra-cluster communication delay, the time required for the message to be delivered is a measure of the time required by the FMG to synchronize itself, since FMGs synchronize themselves by sending themselves messages and awaiting their reception. This synchronization overhead can be compared to the equivalent quantity for other fault tolerant computers. The compute time is assumed to follow a Poisson distribution having a mean of t_{FMG_period}. To model the variation in the execution rate and clock nondeterminism of different members of an FMG, a rectangularly distributed uncertainty of $10^{-3}t_{FMG_period}$[1] is applied to the compute times of different members of the same FMG. This uncertainty or skew has the effect of increasing the average delay experienced by a message, because the message transmission requests emanating from different FMG members have a non-zero probability of straddling the beginning of a SERP exchange, causing the transmission to be delayed until the termination of the next SERP.

The IOEGs execute cyclic behavior in which they scoop at a predefined period. This

[1]The drift rate of typical oscillators is on the order of 10^{-4}. The above model assumes a drift rate roughly one order of magnitude above this value.

period is assumed to follow a Poisson distribution having a mean of t_{IOE_period}. At the end of its scoop, a given IOEG determines if it has received any outgoing inter-cluster messages from FMGs in its own cluster or incoming inter-cluster messages from IOEGs in adjacent clusters. The IOEG transmits all such outgoing messages over the inter-cluster links in the order in which they were received. In the present analysis, the inter-cluster links are assumed to have the same bandwidth as the intra-cluster links. For all such incoming messages, the IOEG determines whether the destination cluster has been reached. If so, it performs a Class 1 transmission of that message to the destination FMG. If forwarding of the message is required, the IOEG performs a Class 1 transmission of that message to itself to mask inter-cluster transmission errors, and then forwards the message to the adjacent cluster which is determined to be closer to the destination. The hypercube inter-cluster topology allows fully distributed routing by comparison of the destination cluster identification field with the identification field of the cluster performing the forwarding, followed by transmission of the message to any neighboring cluster whose identification field has a Hamming distance from the destination cluster which is smaller than that of the forwarding cluster [Wittie81].

To determine the traffic intensity supported by the IOEG of a cluster, it is assumed that the ensemble is in stochastic equilibrium, with the mean number of messages being produced by the FMGs equal to the mean number of messages received by the FMGs. We consider only inter-cluster messages, as these are the only ones impacting IOEG loading. (Intra-cluster message density of course impacts IOEG scoop delay through contention effects, which are modelled by the simulation.) Each FMG generates messages at a mean rate of $\lambda_{FMG}=1/t_{FMG_period}$. We assume an intra-cluster reference probability of ϕ, such that each FMG generates inter-cluster messages at a rate of $(1-\phi)\lambda_{FMG}=(1-\phi)/t_{FMG_period}$. Consider a particular cluster possessing $N_{FMG/CL}$ FMGs. The IOEG subscribing to that cluster must support an outgoing message rate of $\lambda_{out}=N_{FMG/CL}(1-\phi)/t_{FMG_period}$. Since stochastic equilibrium holds, this rate must be equal to the rate of incoming messages,

assuming that all clusters are addressed by an inter-cluster message with equal probability. Therefore $\lambda_{in}=\lambda_{out}$. These quantities give the mean rate at which the IOEG of a cluster must service incoming and outgoing inter-cluster messages.

To compute the message forwarding traffic intensity we multiply the probability that a given inter-cluster message is forwarded by the IOEG under consideration by the total number of inter-cluster messages generated per unit time to get the mean number of messages per unit time that an IOEG must forward. For a given message, the expected number of hops that it traverses on the way to its destination is equal to n times the probability that it travels "n" hops to its destination, summed over all values of n less than or equal to the network's diameter. In the binary hypercube inter-cluster network under consideration each cluster possesses a unique identifier which is $\lg N_C$ bits in length. Denote a cluster i's identifier as

$ID_i=\{x_1,x_2,...x_n\}$, where $n\equiv\lg N_C$ and $x_k\in\{0,1\}$, $1\leq k\leq n$.

By construction, any cluster C_i has $\lg(N_C,1)$ neighbors, each of whose identifiers differs from ID_i by one and only one bit. In other words, the identifiers of the neighbors of C_i have a Hamming distance from the identifier of C_i of one. Let C_j be a neighbor of C_i. The neighbors of C_j possess identifiers differing from ID_j by one and only one bit. Each of these neighbors of C_j (not including C_i) possess identifiers which differ from ID_i by exactly two bits and thus have a Hamming distance from ID_i of two. Thus there are $C(\lg N_C,2)$ such clusters two hops away from C_i. Continuing in this manner reveals that in the binary hypercube inter-cluster topology the number of clusters that are "n" hops away from a given cluster is equal to the number of clusters whose Hamming distance is equal to n, such that

clusters at a distance of n hops$=C(\lg(N_C),n)$.

The cluster farthest away from C_i possesses an identifier field which differs in all $\lg(N_C)$ bit positions from ID_i. There is only $C(\lg(N_C),\lg(N_C))=1$ such cluster, and it is $\lg(N_C)$ hops away from C_i. Thus the diameter of the binary hypercube network is $\lg(N_C)$.

To simplify the calculation, assume that FMGs in other clusters are all addressed with equal probability, that is, that there is no extra-cluster locality of reference. Since each cluster has the same number of resident FMGs under nonfaulty conditions, this implies that each cluster is equally likely to be addressed by an inter-cluster message. Therefore the probability that a message is sent n hops is

$$\text{prob(message is sent n hops)}=C(\lg(N_C),n)/(N_C-1),$$

and the expected value of the number of hops traversed by inter-cluster messages is

$$E(\#\text{inter-cluster hops})=\sum_{n=1}^{\lg N_C} n\,\frac{C(\lg N_C,n)}{(N_C-1)}\,.$$

The expected number of message forwardings required by a given message is

$$E(\#\text{inter-cluster forwardings})=E(\#\text{inter-cluster hops})-1.$$

The probability that a given IOEG forwards a given message is equal to the expected number of clusters which forwarded the message divided by the number of clusters which could have forwarded the message, or

$$\text{Prob(a given IOEG forwards a message)}=E(\#\text{inter-cluster forwardings})/(N_C-2), \text{ or}$$

$$\text{Prob(a given IOEG forwards a given message)}=\left\{\sum_{n=1}^{\lg N_C} n\,\frac{C(\lg N_C,n)}{(N_C-1)}-1\right\}/(N_C-2).$$

The expected value of the rate at which a given IOEG forwards inter-cluster messages is equal to the rate of generation of inter-cluster messages, minus the number sourced by that particular IOEG, minus the number delivered by that particular IOEG, times the probability that the given IOEG forwards a given message. The total inter-cluster message generation rate of the ensemble is $N_{FMG}(1-\phi)\lambda$, where N_{FMG} denotes the total number of FMGs in the ensemble. The former product is then

$$N_{FMG}(1-\phi)\lambda_{FMG} - N_{FMG/CL}(1-\phi)\lambda_{FMG} - N_{FMG/CL}(1-\phi)\lambda_{FMG}$$

$$=(N_{CL}-2)N_{FMG/CL}(1-\phi)\lambda_{FMG},$$

and the latter is given above, to yield $\lambda_{forward}$, the mean arrival rate of messages to be

forwarded;

$$\lambda_{forward}=(N_{CL}-2)N_{FMG/CL}(1-\phi)\lambda_{FMG}\left\{\sum_{n=1}^{\lg N_C} n\,\frac{C(\lg N_C,n)}{(N_C-1)}-1\right\}/(N_C-2).$$

The above expressions for λ_{in}, λ_{out}, and $\lambda_{forward}$ will be used in the simulation to be described below to generate message requests for the IOEG. Note that they follow the same distribution as the FMG request rate.

11.3 Simulation Description

The intra-cluster delay is computed using a discrete-event simulation. The simulation begins by generating an event queue of transmission request times $t_{request}(i)$. There is one entry on the event queue for each FMG in the cluster plus one entry for the IOE group for a total number of requests on the event queue of $N_{FMG/CL}+1$. The FMG inter-transmission request times are distributed according to Poisson distribution which has a mean value t_{FMG_period}. This parameter will be varied over a wide range to quantify the effect of contention on the message delay. The IOEG inter-transmission request times are similarly distributed but with a different mean value, t_{IOE_period}. The sensitivity of the average message inter-FMG delay to t_{IOE_period} will be presented. The initial event queue is sorted and the simulation is begun. The event queue can be represented by the ordered sequence

$eq=\{t_{request}(1),t_{request}(2),\ldots,t_{request}(N_{FMG/CL}+1)\}$, where

$t_{request}(i)\leq t_{request}(j)$, $i<j$.

The simulation is iterative. In its first loop, the simulation computes

$t_{SERP}=\lceil t_{request}(1)/\Delta t_{SERP}\rceil$,

the initiation time of the first SERP occurring after $t_{request}(1)$, the first transmission request time on the event queue. Then, for each message transmission request "i" occurring before t_{SERP}, the simulation calculates the delivery time of that message as

$t_{delivery}(i) = t_{SERP} + \Delta t_{SERP} + \Delta t_{think} + i*\Delta t_{delivery}$.

The delay experienced by the transmission requested at time $t_{request}(i)$ is

$$t_{delay}(i)=t_{delivery}(i) - t_{request}(i).$$

This value is added to the delay to be accumulated over the trial and the number of intra-cluster messages delivered by the Network Element aggregate is incremented by one. Then the simulation computes the next transmission request time for that message source as

$$t_{request}(i)= t_{delivery}(i) + \text{Poisson}(t_{FMG_period})$$

if the transmission request emanated from an FMG, and

$$t_{request}(i)= t_{delivery}(i) + \text{Poisson}(t_{IOE_period})$$

if the transmission request emanated from an IOEG, where Poisson(x) denotes a sample from a Poisson distribution having mean x. When all requests i, $t_{request}(i) \le t_{SERP}$, have been serviced and the delay accumulated, the simulation sorts the event queue and, if the requisite number of iterations has not been performed, loops back to perform the next iteration. Let k denote the index in the event queue of the last message to be delivered in a given Network Element cycle. The message corresponding to this index was delivered at $t_{delivery}(k)$. Henceforth, the new SERP time is computed as

$$t_{SERP}=t_{delivery}(k) + \max(\,0, \lceil (t_{request}(1)-t_{delivery}(k))/\Delta t_{SERP} \rceil\,).$$

At the end of the trial, the accumulated intra-cluster delay is divided by the number of message transmission requests serviced in the trial to yield the mean intra-cluster delay, $E(t_{delay})$.

We now describe the calculation of the mean inter-cluster delay. Recall that the IOEG members subscribing to a cluster execute synchronous cyclic behavior in which they periodically exchange their message directories to determine if an inter-cluster message needs to be transmitted, forwarded, or delivered. Depending on the results of this determination, the IOEG performs the appropriate action. Each action will be described in turn.

Figure 11.2 depicts the IOEG's actions upon detection of an outgoing inter-cluster message. The IOEG performs a scoop, then each member investigates its In FIFO for

outgoing messages. If any arrived prior to the completion of the scoop, i.e., in the shaded region in Figure 11.2, the BRVC abstraction guarantees that all IOEG members have received that message and in addition all IOEG members have received the same copy of that message. Furthermore, if more than one message is scooped, the total ordering of the message arrivals is identical at all IOEG members. Therefore each IOEG member can immediately perform the inter-cluster transmission of any outgoing messages received prior to the scoop reception to the appropriate neighboring cluster in the order in which they were received.

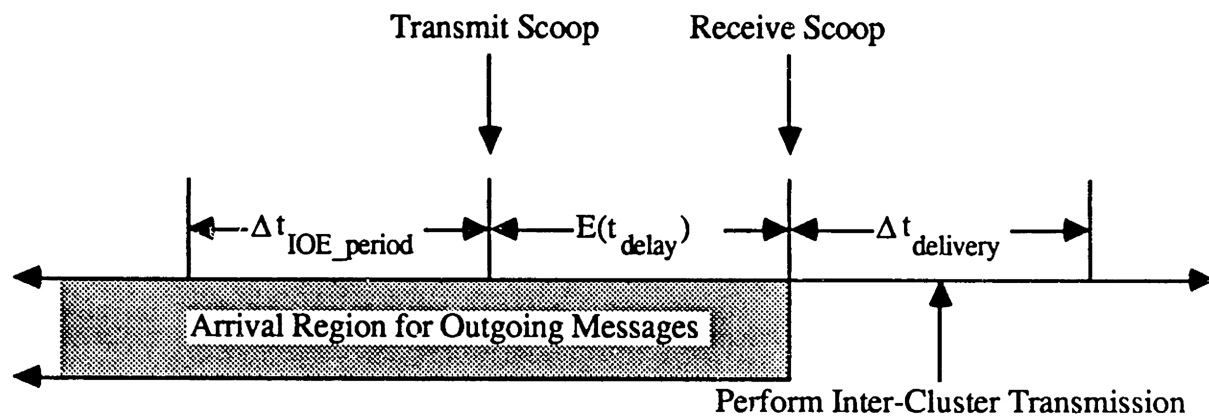


Figure 11.2

Sourcing an Inter-Cluster Message Transmission

Because of the asynchrony between FMGs and the IOEG, outgoing messages may arrive at the IOEG at any time in its cycle. However, they are noticed and transmitted only upon the completion of an IOEG scoop. Therefore the average time required to source an inter-cluster message, Δt_{send}, is a function of the average time in the IOEG cycle that the message arrived at the IOEG, the mean intra-cluster delay required for the IOEG to complete its scoop, the average IOEG cycle period, and the inter-cluster bandwidth. The above IOEG behavior and asynchronous relationship is modeled in the simulation and

Δt_{send} computed as the mean delay between the time at which the source FMG requests a message transmission to the IOEG and the time at which the message is delivered at the IOEG of the adjacent cluster.

Forwarding of an inter-cluster message follows a different protocol. Again, the IOEG performs a scoop containing the contents of each IOEG member's message directory. Upon the reception of the scoop, each IOEG member reads its consistent copy of each other IOEG member's message directory to determine whether a sufficient number of its channels have received inter-cluster messages on its incoming inter-cluster links. This determination is positive if a sufficient number of corresponding messages have arrived on incoming inter-cluster links before the initiation of the directory scoop. This region of time is depicted by the shaded region in Figure 11.3. In the case of a positive determination, the IOEG immediately schedules another scoop whose purpose is to perform a Class 1 vote of the value to be forwarded. Upon reception of this voted value from the Network Element aggregate, each IOEG member forwards its copy of the message to the appropriate neighboring cluster.

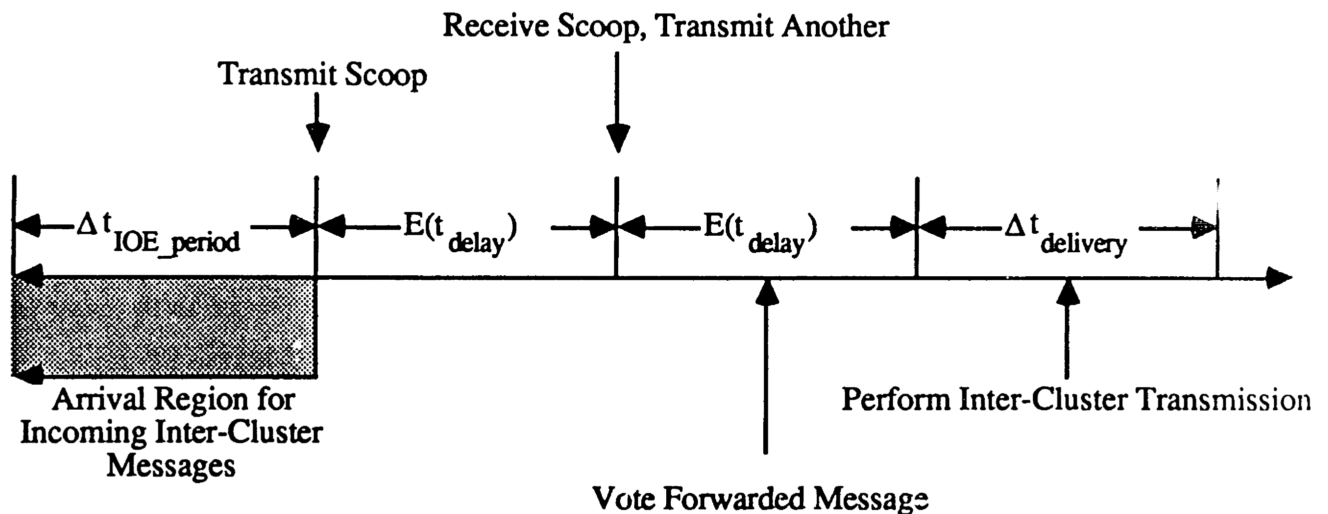


Figure 11.3

Forwarding of an Inter-Cluster Message

Again, because of the totally asynchronous relationship between message arrival at the IOEG and the IOEG's cycle, the mean delay incurred by a forwarded message, $\Delta t_{forward}$, is a function of its mean time of arrival with respect to the cycle of the IOEG performing the forwarding.

If, after investigation of the IOEG members' message directories, the IOEG determines that they have received an incoming message destined for an FMG resident on the IOEG's cluster, they immediately perform a Class 1 transmission to the destination FMG to effect delivery of a validated copy of the message (Figure 11.4). The mean time required to perform this delivery will be denoted by $\Delta t_{deliver}$, and is calculated as the mean elapsed time between the lodging of a message delivery request to the IOEG and delivery of that message to the destination FMG's In FIFO.

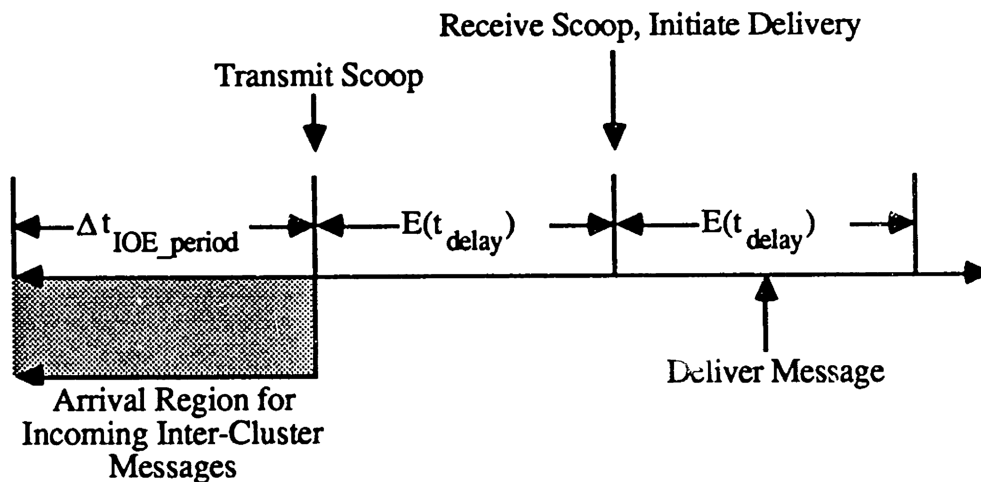


Figure 11.4

Delivery of an Inter-Cluster Message

Using these relationships the mean time required for a message to be delivered to an FMG in a cluster that is "n" clusters away from the source cluster in the inter-cluster network is

$\Delta t_{inter\text{-}cluster}(n) = \Delta t_{send} + (n-1)\Delta t_{forward} + \Delta t_{deliver}$.

We desire to calculate the mean delay for inter-cluster messages. Recall that the probability that an inter-cluster message is sent "n" hops in the binary hypercube inter-cluster network is

prob(message is sent n hops)=$C(\lg(N_C),n)/(N_C-1)$.

Use of this fact gives the result that the expected value of the delay incurred by inter-cluster messages is

$$E(\text{inter-cluster delay})=\sum_{n=1}^{\lg N_C}\frac{C(\lg N_C,n)}{(N_C-1)}\{\Delta t_{send}+(n-1)\Delta t_{forward}+\Delta t_{deliver}\}.$$

We assume that by virtue of locality of reference due to intelligent partitioning of the application an FMG will tend to send messages to other FMGs in its own cluster preferentially to those in other clusters. Let ϕ denote the probability that a message is sent to an FMG in the sending FMG's own cluster. Then $1-\phi$ is the probability that the message is sent to another cluster, and the mean value of the inter-FMG delay is

E(inter-FMG delay)=$\phi E(t_{delay}) + (1-\phi)$E(inter-cluster delay).

11.4 Simulation Results

Figures 11.5 to 11.11 show the results of the simulation for the following cases: 4, 8, 16, 32, and 64 FMGs per cluster, corresponding to 16, 8, 4, 2, and 1 clusters per 64-FMG ensemble, respectively. We show $E(t_{delay})$, the mean intra-cluster delay and E(inter-FMG delay), the average inter-FMG delay as a function of t_{FMG_period} for fixed values of t_{IOE_period} (10 μsec) and ϕ (0.90). Further analyses will display the sensitivity of the mean delays to these latter quantities.

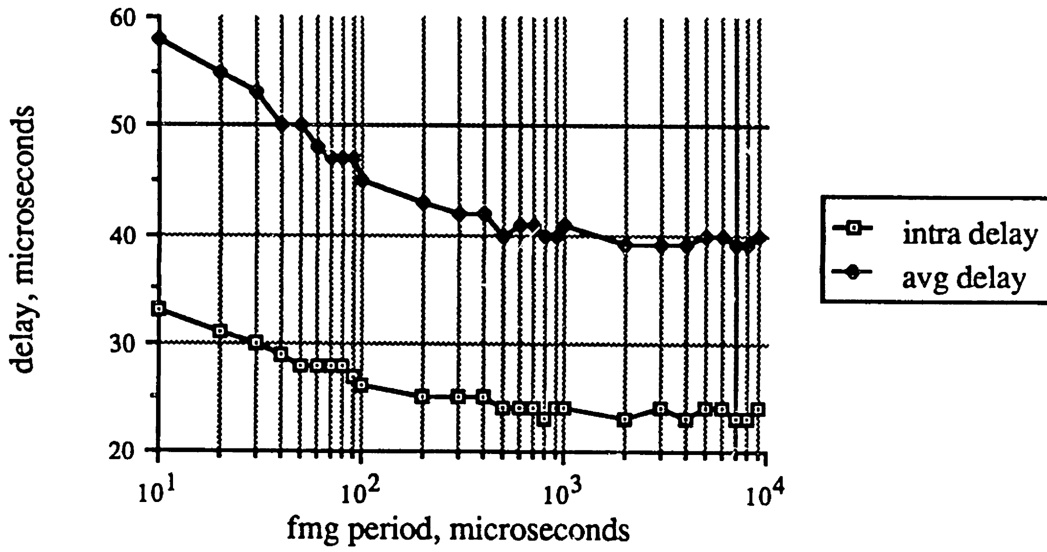


Figure 11.5

Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 4$

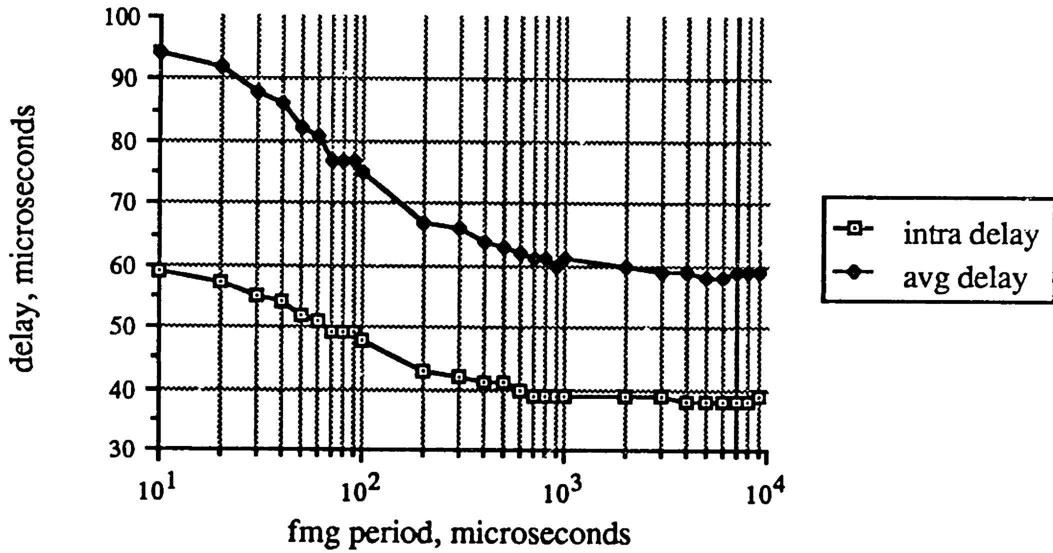


Figure 11.6

Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 8$

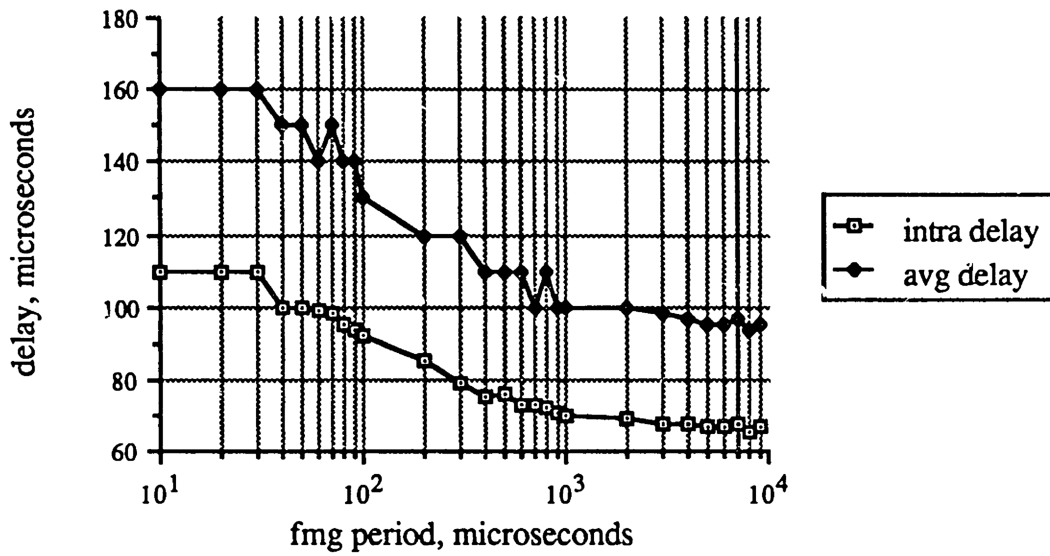


Figure 11.7

Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 16$

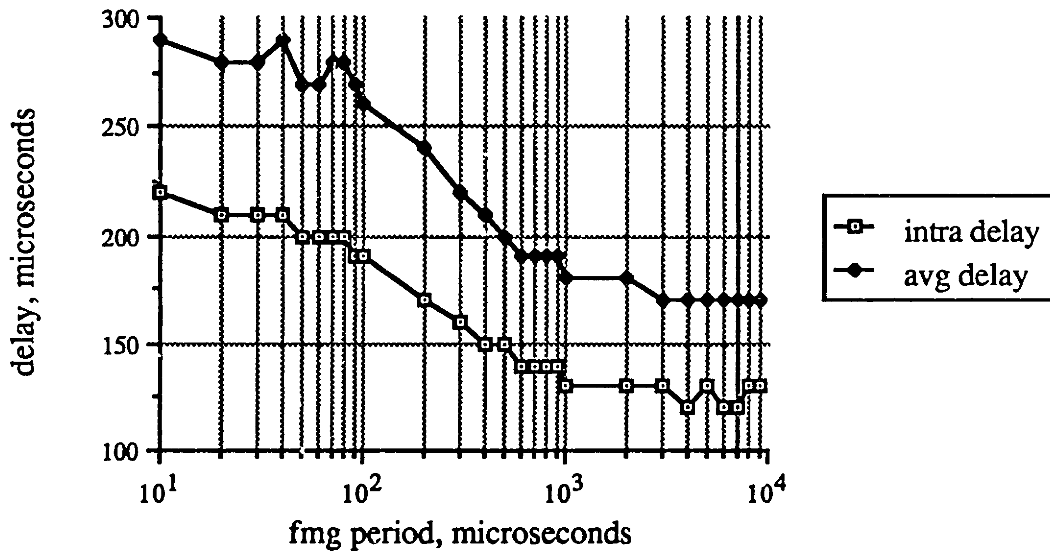


Figure 11.8

Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 32$

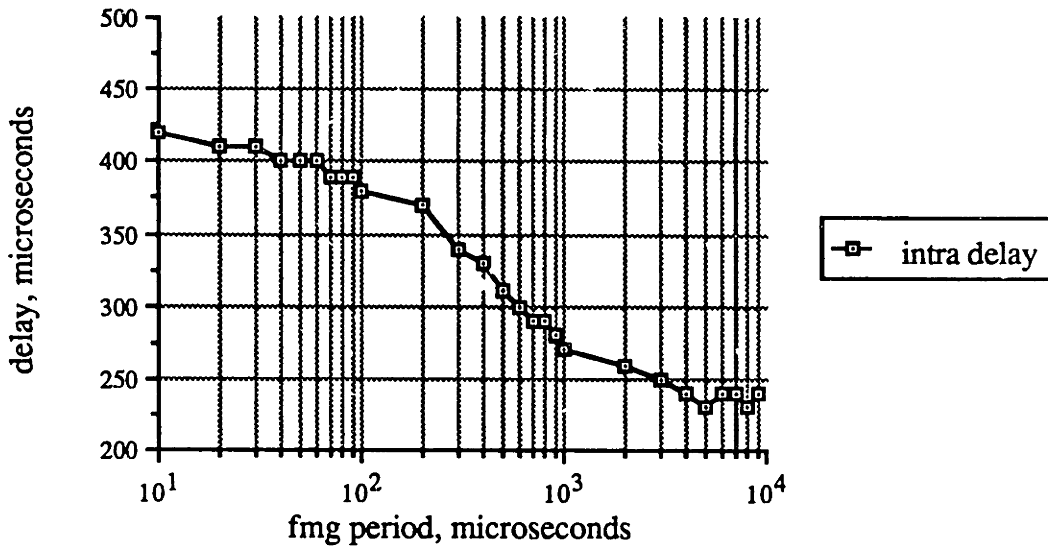


Figure 11.9

Mean Delay vs. t_{FMG_period} for $N_{FMG/CL} = 64$

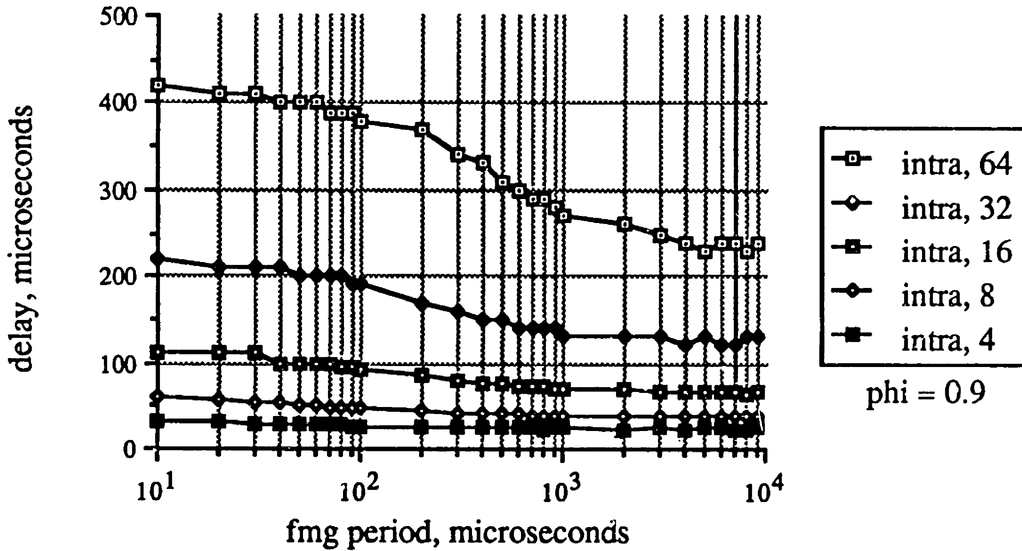


Figure 11.10

Comparison of Mean Intra-Cluster Delay vs. t_{FMG_period} for all Cluster Sizes

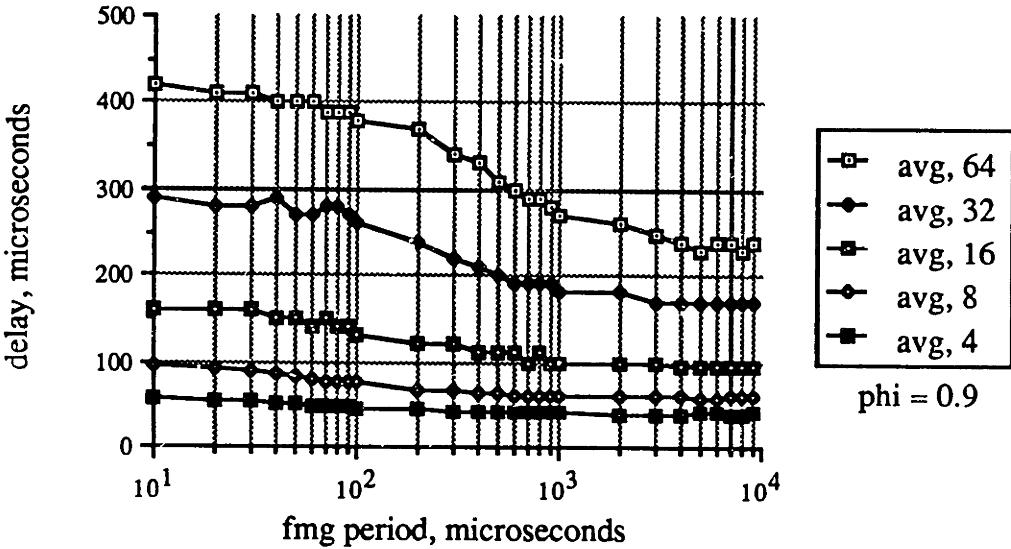


Figure 11.11

Comparison of Mean Inter-FMG Delay vs. t_{FMG_period} for all Cluster Sizes

In such a Monte-Carlo simulation it is necessary that the results depict the steady-state performance of the system being simulated as opposed to containing anomalies due to startup transients. Figures 11.12 through 11.14 show the development of the mean delay times as the simulation progresses. Examination of the figures shows that the mean intra-cluster delay stabilizes after about 300 intra-cluster transmission requests have been serviced, and inter-cluster and mean inter-FMG delay stabilize after about 100 inter-cluster transmission requests have been serviced. If the Monte Carlo runs are long enough so that the number of messages serviced exceeds these values by a significant margin, it can be concluded that the data represent steady state values.

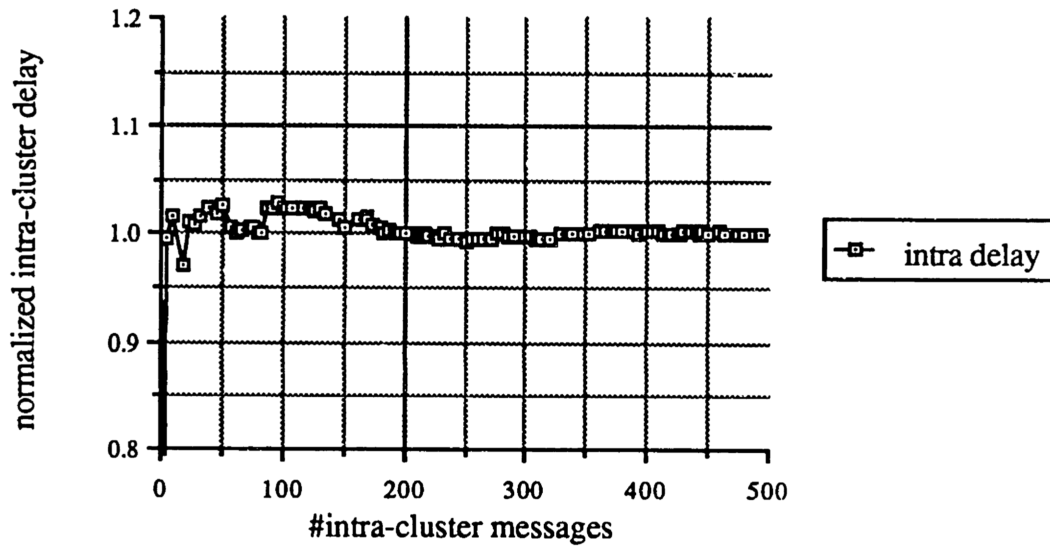


Figure 11.12

Convergence of Mean Intra-Cluster Delay vs. Number of Monte Carlo Trials

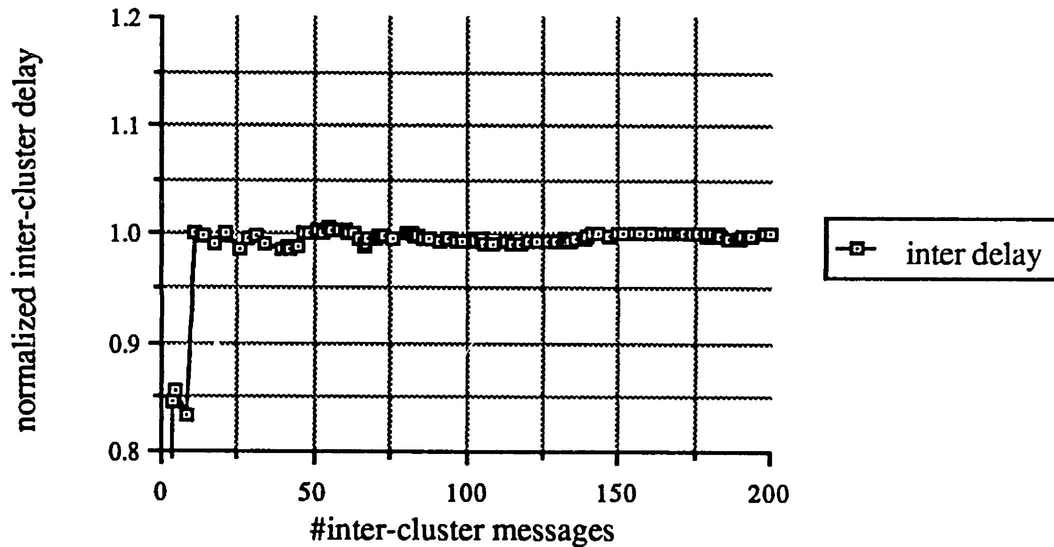


Figure 11.13

Convergence of Mean Inter-Cluster Delay vs. Number of Monte Carlo Trials

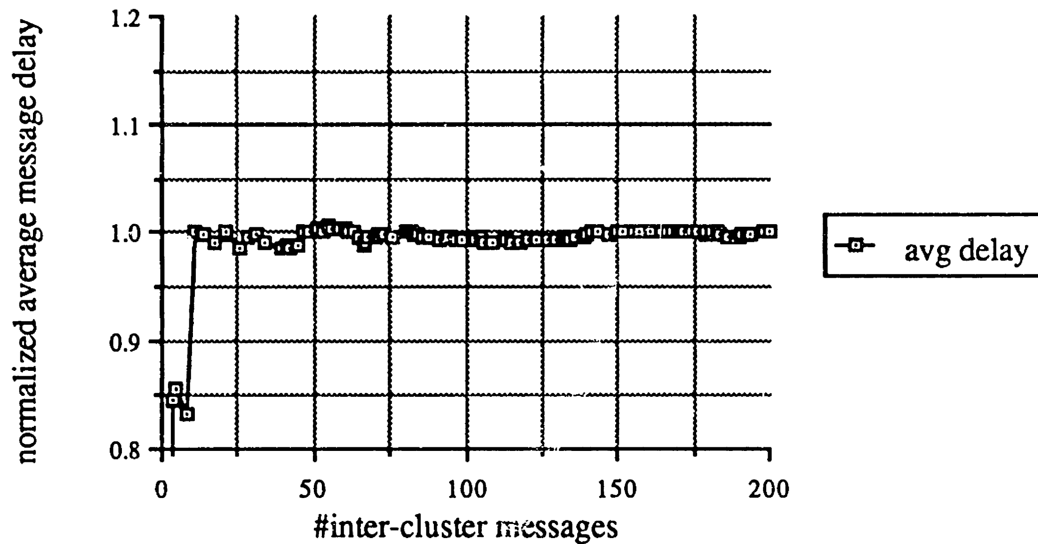


Figure 11.14

Convergence of Mean Inter-FMG Delay vs. Number of Monte Carlo Trials

11.5 Conclusions

Referring to the comparative results pictured in Figures 11.10 and 11.11, several conclusions may be drawn. First, intra-cluster messages in ensembles composed of smaller clusters experience smaller delays than messages in larger clusters. This is a direct result of the linear proportionality of Δt_{SERP} and Δt_{think} to the number of FMGs per cluster, resulting in the unsurprising conclusion that significant architectural and design effort should be expended in increasing the intra-Network Element bandwidth and reducing the message processing time. Note that, even when the average inter-FMG delay is computed, which takes into account the delay in a message transiting from one cluster to another, the smaller clusters (which have a larger mean inter-cluster transit distance) still possess lower average inter-FMG message delays.

Second, at an FMG period of around 1 msec, the contention for the Network Element aggregate has been reduced to the point that the delay is constant and equal to a value easily calculated to be the delay incurred when the FMG's message gains immediate access to the

network. In this "low-contention limit", contention effects are negligible. Note that the low contention limit nevertheless corresponds to a relatively high iteration rate on the part of the FMGs, leading to the hope that the proposed architectural approach might in fact be useful in real-time applications. To obtain anecdotal comparison of the proposed architectural approach to existing approaches, we compare the former's performance parameters to those of two existing fault tolerant computers under comparative operational conditions.

The SIFT computer [Palumbo86] executed an aircraft control function at an iteration rate of about 30 Hz. On each iteration, 18 variables were voted, 21 variables were input from the airframe simulation, and 9 variables were output to the airframe simulation (in the SIFT all words are 16 bits in length). Under these conditions the system overhead attributed to interactive consistency was 39% or about 13 msec, while the time required to vote four data values took a little over 1.6 msec. For rough comparison purposes the simulation developed in this chapter was executed using parameters derived from this SIFT application. The mean FMG iteration period was set to 30 msec, the packet size was set to (18+21+9)*2=76 bytes, and the ensemble was assumed to be composed of 8 clusters having 8 FMGs each. The results from the simulation indicated that the mean intra-cluster message delay was 48 μsec, the mean inter-cluster delay was 290 μsec, and the mean inter-FMG delay was 71 μsec. Depending on the location of the input sources with respect to the message destination(s), the ratio of the time required to obtain data consistency to the mean iteration time of the FMGs ranged from .048msec/(30+.048)msec*100≈0.16% when the information source is in the same cluster as the destination FMG, to .290msec/(30+.290)msec*100≈0.95% when the information source is the average distance away from the destination in the inter-cluster network.

A second point of comparison may be obtained using the requirements for a turbine control application described in [Smith83]. Although the architecture in this thesis clearly was not initially intended for such control applications, their requirements represent a relatively taxing set of iteration and data transfer rates for fault tolerant computers and thus

serve as a useful benchmark. The requisite iteration rate is 100 Hz, with 60 words (16 bits/word) per iteration transferred from the control processor to itself for validation purposes. This data transfer required roughly 10% of the processor's throughput in the implementation described in [Smith83]. The simulation developed in this chapter was used to compare how the FMGs of a cluster fare when confronted with such operational requirements. We modified the simulation to utilize 120 bytes per packet, assumed a 10 msec iteration rate, and performed the simulation for an 8 FMG per cluster 8-cluster architecture. The output from the simulation indicates that the mean intra-cluster delay is 55 μsec, the mean inter-cluster delay is 350 μsec, and the mean inter-FMG delay is 85 μsec. Assuming that the control tasks utilize intra-cluster messages to perform the data input and voting in the turbine control application, the ratio of the time required for such data distribution and validation functions to the total iteration time of the control FMG is $.055\text{msec}/(10+.055)\text{msec}*100 \approx 0.6\%$. If inter-cluster messages are required, which may be the case when the input sensors reside on another cluster, the mean inter-cluster message delay of 350 μsec results in 3.5% of the FMG's cycle time required for input data distribution and validation.

From the above observations it is possible to conclude that there is a high likelihood that such an architecture can indeed operate in the low contention limit. This is fortunate not only in that it minimizes the message delay of any given architecture, but that it also allows us to fix t_{FMG_period} while we vary other parameters such as t_{IOE_period} and ϕ while observing their effect on delay.

It is tempting to select the architecture having the smallest cluster size of 4 FMGs per cluster as the optimal choice for further analysis, as that is the architecture having the lowest message delay. However, referral back to Table 6.2 in Chapter 6 shows that this architecture possesses inferior attrition resiliency compared to that of an architecture having a slightly larger cluster size. An absolute judgement regarding these tradeoffs is not possible to make, but we shall assume that the attrition resiliency of the smallest cluster-

sized architecture is inadequate, and the architecture having 8 FMGs per cluster will be used for further studies.

First we will study the effect of varying the IOEG scoop rate on the message delay. One would expect to observe a reduction in inter-cluster delay as t_{IOE_period} is decreased because inter-cluster messages would wait less time for the IOEG to scoop them and thereby transmit them. This trend would continue until t_{IOE_period} was reduced to the point that inter-cluster message service time was dominated by the SERP, think, and transmission times, at which point further reduction of t_{IOE_period} would have no effect. Also, as t_{IOE_period} is decreased, the intra-cluster delay is expected to increase, because it becomes more likely that an intra-cluster message entering the Network Element aggregate would find it occupied by an IOEG scoop. With the goal of corroborating these ideas, the delay parameters were calculated using the simulation for a variety of IOEG scoop rates. The low contention limit operational regime, an architecture possessing 8 clusters each having 8 FMGs per cluster, and an intra-cluster reference probability of ϕ=0.90 were assumed.

Figure 11.15 shows the results of this study. The mean intra-cluster, inter-cluster, and average message delay are presented. The curves behave as expected. The effect of IOEG period on intra-cluster delay is very slight, while at roughly t_{IOE_period}=20μsec the effect of increasing t_{IOE_period} results in a marked increase in the inter-cluster and average message delay. Also, we observe that reduction of t_{IOE_period} to values below around 10 μsec provides no improvement or degradation in the delay parameters. Therefore we will utilize the value of t_{IOE_period}=10μsec for further studies.

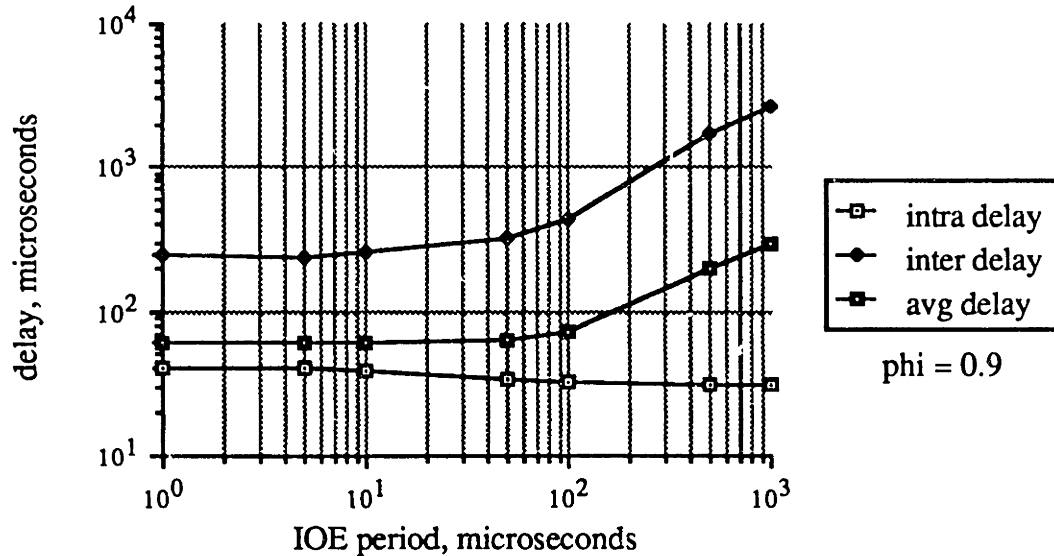


Figure 11.15

Sensitivity of Mean Delay to t_{IOE_period} [2]

As a final study using the simulation, we will fix all other parameters and vary the probability of intra-cluster reference ϕ. This clearly is not an architectural parameter but instead is an attribute of the applications code and to some degree a measure of the cleverness of the algorithm generation process. However, it is interesting to see how sensitive the architecture is to this quantity. One expects the locality of reference probability to have little effect on the intra-cluster delay, because operation is in the low contention limit, and increasing or decreasing the utilization of the Network Element aggregate by increasing or decreasing the rate at which the IOEG's service inter-cluster forwards and deliveries should not impact contention too severely. Where the effect is likely to come into play is in the calculation of average message delay, where one expects a linear relationship to ϕ. Figure 11.16 shows the results of the simulation, which indicate the near-linear relationship of average delay to ϕ, and the insensitivity of the other performance parameters to ϕ.

---

[2] 8 FMG/Cluster, t_{FMG_period} = 1msec.

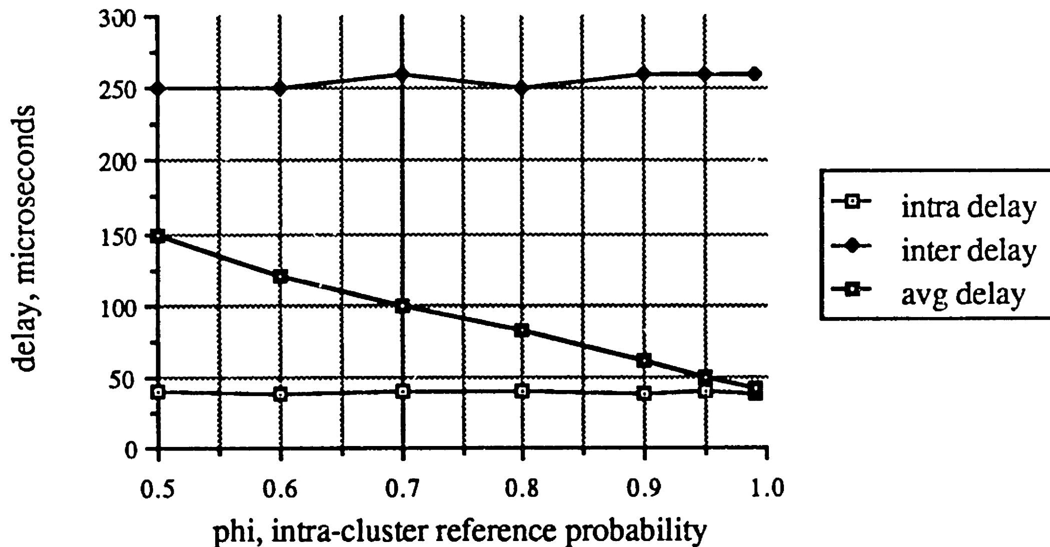


Figure 11.16

Sensitivity of Mean Delay to Locality of Reference Parameter ϕ[3]

Calculation of the effective throughput of a parallel computer is a risky business. Throughput measures (speedup, utilization, effective MIPS, execution time, etc.) depend strongly on the algorithmic structure and how efficiently that structure is mapped onto the hardware. What is commonly done in an attempt to obtain an implementation-independent (but not necessarily useful) indicator of the performance of a computer is to present the raw throughput of the machine. In the case of a parallel ensemble, it is simply the number of Processing Elements times the useful throughput of each PE. As an optimistic overestimate, we will take this approach, and present the raw throughput T_{raw} of an ensemble constructed according to the proposed approach. We have calculated that message transmission delay consumes under 5% of the throughput of an FMG[4]. If a given FMG has an execution rate of 3 MIPS[5] without taking into account a 5% message transmission

[3] 8 FMG/Cluster, t_{FMG_period} = 1msec, t_{IOE_period} = 10μsec.

[4] 8 FMG/Cluster, t_{FMG_period} = 10 msec, t_{IOE_period} = 10 μsec, ϕ = 0.9, 120 bytes/packet.

[5] This is the approximate raw throughput of the Processing Elements in the CSDL FTPP.

delay, then it would have a throughput of 0.95*3=2.85 MIPS if that delay is taken into account. Since we assume that there are 64 FMGs at the beginning of the mission, the raw throughput of the ensemble at the beginning of the mission is T_{raw}=64*2.85=182.4 MIPS.

As a (probably pessimistic) estimate of the average throughput of the the architecture, we will compute the average throughput T_{avg} according to the N/lnN relationship derived in [Hwang84]. Using this technique, the average throughput of the 64-FMG ensemble is T_{avg}=64/(ln64)*2.85= 64/4.159*2.85= 43.9 MIPS. Finally, we can compute the worst-case throughput T_{worst} based on Minsky's lgN conjecture, as T_{worst}=lg64*2.85=6*2.85=17.1 MIPS. These three points are plotted on the performability chart in Figure 12.1. The best case throughput exceeds the target throughput by a comfortable margin, the average throughput falls in the middle of the target range, and the worst case throughput falls at the lower end of the target range. It is likely that the actual throughput obtained from a 64-FMG FTPP would fall somewhere between these values, but much more algorithmic development is required before a more accurate estimate can be made.

Chapter 12
Conclusions and Recommendations for Future Work

12.1 Conclusions

The results of the reliability analyses performed in this thesis demonstrate that a parallel computer comprising fault masking processing groups is necessary to achieve the reliability levels required for life and mission-critical, computationally intensive information processing tasks. A fault masking architecture is also necessary to permit a tractable programming model which avoids the need for checkpoint maintenance and complex software validation. In the course of the analysis, it was found that the short-term ensemble failure probability is proportional to the square of the processor failure rate and linearly proportional to mission time, implying that reductions in processor complexity pay large dividends in short-term reliability. It was also shown that the long-term reliability, as represented by the Mean Time to System Failure, is proportional to the inverse of the processor failure rate and, to a limited extent, responds favorably to the ability to reconfigure the redundant processing groups as failures occur.

Fault masking with near-unity probability presumes the ability to tolerate arbitrary failure behavior of a specified number of components, which in turn hinges on the capability to distribute bitwise identical inputs to the redundant sites in the presence of such faults. Theoretical studies of this well-known "Byzantine Generals Problem" have resulted in demonstrable lower bounds on the number of fault sets required in the computer, the inter-fault set connectivity, the degree of allowable fault set asynchrony, and the amount of inter-fault set information transfer. This thesis identifies the constraints of connectivity and synchronization as the major unsolved problems in bringing Byzantine Resilience to a parallel computer, and presents ways to provide these requisite quantities in an efficient and implementable way.

In this thesis the requisite inter-fault set connectivity is provided to a large number of processors through subscription to a smaller number of Network Elements. Network Elements are low-complexity interfaces to a region of connectivity smaller than full inter-processor connectivity, but which provides enough inter-processor connectivity to allow a degree of reconfigurability in the arrangement of the processors into redundant sites. Using this feature, redundancy management strategies can defer attrition via parallel-hybrid redundancy. In addition, the use of the low-complexity Network Elements as brokers to small regions of connectivity reduces the processor complexity, which includes

communications ports, to its absolute minimum, significantly lowering the ensemble's short-term failure rate and increasing its MTTSF.

The requisite synchronization of redundant processors is provided by defining a new synchronization method called functional synchronization. In functional synchronization, redundant processors utilize naturally-occurring interactions with themselves and the other redundant groups in the ensemble as synchronizing acts. These synchronization points occur as the members of redundant groups transmit messages, read input message buffers, and participate in a number of distributed protocols. They may also be inserted by a programmer, the compiler, or by the operating system. Functional synchronization provides several benefits. The members composing the redundant sites need not obey the homogeneity constraints imposed by previous synchronous fault tolerant computers' synchronization schemes. In particular, clock-nondeterministic processors may be used as members of a redundant group, the use of hardware and software design diversity to overcome generic faults is facilitated, and the integration of heterogeneous processing units into a theoretically sound architecture is made possible. In addition, the redundant groups utilize the shared region of connectivity provided by the Network Elements only as needed for synchronization, further permitting it to be effectively shared by a number of redundant groups.

To shield the programmer from the underlying complexity of the redundant system, the Byzantine Resilient Virtual Circuit (BRVC) abstraction is introduced. This abstraction is a set of Byzantine Resilient guarantees that copies of a message delivered to non-faulty members of a redundant group are identical, received in identical order with respect to other messages, and delivered with a bounded skew at each non-faulty member. In addition, such copies are identical to the message sent by non-faulty members of a source group, and are delivered in the order sent by the non-faulty members of a source group. In the course of developing the operating system and programming model for the Fault Tolerant Parallel Processor, we have found that this abstraction has vastly simplified the development of all the protocols that we have developed to date.

To obtain a quantitative estimate of the penalty incurred in sharing a region of connectivity among several redundant groups, a Monte Carlo calculation of the mean inter-processing group message transmission delay was performed. The results of this calculation show that even under a stressful control application less than 5% of a processor's throughput is spent waiting for completion of inter-group message transmissions. Using this fact, the throughput of a 64-FMG computer was estimated to lie between 17 MIPS (Minsky's lgN conjecture) and 182 MIPS (raw throughput measure).

The architecture developed in the thesis fits on the performability chart in the region depicted in Figure 12.1, which shows that the throughput and reliability goals for the life and mission-critical, computationally intensive tasks can be met using the architectural approach proposed in this thesis.

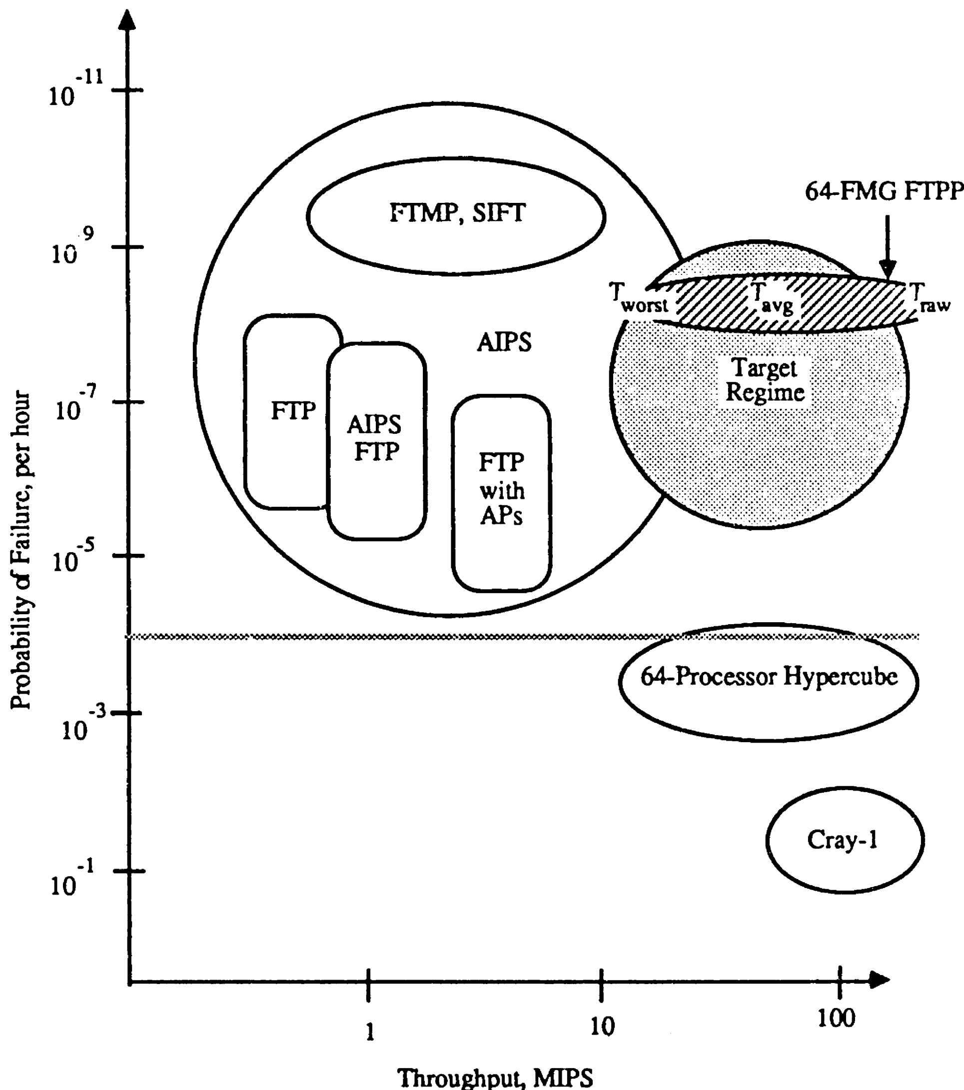


Figure 12.1
Performance and Reliability of Proposed Architecture

12.2 Recommendations for Future Work

Several topics have come up in this research which merit further study. From this set, three have been selected as particularly interesting.

We have begun work on the development of a functional programming model based on Asynchronous Remote Procedure Calls (ARPCs) [Troxel87]. In a functional programming

model, the entire state of computational segments may be described by the arguments, the function, and the destination of the result(s). Saving of this information upon invocation of the function is a way to obtain checkpoints at a low cost. The saving of the information can be totally transparent to the programmer, as can the distributed rollback mechanism, while the referential transparency property of functional computations allows an aborted computation to be restarted on any other processor in the system. The result of a restarted computation will be identical to that of the original computation, merely delayed in time. Use of this programming model makes a Byzantine Resilient duplex processing site possible if the members of the duplex subscribe to the Byzantine Resilient Network Element core. This results in a 33% reduction in the number of PEs needed to obtain a given throughput, while maintaining Byzantine Resiliency. As another advantage, the system can "undo" segments of the calling tree to free resources and thus alleviate deadlock arising from excessive concurrency generation, followed by restarting the execution of the aborted segments when the deadlock condition no longer exists. The programming model also allows work to be transferred to other processors without the accompaniment of large amounts of mutable state, providing the potential for an efficient load balancing method. In addition, because the computational model is insensitive to the number of processors in the system, graceful degradation is obtained as an integral feature of the programming model. We think that the benefits of this programming model make its further study and development worthwhile.

In many applications certain functions and data structures are more critical than others. While the author believes that this thesis presents a viable solution to the problem of physical integration of mixed redundancy such that the lower redundancy level groups cannot corrupt the higher-level groups, how to perform the logical integration of that mixed redundancy is an open question. For example, in terms of the ARPC programming model, one might ask whether a parent RPC can dispatch a child RPC to a group having a lower redundancy level, i.e., a triplex parent to a duplex child. What does the parent do in the event of a child fault? Can a duplex group always ask a triplex group to perform an RPC for it? Upon what basis does the system vary the number of groups of a given redundancy level? Answers to these questions are likely to be application dependent, but a consistent and well defined framework for stating and solving the logical mixed redundancy problem could result in a more efficient use of the resources of the machine by allocating the appropriate resources to a given application segment.

It is the opinion of the author that we have solved the fault masking problem, in that we know how to efficiently meet the theoretically demonstrable requirements for a system that can tolerate arbitrary failure behavior on the part of some of its components. However,

unless faulty components can be purged from the fault masking system as they occur they will eventually overwhelm the non-faulty components. Intuitively, a diagnosis can guarantee that a given fault lies in one of two fault sets, but in the presence of two-faced Byzantine behavior it cannot tell which contains the fault. First, a rigorous proof of this observation is needed. More importantly, we need a theory of diagnosis for Byzantine behavior commensurate with the theory of consensus and synchronization. There are many theories of diagnosis around, but to the author's knowledge all rely on restrictive hypotheses about how a faulty unit may behave, and therefore fall apart in the presence of malicious behavior. A Byzantine systems diagnosis theory would among other things tell us how to construct the fault sets such that external observation and testing of that unit would provide a quantifiable indication of the likelihood that it is faulty, even in the face of malicious behavior. It is possible that such a theory could be grounded in the areas of Boyer-Moore logic [Boyer79], the Logic of Computable Functions-Sequential Logic Machines (LCF-SLM), or some other formal automata description methodology.

References

[Adams86] Adams, M. B., Beaton, R. M., "Automated Mission and Trajectory Planning", Pilot's Associate Planning Conference, Aspen, Colorado, September 16 1986.

[Anderson81] Anderson, T., and Lee., P. A., Fault Tolerance-Principles and Practice, Prentice-Hall International, London, 1981.

[Arlat83] Arlat, J., and Laprie, J. C., "Performance-Related Dependability Evaluation of Supercomputer Systems", *Proc. 13th Int'l Symp. Fault-Tolerant Computing*, June 1983, pp. 276-283.

[Bouricius71] Bouricius, W. G., et al, "Reliability Modeling for Fault-Tolerant Computers", *IEEE Trans. Computers*, Vol. C-20, No. 11, November 1971, pp. 1306-1311.

[Boyer79] Boyer, R. S., Moore, J. S., *ACM Monograph Series: A Computational Logic*, Academic Press, 1979.

[Browning80], Browning, S. A., The Tree Machine: A Highly Concurrent Computing Environment, PhD Thesis, California Institute of Technology, January 1980.

[Cervantes86] Personal communication with Jim Cervantes, developer of the waypoint planner.

[CSDLlore] Many conversations with persons working in the area of fault tolerant systems at CSDL.

[CSDL84] *Advanced Information Processing System (AIPS) System Specification*, prepared for Johnson Space Center by C. S. Draper Laboratory, Cambridge, MA, May 15, 1984.

[Dally86] Dally, W. J., and Seitz, C. L., "The Torus Routing Chip", *Journal of Distributed Computing*, Vol. 1, No. 3, 1986..

[Dennis84] Dennis, J. B., *Data Flow Models of Computation*, Technical Report, Massachussets Institute of Technology Laboratory for Computer Science, 1984.

[Deo80] Deo, N., Pang, C. Y., Lord, R. E., "Two Parallel Algorithms for Shortest Path Problems", IEEE CH1569-3/80/0000-0244.

[Depaula82] Depaula, A., "Evaluation and Reliability Estimation of Distributed Architectures for On-Board Systems", PhD Thesis, Computer Science Department, University of California, Los Angeles, CA, June 1982..

[Dolev82] Dolev, D., "The Byzantine Generals Strike Again", *Journal of Algorithms*, Vol. 3, 1982, pp. 14-30.

[Dolev84] Dolev, D., Dwork, C., Stockmeyer, L., "On the Minimal Synchronism Needed for Distributed Consensus", IBM Research Report RJ 4292 (46990), 5/8/84.

[El-Dessouki80] El-Dessouki, O. I., Huen, W. H., "Distributed Enumeration on Network Computers", *IEEE Trans. Computers*, Vol. C-29, September 1980, pp. 818-825.

[Fischer82] Fischer, M. J., Lynch, N. A., "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, Vol. 14, No. 4, 13 June 1982, pp. 183-186.

[Fischer83] Fischer, M. J., Lynch, N. A., Paterson, M. S., "Impossibility of Distributed Consensus with One Faulty Process", *Proc. 2nd ACM Symp. on Principles of Database Systems*, 1983.

[Flynn66] Flynn, M. J., "Very High Speed Computing Systems", *Proc. IEEE*, Vol. 54, 1966, pp. 1901-1909.

[Friend86] Friend, S. A., "Process Synchronization Within a Loosely Coupled Fault Tolerant Parallel Processing System", MS Thesis, Northeastern University, December 1986.

[Gauthier86] Gauthier, R. J., "The Airlab Fault-Tolerant Processor: Physical Implementation", CSDL Report CSDL-R-1928 prepared for NASA Langley Research Center, December 12, 1986.

[Gjermundsen87] Gjermundsen, E. I., "Reliability Modelling for a Fault Tolerant Parallel Processor", MS Thesis, Massachusetts Institute of Technology, January 1987.

[Gray79] Gray, J. N., "Notes on Database Operating Systems", pp. 393-481, Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[Hopkins78] Hopkins, A. L., Smith, T. B., Lala, J. H., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978.

[Hosseini83] Hosseini, S. H., Kuhl, J. G., and Reddy, S. M., "An Integrated Approach to Error Recovery in Distributed Computing Systems", *Proc. 13th Int'l Symp. Fault-Tolerant Computing*, June 1983, pp. 56-63.

[Houtchens83] Houtchens, S. P., "A Candidate Architecture for an Integrated Avionics System", MS Thesis, Massachusetts Institute of Technology, September 1983.

[Hwang84] Hwang K., Briggs, F. A., Computer Architecture and Parallel Processing, McGraw-Hill, 1984, pp. 27-28.

[Ihara86] Ihara, H., "Autonomous Decentralized System", The Evolution of Fault Tolerant Computing, Avizienis, A., et. al., ed., Baden, Austria, June 30 1986.

[Intel86] Intel iPSC central controller is a good example.

[Jagannathan86] Jagannathan, R., Ashcroft, E. A., "Fault Tolerant Aspects of the Eduction Model and Architecture", IEEE CH2359-8/86/0000-0515.

[Karnin84] Karnin, E. D., "A Parallel Algorithm for the Knapsack Problem", *IEEE Trans. Computers*, Vol. C-33, No. 5, May 1984, pp. 404-408.

[Krishna83] Krishna, C. M., Shin, K. G., "Performance Measures for Multiprocessor Controllers", *Performance '83*, edited by A. K. Agrawala and S. K. Tripathi, North Holland, pp. 229-250, 1983.

[Krishna85] Krishna, C. M., Shin, K. G., Butler, R. W., "Ensuring Fault Tolerance of Phase Locked Clocks", *IEEE Trans. Computers*, Vol. C-34, No. 8, August 1985, pp. 752-756.

[Kuhl86] Kuhl, J. G., and Reddy, S. M., "Fault Tolerance Considerations in Large, Multiple-Processor Systems", *IEEE Computer*, March 1986, pp. 56-67.

[Lala83] Lala, J. H., "Fault Detection, Isolation, and Reconfiguration in FTMP: Methods and Experimental Results", *Proceedings of the IEEE/AIAA 5th Digital Avionics Systems Conference*, 83CH1839-0, Seattle, WA, Nov. 1983.

[Lala86a] Lala, J. H., "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications", 16th Annual International Symposium on Fault Tolerant Computing Systems, Vienna, Austria, 1-4 July 1986.

[Lala86b] Lala, J. H., Alger, L. S., Gauthier, R. J., Dzwonczyk, M. J., "A Fault Tolerant Processor to meet Rigorous Failure Requirements", IEEE/AIAA 7th Digital Avionics Systems Conference, Fort Worth, Texas, October 13-16, 1986.

[Lala86c] Lala, J. H., "Test-Bed Support for Fault-Tolerant Computer Evaluation", CSDL-R-1900, prepared for The System Development Corporation, September, 1986.

[Lin86] Lin, F. C. H., Keller, R. M., "Distributed Recovery in Applicative Systems", IEEE 0190-3918/86/0000/0405.

[Lin65] Lin, S., "Computer Solutions for the Traveling Salesman Problem", Bell Systems Technical Journal, 44(1965), pp.2245-2269.

[Martin78] Martin, D. L., Gangsaas, D., "Testing of the YC-14 Flight Control System Software", *AIAA Journal of Guidance and Control*, Vol. 1, No. 4, July-August 1978.

[Mohan83] Mohan, J., "Experience with Two Parallel Programs Solving the Traveling Salesman Problem", IEEE 0190-3918/83/0000-0191.

[Palumbo86] Palumbo, D. L., Butler, R. W., "A Performance Evaluation of the Software-Implemented Fault-Tolerance Computer", *J. Guidance*, Vol. 9, No. 2, March-April 1986, pp. 175-180.

[Pease80] Pease, M., Shostak, R., Lamport, L., "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, Vol. 27, No. 2, April 1980, pp. 228-234.

[Pierce80] Pierce, J. R., An Introduction to Information Theory: Symbols, Signals and Noise, Dover Publications, New York, 1980.

[Preparata67] Preparata, F. P., Metze, G., Chien, R. T., "On the Connection Assignment Problem of Diagnosable Systems", *IEEE Trans. Electron. Comput.*, Vol. EC-16, December 1967, pp. 848-854.

[Reed83] Reed, D. A., Schwetman, H. D., "Cost-Performance Bounds for Multimicrocomputer Networks", *IEEE Trans. Computers*, Vol. C-32, No. 1, January 1 1983, pp. 83-95.

[Rennels 84] Rennels, D. A., "Fault Tolerant Computing - Concepts and Examples", *IEEE Trans Computers*, Vol. C-33, No. 12, December 1984, pp. 1116-1129.

[Rosch84] Rosch, G., Schabowsky, R., Weinstein, W., Harper, R., "Architecture and Reliability Evaluations for the GE/CE2 Fault-Tolerant Engine Controller", CSDL Technical Report CSDL-R-1734, August 1984.

[Schneider86] Schneider, F. B., "Abstractions in Fault Tolerance in Distributed Systems", Proc. 10th World Computer Congress, (IFIP Congress '86), Dublin, Ireland, September 1986.

[Seitz85] Seitz, C. L., "The Cosmic Cube", *Comm. ACM*, 28:1, January 1985, pp. 22-33.

[Sherron82] Sherron, R. C., Savage, C., Ashtaputre, S., "Graph Search with a Systolic Multiqueue", IEEE CH1813-5 82 0000 0407.

[Shin84] Shin, K. G., Lee, Y-H., "Error Detection Process-Model, Design, and Its Impact on Performance", *IEEE Trans. Comp.*, Vol. C-33, No. 6, June 1984, pp.529-540.

[Shin86] Shin, K. G., Ramanathan, P., "Transmission Delays in Hardware Clock Synchronization", June 18, 1986.

[Siewiorek82] Siewiorek, D. P., and Swarz, R. S., The Theory and Practice of Reliable System Design, Digital Press, Bedford, MA, 1982.

[Smith83] Smith, T. B., "Fault Tolerant Processor Concepts and Operation", CSDL-P-1727, May 1, 1983.

[Sundstrom74] Sundstrom, R. J., "On-Line Diagnosis of Sequential Systems", PhD Thesis, University of Michigan, 1974.

[Tamir84] Tamir, Y., Sequin, C. H., "Error Recovery in Multicomputers Using Global Checkpoints", *Proc. 1984 Int'l Conf. Parallel Processing*, Aug. 1984, pp. 32-41.

[Tanenbaum81] Tanenbaum, A., Computer Networks, Prentice Hall, New Jersey, 1981.

[Troxel87] Troxel, G. D., "Detection of and Recovery from Deadlock in a System using Remote Procedure Calls", BS Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June, 1987.

[Tsai82] Tsai, W. H., "Graph Matching Problems: A Survey and Tutorial", *Proceedings of First Conference on Computer Algorithms*, Hsinchu, Taiwan 300, Republic of China, July 1982.

[Twain85] Twain, M., The Adventures of Huckleberry Finn, 1885.

[von Neumann56] von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", Automata Studies, 1956.

[Wah84], Wah, B. W., Ma, Y. W. E., "MANIP- A Multicomputer Architecture for Solving Combinatorial Extremum Search Problems", *IEEE Trans Computers*, Vol. C-33, May 1984, pp. 377-390.

[Walter85] Walter, C. J., Kieckhafer, R. M., Finn, A. M., "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems", *Proc. IEEE Real Time Systems Symposium*, Dec 1985.

[Wensley78] Wensley, J., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proc IEEE*, Vol. 66, Oct 1978, pp. 1240-1255.

[Wittie81] Wittie, L. D., "Communication Structures for Large Multimicrocomputer Systems", *IEEE Trans. Computers*, vol. C-30, no. 4, pp. 264-273, April 1981.

[Wulf81] Wulf, W. A., "Compilers and Computer Architecture", *IEEE Computer*, July 1981, pp. 41-47.

Appendices

Appendix A Ensemble Loss Curves for Network Element-Based Clusters..... A-1

Appendix B Relative Complexities of Processing, Network, and IO Elements B-1

Appendix A

System Loss Probability Curves for Network Element-Based Clusters

Caption : "Data from "$N_{NE/C}$ $N_{PE/NE}$ N_C data" ", where

$N_{NE/C}$ = number of NEs per cluster;

$N_{PE/NE}$ = number of PEs per NE;

N_C = number of clusters in ensemble.

Legend:

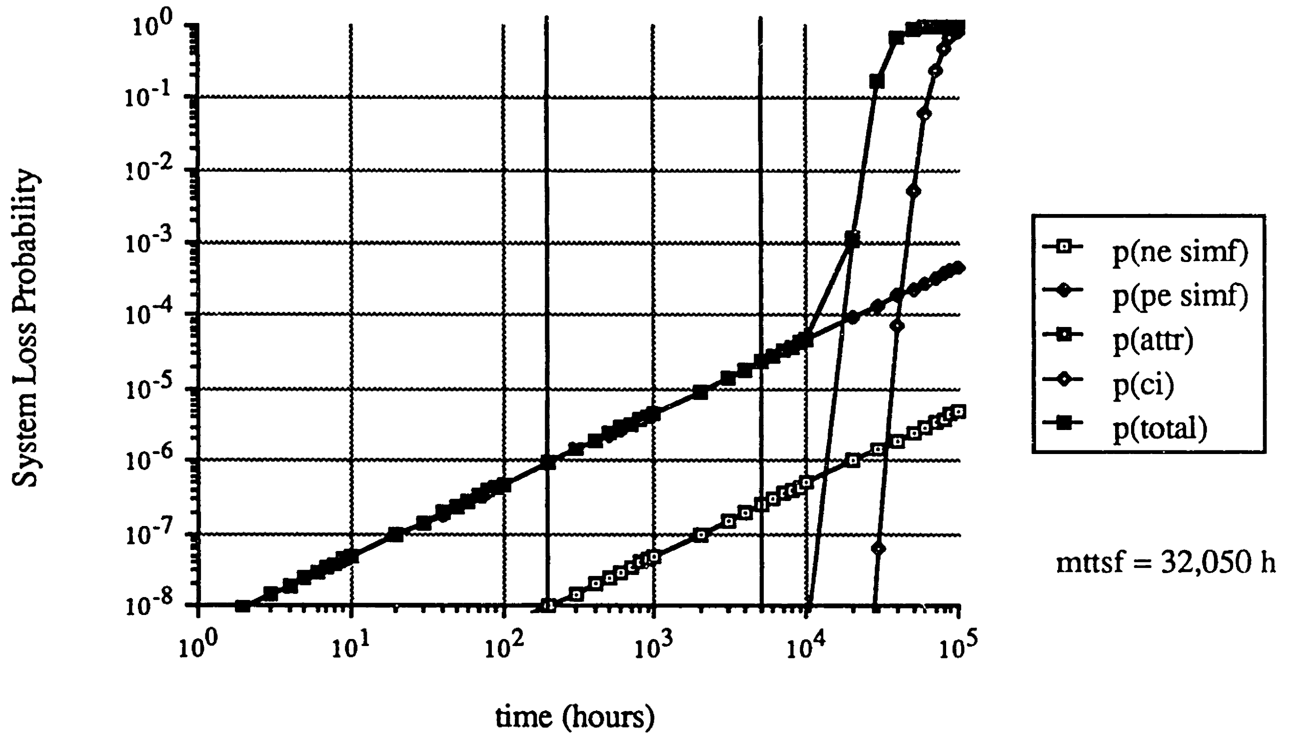
p(ne simf) = probability of ensemble loss due to near-simultaneous Network Element failures;

p(pe simf) = probability of ensemble loss due to near-simultaneous Processing Element failures;

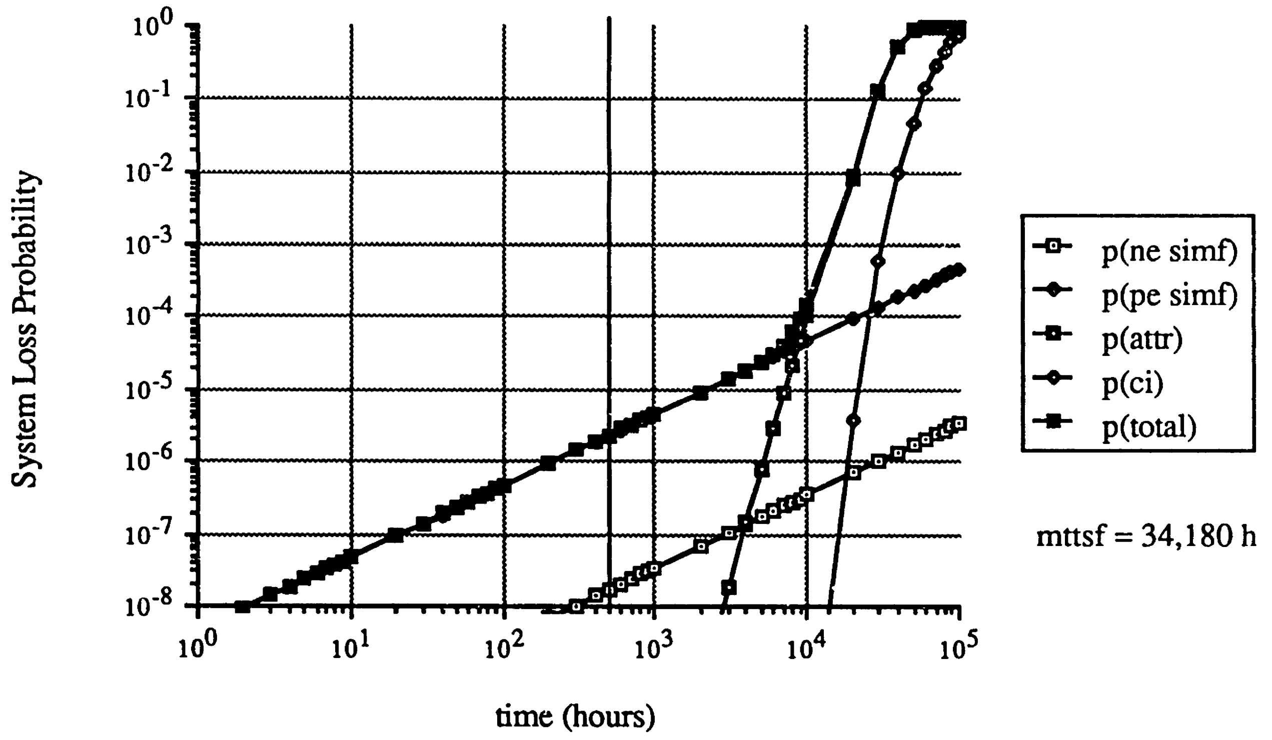
p(attr) = probability of ensemble loss due to Network Element/Processing Element attrition;

p(ci) = probability of ensemble loss due to isolation of all clusters;

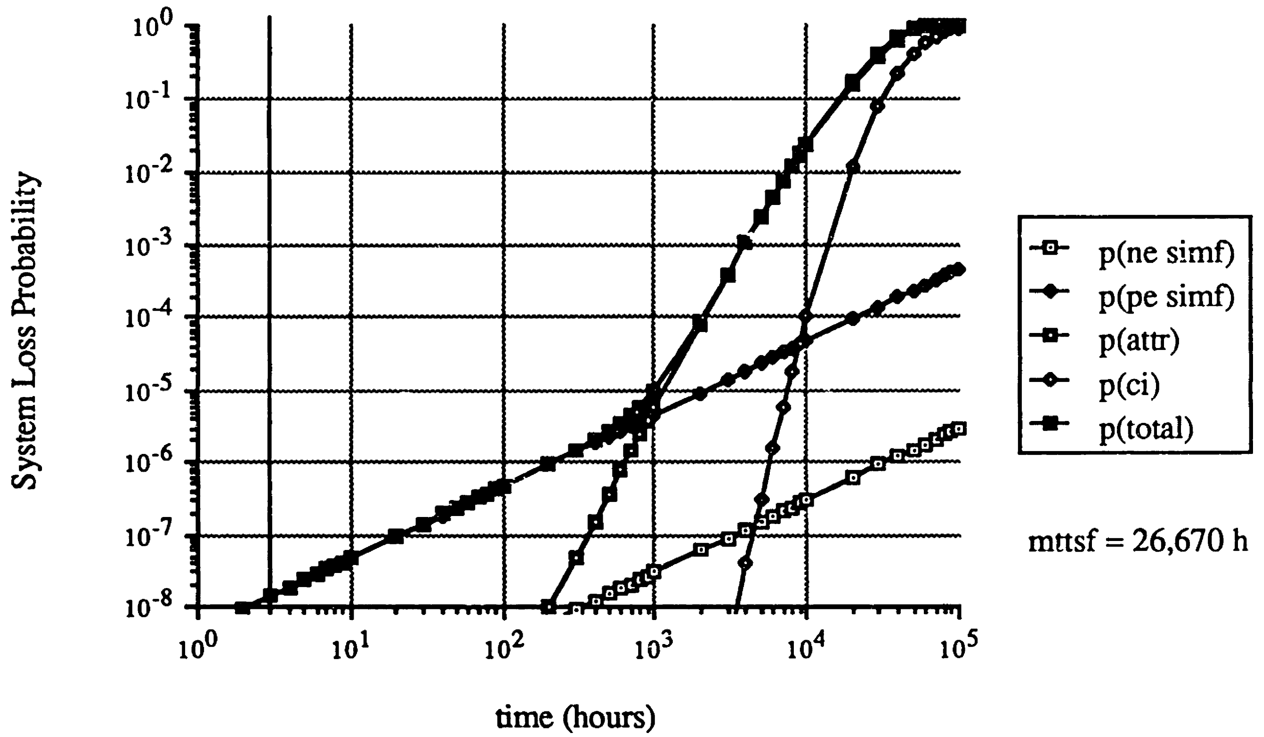
p(total) = total probability of ensemble loss.



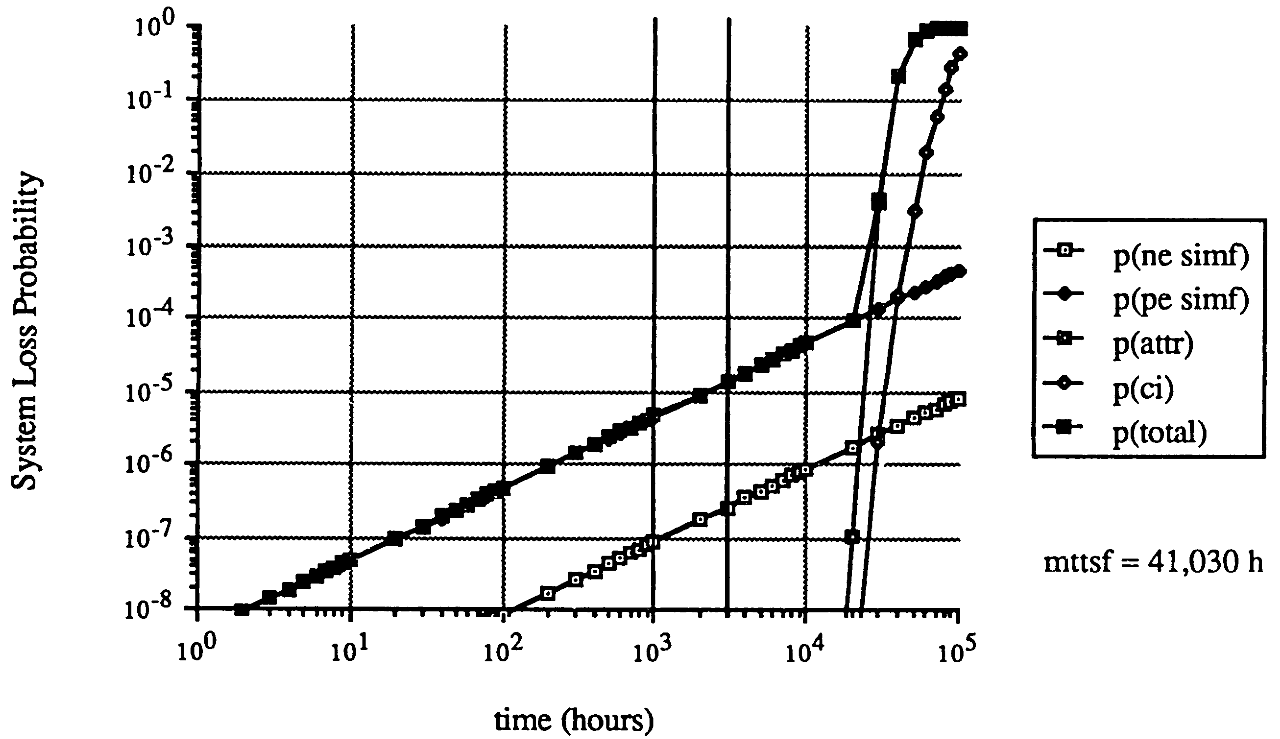
Data from "4 3 16 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 32,050 h



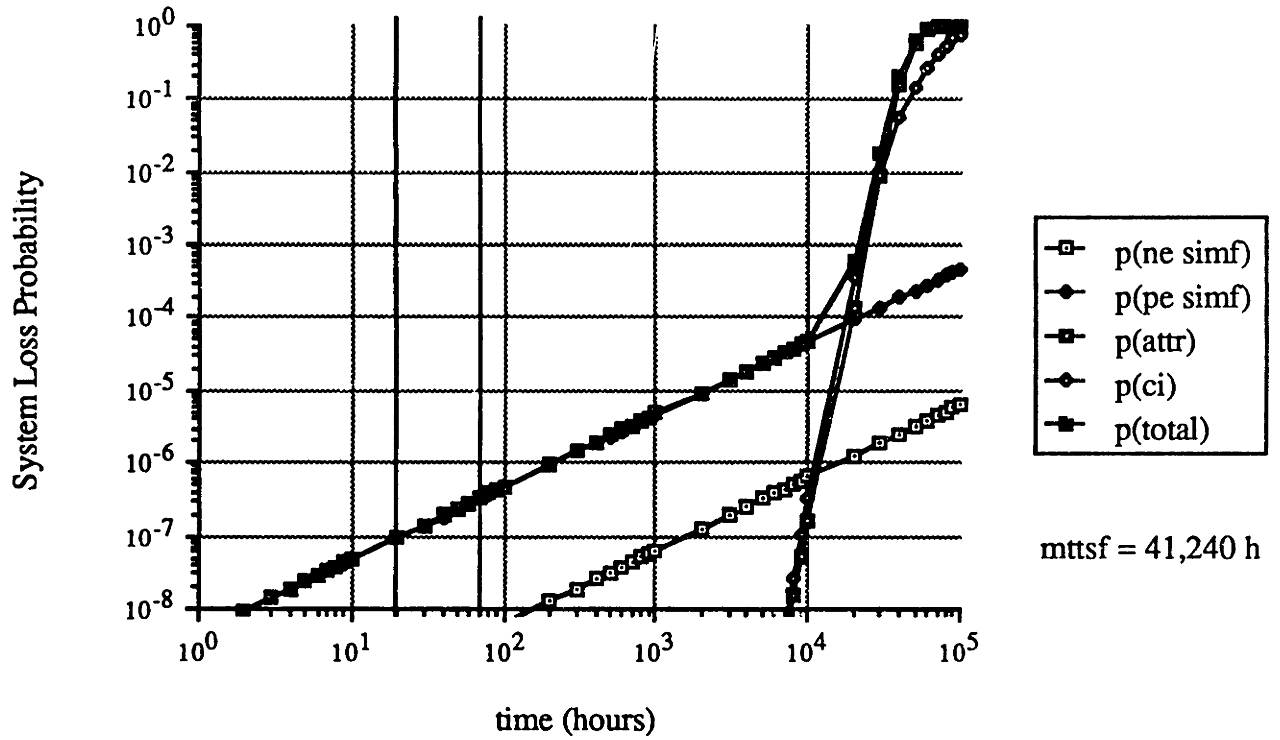
Data from "4 6 8 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 34,180 h

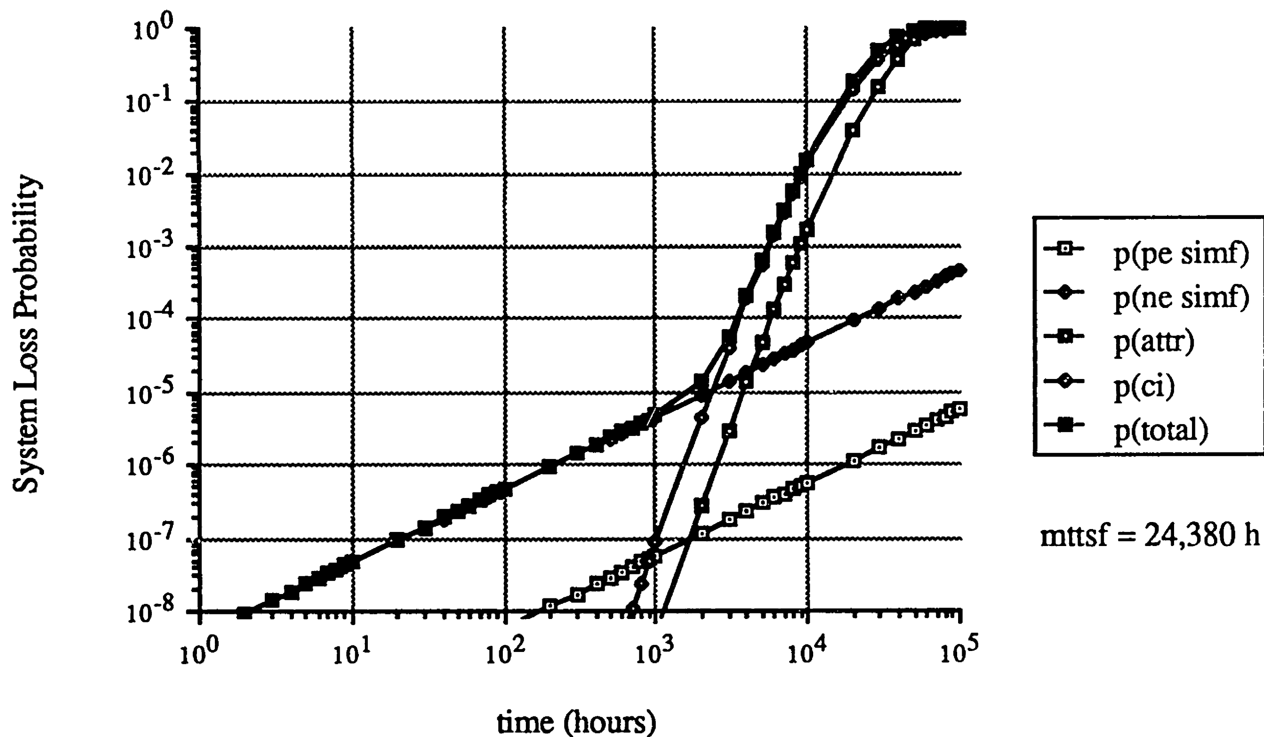


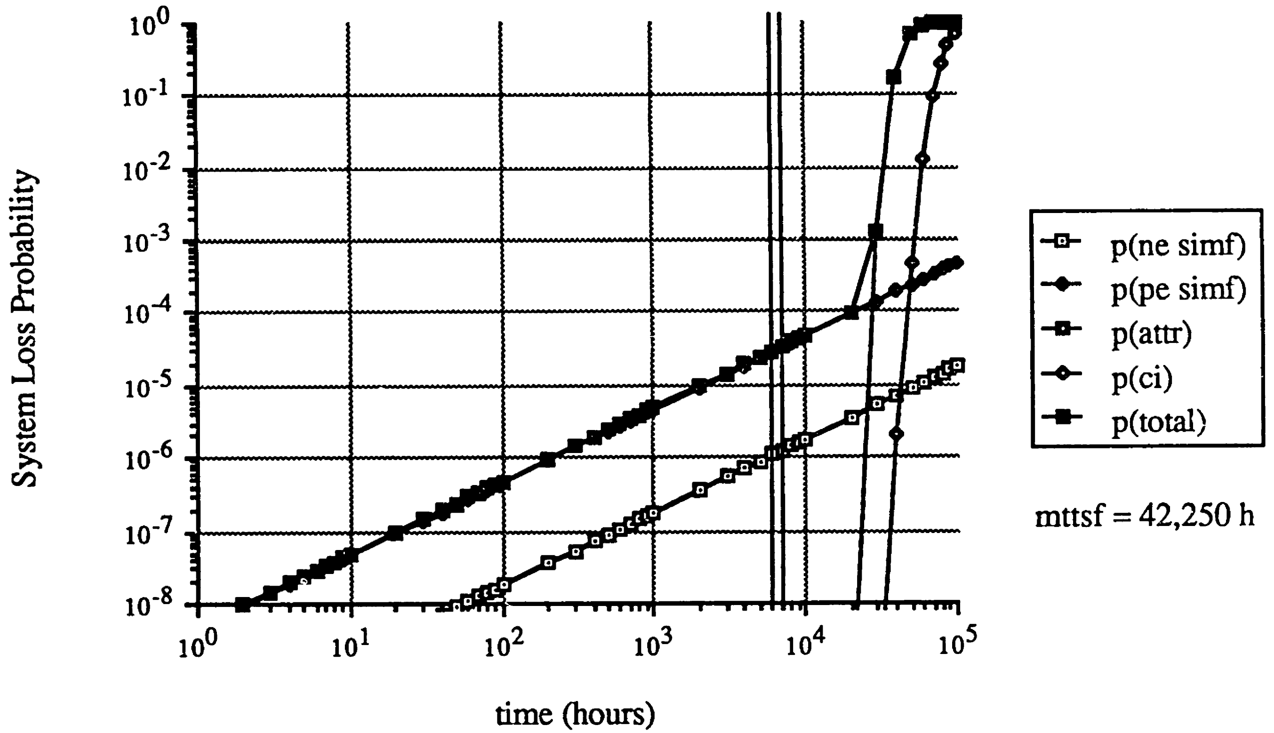
Data from "4 12 4 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 26,670 h

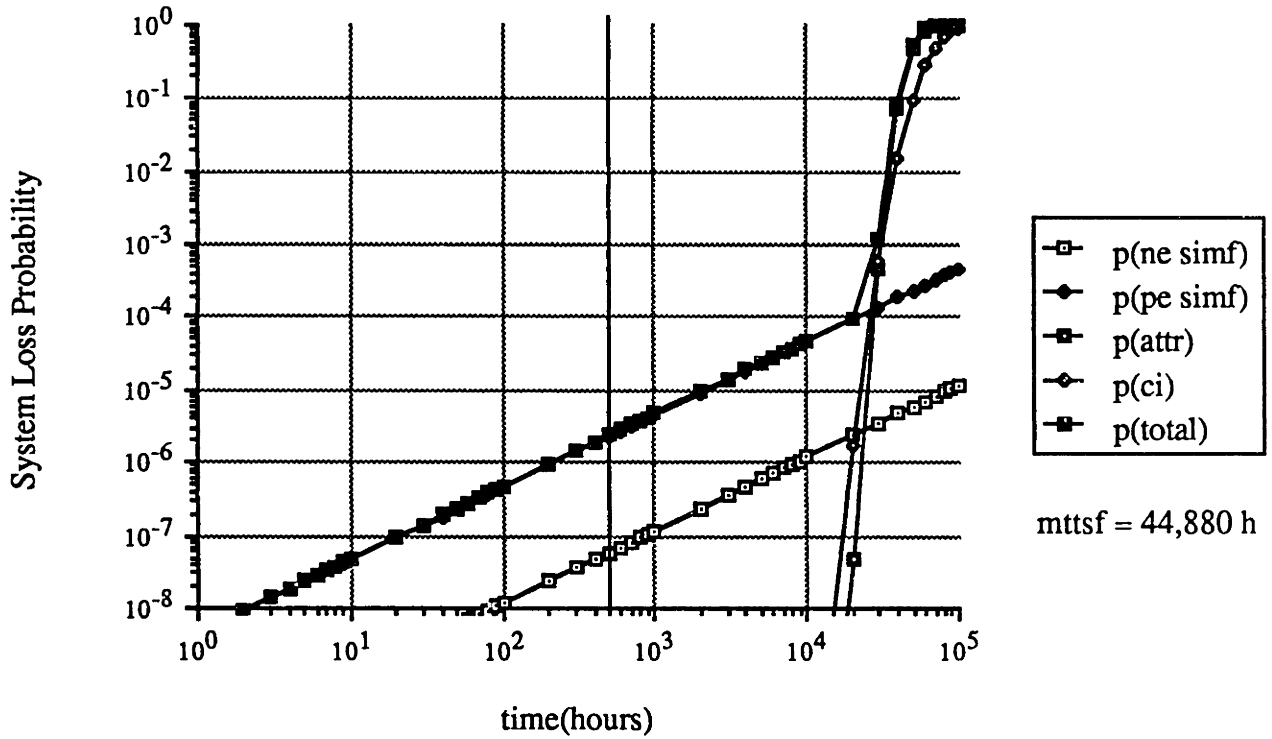


Data from "6 4 8 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 41,030 h

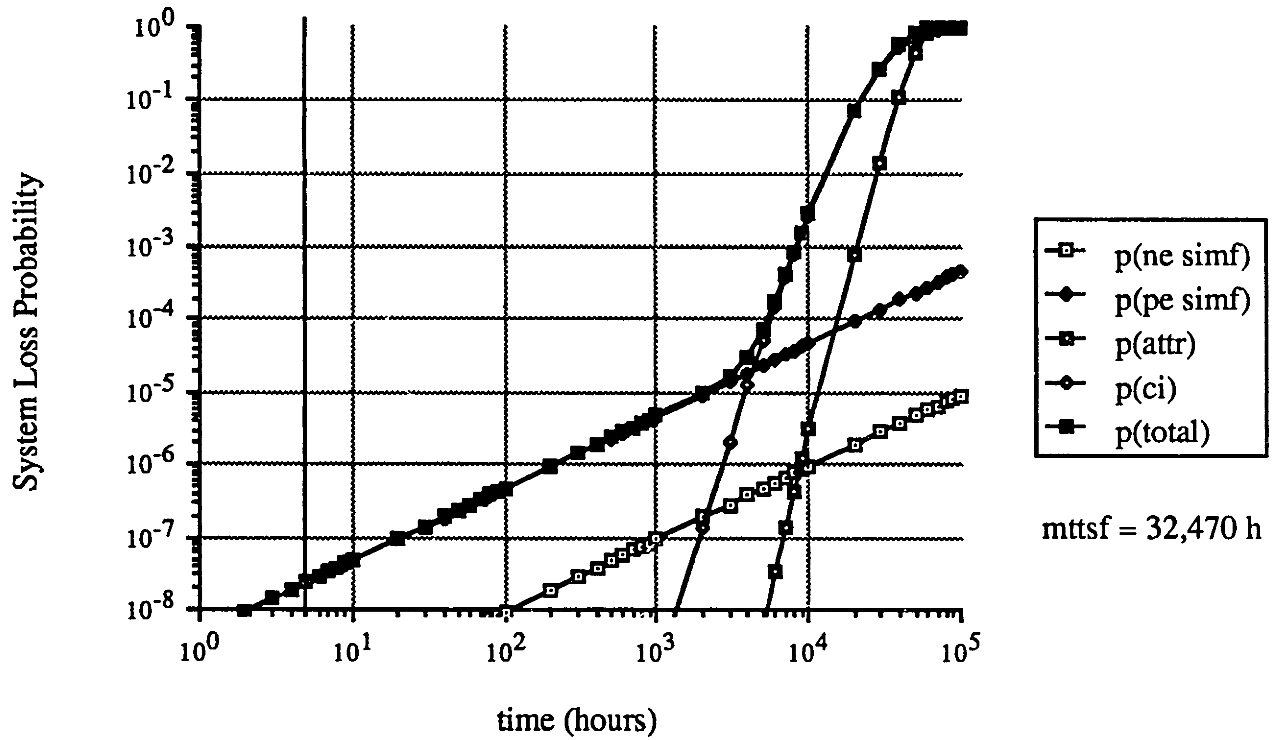

Data from "6 8 4 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 41,240 h


Data from "6 16 2 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(pe simf)
p(ne simf)
p(attr)
p(ci)
p(total)
mttsf = 24,380 h

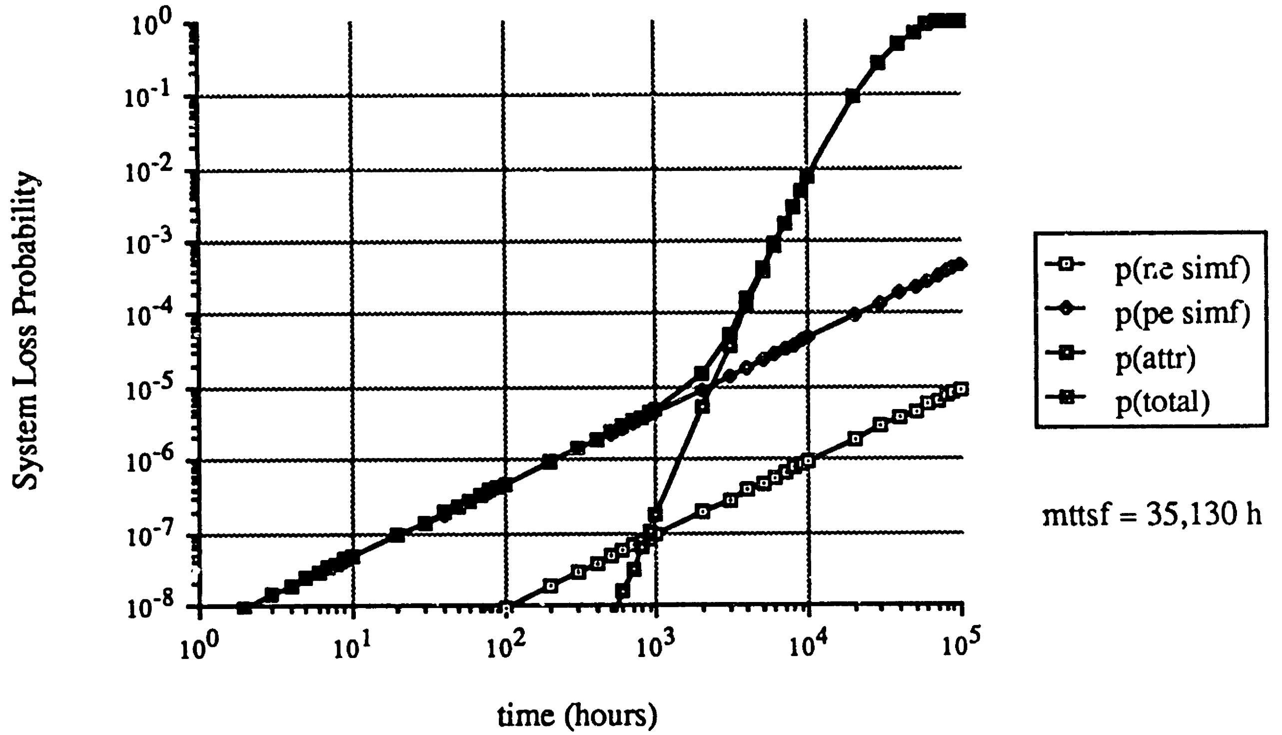

Data from "8 3 8 data ez"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 42,250 h



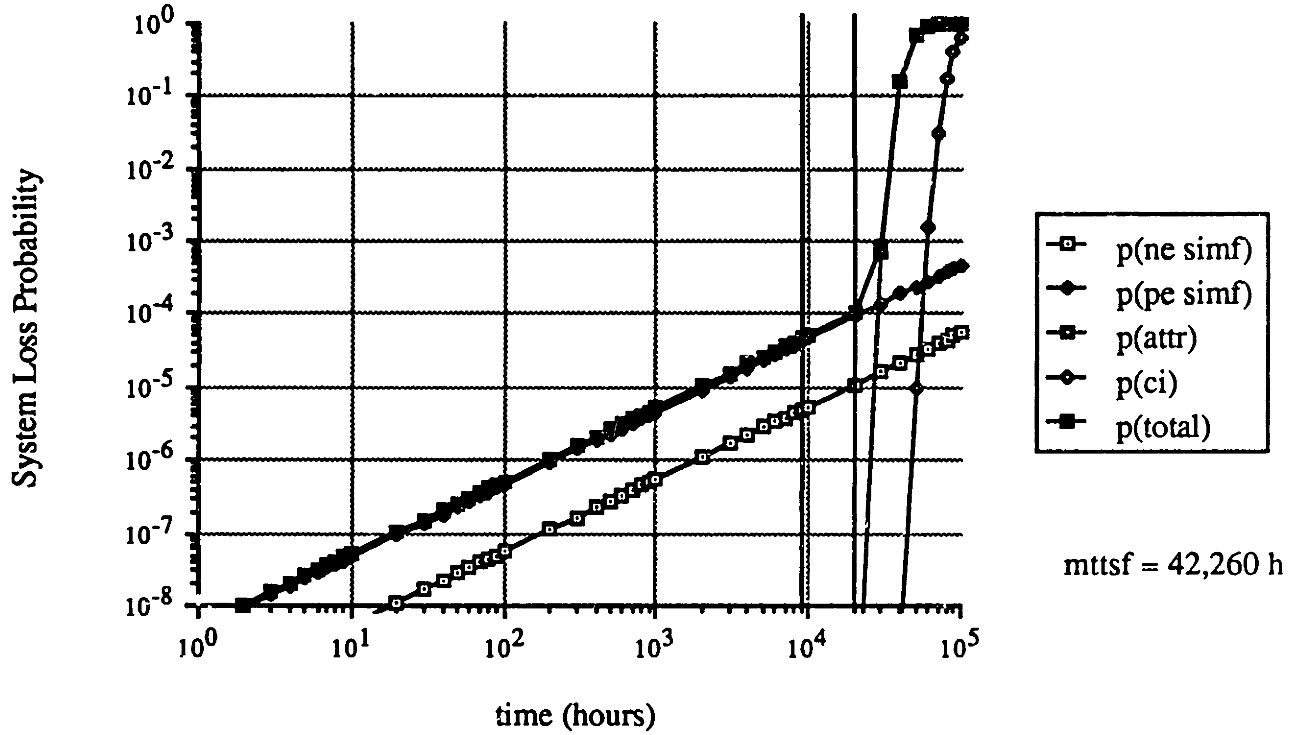
Data from "8 6 4 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time(hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 44,880 h

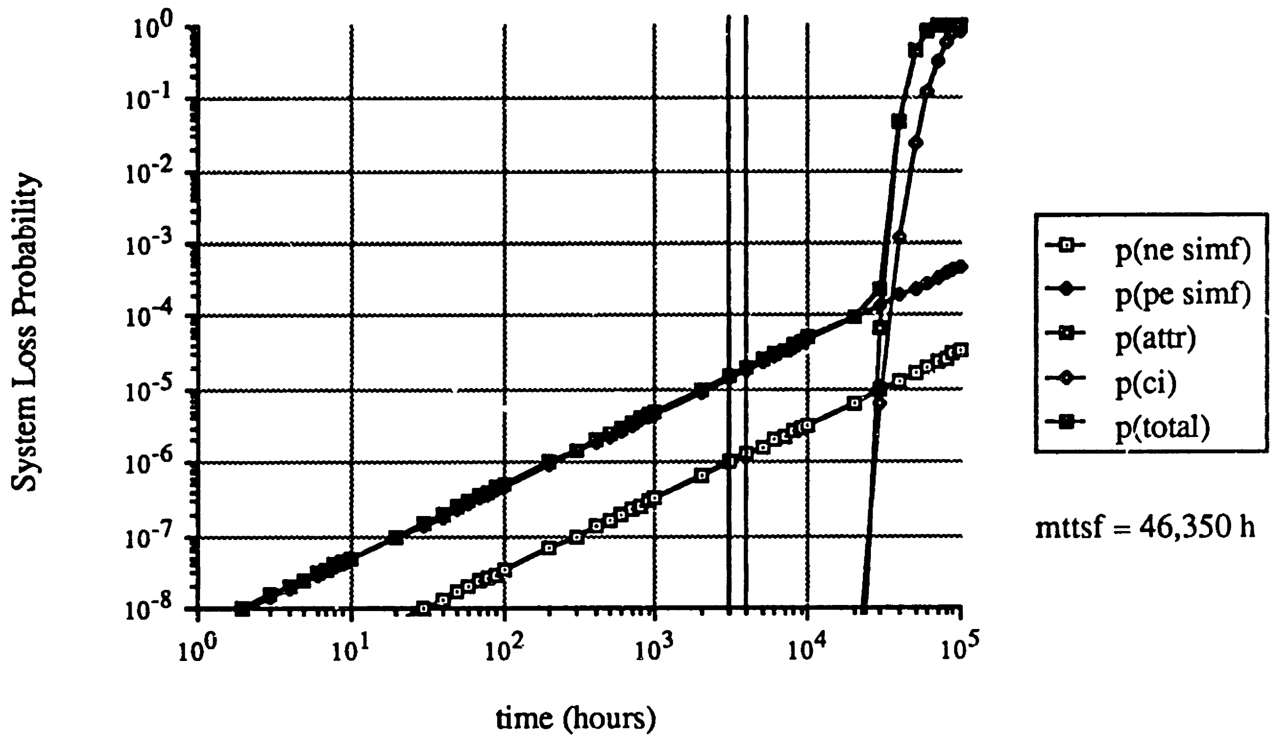


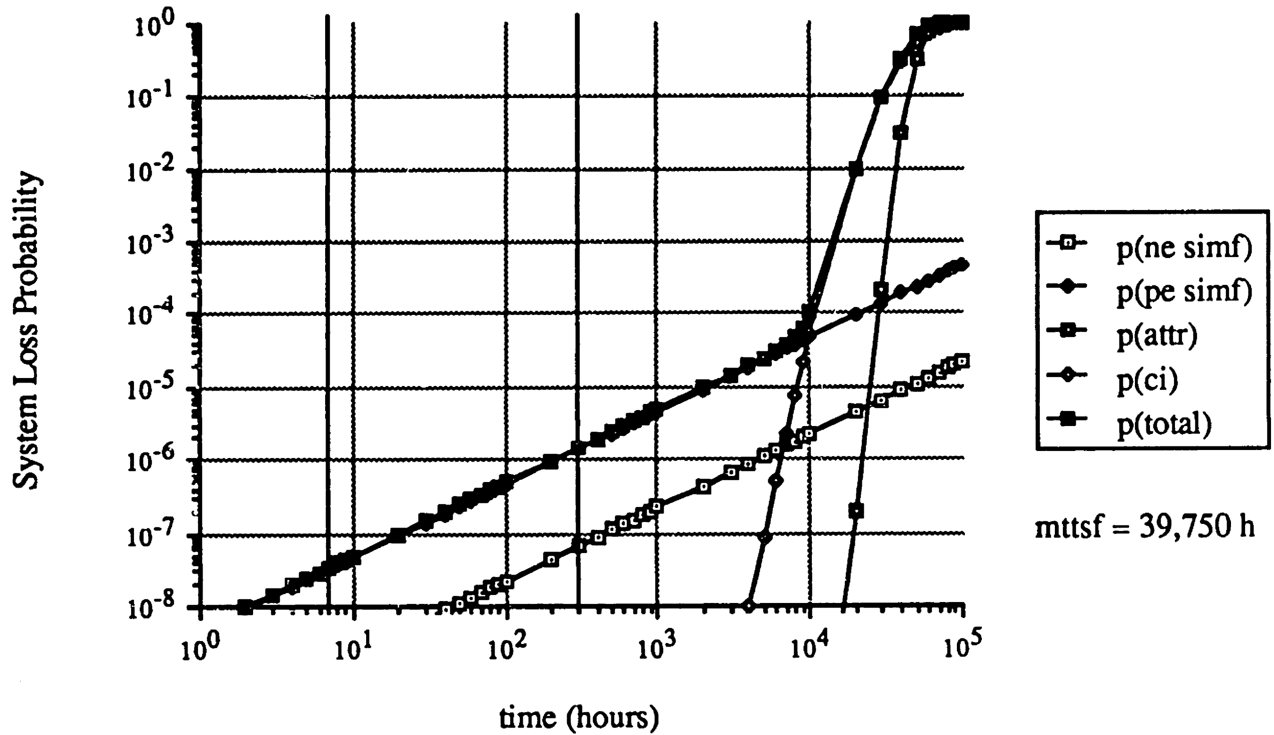
Data from "8 12 2 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 32,470 h



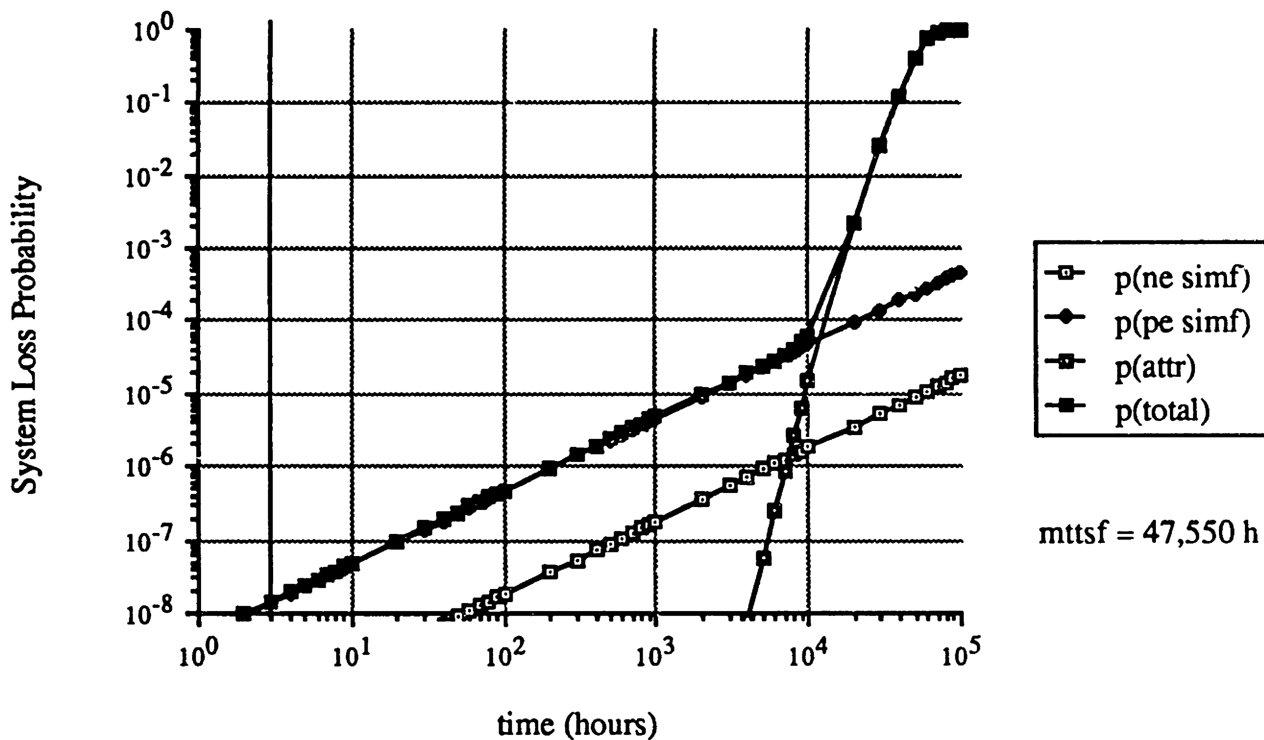
Data from "8 24 1 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(r.e simf)
p(pe simf)
p(attr)
p(total)
mttsf = 35,130 h

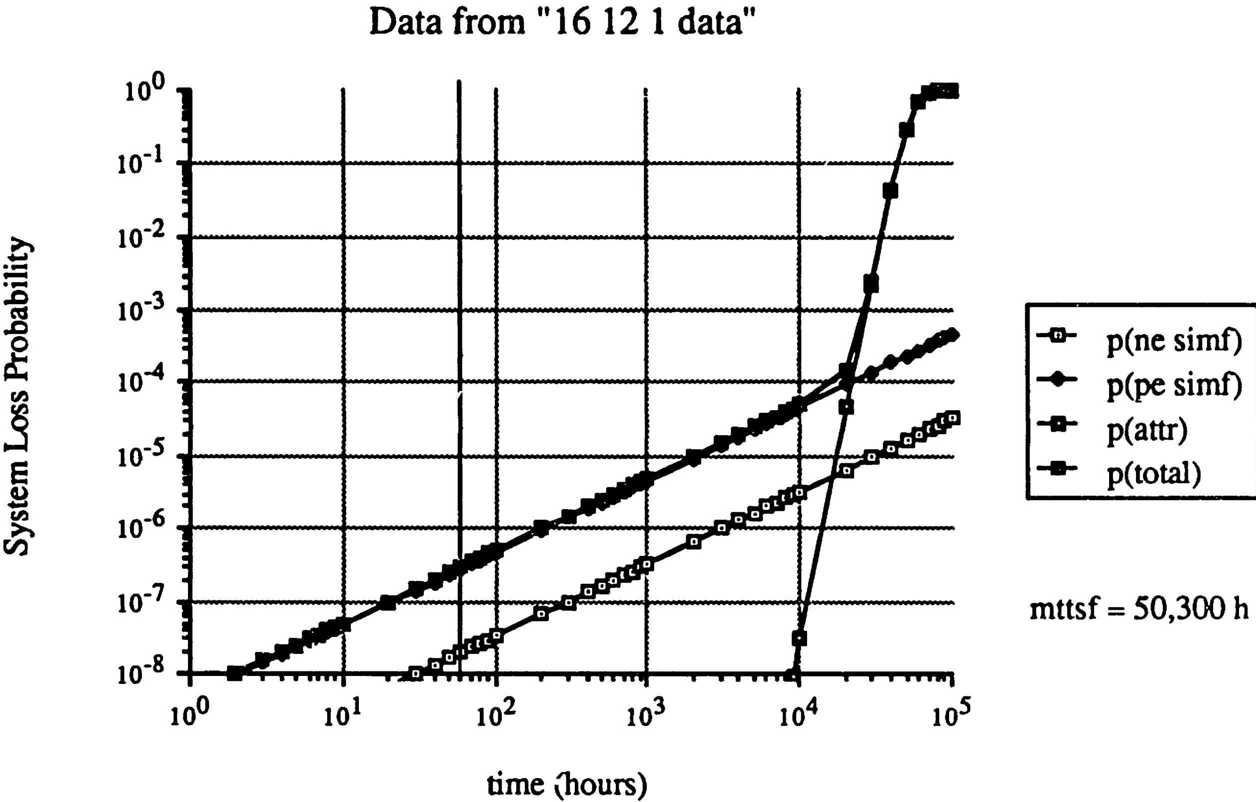

Data from "12 2 8 data"
System Loss Probability
10^0
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
10^-8
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 42,260 h

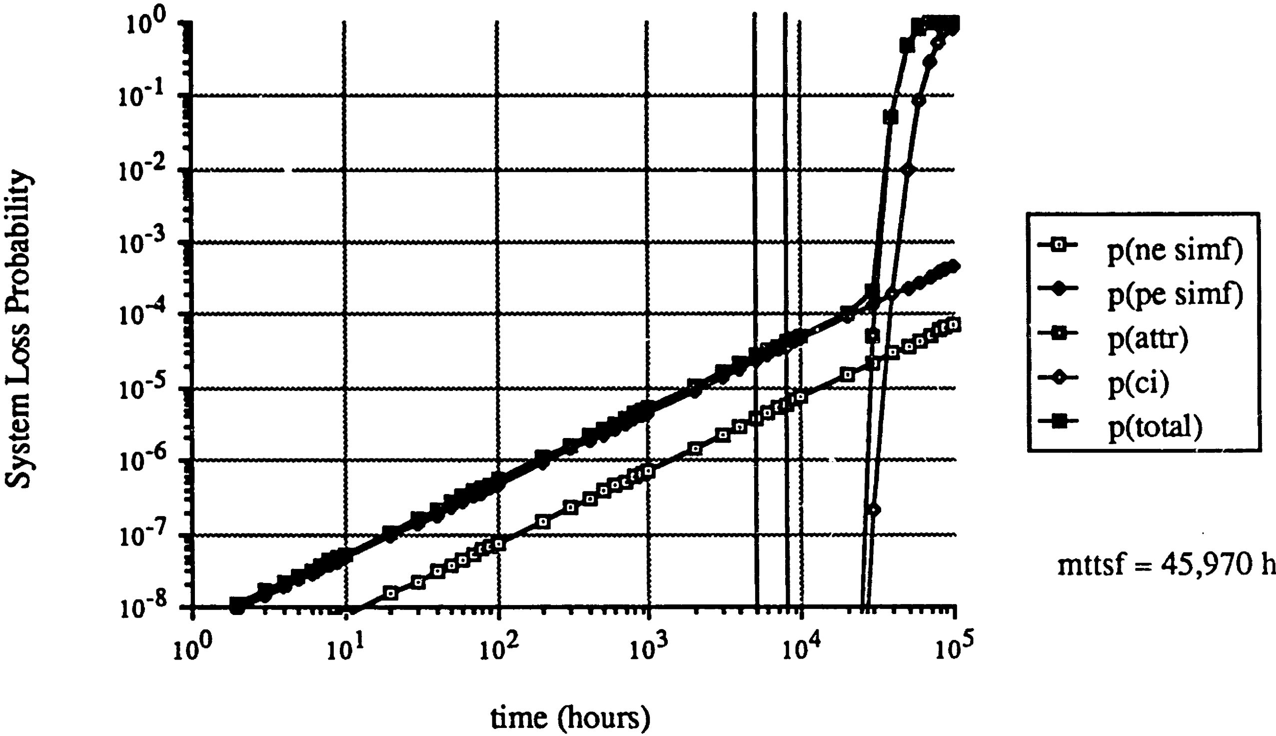

Data from "12 4 4 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 46,350 h

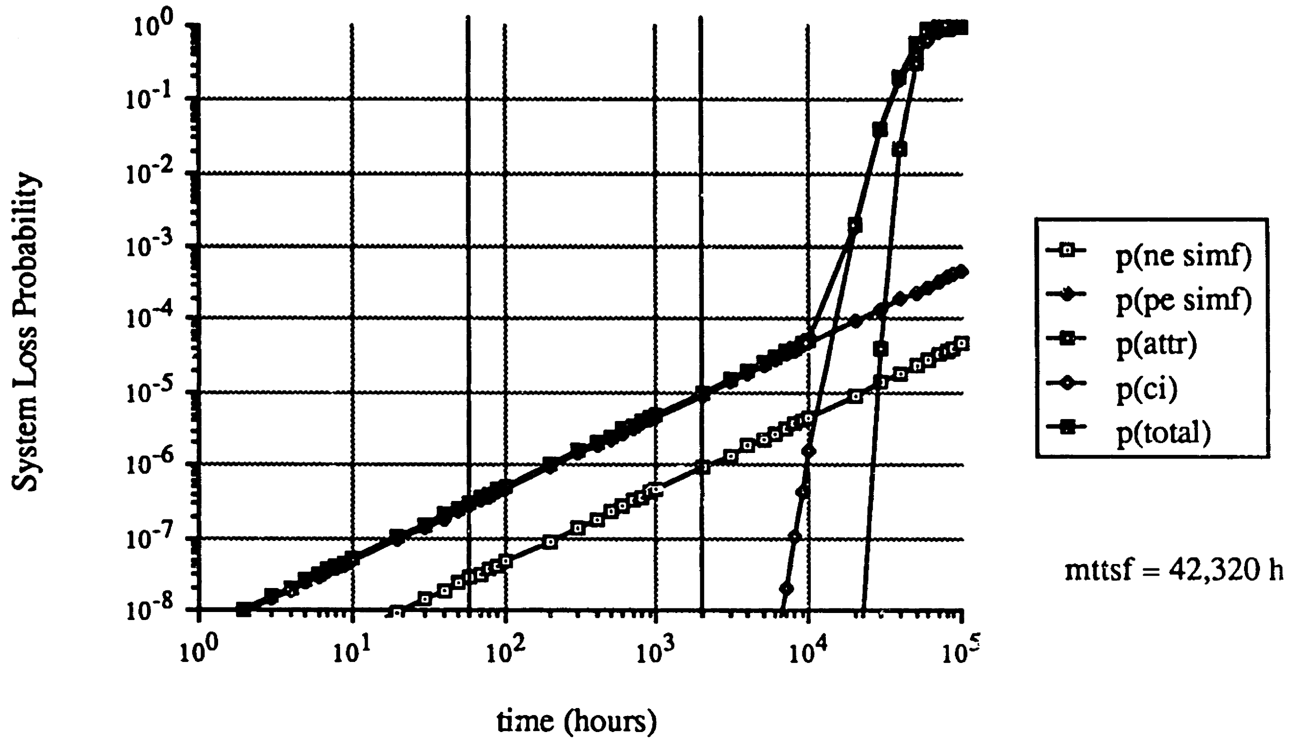


Data from "12 8 2 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 39,750 h

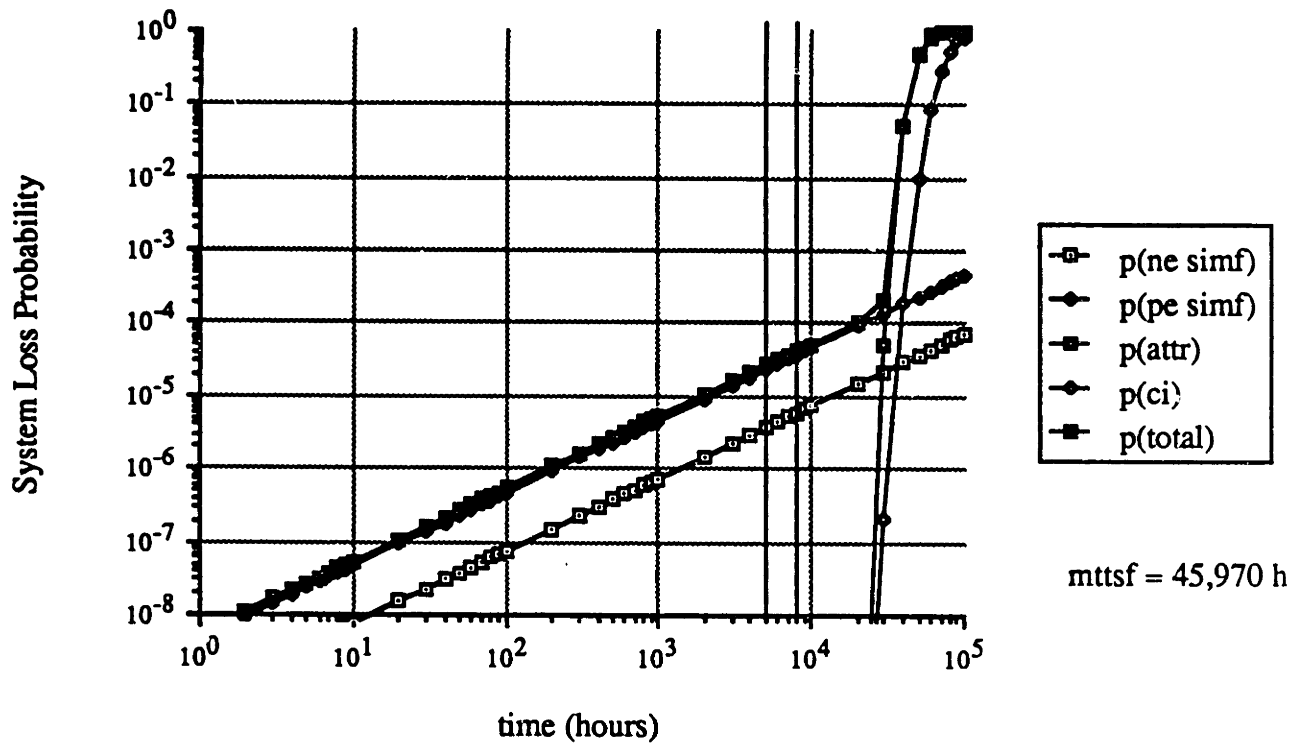

Data from "12 16 1 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(total)
mttsf = 47,550 h

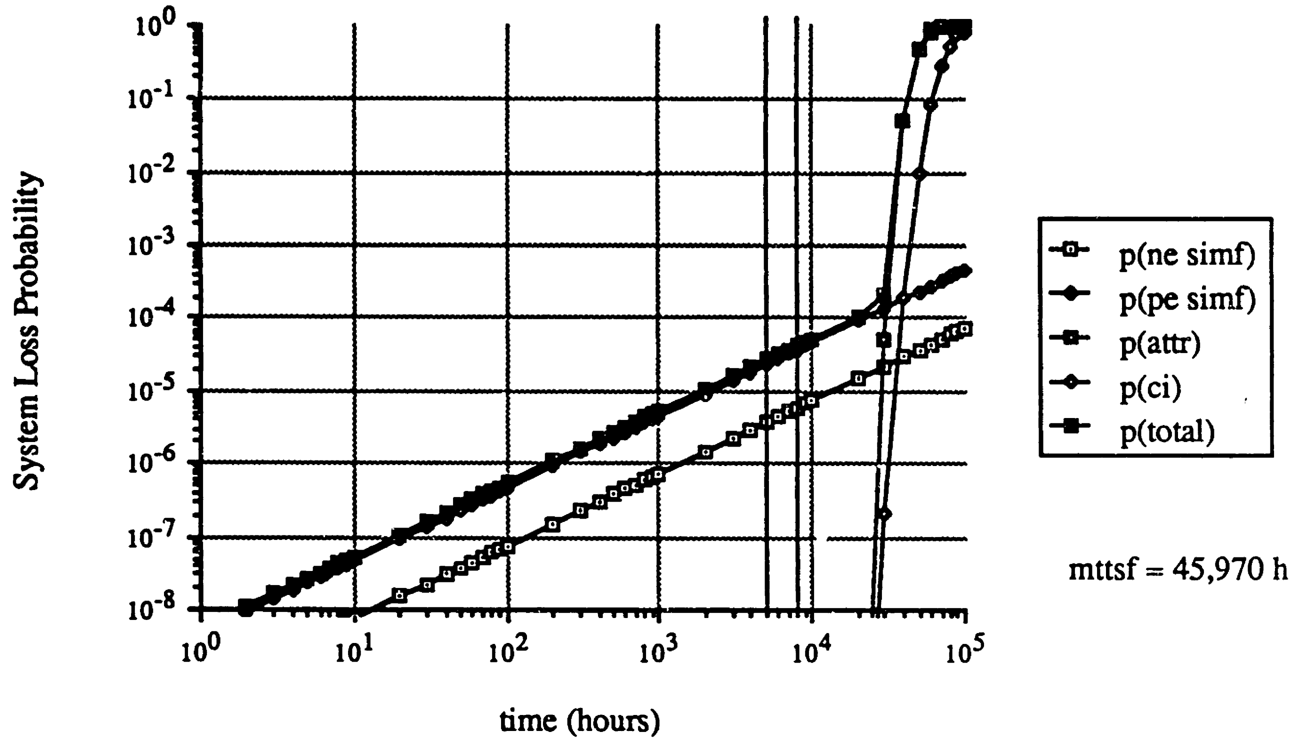

Data from "16 12 1 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(total)
mttsf = 50,300 h

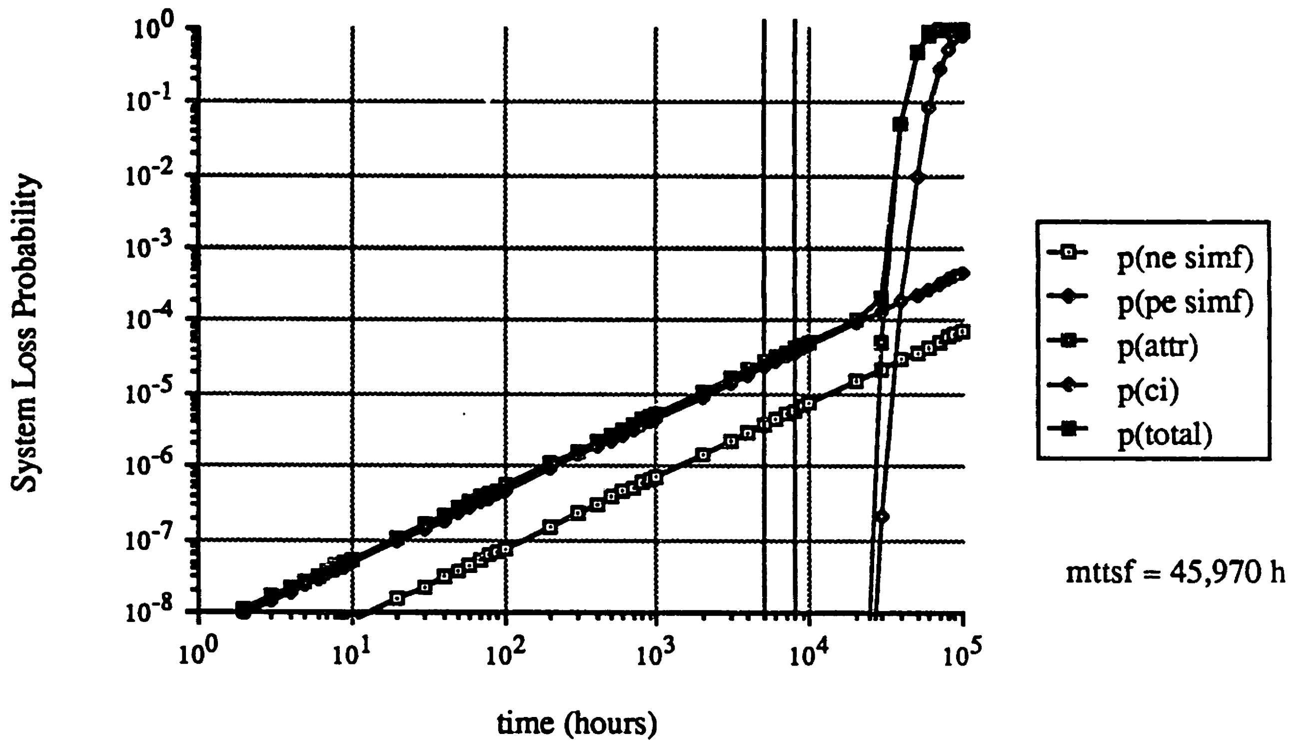

Data from "16 3 4 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 45,970 h



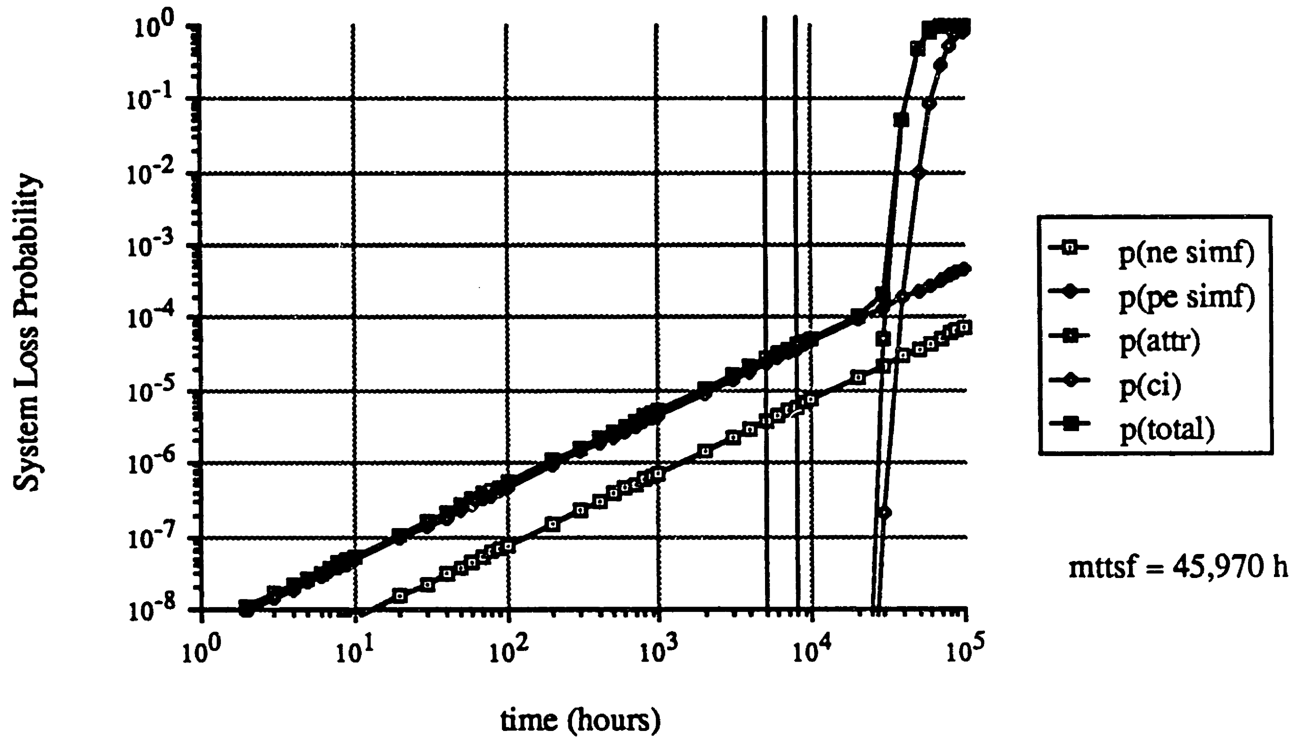
Data from "16 6 2 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 42,320 h


Data from "16 3 4 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 45,970 h


Data from "16 3 4 data"
System Loss Probability
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3
10^4
10^5
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 45,970 h



Data from "16 3 4 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 45,970 h


Data from "16 3 4 data"
System Loss Probability
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
time (hours)
p(ne simf)
p(pe simf)
p(attr)
p(ci)
p(total)
mttsf = 45,970 h

Appendix B

Relative Complexities of Processing, Network, and IO Elements[1]

Heurikon V20 68020 Processing Element

Component:	# transistors
CPU, FPU	600,000
1 Mbyte Memory	8,000,000
ROM	2,000,000
Glue Logic	50,000
Total	10,650,000

Network Element[2]

Component:	# transistors
FIFOs for I/O buffering To/From PEs and other NEs	500,000
Glue Logic	40,000
Microsequencer Control PROMS	230,000
Internal RAM	130,000
Total	900,000

IO Element[3]

Component:	# transistors
UARTS To/From other IOEs	1,120,000
FIFO To/From NE	70,000
Glue Logic	10,000
Microsequencer Control PROMS	20,000
Total	1,220,000

[1]This data generously provided by Stuart Adams.
[2]4 NEs per cluster, 4 PEs per NE.
[3]16 Clusters per ensemble.