Adaptive Fault-Resistant Systems[1]

Jack Goldberg, Li Gong, Ira Greenberg
SRI International

Raymond Clark
Concurrent Computer Corportion

E. Douglas Jensen
Digital Equipment Corporation

Kane Kim
University of California, Irvine

Douglas Wells
Connection Technologies

[1]This work was sponsored by Rome Laboratories, United States Air Force Base, New York, Contract 13441-4505

Abstract

A research team led by SRI International has completed a 1.5 year period of work in adaptive, distributed, fault-resistant systems. This research has been motivated by the increasingly complex and dynamic nature of working environments for modern systems, especially distributed, real-time systems. Operating conditions in such environments vary greatly in the types and distributions of faults and input data, in user requirements in changing service situations, and in possible losses of computing resources. The traditional approach—applying given resources in a fixed system configuration to meet worst-case operating conditions— is becoming less tenable. Adaptive systems can track changes in the environment by modifying the way computing resources are organized and utilized. The goal of the research was to establish a foundation for a general methodology of design for adaptive, distributed, real-time, fault-resistant systems.

This report presents a general theory and architecture, a taxonomy of design approaches, and examples of concrete architecture and design techniques. A core approach is the use of a control-theory model for adaptive computer systems; key issues derived from the model are the need for accurate state evaluation and prediction and incremental control to assure adaptation stability. The study investigated general frameworks for specifying trade-offs among service attributes such as timeliness, accuracy and precision and examined how such trade-offs can be managed during adaptation.

Several new issues and opportunities in fault-tolerant computing were uncovered, including the use of formal models for specifying and predicting adaptive fault-resistant systems, reflective architecture for recursive control of fault tolerance implementations, and multihypothesis fault diagnosis to reduce the ambiguity and diagnosis latency in real-time, distributed systems.

Several case studies are presented, including Adaptive, Distributed Recovery Blocks (ADRBs), a scheme for exchanging processing resources for recovery speed, Adaptive Distributed-Thread Integrity (ADTI), a scheme for dynamically selecting appropriate detection and recovery protocols for managing node and link failures in the Alpha programming model, and Adaptive Fault Tolerance for Hybrid Faults (AFTHF), an efficient scheme for tolerating faults with a wide range of complexity.

Contents

1 Overview **1**

1.1 Motivations and Objectives 1
1.2 Work plan 2
1.2.1 Theory and General Architectural Principles 2
1.2.2 Specific design techniques 2
1.2.3 Communication of Results 4
1.3 Summary 4

2 Concepts and Techniques **5**

2.1 Objectives and Basic Concepts 5
2.1.1 Project Scope 5
2.1.2 Expanding the Envelope 6
2.1.3 Adaptation Effects in Distributed Systems 6
2.1.4 Environmental Change Properties and Service Attributes 7
2.2 Adaptation Triggers and Responses 11
2.3 Adaptation Example: Adaptable Distributed Recovery Blocks 12
2.4 Adaptation Models 14
2.4.1 A Model for Specifying Adaptation 14
2.4.2 A Performance Model for Adaptive Systems 16
2.5 A General Adaptation Scheme 17
2.6 Adaptation Strategies for Fault Tolerance 18
2.7 Attribute-Based Technique Selection 20
2.8 Diagnosis and Control 21
2.8.1 The Role of Diagnosis in Adaptive Control 21
2.8.2 A Simple Control Scheme 22
2.8.3 Incremental Diagnosis and Control 23
2.9 Reflective and Hierarchical Architecture 25
2.9.1 Reflective Architecture 26
2.9.2 Multilayered Changes in a Function-Support Hierarchy 27
2.10 Results and Future System Design Issues 27

3 The Adaptive Distributed Recovery Block Scheme **29**

3.1 The Role of ADRBs in Adaptive Fault Tolerance 29
3.2 Basic Principles of the DRB Scheme 31
3.2.1 Primary-shadow pair of self-checking processing nodes 31

3.2.2 Replication of recovery blocks . . . 35
3.2.3 Recursive shadowing with N (> 2) try blocks . . . 37
3.2.4 Supervisor station . . . 38
3.2.5 A DRB Implementation Structure for Multicast LAN-Based Systems 39
3.2.6 Principles and Implementation Structures of the ADRB scheme . . . 44
3.2.7 Summary and Remaining ADRB Research Issues . . . 64
3.2.8 Partial Validation of a DRB Implementation . . . 64
3.2.9 A Modular Implementation Model for the DRB Scheme with a Configuration Supervisor . . . 64
3.2.10 An Implementation Model for a Worker DRB Computing Station . . 68
3.2.11 An Experimental Validation of the Modular Implementation Model . 74

4 Adaptive Distributed Recovery Block Demonstration 75
4.1 Adaptive Distributed Recovery Blocks . . . 76
4.2 ADRB Behavior . . . 77
4.2.1 Operational ADRB Mode Characteristics . . . 77
4.2.2 Dynamically Changing ADRB Modes . . . 78
4.2.3 Experimental Measurements . . . 78
4.3 ADRB and Application Structure . . . 81
4.3.1 Separation of Application and Adaptivity Software . . . 81
4.3.2 Structure of Adaptivity Software . . . 82
4.4 Lessons Learned . . . 83
4.4.1 Desirable Application Function Properties . . . 83
4.4.2 ADRBs and Databases . . . 84

5 Adaptive Distributed-Thread Integrity 87
5.1 Alpha Thread Maintenance and Repair . . . 88
5.2 The Alpha Programming Model . . . 88
5.2.1 Objects . . . 89
5.2.2 Distributed Threads . . . 89
5.2.3 Invocation and Thread Creation . . . 90
5.2.4 Illustration of Alpha Programming Model Concepts . . . 90
5.3 Distributed-Thread Integrity . . . 90
5.3.1 System Assumptions . . . 92
5.3.2 Components of the Thread Trimming Approach . . . 93
5.3.3 Alternative Task Designs . . . 94
5.3.4 Interactions with Other System Components . . . 95
5.4 The Alpha TMAR Protocol . . . 96
5.4.1 Alpha TMAR Assumptions . . . 96
5.4.2 The Alpha TMAR Protocol . . . 97
5.5 Alternative TMAR Protocols . . . 99
5.6 The Node Alive TMAR Protocol . . . 101
5.6.1 Node Alive TMAR Assumptions . . . 101
5.6.2 Description of the Node Alive TMAR Protocol . . . 103
5.7 Adaptive TMAR . . . 105

5.7.1 An Adaptive Thread Polling Protocol . . . 105
5.7.2 An Adaptive Node Alive Protocol . . . 106
5.7.3 Switching between Thread Polling and Node Alive Protocols . . . 107
5.8 Adaptivity Functions . . . 108
5.8.1 Monitoring . . . 109
5.8.2 Diagnosis . . . 109
5.8.3 Control . . . 110
5.8.4 Metacontrol . . . 111
5.9 Simulation . . . 111
5.9.1 Simulation System . . . 111
5.9.2 Assumptions . . . 112
5.9.3 The TMAR Protocols . . . 113
5.9.4 Adaptation Control Strategy . . . 114
5.9.5 User Interface . . . 115
5.9.6 Adaptive Control Experiment . . . 119
5.9.7 Further Development of the Simulation System . . . 122
5.10 Conclusions and Recommendations . . . 123

6 Implementing Adaptive Fault-Tolerant Services for Hybrid Faults 125
6.1 Introduction . . . 125
6.2 An Adaptation Strategy . . . 126
6.3 Two Adaptive Fault Tolerance Algorithms . . . 127
6.3.1 System Model . . . 127
6.3.2 Manifest versus Symmetric Faults . . . 128
6.3.3 Manifest versus Asymmetric Faults . . . 132
6.4 Related Work . . . 134
6.5 Summary and Future Work . . . 135

7 Conclusions 137

A Adaptive Fault Tolerance 145

List of Figures

2.1 Expanding the envelope of operations 6
2.2 Adaptation effects in distributed systems 7
2.3 Operating-environment change factors 8
2.4 Service attributes and interfaces 9
2.5 Adaptation triggers and responses 11
2.6 Adaptable distributed recovery blocks 13
2.7 State model of adaptive systems 15
2.8 Markov model of fault and work load adaptation 16
2.9 A model-based control system 17
2.10 A general adaptation scheme 18
2.11 Implementation strategies 19
2.12 Design-time analysis of technique attributes 21
2.13 Attribute-based technique selection 22
2.14 A Simple Control Scheme 23
2.15 Incremental diagnosis and control 24
2.16 Reflective architecture 26
2.17 Multilayer changes for adaptation 27

3.1 PSP-structured computing station 31
3.2 Detailed view of a PSP-structured computing station 33
3.3 Successor PSP-structured computing station 34
3.4 A DRB combines PSPs and replicated RBs 36
3.5 A DRB station with recursive shadowing 38
3.6 Using a DRB station as supervisor 39
3.7 A fault-tolerant LAN-based system consisting of a supervisor station and DRB stations 39
3.8 Achieving reliable data input 42
3.9 Basic operations under the ADRB scheme 45
3.10 Basic Components of an ADRB station 47
3.11 Three execution modes of an ADRB station 47
3.12 Ordering of execution times under different modes 49
3.13 Typical adaptation scenarios 50
3.14 A basic configuration of an ADRB station 51
3.15 ADRB transition protocols I 52
3.15 ADRB transition protocols II 53

3.15 ADRB transition protocols III 54
3.16 Adapatation possibilities for various combinations of equipment and time . 56
3.17 Logical NCM function 57
3.18 Four different execution modes for NCM 58
3.19 Basic principles guiding the execution modes 59
3.20 Performance characteristics of the four NCM execution modes 60
3.21 A scenario for adaptation of the NCM server 62
3.22 Structuring of the freeway monitoring application 65
3.23 Details of the DRB protocols for the primary node from the freeway example 66
3.24 Details of the DRB protocols for the shadow node from the freeway example 67
3.25 Structure of the modular implementation model for a node in a DRB station 69
3.26 Detailed structure of the modular implementation model for a primary node 70
3.27 The structure of the implementation model for a supervisor node 73

5.1 Illustration of the Alpha programming model concepts 91
5.2 Effects of a node failure on executing threads 92
5.3 Repair actions performed by TMAR 93
5.4 Example control function using hyst.pseresis 110
5.5 Sample simulation program configuration file 116
5.6 Results of the adaptive control experiment 121

Chapter 1

Overview

A 1.5 year period of work in adaptive, distributed, fault-resistant systems was conducted by a team of researchers at Concurrent Computers, University of California at Irvine, SRI International, and two consultants, and was led by SRI International.

We hope that this report will communicate our view that incorporating adaptation into operating environments as a fundamental characteristic of computer system architecture offers (1) an important advance in computer architecture, and (2) a fruitful area for technology development.

1.1 Motivations and Objectives

In early applications and in current high-criticality applications, great efforts are made during design to isolate a computer system from its environment except for a carefully managed data flow. Designs are intended to provide a predetermined service to the users, using a fixed set of algorithms for all data and environmental conditions within allowed ranges. In the design, sufficient resources are provided to meet the worst possible combination of data and environmental events; resolution of conflicts in resource allocation due to unpredictable data events is relegated to a scheduler.

As computer systems become ubiquitous, the dynamic characteristics of their environment become more significant in determining how well (or poorly) a system serves its users. The combined effects of faults and resource failures, wide swings in service demand, and situation-dependent user requirements stress a computer's ability to satisfy its service expectations. This is an especially significant problem in distributed systems that employ unreliable communications, and whose components may operate in different and perhaps harsh physical, data, and usage environments. For such operating situations, the principle of meeting worst-case operational constraints using a fixed design has become increasingly difficult to apply. One of the goals for adaptive design is to allow flexible use of available resources to cover a much wider range of different kinds of environmental variables than could be covered by a fixed, worst-case design.

Adaptivity has been increasingly suggested by researchers as a way to meet the challenge of wide environmental changes. While some adaptive algorithms have been developed and employed for particular computer system functions—the ethernet protocol is a well-known example—adaptivity as a general principle in computer systems is not well understood or

accepted. The purpose of the research was to establish and demonstrate the feasibility of adaptive computer systems by initiating development of a systematic design methodology, consisting of theory, architecture, and practical techniques of design.

Given the newness of the subject, some effort was spent in establishing and clarifying concepts–for example, one might say that an adaptive system responds to its environment, but since all data processing is in some sense uniquely responsive to data values obtained from the environment, at what point does a system cease to be just an implementation of an algorithm, and become an adaptive system? We believe we have answered this question.

We also sought to arouse the interest of the research community, through papers and presentations at conferences and workshops, in order to use the present project as a lever in advancing technology. We are pleased to report that several researchers have responded to our ideas, and are now contributing techniques for adaptivity to the scientific literature.

1.2 Work plan

The research focused on the following tasks:

- Develop a theory of adaptive fault-resistant systems and general principles of architectural design
- Develop specific architecturasl design techniques
- Demonstrate adaptive designs
- Communicate results to the scientific community

1.2.1 Theory and General Architectural Principles

Chapter 2 presents results in theory and general architectural design principles. The discussion is intended to clarify concepts, define key issues, and offer feasible solutions and design approaches. It proceeds through the following subjects: basic concepts of fault-tolerant service and adaptability, examples of adaptation, models for specification and performance prediction, a general architecture for adaptive control, controlling the tradeoffs of service attributes in adaptation, a taxonomy of design techniques, real-time diagnosis, prediction and stable control, recursive-reflexive control architecture, and adaptation relations in layered and distributed systems.

We believe that the abstract specification and performance models, the discussion of the critical role of diagnosis in achieving stable control, and the use of reflexive architecture for fault tolerance are novel contributions to the methodology of adaptive computer design. The use of the control-theory model for adaptive systems has been particularly useful in exposing problems of state assessment (called diagnosis in our treatment to conform with the vocabulary of fault tolerance) and stability of adaptation.

1.2.2 Specific design techniques

The taxonomy of design techniques presented in Chapter 2 suggests the very wide range of new techniques that may be employed to support adaptive architectures. We studied three techniques, whose results are described in separate chapters:

- Adaptive Distributed Recovery Blocks (ADRBs), a multiple-mode scheme for error detection and recovery, useful for both hardware and software faults; the scheme allows an exchange of processing resources for error-recovery speed (Chapters 3 and 4)
- Adaptive fault tolerance for hybrid faults, an economical technique for tolerating both simple and complex fault types (Appendix A)
- Adaptive distributed-thread integrity, a technique for detecting and repairing thread breaks in a wide range of operating environments using the Alpha programming model (Chapter 5)

Our examples demonstrate specific design techniques and illustrate the service tradeoffs that are characteristic of adaptive designs. Models of service attribute tradeoffs are reviewed, such as the triad basis of timeliness, precision, and accuracy. The examples show that there is an abundance of types of functional adaptation, and that adaptive design can be very straightforward. A remaining challenge is to find very economical design solutions that can limit the added complexity and performance overhead introduced by adaptation. One example shows that adaptation can actually *reduce the normal processing overhead* for systems that must cope with complex as well as simple faults.

Demonstration of adaptive distributed recovery blocks We developed a demonstration of the ADRB scheme, using the Alpha programming model and designed to run on an Alpha testbed. The system is intended to demonstrate the tradeoff of resource utilization (with impact on possible throughput) and error-recovery speed. Three modes are exhibited: (1) single processor, serial recovery, (2) dual processor, concurrent recovery, and (3) single processor, default output. Mode 1 features low processor cost—hence high throughput for a given set of processors, mode 2 features rapid recovery but higher processor utilization, and mode 3 features low processor cost but low accuracy. One of the challenges encountered was to create a design that would make seamless transitions between operating modes.

Adaptive fault tolerance for hybrid faults We developed a novel algorithm for using redundant processors to tolerate hybrid faults—that is, faults of several types. The types considered include crash faults (silent processors) and value faults of two kinds: symmetric (all faulty processors produce the same wrong value) and asymmetric (faulty processors produce arbitrary values). In contrast to other hybrid-fault tolerance schemes, the algorithm has very low processing overhead in the normal, fault-free case. The algorithm may be seen either as a very economical way to broaden the fault coverage of classical primary-backup processor systems or to lighten the average processing burden of classical consensus-based processor systems. The contribution of adaptivity is that errors are diagnosed and a decision is made as to the proper fault tolerance.

Adaptive distributed-thread integrity The Alpha distributed, object-oriented programming model employs threads of control that may span several nodes. A distributed thread has a root and a point of activity that may move from node to node as objects are invoked. Control ultimately returns to the root as invocations complete. Node failures may break a distributed thead into several disjoint pieces. The system is responsible

for identifying broken threads, safely terminating orphan thread segments, and properly restarting thread activity. Different strategies are better suited for maintaining thread integrity, depending on the nature of the thread workload, the system resources, the fault model, and the application requirements. We have developed an adaptive algorithm for this maintenance function, and have simulated its behavior.

1.2.3 Communication of Results

Project work has been reported in the following communications:

- A paper at the IEEE Workshop on Advances in Parallel and Distributed Systems, November 1993, Princeton [14]—included as an appendix to this report
- A manuscript submitted to the 1994 Symposium on Reliable and Distributed Systems, November 1994
- Oral presentations at two 1993 Rome Laboratory Technical Exchanges
- Talk at a seminar of the University of Arizona Computer Engineering Department, January 1993
- Talk at IEEE Workshops on Fault and Error Injection, Gotheborg, Sweden, June 1993, and Annapolis MD, April 1994
- Talk at the Bay Area System Seminar, Menlo Park CA, July 1993
- Talks at IFIP Working Group 10.4 meetings, June 1993, and January 1994
- Outline and material for a paper on distributed ADRB control

1.3 Summary

Adaptative fault resistance is a new direction in computer architecture. Adaptivity allows decisions about algorithms and resource allocation to be made at operation time that usually are fixed at design time. The goal of adaptive design is to make it possible for a fault-tolerant computer to cover a wider range of environmental conditions, such as variations in fault types and distributions, changes in user requirements, variations in workload, and variations in resource availability, than could be served by a fixed design.

This report describes work of a multiorganization team in theory, system architecture, and methodology of design, including case studies, novel algorithms, and a demonstration. The work is intended to build a foundation for a methodology of system design. The results have been reported in various professional meetings, resulting in some new investigations by other researchers.

Chapter 2

Concepts and Techniques

The major technical issues and principles for adaptive fault resistance that have been developed during this project are discussed below. . Since our previous interim report, we have introduced more formalism in the definition and behavioral description of adaptation, sharpened our description of key issues such as attribute-based technique selection, and developed several general and concrete architectural models for adaptive systems.

2.1 Objectives and Basic Concepts

The scope of the project was determined by the general motivations for, and concepts of adaptation for, distributed fault-tolerant systems.

2.1.1 Project Scope

The basic technical concept of adaptive fault resistance (AFR) is that a dependable computing system should have the ability to adapt its structure autonomously to changing operational conditions, so that its service range will be larger than could be provided by the same resources in a static structure.

This may be envisioned as an expansion of the envelope of dependable service that is determined by user requirements, computing resources, work load, and fault distribution. A major area of application for adaptive fault resistance is distributed real-time systems, which typically operate in environments that are highly variable and difficult to control.

The adaptation concept differs from current design methodology, which assumes that a system is given certain behavioral capabilities, typically based on some worst-case assumption, that are invariant during operation. We believe that in the dynamic environments in which real-time distributed systems operate, it will be rare for the worst-case conditions to occur simultaneously in all dimensions. Adaptation seeks to take advantage of this non-concurrency by reconfiguring system resources to meet the current combination of service demands and fault conditions.

The objective of the research is to develop principles, methods, and techniques of design that will make adaptive behavior a fundamental feature of dependable, real-time distributed systems.

2.1.2 Expanding the Envelope

We assume an initial, classic design in which a system operates within a given domain that is determined by a set of requirements and expected operational conditions; it is further assumed that any combination of requirements and operating conditions may occur at any time during operation. As illustrated in Figure 2.1, we then assume that a revised requirement has been submitted that significantly expands the domain with respect to some or all the dimensions of user requirements, available resources, work load, and fault distribution, but with the additional possibility that not all combinations of operating conditions may occur simultaneously.

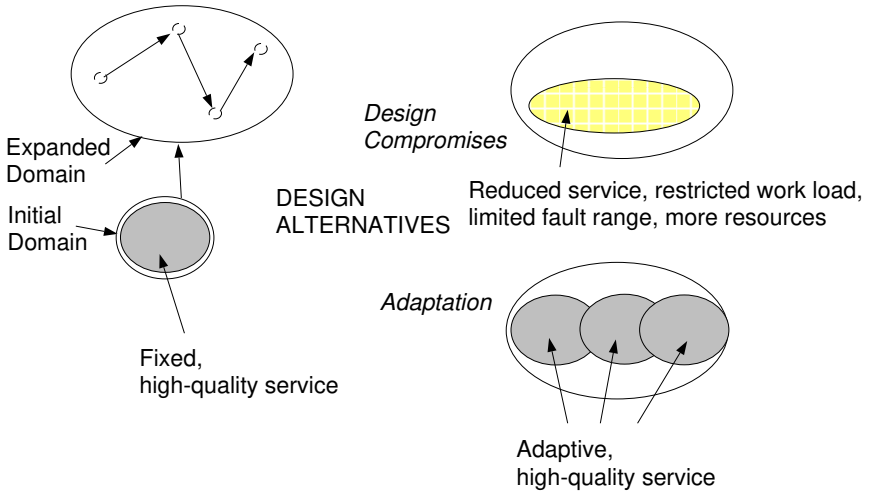


Figure 2.1: Expanding the envelope of operations

We expect such expansions to be increasingly common as dependable, real-time systems move out of their current niche applications, such as ultracritical process control or well-constrained transaction processing, into more complex, real-world applications.

The classic response to the expanded needs of complex environments is to add resources and to compromise the initial requirements so as to attain a buildable and affordable system. The AFR approach is to limit the increase of resources, and to structure the resources in different ways at different times, in accordance with changes in operating conditions. The design goal is that for any given operating condition, the resources will give dependable service comparable to that provided by the same resources for a constrained, fixed operating domain.

The price to be paid for such economy of resources is that it may take time for the system to adapt to new conditions and that some additional design risk and overhead performance cost may be associated with the additional functionality of adaptation.

2.1.3 Adaptation Effects in Distributed Systems

Significant changes in operating conditions for a distributed system may apply globally or locally, in one node or link or in several. We assume that such changes will trigger adaptivity

within and among the components affected. As illustrated in Figure 2.2, the adaptivity may or may not be transparent to the remainder of the system, whose operating conditions have not changed.

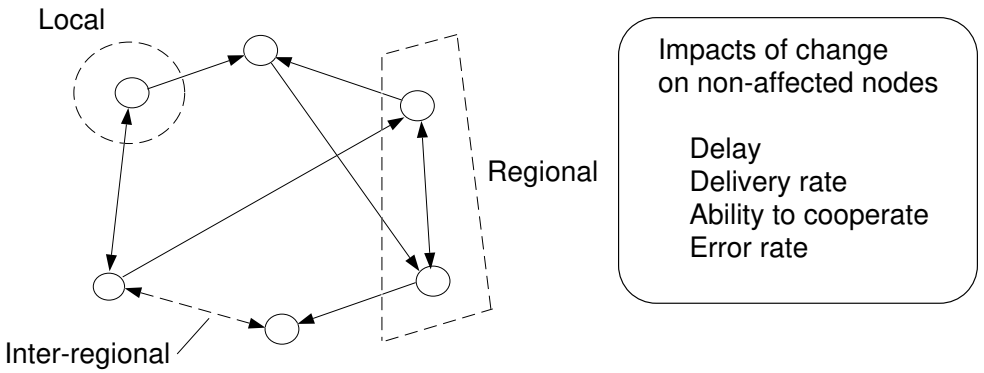


Figure 2.2: Adaptation effects in distributed systems

Externally visible changes resulting from adaptation may include

- Changes in the latency or bandwidth of exchanges between adapted and nonadapted nodes
- Changes in the work load capacity of adapted nodes
- Changes in the error rate of delivered results

We note that the changes may be positive or negative; for example, the rate of fault occurrence may increase or decrease with the appearance or disappearance of an external fault source. Adapting to a reduced fault rate may result in increasing processing capacity, which can benefit the system as a whole.

Externally visible adaptations effectively constitute a change in operating conditions for the nonadapted nodes, and may therefore require them to undertake adaptation — that is, there may be some propagation of adaptation through the system. This is a potentially dangerous phenomenon, because the chain of adaptation may become a significant performance burden or interruption, and if there are cycles in the adaptation path, there may be instability.

An important technical issue is therefore how to limit or control the propagation of adaptation effects within a distributed system. One way this can be expressed is the problem of finding the smallest subset of a distributed system such that adaptation effects within the subset are invisible to the remaining system.

2.1.4 Environmental Change Properties and Service Attributes

Changes in the environment's properties may require system adaptation. We first review a key set of properties, and then consider the criteria for service that will be used to guide the adaptation.

Environmental Change Properties

As illustrated in Figure 2.3, we consider the operating environment to be defined by four dimensions:

- Work load
- User service requirements
- Faults
- Resources

For each dimension, it is assumed that there may be some operation-time variation that stresses the system's ability to meet its requirements, and hence may require the system to modify the way its resources are utilized. The dimension of user requirements is unique, in that its changes may be both the reason for adaptation and the measure of how well adaptation succeeds. Specifications of requirements may be both absolute and relative, and different for each element of a user's service. For example, a user may demand absolute availability for some service element, while allowing a tradeoff between availability and error rate for some other service element.

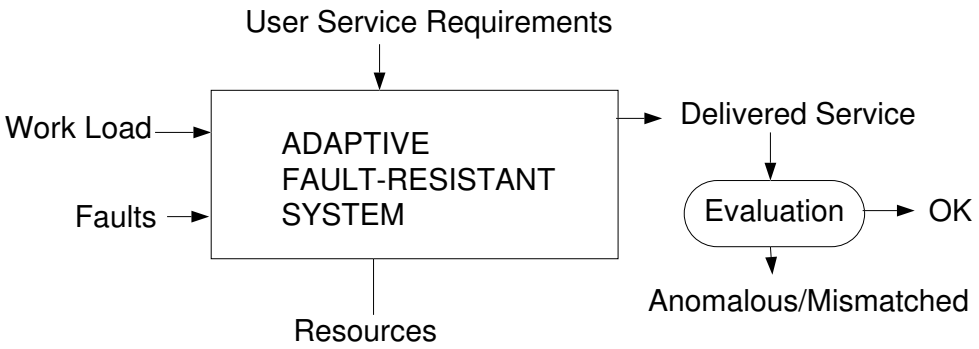


Figure 2.3: Operating-environment change factors

The following list contains some familiar instances and examples of the four dimensions:

- Work load dimensions
  - Data types (alphanumeric, logical, signals, strings)
  - Time and space distribution (rates, intervals, location, clustering)
- User service dimensions (illustrated in Figure 2.4)
  - Data processing functionality
  - Performance (throughput, latency)
  - Timeliness (satisfaction of deadlines, allowed number of missed responses, preservation of order, closeness of synchronization)

  - Accuracy (closeness of the computed to the ideal value, mutual consistency of related data—but see below for a different interpretation)
  - Dependability (reliability, availability, safety, security)

- Fault dimensions
  - Fault types (permanent, transient; design, hardware, operator)
  - Fault time, space and multiplicity distribution (rate, interval, location, clustering)

- Resource dimensions
  - Magnitude of processing, storage, and communication resources
  - Rate of permanent losses
  - Frequency and duration of resource overloads

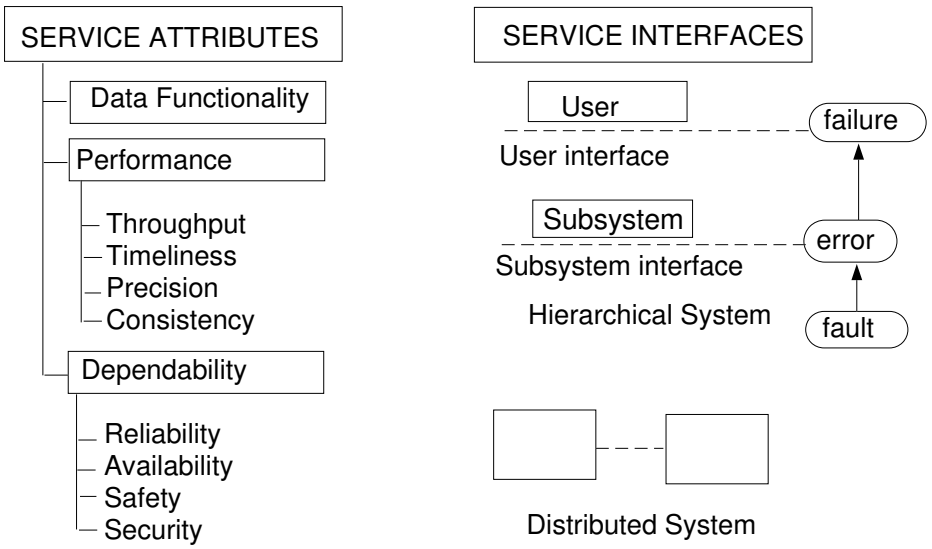


Figure 2.4: Service attributes and interfaces

A Basis Set for Service Attributes

Tom Lawrence, Rome Laboratory, has noted that these service attributes may be interpreted in terms of an economical basis set consisting of Timeliness, Precision, and Accuracy. In this basis set, Precision refers to the amount of information processed, which might include the volume of data or the rate of processing, while Accuracy may include correctness or closeness to correct value. In any system and application environment, the service attributes may be closely interrelated—for example, loss of timeliness in a service delivery may result in a failure to service new data values adequately or at all, with resulting errors or omissions in computed values. In terms of the attribute basis set, such loss of timeliness may cause loss of both precision—in this case, throughput or coverage of input work, and accuracy—in this case, errors of omission or incomplete processing.

Faults, Errors, Failures, and Fault Tolerance

To round out our definitions of the service concept, we present a highly condensed summary of key ideas in fault tolerance. A more extensive definition and explanation of fault tolerance concepts has been published by the IFIP 10.4 Working Group on Dependability and Fault Tolerance [20]. We start by distinguishing faults, errors, and failures and their relations, with specific reference to the point in a system at which they may occur or be observed:

- Fault — A deviation between the specification and implementation of a component's function, either by physical state change or design error. Faults may have many different causes and manifestations—for example, faults may be transient, intermittent or permanent, they may occur in hardware, software or in operation, and they may occur at all levels of a system. Proper treatment of a fault strongly depends on the fault type—for example, repetition may be appropriate for a transient hardware fault, but a waste of time for a software fault.

- Error — A deviation of the state of a computed variable from the value specified by the design, as observed at a system component interface.

- Failure — A deviation of a system component from its specified behavior, as observed at its interface. A failure specification may allow some level of error in the system output to be considered acceptable.

A fault-tolerance monitor can *observe* errors but can only *infer* faults. This is typically done by comparing the results produced by a subsystem, for known inputs, with some reference values, and deducing which computing elements within the subsystem may have caused the error. In some cases, the data that produced an error is not available; in that case, considerable ambiguity may be introduced in determining the fault type and location.

Fault effects can propagate in several ways. Errors may flow horizontally through cascaded chains of functional elements, and vertically through layers of support functions. Errors that change the definition of a function create new faults—that is, malfunctions—which become the source of new errors.

Fault-tolerant computers are designed to prevent faults from leading to failures by

- Bounding the propagation of errors

- Masking or correcting erroneous results

- Isolating faulty components

- Restoring system state to the ideal or acceptable value

Fault tolerance objectives may have to be compromised according to the availability of resources and time. For example, it may only be necessary—or there may only be enough time available—to mask errors rather than to isolate the faulty components that cause the errors.

Given the possibility of fault propagation from point to point and layer to layer, it is possible to repair a fault at its origin without correcting all of its consequences—thus, a

fault that originated in some low-level hardware element may give rise to complex fault situations that must be remedied at a high system level.

Given the great variety of possible fault types and the complexity of determining the nature and location of a fault from the available error information, it is extremely important to be able to determine the type of fault that may have occurred and to predict the faults that are likely to occur in the next computing interval.

2.2 Adaptation Triggers and Responses

The phenomena that give rise to the need for adaptation, and possible adaptive responses, are related to the four key system properties of work load, user service requirements, faults, and available resources.

In an adaptive fault-tolerant system, adaptation will be initiated when there is an excessive mismatch between new operating conditions and the capabilities of the current system, that is, when the currently available resources are not fully utilized to meet current requirements for performance and dependability. Figure 2.5 illustrates various kinds of changes in operating conditions and possible adaptive responses.

Change Type	Triggering Anomaly	Response
Fault type	Too many software errors	Use alternative design versions
	Too many node and link crashes	Use alternative paths
		Increase redundancy
User directive	Reliability sacrificed for application coverage!	Reduce fault-tolerance redundancy levels
Fault distribution	Too many comm-link errors	Switch from optimistic to pessimistic protocols
Data distribution	Too many processor overloads	Employ load balancing
		Switch to memory-intensive algorithm

Figure 2.5: Adaptation triggers and responses

Fault-type triggers For changes in the prevailing fault type, say from mainly hardware to mainly software faults, a possible adaptation-triggering anomaly is the occurrence of an excessive rate of errors, resulting from the use of inappropriate fault tolerance mechanisms. An appropriate response may be to tolerate errors using multiple program versions (a form of software fault tolerance). A different kind of change in fault type may be the occurrence of more node and link crashes than can be satisfactorily handled by the current fault tolerance mechanisms. Possible responses are to use alternative communication paths, to increase the level of data redundancy over multiple nodes, or to switch from optimistic to pessimistic communication protocols.

User-requirement triggers A possible change in user requirements may be a user's decision to sacrifice the reliability of some application, for example, by accepting a higher error rate or lower precision, to be sure that certain vital tasks are adequately served. A possible system response is to shift resources from redundant to nonredundant task service.

Workload triggers Another important type of operating change is in the way input data are distributed among processing nodes. For example, data from external sensors may be concentrated at some node so that the node is occasionally overloaded. The resulting inability to process additional data is equivalent at the system level to a transient node failure. One obvious response is to employ load balancing among nodes. Some adaptation may be sufficient within the node; for example, it may be useful to switch processing algorithms at the node from processing-intensive to memory-intensive (i.e., table lookup) forms or vice versa.

Available-resource triggers As the number of resources available diminishes due to failures or overload, certain fault algorithms may become infeasible to support—for example, there may not be sufficient processing resources to provide desired levels of replication. Algorithms that require a certain balance of processing, storage and communication may become infeasible—for example, algorithms that depend on high bandwidth communication to maintain close synchronization between replicated processes may be impractical if certain timeliness and data consistency requirements must be satisfied.

These are only a few examples of adaptation-triggering anomalies and possible responses. We note that if a system were required to handle all of these changes simultaneously, as a worst-case condition, and without adaptation, the complexity and processing overhead costs of the fault-tolerant processing algorithm might be intolerable.

2.3 Adaptation Example: Adaptable Distributed Recovery Blocks

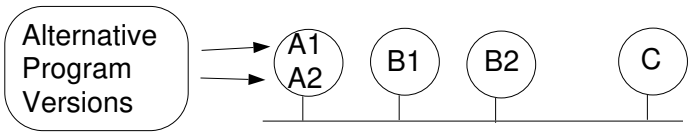
As a study vehicle, we have developed the concept of adaptable distributed recovery blocks (ADRB), and have demonstrated it in a simple testbed. An ADRB, which was conceived by Kane Kim, a project contributor, is an extension of a distributed recovery block (DRB), also by Kim, which, in turn, is an extension of a recovery block (RB), which was conceived and developed by Brian Randell of the University of Newcastle, UK. All of the RB versions are intended to tolerate both software design faults and hardware faults. RBs can tolerate transient hardware faults, but DRBs can tolerate both transient and permanent hardware faults. The basic DRB concept is to provide two or more versions of a program and to select the results of a program if it satisfies a user-designed acceptance test (AT). The reliability of the results depends on the precision of the AT, which is an application-level concern, and not a system-level concern. Practical ATs provide less than perfect assurance of correctness.

In the basic RB scheme, versions are tried sequentially until a version passes its AT. If all ATs fail, control is returned to a higher system level. In the DRB scheme, versions are given a rank ordering; all versions are executed concurrently and output is chosen from the

version of highest rank that passes its AT. Given the diversity of designs, the versions may not all complete at the same time.

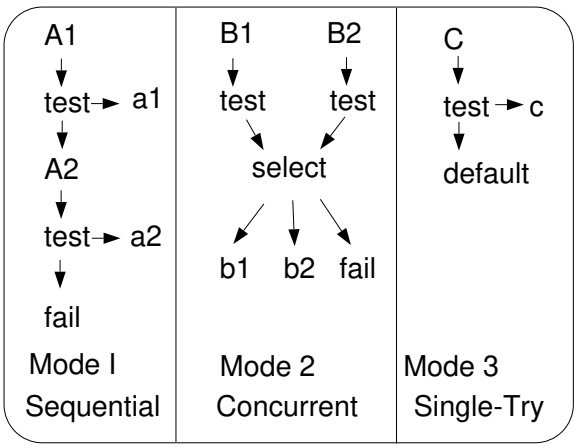
A third recovery block mode may be employed, which we call single-try (ST). ST attempts only a single program version; if the results fail the AT, they are discarded and control passes to the next input data item. Clearly, this default action constitutes a degradation of service.

The ADRB scheme, illustrated in Figure 2.6, permits a collection of processing nodes to be organized according to the RB, DRB, or ST scheme, depending on the desired system attributes. The figure illustrates that a collection of communicating processors might be set to operate in different modes simultaneously; for example, versions A1 and A2 are co-resident, and support the RB mode, versions B1 and B2 are in different processors, supporting the DRB mode, and single-version C is in a single processor, supporting the ST mode.



Mode	Attributes		
	Resource Utilization	Recovery Speed	Reliability
1	high	low	high
2	low	high	high
3	high	high	low

Attribute Trade-Offs



Alternative Error-Recovery Schema

Figure 2.6: Adaptable distributed recovery blocks

As shown in the table of attribute trade-offs in the figure, the modes exhibit different service attributes. RB mode has high resource efficiency (its only redundancy consists in the execution of the AT), but it suffers from a low speed of recovery from errors because of its need to re-execute a computation following an AT failure. This may result in a system error if a fault occurs during the execution of a task that has a tight deadline. In contrast, DRB mode is advantageous in fault-recovery speed, because the results of a second version are available immediately on the failure of the AT for the higher-ranked version, except for possible differences in execution of diverse versions. DRB's disadvantage is that it consumes twice the processing resources of RB, which diminishes the resources that may be needed

to serve a high work load. The ST scheme has high resource efficiency and high recovery speed (there is no second version to try again), but it suffers in reliability compared to RB and DRB because failed ATs result in failed service.

High recovery speed can have an important impact in reducing error rate, as follows: for high input data rate, it may occur that recovering from an error in processing one input datum using the serial method (standard RB) may prevent a processor from responding to a new input datum. The result may be the omission of output for the new datum or an imperfect computation. This corresponds to an interaction of Timeliness and Accuracy in Lawrence's attribute basis set.

The different attribute sets of the several modes present trade-off choices to the operator, in this case, resource efficiency, error recovery speed, and reliability. The operator may choose to use RB, DRB, or ST, depending on the relative importance of the three attribute characteristics. For a multiple-processor system, this choice may be made independently for different tasks and different processors.

In summary, we have described an example of an adaptable system with several modes that offers the operator alternative combinations and degrees of service attributes. Section 2.8.3 discusses how the adaptation choice itself may be made and put into effect.

2.4 Adaptation Models

Two abstract models for adaptation are described here. The first model, which is state based, is intended to allow specification of essential adaptive behavior, such as probability of successful and unsuccessful adaptation, adaptation thresholds, and transition times. The second model, which is Markovian [48], is intended to allow prediction of performance, both successful and unsuccessful, given probabilities of certain fault and work load experience and adaptive responses.

2.4.1 A Model for Specifying Adaptation

An engineering methodology for adaptive fault resistance design must include some means for specifying precisely when and how the system will perform adaptation. Figure 2.7 presents a simple, abstract model that suggests the possible form of such a specification.

An adaptive system is represented by the interaction between two state spaces: the operating-condition state space and the system-configuration state space. States in the operating-condition state space are themselves points in a vector space, with dimensions such as user requirements, work load, fault mode, and computing resources, and states in the system-configuration state space represent choice of data processing algorithm and resource configuration. The quality of match between operating-condition state and system-configuration state is represented by the match/anomaly function space.

In the figure, it is assumed that an extraneous condition has caused a change in the operating-condition space from state OC_i to OC_j. For a given initial system-configuration state, SC_p, the match function value, will then change from $M(OC_i, SC_p)$ to $M(OC_i, SC_p)$. It is assumed that the distance between the two match functions is sufficient to trigger a change in system configuration state from SC_p to SC_q, with a resulting change in match function from $M(OC_j,SC_p)$ to $M(OC_j, SC_q)$. In making a change in configuration state,

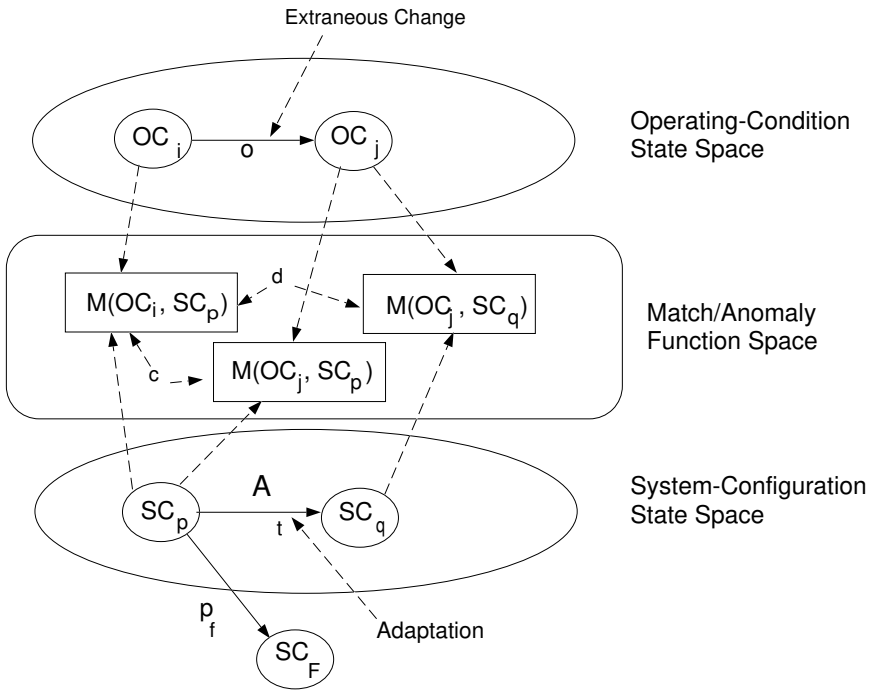


Figure 2.7: State model of adaptive systems

it must be considered that a harmful transition to a degenerate state, identified as SC_F, may occur; the degeneration may be static, for example, an unrecoverable partition or loss of control integrity, or dynamic, for example, a thrashing among adaptation states that interferes with delivery of service.

This model allows a buyer and vendor of an adaptive fault-resistant system to specify adaptation behavior in terms of certain relationships among the states and state transitions; for example, it may be specified that

- t, the transition time from SC_p to SC_q, must be less than some value T.
- c, the difference in quality of match following a change o in operating states sufficient to cause a change in configuration, must be less than some value C.
- d, the difference in quality of match following an adaptation, must be less than the initiating mismatch c, by some value D.
- P_f, the probability of failure into a degenerate state, must be less than some value P_d.

Such models are useful in exposing the performance and reliability issues that must drive a practical design. The model also may help a customer and vendor to negotiate the difficult trade-offs of a system design. The model can serve as a point of reference for evaluating the success of an adaptive design. This particular model abstracts out the performance of an adaptive system; other models are needed for that important property.

2.4.2 A Performance Model for Adaptive Systems

Markov models have been used to predict the behavior of fault-tolerant systems, given probabilities of elementary events, such as faults and fault recovery actions. When the behavior of interest is reliability or availability, the model is used to predict expected failure-free life from initialization (reliability), or mean service time (availability). Such models also have been used to study performability, which is the expected performance of a system that is subject to temporary or permanent loss of resources.

The model depicted in Figure 2.8 is a Markov model for systems that adapt to changes in fault type and work load. As with classic Markov models, it describes a system as a set of states and state transitions. Each state of the described system assumes some level of computing resources and a level of performance efficiency for those resources. The following events are represented by transitions in one of three dimensions, depicted by the three-axis cartoon:

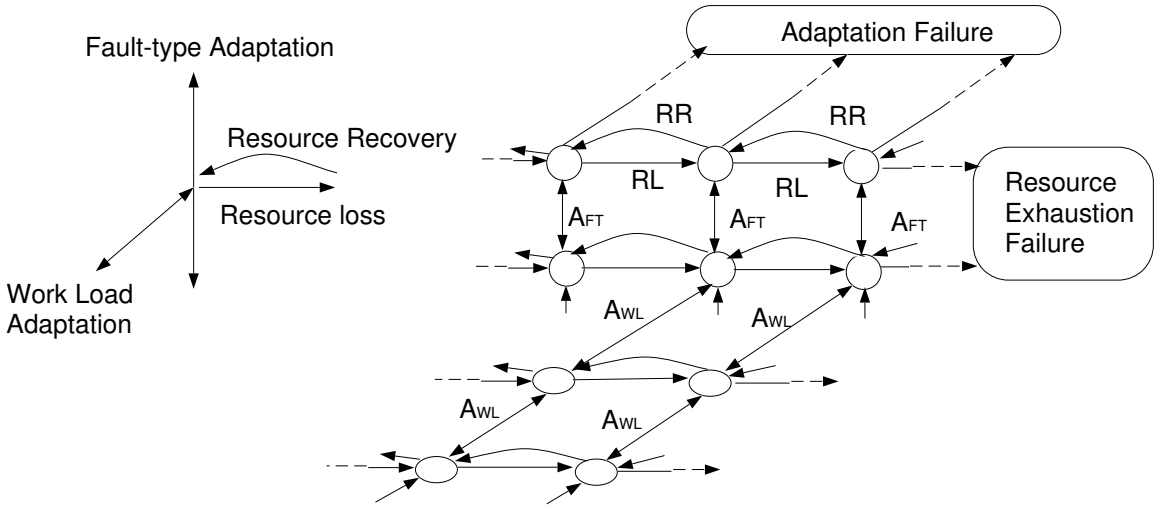


Figure 2.8: Markov model of fault and work load adaptation

- Resource loss because of a transient or permanent component failure
- Resource recovery, either intrinsic, following a transient fault, or logical, provided by a fault tolerance mechanism
- Fault-type adaptation, intended to improve the effectiveness of the system's fault-tolerant responses to changing fault types
- Work load adaptation, intended to improve the computing effectiveness of a system's resources for a changing work load

The behavior of the system may be visualized as an ongoing transition between nodes of the state graph resulting from the system's response to faults, changing fault types, and changing work loads.

Two kinds of system failures are recognized. The first failure results when the computing resources are exhausted because of unrecovered component failures. The second failure is a breakdown of integrity of the system because of improper adaptation, such as sending the system into a degraded computing configuration, or entering into an unproductive series of adaptations.

Such a model may be useful both for absolute predictions of performance and failure and for comparisons of alternative designs. Getting realistic numbers for the state transition rates may be expensive, but some estimation of these rates will be necessary to justify the effectiveness of a proposed design.

2.5 A General Adaptation Scheme

We have found that the adaptive control model used in modern control system theory is quite useful in characterizing the behavior of adaptive computer systems. Issues of stability and the use of models to analyze and predict behavior are translatable with little essential difference. A simplified illustration of such a control system model is shown in Figure 2.9. The controller uses models of the system and the environment to translate goal commands into control commands for the system under control. The controller also updates the models using the difference between the predictions and the observations. For simplicity, the illustration does not assume predictions of the control goal, but that is entirely feasible.

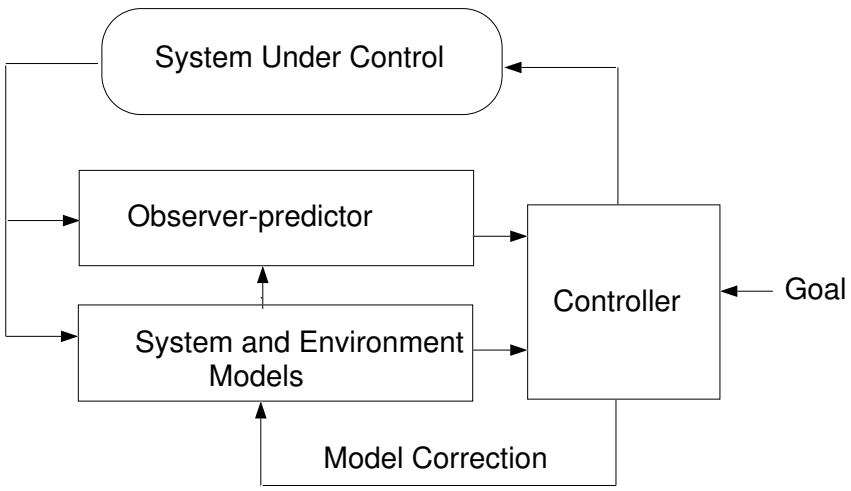


Figure 2.9: A model-based control system

Figure 2.10 shows a general architectural scheme for adaptation, which translates the general scheme of an adaptive control system into relevant computer objects. The system under control is represented as a current fault-tolerant implementation of a set of user requirements for logical service, performance, and dependability, including a function that reports the system's service behavior; the system is driven by work load data and by faults. The implementation is governed by the fault tolerance scheme selection and control function of an adaptation controller. That function is driven by two sources: (1) an adaptation control function, and (2) a report on the inventory of available computing resources. The

adaptation control function integrates user requirements for performance and dependability and a diagnosis of the behavior of the system under control, both its current state and predictions based on environmental data.

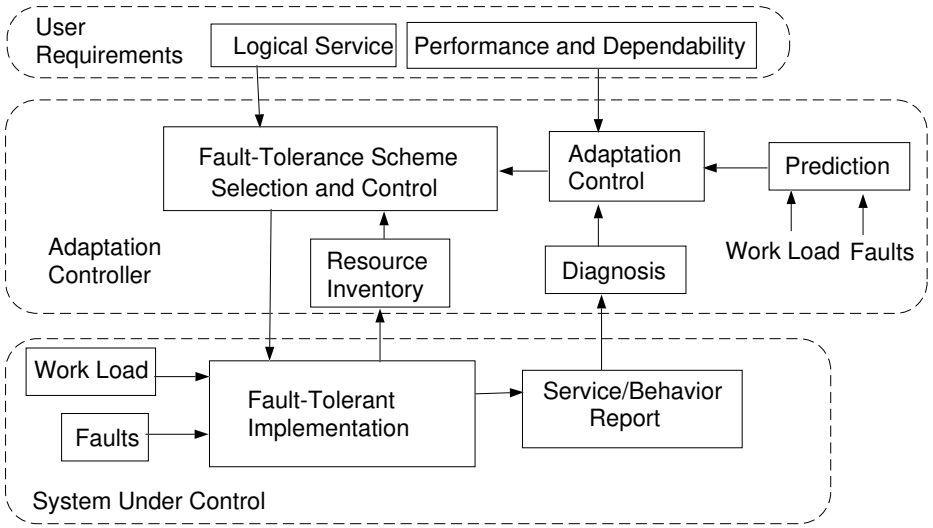


Figure 2.10: A general adaptation scheme

The scheme is formally equivalent to a control system in which a control function, derived from the system output, is applied to a system in order to make it meet some externally specified goal in the presence of environmental disturbances. In the adaptation scheme, the goal is to satisfy user requirements for logical service, performance, and dependability. The external disturbances are changes in user requirements, work load, faults, and available computing resources. Adaptive versions of such control systems, such as the Kalman Filter, provide for the prediction of changing environmental and system properties based on continually updated models. The same paradigm is appropriate for an adaptive computing system.

We note that in the general scheme as described, the system and the controller are monolithic. We have noted earlier some of the problems of distributed systems—for example, the need to bound the propagation of adaptation effects. In Section 2.9, we discuss how the notion of reflective architectures may be applied to layered systems.

Some key issues in the realization of this general scheme are strategies for adaptive implementations, techniques for selecting implementations to meet attribute requirements, and incremental techniques for diagnosis and control.

2.6 Adaptation Strategies for Fault Tolerance

The set of modifiable fault tolerance techniques, illustrated taxonomically in Figure 2.11, is quite rich. We note three broad classes of modification, alternative fault-tolerant algorithmic scheme selection, alternative service algorithm selection, and parameter variation. Fault-tolerant algorithm options include (1) the fault anticipation policy of an algorithm, which

may be optimistic or pessimistic anticipations about the likelihood of occurrence of a fault during the algorithm's execution, (2) recovery policy, which may be of the forward or backward type, (3) management of group redundancy—in practice, the use of primary/backup or consensus fault masking, and (4) choice of fault isolation or error masking. Service algorithm options will affect the amount of resources required for user service, and indirectly affect the amount of resources available for fault tolerance. Options include (1) the types of concurrency–serial or parallel, (2) the degree of distribution of control—centralized or distributed, (4) the logical model for processing—such as imperative, functional, model-based, or rule-based, (5) the balance or mix of processing, storage, and communication resources used by a given algorithm, and (6) whether processing is optimized at design and compile times, or at run time.

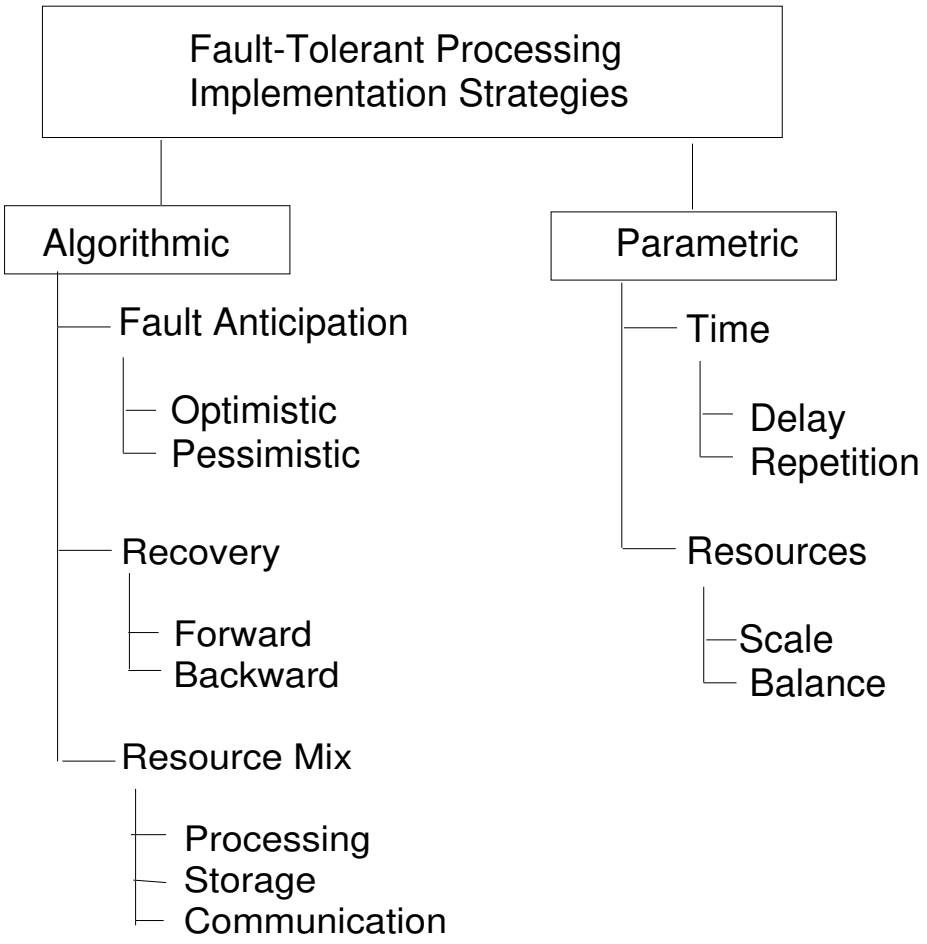


Figure 2.11: Implementation strategies

Parametric changes may include modification of (1) time parameters, such as the time allowed for a failed but possibly recoverable process to recover, or the number of repetitions attempted before a fault process is considered to be unrecoverable, and (2) resource parameters, such as the level of redundancy applied to a given fault tolerance scheme, or how load is distributed among a set of resources.

These few examples indicate that there is little limitation in finding feasible strategies for adaptation. There are, however, several important challenges in selecting and applying sets of fault tolerance techniques, such as

- When should changes be made?
- How can changes be made safely (to avoid failure or thrashing), incrementally (to allow assessment of correctness of the diagnosis that triggered the change), and reversibly (to cope with incorrect adaptation decisions)?
- How can the overhead cost of multiple techniques be minimized?

In Section 2.8.3, we argue that the amount of change attempted in an adaptation should be a function of the confidence in the diagnosis of mismatch conditions. This implies that useful adaptation strategies should not only offer many alternatives in fault tolerance technique, but that the techniques should allow different levels of change, and that if an adaptation proves to be unfruitful or harmful it should be possible to restore the system's initial configuration with minimum loss. Overhead cost includes the performance cost of the selection mechanism and the possible temporary reduction in service during changes.

These criteria indicate that despite the abundance of alternative techniques for modifying system implementation, the selection of sets of economical, safe, and effective alternative techniques is far from trivial.

The use of parametric changes clearly provides opportunity for making incremental changes. The method of changing algorithms, while discrete, also may be applied incrementally. For example, if there is some uncertainty about the nature of the current fault type, a small fraction of the current workload might be processed with a different algorithm. Success would encourage increasing that workload fraction incrementally.

2.7 Attribute-Based Technique Selection

In the ADRB example in Section 2.3, the different modes provide the designer with different levels of service attributes such as resource utilization, fault-recovery time, and reliability. The different attribute levels may be helpful in responding to changes in user or operator requirements, but they also may constitute a difficult challenge in satisfying a given set of requirements with diminished resources; a given scheme may satisfy one attribute, but not another. The general adaptation scheme calls for the adaptation controller to find a fault-tolerant processing scheme that best satisfies an operating condition. This is essentially a problem in design, but one that must be solved at operation time.

To solve this problem, we assume, as shown in Figure 2.12, that a human designer generates, as part of the design process, a set of fault-tolerant processing techniques to cover the expected range of changes, and for each technique derives a characterization of all pertinent service attributes. Generally, such a characterization will be parameterized, showing the effect of applying different levels of resources to the technique. For example, the technique of multichannel, majority-logic voting may be characterized by throughput, latency, reliability, and availability functions, parameterized by the number of channels.

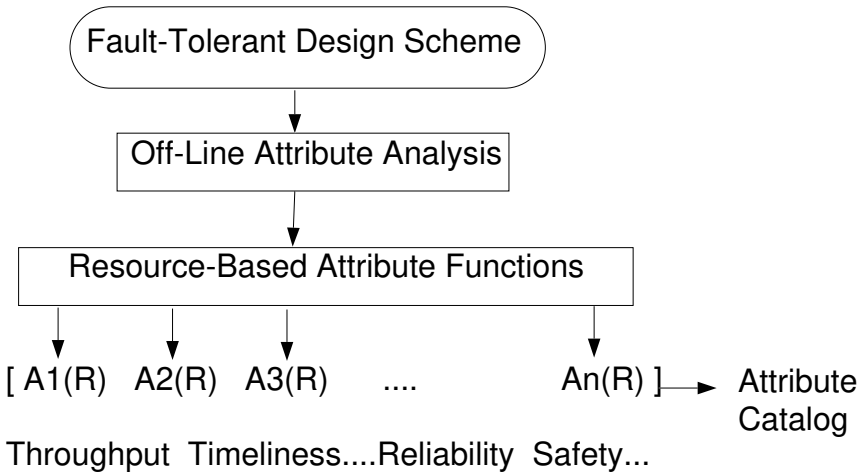


Figure 2.12: Design-time analysis of technique attributes

The resulting attribute profiles will be stored in the adaptive controller for use at run time. Figure 2.13 illustrates how a set of service attribute requirements may be evaluated to determine if a given technique is feasible for serving a particular set of run-time-developed service attribute requirements. The feasibility of a technique is assessed for varying levels of resources for which the scheme has been parameterized. That technique should be selected that satisfies the given attribute requirements with the lowest assignment of resources.

The assignment of resources should satisfy global requirements as well as the demands of the current adaptation circumstances; that is, it may be necessary to compromise some requirements in favor of others.

2.8 Diagnosis and Control

In defining our approach to diagnosis and control of adaptive fault-tolerant systems we examine the general role of diagnosis in system control, discuss a very simple diagnosis and control scheme, and present a more general approach to diagnosis and control.

2.8.1 The Role of Diagnosis in Adaptive Control

The function of adaptation is to execute a change in the way a computing system accomplishes a specified service. This may be seen as a kind of computation in which the data is not an observation of the real world, but rather a concrete characterization, or *diagnosis* of the effectiveness of the current service implementation—the product of the computation is a new implementation. In control theory, such characterization is called *System Identification* [29]. Implementation-characterization, which we will refer to as *diagnosis* in the remaining discussions, is one of the central issues in adaptation, because it, together with the service requirement, provides the information as to what problem an adaptation must solve.

Diagnoses should be fast and accurate in order to both achieve the highest level of service implementation and to avoid instability in making changes. Speed is important to avoid instability caused by excessive lags in following rapid changes in the environment.

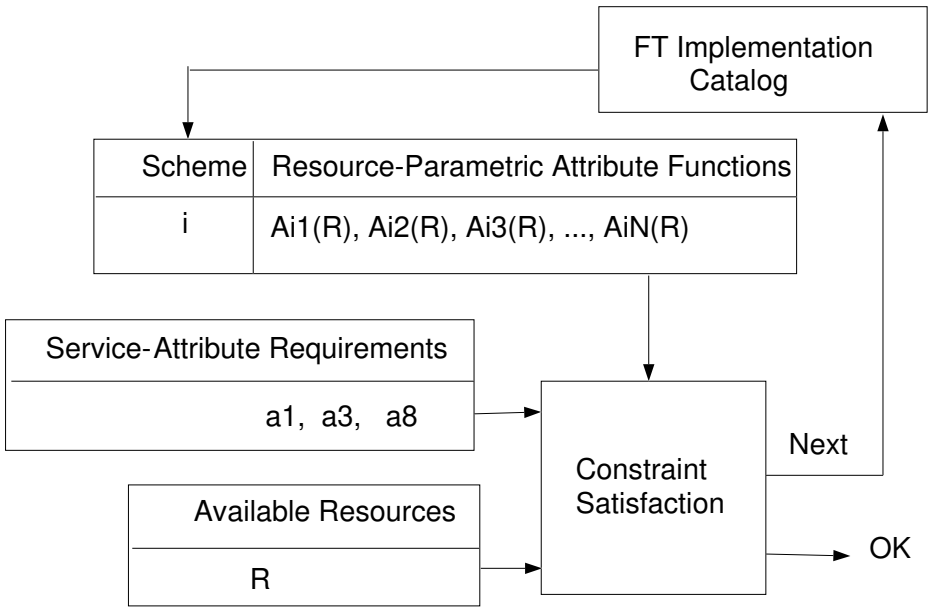


Figure 2.13: Attribute-based technique selection

Accuracy is important to avoid corrections that cause the system to depart from, rather than to approach, the desired state and hence diminish the prospects and increase the rate of convergence to the correct new system state.

We suggest two approaches to diagnosis and control. The first is a very simple scheme, in which diagnosis is a threshold function, with hysteresis, driven by a single, filtered adaptation variable. The second is a more complex scheme that aims to deal with ambiguous evidence about system effectiveness and, furthermore, to obtain high-quality diagnoses as rapidly as possible.

2.8.2 A Simple Control Scheme

A simple control scheme is illustrated in Figure 2.14. In this scheme, some adaptation variable, say the occurrence of errors, is observed and smoothed with a low-pass filter. The result is tested by a threshold detector with hysteresis—that is, a change is triggered when the variable rises above the high threshold and falls below the low threshold. The filter tends to produce an output only when there is a long-term shift in the effectiveness of the system service implementation, and the hysteresis serves to avoid changes due to momentary variations.

The scheme allows some parameter setting by a higher level of control—including the time constant of the filter and the level of the two thresholds.

This simple scheme might be used for low-level system adaptation, where the data about adaptation effectiveness is relatively unambiguous. For example, two nodes may communicate using either an optimistic or a pessimistic protocol, depending on the error rate. In this case, the error rate is the only adaptation variable of interest, and the choice of implementation modes is simple. This simple scheme would be satisfactory for a Poisson

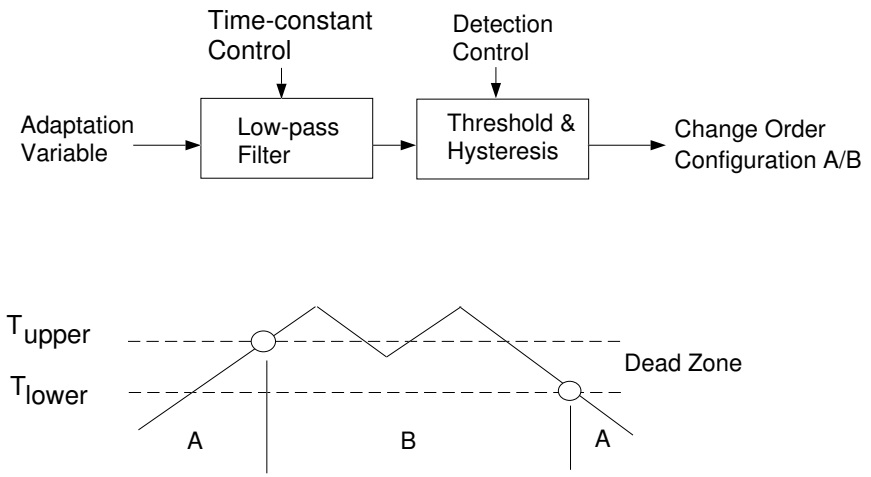


Figure 2.14: A Simple Control Scheme

error distribution, but it might not be satisfactory if the distribution is bursty or has predictable fluctuations.

2.8.3 Incremental Diagnosis and Control

In complex systems, information about the effectiveness of a service implementation may not be easy to interpret. In analyzing service anomalies, it may be difficult to distinguish the contributions of faults and service overloads, and to distinguish the various types of faults and overloads themselves. A given anomaly may have many explanations, whose remedies may be very different, and even opposed. The problem is complicated by the fact that causes usually cannot be observed directly, and can only be inferred from weak and fluctuating evidence.

A further difficulty arises from the need to make decisions about adaptations as soon as possible after operating changes occur, in order to maintain performance and avoid possible catastrophic failure.

In response to these difficulties, we suggest a diagnosis approach that has three components:

- Multiple concurrent diagnoses, to deal with ambiguity in error evidence
- Prediction, to maximize a priori knowledge about environmental variations
- Incremental decision making, to achieve most rapid possible adaptation to rapidly-changing operating conditions

Multiple Concurrent Diagnosis and Incremental Decision Making

These approaches are illustrated in Figure 2.15. An incremental and differential diagnosis unit observes error reports from the system under control and presents one or more possible diagnoses, each with some measures of confidence and precision (specificity of the size of

the region that contains the fault). Diagnoses are attempted at every error report, and as new evidence is obtained, the likelihoods of the several candidate fault diagnoses may be modified (shown by the feedback of weight adjustment) in accordance with the changing weight of evidence for each fault hypothesis.

As faults change from one type to another, the level of confidence in the current diagnosis should change, initially decreasing as new evidence arrives that is inconsistent with the prevailing theory, and then increasing as new error information arrives and new hypotheses are strengthened. At some point in the transition between dominant hypotheses, the adaptation controller will have to decide when the confidence level in a new hypothesis justifies a change in fault tolerance technique, and, for any given confidence level, what amount of commitment of the system to a new technique is justified.

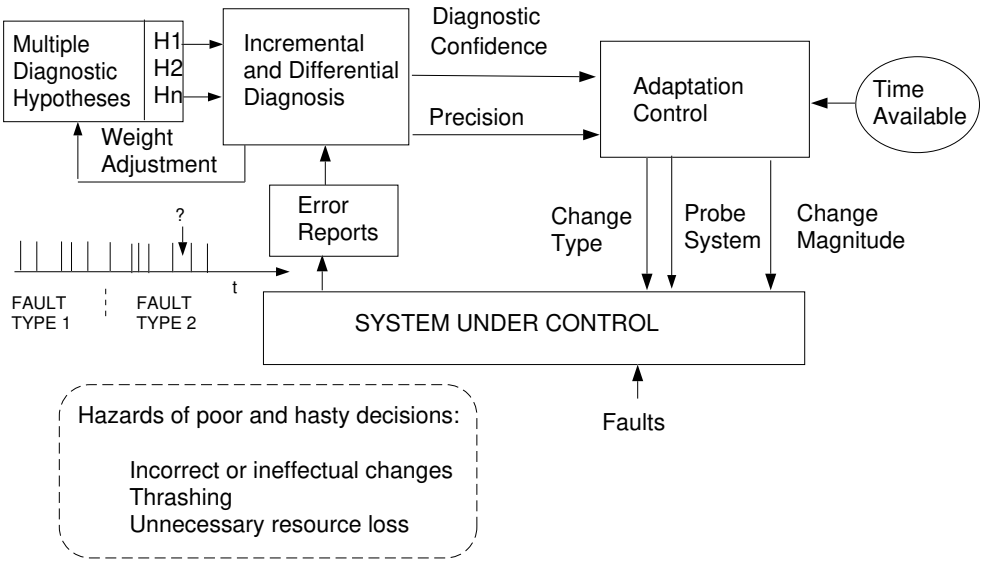


Figure 2.15: Incremental diagnosis and control

Given the uncertain and data-dependent arrival of error information in real systems, we assume that the adaptation controller is capable of periodically probing the system with tests to quickly reveal the presence of faults.

There is surprisingly little literature on the subject of real-time diagnosis—that is, diagnosis that attempts to achieve useful analysis of a system that operates within a changing data and fault environment. Most diagnosis results are given for static situations—that is, situations in which time is not a limiting factor in the analysis. For most diagnosis technology, the problem to be solved is to determine complexity–the number of tests required to analyze a system of a given size, or coverage—and the fraction of faults that may be uncovered for tests of a given length. By contrast, the problem of real-time testing is to determine the function relating accuracy of testing and number of observations, in order to allow the earliest possible estimation of system state.

Prediction for Diagnosis and Adaptation Decisions

Modern control systems make substantial use of prediction in generating control commands. Predictions are based on some model. In simple control systems the predictions may be based on a fixed model—say a filter that integrates recent observations. In advanced, adaptive control systems, the model is dynamic—that is, it is updated by comparing the predictions it generates with experience.

Some opportunities for prediction in distributed computing systems are

- Prior knowledge about the operation, including the dynamics of data arrival for the normal working day and for different operating situations
- Knowledge about trends in fault behavior, such as gradual breakdowns of storage and communication media, models of crash behavior, models of overloads and recoveries, and histories of failures of particular subsystems
- Knowledge about user priorities for work in different operational situations

Such knowledge can be very valuable both in diagnosing system conditions on the basis of partial evidence, and in deciding whether or not an adaptation is justified.

For diagnosis, prediction may help to distinguish random from burst faults, and physical from design faults, which can be crucial in chosing fault tolerance remedies. For adaptation decisions, it is clearly beneficial to avoid system changes when, after environmental changes are detected, it can be predicted that changes are only temporary.

Benefits of fault and environmental prediction have been discussed informally in the fault tolerance research literature, but there is no existing theory or systematic methodology for exploiting it in practical systems.

Multiple-dimension diagnosis

Aspects of the operating environment such as faults, workload, resources and requirements, may be characterized individually by various criteria. For example, faults may be characterized by type and by spatial and temporal distribution; workload may be characterized by rate, object size, and spatial concentration, and so on. Quantified indicators exist that can serve as the basis for adaptation for any single aspects..

Since a given set of resources must be configured to serve all of these aspects, the best use of the resources would likely result from using a characterization that integrated all relevant dimensions. The use of techniques from the fields of pattern recognition and neural networks may be useful here. Neural techniques offer the additional possibility of learning from actual system experience.

2.9 Reflective and Hierarchical Architecture

A recently developed principle of system hierarchy, called *reflection*, can be used for organizing adaptive control and techniques for managing adaptation in hierarchical systems.

2.9.1 Reflective Architecture

The general adaptation scheme discussed previously assumes that both the system under control and the adaptation controller are monolithic. In practice, designers employ a layered structuring for their systems in order to manage complexity, and we expect that practical adaptive systems will be so layered. We further expect that the control logic required for adaptation will itself be so complex as to justify some degree of layering. We find the relatively new notion of reflective architecture discussed in recent literature to be attractive, both for the layering of adaptive control and for applying adaptive control to layered systems.

Reflective architecture is based on a hierarchical relation, which is illustrated in the first part of Figure 2.16. The figure shows layer R observing layer Q's behavior and subsequently directing Q to change the way it implements some function, such as fault tolerance. The same principle has been applied to balancing load in a multiprocessor. The second part of the figure shows a combination of a conventional Uses-based hierarchy and a reflective hierarchy. Layer C uses layer B as, for example, an application program uses an operating system utility — but layer B is subject to the control of a reflective hierarchy, shown as B(R1), B(R2), and so forth.

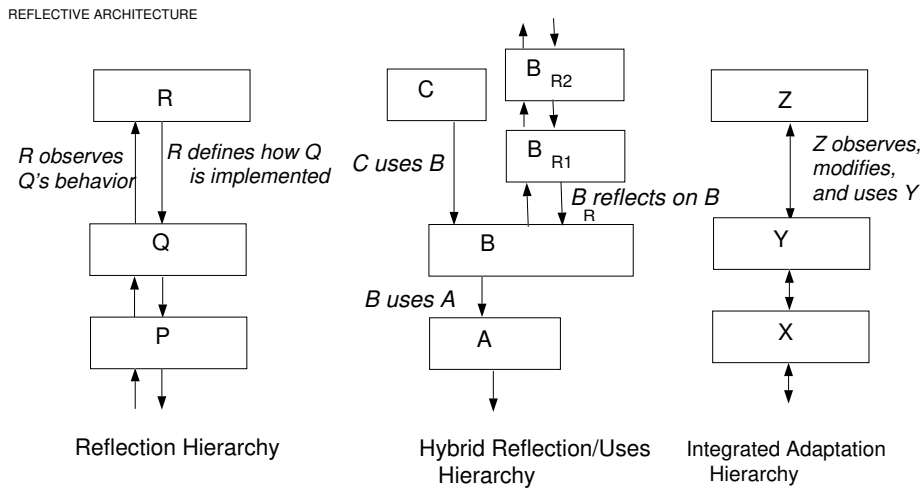


Figure 2.16: Reflective architecture

The B() hierarchy separates out several adaptivity concerns; for example, layer B(R1) may be concerned with selecting appropriate fault tolerance schemes, while layer B(R2) may be concerned with when an adaptation should be attempted; that is, it moderates the action of B(R1). Layer B(R3) might be concerned with how aggressively an adaptation should be carried out, given the current level of diagnostic confidence and the current user policy on how adaptation risk and service urgency are to be balanced. Such layering of adaptation concerns may help to simplify adaptive designs and to allow orderly growth of capabilities with experience.

In some cases, the two kinds of hierarchy might usefully be integrated, as suggested in the third part of Figure 2.16, where layer Z observes, modifies, and uses layer Y.

Reflection does not solve the algorithmic problems of adaptive control, but it offers a general structure for organizing complex adaptive control functions.

2.9.2 Multilayered Changes in a Function-Support Hierarchy

In layered systems, a change of system configuration may not easily be restricted to a single layer. Figure 2.17 illustrates a layered adaptive system, where alternate functions are available at each level, and functions at one level support functions at higher levels. It is assumed that a function at one level may not be able to support all functions at the next higher level, that is, the dependencies from level to level are incomplete. In the figure as constructed, a change from function S(2,1) to function S(2,3) at level 2 is assumed to be required to accomplish an adaptation originating at level 2. As shown, such a change will require level-3 changes from function S(3,1) to function S(3,3), inasmuch as S(3,1) is not supported by S(2,3).

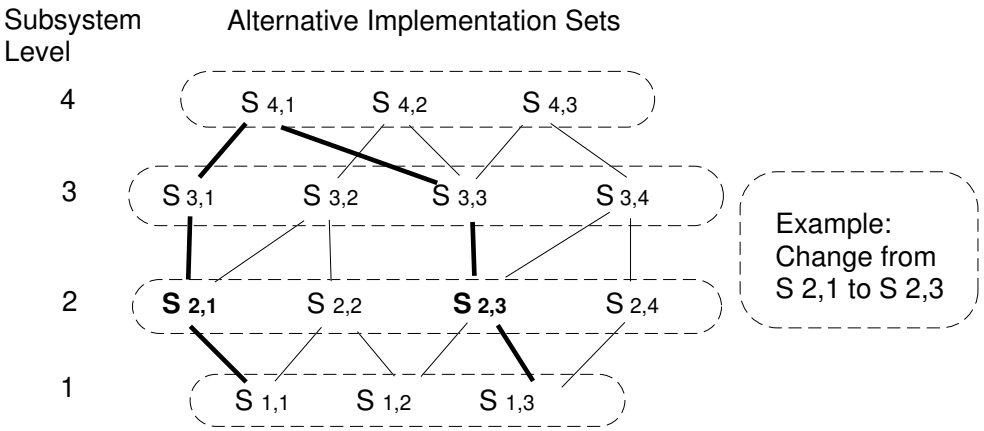


Figure 2.17: Multilayer changes for adaptation

We conclude that changes made to accomplish adaptation in a multilevel system may have to extend over several functional system layers, and that changes among levels may have to be coordinated at design time.

2.10 Results and Future System Design Issues

This review has extended and formalized the statement of general principles and discussion of examples and technical approaches presented in previous reports. We have differentiated adaptive fault-resistant systems from traditional fault-tolerant systems by emphasizing an adaptive system's need to have autonomous awareness of an anomalous mismatch between its current implementations of fault tolerance and the demands of a dynamically changing set of requirements. We have presented several schemes for design that have a high degree of generality. The following technical insights and problem areas have risen from this viewpoint

- Abstract models based on finite-state machines are available for specifying adaptive behavior and predicting performance and misadaptations.
- Process-control models are very fruitful analogues for adaptive computer system control; directly appropriate issues include gain control, prediction, instability, and adaptation failures.
- Uncertainty of evidence is a critical issue in real-time system state analysis and fault diagnosis, and has a central impact on making adaptation decisions. Useful techniques include concurrent multihypothesis diagnosis, predictive diagnosis, and incremental decision making.
- reflective architecture, developed originally to allow extendibility in computer languages, provides an excellent and general structure for adaptive architecture.
- There is an abundance of design families within which adaptation choices may be made.
- Selecting particular fault tolerance techniques to meet changing user requirements may be based on predetermined service attributes of candidate techniques.
- Similar to the need for bounding fault propagation in fault-tolerant systems, there is a need to bound (1) the propagation of adaptation responses in a distributed system, and (2) vertical propagation of changes in multilayer support functions.

We believe that these general and particular results offer a useful basis for a methodology of design for future systems.

We have used one case study, ADRBs, to illustrate the need for techniques for selecting alternative implementations based on possibly changing service attributes. Further details on ADRBs are presented in Chapters 3 and 4, and a case study on distributed thread integrity is discussed in Chapter 5.

Chapter 3

The Adaptive Distributed Recovery Block Scheme

In many challenging applications, environmental conditions that affect fault tolerance requirements imposed on computer systems change dynamically. As significant changes in environmental conditions or in internal computing resource conditions occur, the effective set of fault tolerance mechanisms also changes.

3.1 The Role of ADRBs in Adaptive Fault Tolerance

The purpose of adaptive fault tolerance (AFT) is to meet the dynamically and widely changing fault tolerance requirement by efficiently and adaptively using a limited and dynamically changing amount of available redundant processing resources [21].

When the fault tolerance requirement reaches a highly stressful state (that is, at or near the peak) in an application in which the fault tolerance requirement fluctuates widely, the processing resources available in the computer system are typically not sufficient to support all the fault-tolerance mechanisms needed without adjusting the set of services provided by the computer system. In addition, given that the resources (processing, communication, and data storage) of a computer system are finite and that under increasing stress the availability/usability of these resources will decrease because of failures, the questions are: Toward what objective will the remaining resources be directed? And is the current strategy for fault tolerance the most effective under this greater stress? The system resource manager may decide to decrease the *functionality* (that is, total set of functions supported) to maintain the level of *timeliness* (that is, the ability to produce critical responses during the required time periods) and the level of *consistency* (that is, the degree of deviation from the intended relationship among the states of the different parts of the computer system and the environment). Or it may decide to give up some consistency for functionality and timeliness. The system resource manager must, therefore, trade-off functionality, timeliness, and consistency in order to maintain an optimal system operation having decreased resources. As a part of this trade off, the set of fault tolerance mechanisms activated may need to be dynamically adjusted. Hence comes the notion of adaptive fault tolerance.

The distinctive nature of AFT as compared to more conventional fault tolerance becomes more evident when the fault tolerance requirement or the resource availability changes "in a noticeably discrete fashion," that is, from one mode to another mode. In response to such mode changes, the adaptive fault-tolerance management system (FTMS) adjusts its operating strategy accordingly, that is, entering a new mode of operating fault tolerance capabilities. Such an adaptive *multimode* FTMS is bound to have a highly modularized structure and is thus easier to implement reliably than monolithic static FTMSs that may execute all available fault tolerance mechanisms all of the time. In fact, in most challenging distributed computer system (DCS) applications, it already becomes prohibitively expensive to operate the DCS with continuous activation of the fault-tolerance mechanisms needed only in several highly stressing modes, let alone operate with all the available fault tolerance mechanisms activated. An adaptive FTMS reallocates its resources to activate a set of fault tolerance mechanisms effective in a new mode of the environment and the computing resource.

A specific instance of an AFT technique, the *adaptive distributed recovery block* (ADRB) scheme is a major extension of the basic *distributed recovery block* (DRB) scheme developed in [23, 26, 15, 25, 24]. The DRB scheme was adopted as the basic structure for designing fault-tolerant real-time DCSs because of its wide applicability and ability to handle both hardware and software faults with no loss of real-time task executions. One fundamental software approach to realizing real-time fault tolerance capabilities in DCSs is parallel redundant execution, which is to have multiple processing nodes execute the same real-time task in a redundant fashion. The DRB scheme is a practical and broadly applicable formalization of the parallel redundant execution approach. The scheme is essentially an approach to structuring a duplex redundant computing station, called a *DRB station*, dedicated to execution of one or a few real-time application processes and capable of handling both hardware and software component failures with the effect of real-time forward recovery.

The ADRB scheme extends the DRB scheme in two major ways. First, a critical real-time task can be executed not only (1) in *the parallel redundant mode*, which is the standard mode used in the basic DRB station, but also (2) in *the sequential backward recovery mode*, which is the execution mode adopted in the original *recovery block scheme* [19, 40], and (3) in *the sequential forward recovery mode*, which has been considered in many previous projects on exception handling. Therefore, an *ADRB station* dynamically switches its operating mode in response to significant changes in the resource and application modes. Secondly, the supervisor station under the DRB scheme is basically responsible for three functions: detection of node crashes, detection of misjudgments by the nodes in DRB stations about the status of their partner nodes, and network reconfiguration including task redistribution. Under the ADRB scheme the supervisory function is not necessarily concentrated in a particular node or computing station. Moreover, the supervisory function station has an additional dimension—that is, changing the set of real-time tasks to be executed. The supervisor function can be executed not only in the centralized mode but also in the decentralized mode. Therefore, again, the system can dynamically switch between the centralized supervisory mode and the decentralized cooperative monitoring and control mode. The algorithms and execution modes for accomplishing the three basic functions can thus be adjusted as significant changes in the resource and application modes occur.

Although the basic DRB scheme has been evolving over the past ten years, exploration

of various possible implementation structures has thus far taken place more in the context of highly parallel multicomputer networks [26, 25]. Only in recent years have some concrete implementation structures and prototypes of DRB stations for use in real-time LAN-based systems been studied [15, 24]. In order to partially validate one version of a DRB implementation structure and identify detailed implementation issues, a simple experimental implementation of DRB stations in a small-scale LAN-based DCS testbed was conducted.

issues are discussed in Section 3.2.7.

3.2 Basic Principles of the DRB Scheme

The most basic and important problem in constructing a real-time fault-tolerant DCS is to construct highly reliable and fault-tolerant constituent computing stations. One approach to realizing this is by parallel replicated execution of real-time tasks. A practical and basic approach that keeps the amount of data communication between replicated processing nodes to a minimum is to structure a computing station in the form of a *pair of self-checking processing (PSP) nodes*, each processing node possessing the capability of judging the reasonableness of its task execution results. The PSP scheme is a core component of the DRB scheme.

We concentrate here on two instances of the PSP scheme. The first scheme is intended to tolerate primarily hardware faults (although some operating system faults are also tolerated) by using identical, replicated software and hardware. The second scheme tolerates software faults as well as some hardware and operating system faults, by using nonidentical software (intended to produce equivalent results for the same inputs), running on identical hardware.

3.2.1 Primary-shadow pair of self-checking processing nodes

An abstract representation of a PSP-structured computing station is given in Figure 3.1.

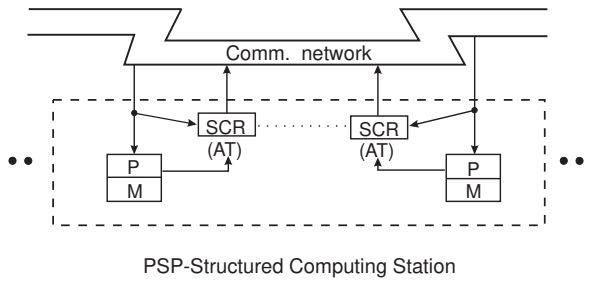


Figure 3.1: PSP-structured computing station

There are largely two basic approaches to implementing the self-checking functions, one through hardware support and the other in software. Self-checking hardware mechanisms have been extensively developed [47, 51]. Self-checking software mechanisms are not necessarily substitutes for self-checking hardware mechanisms, but rather the former can be supplements to the latter. One of the most versatile and flexible self-checking software mechanisms is the *acceptance test*, which is a routine for checking the acceptability of the execution results of a task [19, 40]. Use of this mechanism, possibly in conjunction with

some self-checking hardware mechanisms, is the approach adopted in the DRB scheme. It can be viewed as an instance of an executable assertion.

Figure 3.2 illustrates a PSP-structured computing station based on the self-checking function implemented in the form of an acceptance test routine. Each node has its own local database. Node A is the initial *primary node*, and node B is the initial *shadow node* within this computing station. Each node has its own local data base for application processing. Both nodes obtain input data from the multicast channel (built on a system-wide multiaccess communication network), and the primary node A informs the shadow node B of the ID of the data item that node A selected for processing in the current task execution cycle. Nodes A and B process the data and perform their self checking concurrently by using the same acceptance test routine. Because node A passes the test here, it informs the shadow node B of its success and then delivers the results to the successor computing station(s). Upon successful delivery, node A informs node B of the success, and both nodes proceed to enter the next task execution cycle.

Processing of task input data usually requires reference to a database. It is essential that the database be protected from unacceptable results that may be generated by either of the processors. There are several ways to achieve this protection. One way is to provide some means for undoing a change to a local database–say by using a buffer that records all changes as a local *state vector*. Another way is to use an external persistent store for recording all acceptable results. The second way is simpler to manage, but it imposes the burden of downloading data into a processor for every new task.

It also may be desirable to protect the local database information from loss resulting from processor faults. In this case, a separate data base may be provided for each processor. For this solution, it must be possible to copy the contents of the unfailed database into a corrupted database.

Suppose that the PSP-structured computing station in Figure 3.3 is the successor station. Upon receiving data from the predecessor station (in Figure 3.2), the primary node C informs node D of the data item to process next, for example, via transmission of the ID of the data item. The nodes process the data and perform their self checking concurrently, but this time the primary node C fails while the shadow node D passes. Node D will learn of the failure of node C via either an explicit notice from node C or a time-out (if node C has crashed). Node D then becomes the new primary node, delivers its task execution results to the successor computing station(s), and notifies the partner node, if alive, of the successful delivery. Meanwhile, node C, if alive, attempts to become a new shadow node by trying again to process the data item. If node C passes the self-checking test this time, it can then continue as a shadow node; upon learning of the successful delivery of the result by the partner node D, it proceeds to the next task cycle as the shadow node.

To realize the full potential of this PSP scheme, cost-effective mechanisms must be provided for ensuring that all versions get the same data in each task execution cycle, and for reliably saving some relevant task execution results into persistent store upon successful completion of acceptance tests. These issues are discussed in Section 3.2.5.

This *primary-shadow* scheme is thus capable of handling hardware faults with the effect of real-time forward recovery and is a core component of the DRB scheme.

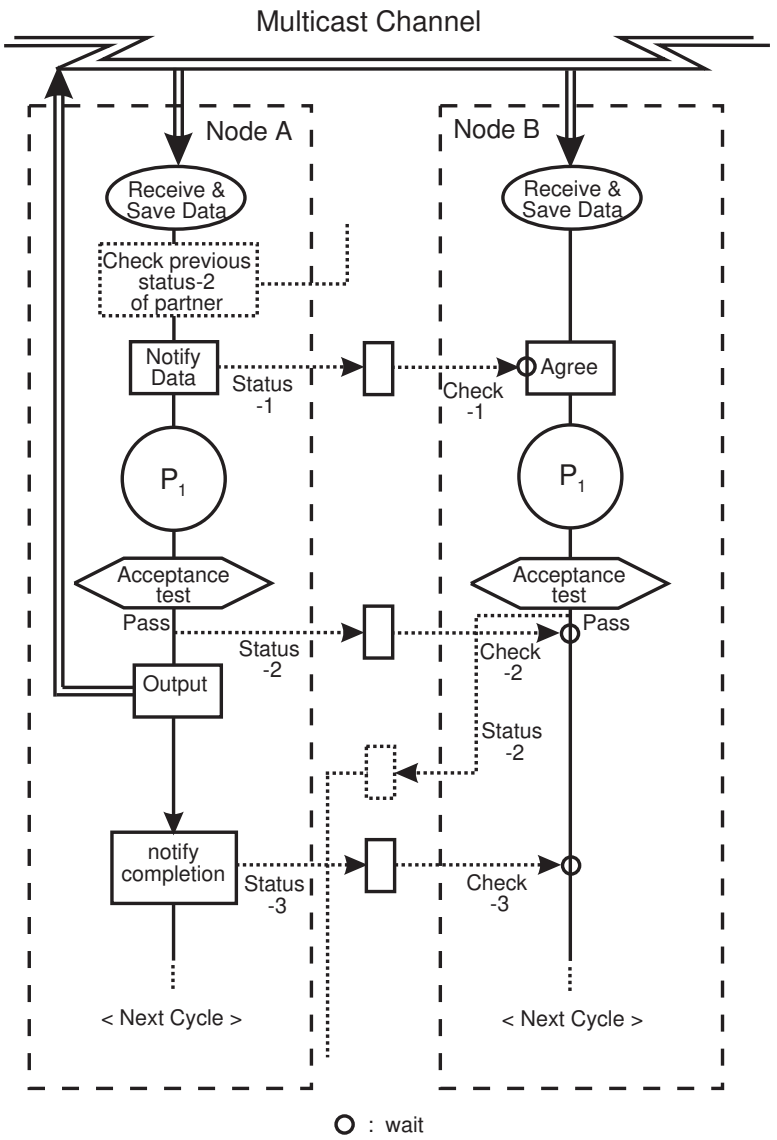


Figure 3.2: Detailed view of a PSP-structured computing station

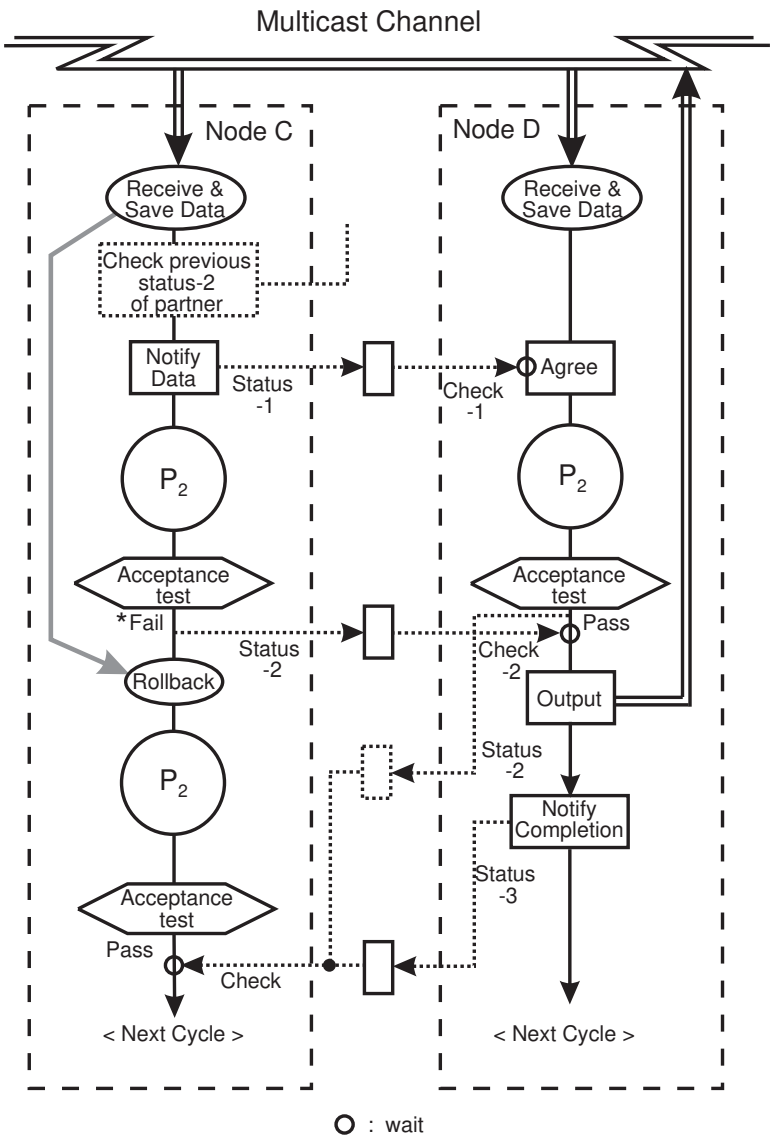


Figure 3.3: Successor PSP-structured computing station

3.2.2 Replication of recovery blocks

If the system designer is concerned with not only hardware faults, but also with software faults, the primary-shadow scheme discussed with Figures 3.2 and 3.3 can be extended by incorporating the idea of using multiple versions of the application task procedure for each task. These multiple versions and a task-specific acceptance test related to the same task can be structured together in the form of a *recovery block* (RB) [19, 40].

The syntax of a recovery block is: *ensure* T *by* B1 *else by* B2 ... *else by* Bn else error. Here, T denotes the *acceptance test* (AT), B1 the *primary try block*, and Bk, $2 \leq k \leq n$, the *alternate try blocks*. All the try blocks are designed to produce the same or similar computational results. The acceptance test is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A *try* (that is, execution of a try block) is thus always followed by an acceptance test. If an error is detected during a try or as a result of an acceptance test execution, then a rollback-and-retry with another try block follows.

The extended scheme is the *distributed recovery block* (DRB), and under this scheme a recovery block is replicated into multiple nodes forming a *DRB* station for parallel redundant processing. In most cases, a recovery block containing just two try blocks, the primary and the alternate, is designed and then assigned to two different nodes, the primary and shadow nodes, as depicted in Figure 3.4. The specification of the maximum execution time allowed for each try block is an integral part of the DRB scheme. A try block that is not completed within the time, because of hardware faults or excessive looping, is treated as a failure. Therefore, the acceptance test can be viewed as a combination of both *logic* and *time* acceptance tests. The roles of the two try blocks are assigned differently in the two nodes. The governing rule is that the primary node tries to execute the primary try block whenever possible, whereas the shadow node tries to execute the alternate try block. Therefore, primary node X initially uses try block A as the *first* try block, whereas shadow node Y initially uses try block B as the first try block. Until a fault is detected, both nodes receive the same input data, process the data using two different try blocks (that is, block A on node X and block B on node Y), and check the results using the acceptance test. Both nodes perform all these tasks concurrently. The *time acceptance test* (that is, the time-out mechanism) is used to ensure the timely behavior of both nodes.

In a fault-free situation, both nodes will pass the acceptance test with the results computed with their first try blocks. In such a case, the primary node notifies the shadow node that it successfully passes the acceptance test. Thereafter, only the primary node sends its output to the successor computing station(s). Both nodes then proceed to the next task cycle. However, if the primary node fails and the shadow node passes its own acceptance test, the shadow node assumes the role of the primary node; that is, the nodes exchange their roles. These actions by the two nodes are done asynchronously as explained in Figure 3.3. On the other hand, if the shadow node fails first, the primary node need not be disturbed. In both cases, the failed node attempts to become an operational shadow node; it attempts to roll back and retry with its second try block to bring its application computation state including its local database up to date. This attempt does not disturb the primary node.

The DRB scheme imposes some restrictions on the use of the recovery block scheme. A recovery block to be used in the DRB scheme should be two-phase structured; it should

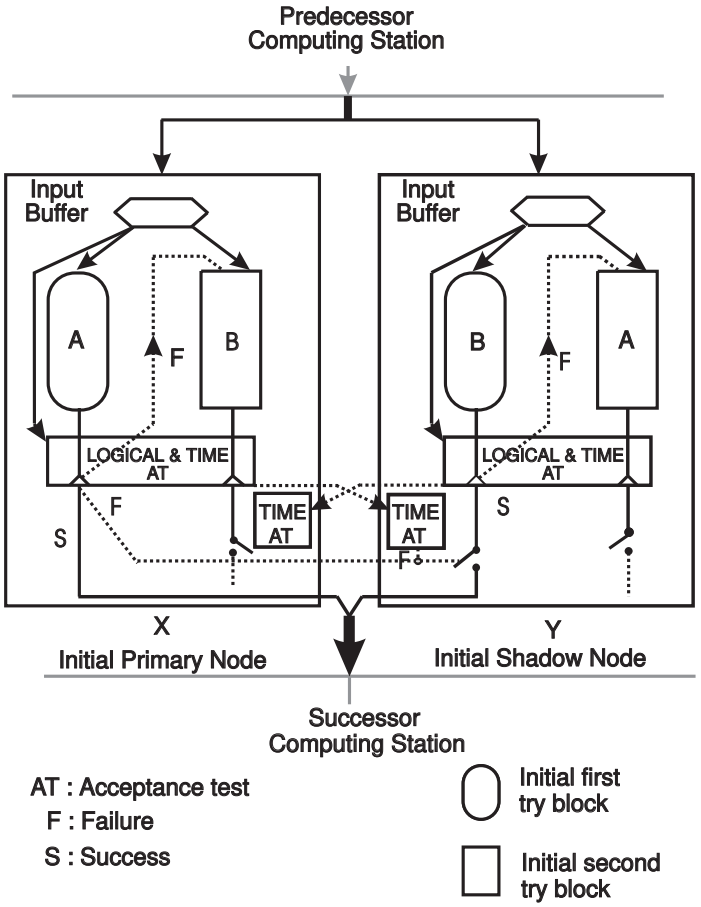


Figure 3.4: A DRB combines PSPs and replicated RBs

consist of one input acquisition phase and one output phase. During the input phase, the recovery block must not involve any output step (i.e., sending computation results to the outside) while it may involve multiple input steps. Similarly, during the output phase, the recovery block may involve multiple output steps but not a single input step. This restriction is essential to prevent interdependency among different DRB stations for recovery from being formed.

There is also the possibility that the two local databases, each belonging to a different partner node, may diverge in their contents. The goal of the DRB scheme is to keep these local databases in acceptable states. If the acceptability criterion used here is a rigorous one and if the two local databases are in acceptable states, the differences in the contents of the local databases will be limited by the acceptability criterion, and hence not problematic. Therefore, the quality of the acceptance test in a DRB station is very important. Fortunately, experience has indicated that design of good-quality acceptance tests is much easier in real-time application fields than in non-real-time data processing applications.

Under the DRB scheme, real-time forward recovery is achieved regardless of whether faults occur in the hardware or software components. During fault-free operation, the execution overhead is very small because the actions of the primary node do not depend on those of the shadow node. By adopting the RB structuring as its component, the DRB scheme supports flexible incorporation of algorithmic redundancy because the two try blocks are not required to produce identical results and the second try block need not be as sophisticated as the first try block. When the designers cannot accommodate the costs of developing alternative try blocks, they can still use the PSP portion of the DRB scheme to facilitate real-time hardware fault tolerance.

3.2.3 Recursive shadowing with N (> 2) try blocks

In some highly safety-critical applications, the system designer may design more than two try blocks into a recovery block to further increase reliability. Although several approaches to structuring a DRB station that uses three try blocks are conceivable, one of the most natural approaches is *recursive shadowing*, which is to treat the third node as a shadow node for the team of the first two nodes as depicted in Figure 3.5 [25].

Node Z in the figure will normally use try block C as its primary try block and deliver its results only when both X and Y fail to produce acceptable results in time. Nodes X and Y behave like a single functional node with respect to interfacing with their shadow node Z. They must share responsibilities for providing their status information to node Z at various points as well as responsibilities for understanding the "useful/useless shadow" status of node Z. If node X or Y crashes, then it can be replaced by node Z and thus the computing station can start functioning as an ordinary two-node DRB station. Similarly, crash of node Z will result in the computing station functioning as an ordinary two-node DRB station. If both X and Y fail at their acceptance tests but are alive, then node Z becomes the new primary node and one of the two failed nodes (X and Y) should become the new secondary node (a shadow for node Z) and the other should become the third node (a shadow for the team of Z and the secondary node). In an n-node DRB station, the nth node functions as a shadow for the team of the first n-1 nodes. A natural consequence of this recursive

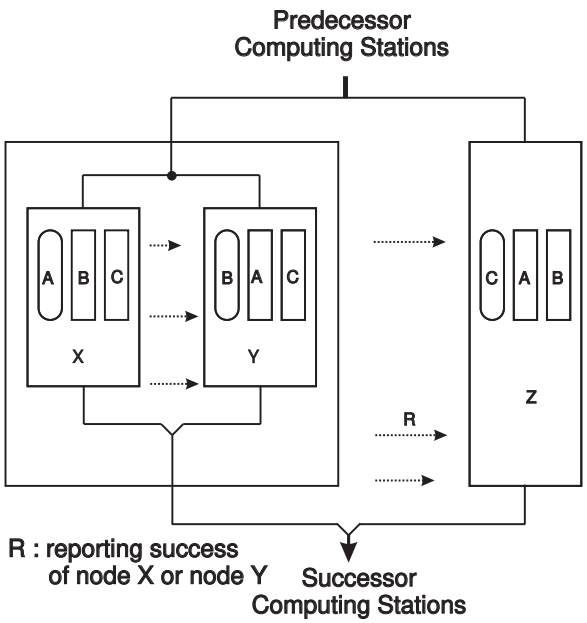


Figure 3.5: A DRB station with recursive shadowing

shadowing organization is the modest increase in the implementation complexity as the number of nodes used in a DRB station increases. For the sake of simplicity in discussion, these cases of using more than two try blocks in a DRB station will be treated as special cases throughout the remainder of this report.

3.2.4 Supervisor station

A major extension of the basic DRB scheme made by Hecht et al. [16, 15] was in incorporating a supervisor computing station into the LAN-based system. A centralized form of a supervisor station was incorporated in [15], but many of the supervisor station functions can also be decentralized. Demonstrations of decentralized implementations have yet to take place.

The supervisor station is basically responsible for three functions:

- Detection of node crashes
- Detection of misjudgments by the nodes in DRB stations about the status of their partner nodes
- Network reconfiguration, including task redistribution

To make the supervisory function highly robust, it is useful to dedicate a DRB station (rather than a nonredundant computing station) to the supervisory function (see Figure 3.6).

Interactions between the supervisor station and "worker" DRB stations must be implemented efficiently. Many different forms of interactions are conceivable.

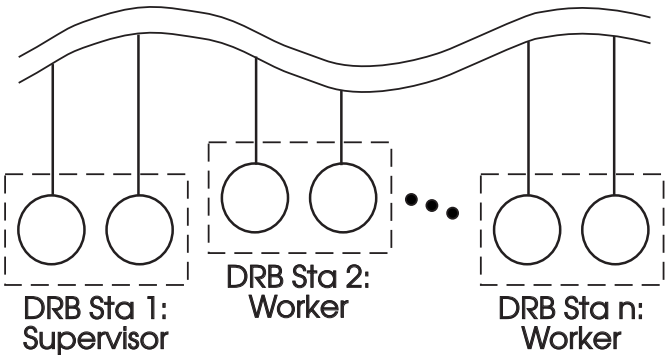


Figure 3.6: Using a DRB station as supervisor

3.2.5 A DRB Implementation Structure for Multicast LAN-Based Systems

A node in a DRB station basically engages in two types of inter-node communication:

- Data communication with other (predecessor and successor) computing stations, which may be DRB stations
- Status exchanges with partner node(s) within the same DRB station

The timings of the two types of communication are different and the status messages are short signals unlike the data messages, which may be substantial and of variable length.

Therefore, in choosing efficient implementation structures for DRB stations, the communication subsystem architecture of the given LAN-based system must be reflected. For the remainder of this chapter, we mainly assume the use of a LAN architecture with a single multiaccess broadcast network, as depicted in Figure 3.7.

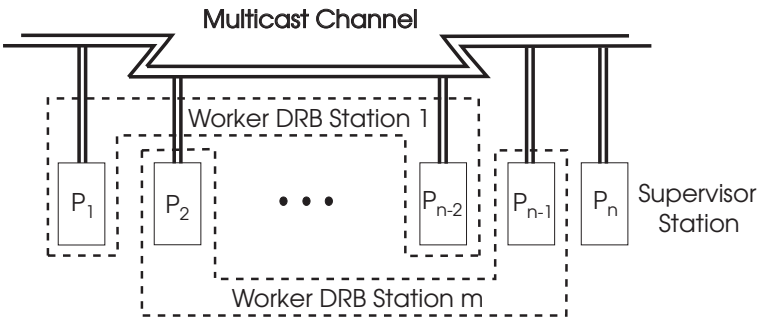


Figure 3.7: A fault-tolerant LAN-based system consisting of a supervisor station and DRB stations

We are focusing on the cases of using highly decentralized (HD) easily expandable LAN system architectures. Popular LAN communication structures such as the CSMA (carrier-sensing multiple access) bus and the token ring possess strong expandability characteristics

at the communication network level. To minimize the impacts of frequently changing hardware configurations on the software, it is useful to design and implement the software such that physical node addresses of each node are used only inside a small kernel module within the node. With such design, interaction among the processes distributed over multiple nodes does not involve use of physical node addresses.

Going a step further in this direction, Mori et al. [34, 33] developed a scheme called the *data field* for dynamically establishing "logical multicast channels" shared among the concurrent distributed processes without requiring the processes to know the identities of other cooperating processes, or the identities of the nodes running the processes. The only thing that a group of cooperating processes needs to know in advance for message communication is the name of the logical multicast channel through which the message will be communicated. Each of those processes can simply educate the communication subsystem in the host node about the logical multicast channel to be used. Therefore, a process wanting to send a message to other cooperating processes will execute a primitive "Multicast the data D_i over the channel C_x". Then all other processes designed to share channel C_x will pick up the message. Dynamic creation and use of logical multicast channels for interprocess communication is an attractive feature to incorporate in HD LAN system architectures.

3.2.5.1 Major Design Parameters of DRB Stations

Five design parameters that must be chosen carefully to obtain a cost-effective DRB station, and that may be impacted by the types of communication architectures used, are

- **Mechanisms for ensuring input data consistency.** Suppose each of the two fault-free partner nodes in a DRB station picks a new data item for the same task execution cycle. If these two data items have the same ID, then the two nodes are said to be preserving input data consistency. Complications can arise if the communication links between some nodes and the multicast channel are not reliable; certain data messages may arrive at one partner node but not at the other partner node. In general, it is necessary to take actions that explicitly ensure input data consistency.

- **Mechanisms for sharing acceptance test results.** The shadow node in a DRB station needs to learn the acceptance test result of its primary partner node with an acceptable delay. On the other hand, it is not essential in principle for the primary node to know the acceptance test result of the shadow node because as long as the primary node does not fail, it alone can satisfactorily meet the application requirements. However, it is possible that the shadow node could temporarily lag more than one task execution cycle behind the primary node. Therefore, the primary node must check to see if the shadow severely lags for longer than a tolerable time period. If so, the primary node may judge the shadow node to have become useless and thus ask the supervisor station to replace the shadow node. Another option for facilitating the detection of the fallout of the shadow node is to have the supervisor station detect it. To do this will require the shadow node to periodically announce its progress.

- **Mechanisms for reliable communication of result data messages.** Successful delivery of the result data message by the primary node to the successor computing

station(s) must be confirmed by both partner nodes in the producer DRB station. In case of a failure, the primary node must learn it and then either make a retry for delivery or give up and become a new shadow node, whereas the shadow node must learn it so that it can decide whether or not to deliver its own result data message. This means that delivery of a result data message by the primary node must be followed by a reply with an acknowledgment message(s) by the successor computing station(s).

- **Mechanisms to support recursive shadowing with N (> 2) try blocks.**

  It is desirable to implement recursive shadowing without incurring a nonlinear surge in the message traffic among the member nodes of a DRB station.

- **Mechanisms to support a supervisor station.**

  It is desirable to have a supervisor station perform its functions with minimal disturbance to the nodes of worker DRB stations.

3.2.5.2 Approaches for Ensuring Input Data Consistency

When the data items to be processed always arrive at both partner nodes belonging to the same DRB station in the same order, the input data consistency requirement is easily met. However, complications can arise if the communication links between some nodes and the multicast channel are not reliable; certain data messages may arrive at one partner node but not at the other partner node. With many LAN architectures, it is not easy to ensure uniform ordering of arriving messages at all the receiving nodes without involving an interaction among the receiving nodes.

Because the multicast channel is the only communication path available in the system architecture considered here, it is important to keep the number of messages exchanged during a task execution cycle of a DRB station to the minimum or a near-minimal number. Two approaches for ensuring input data consistency are conceivable. One approach is to simply have the primary node send an explicit message containing the ID of the data item selected for processing to the shadow node as exemplified by the Status-1 message-send action of node A (the primary node) and the Check-1 message-receive action of node B (the shadow node) in Figure 3.2.

The other approach is to piggyback the information on the data item selected onto the acknowledgment message that the primary node needs to send to the producer station of the selected data item. Although it is not shown explicitly in Figures 3.2 and 3.3, the "Receive & Save Data" action by a node must be followed by an acknowledgment action. Because the acknowledgment message must also be transmitted through the multicast channel, it is a good choice to multicast the message not only to the producer computing station (of the received data item), but also to the shadow node and the supervisor station. This piggybacking approach is depicted as the "A & S-1" (Acknowledgment & Status-1) message-send action of the primary node A in Figure 3.8. By checking this message, the shadow node can tell if both its the primary partner and itself have the same message-receiving experiences and can process the same data item. This action is depicted by the "C-1" (Check-1) message- receive action of node B in Figure 3.8.

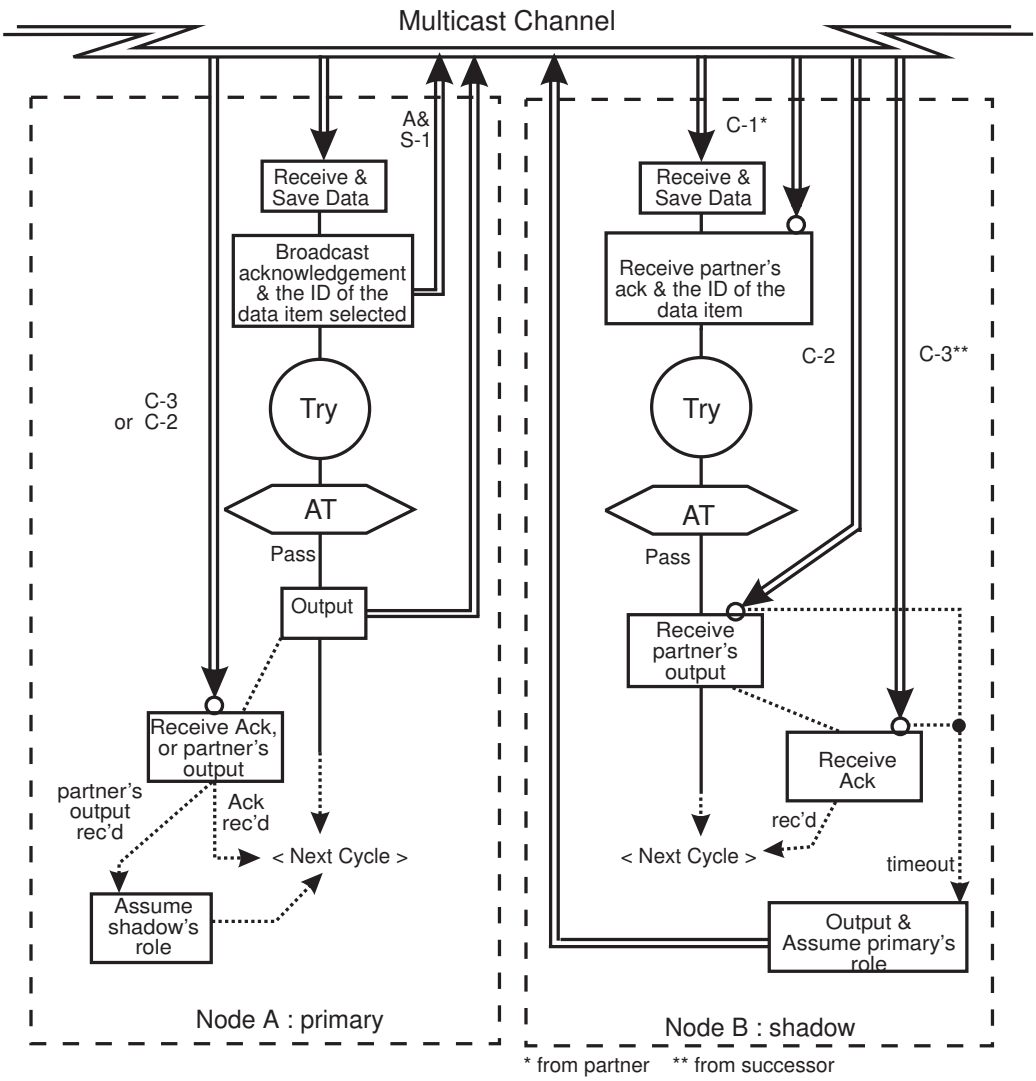


Figure 3.8: Achieving reliable data input

3.2.5.3 Approaches for Sharing Acceptance Test Results

An attractive approach is to have the shadow node check to see if the result data message from the primary partner node is multicasted, thereby judging the acceptance test success or failure of the primary partner. An absence of the multicast is interpreted as an indication that the primary partner has crashed. If the primary node fails in its acceptance test but remains alive, then it takes the step of sending a notice of the failure to the shadow node (and possibly to the supervisor station) in place of the step of multicasting a result data message. This *output monitoring* approach is depicted as the result output action of node A and the C-2 (Check-2) message-receive action of node B in Figure 3.8.

The primary node and/or the supervisor station needs to detect the fallout of the shadow node. The Status-2 message-send action of node B, which involves sending the acceptance test result to be checked by the primary node in a future task execution cycle, and the Check-previous-Status-2 action of node A, which involves checking a set of recent Status-2 messages from the shadow node, illustrates one way to facilitate the detection of the fallout of the shadow node. Another option is to require the shadow node to announce its progress periodically, in the form of a Status-2 message in each task execution cycle or less frequently. The progress report can then be monitored by the primary node and/or the supervisor station.

3.2.5.4 Approaches for Reliable Communication of Result Data Messages

One approach for achieving reliable communication of result data messages is depicted in Figure 3.2. The output action of node A includes receiving acknowledgment(s) from the successor computing station(s). Receipt of the acknowledgment(s) together with the Status-3/Check-3 protocol, ensures reliable delivery of a result data message.

A more attractive approach, however, is to have the successor computing station multicast an acknowledgment message and have both the shadow node and the primary node verify that the acknowledgment arrives. The C-3 (Check-3) actions of node A and node B in Figure 3.8, together with the A & S-1 (Acknowledgment & Status-1) action to be taken by the successor computing station, represent this *acknowledgment monitoring* approach. Note that in Figure 3.8 a nonblocking approach for receiving the acknowledgment message was also adopted in both partner nodes. This is an optional feature and it can be adopted when it is desirable to allow the nodes to "look ahead" into the next task execution cycle while the acknowledgment message is on the way.

3.2.5.5 Approaches for Recursive Shadowing with N (> 2) Try Blocks

Implementing the recursive shadowing involves a recursive arrangement of the approaches discussed earlier for implementation of three of the five basic design parameters.

3.2.5.6 Approaches for Implementing a Supervisor Station

With the basic types of LAN-based system architectures, maximum flexibility exists in implementation of supervisor stations because it is easy to enable every processor to hear any data or status message communicated between nodes of "worker" computing stations.

As mentioned early in Section 3.2.4, many of the supervisor station functions can be decentralized. However, the advantages and disadvantages of decentralized implementations have not been fully understood. Of the three basic functions of the supervisor station, the detection of node crashes, is the easiest to implement in a decentralized form. When a centralized supervisor station is adopted, it is also necessary to make arrangements for each worker station to periodically cast its opinion on the health status of the supervisor station. The interval between such opinion casting can be made much larger than a typical task execution cycle.

3.2.5.7 Modular Implementation Models and Experimental Validations

To partially validate the DRB implementation structure discussed throughout Section 3.2.5 and identify detailed implementation issues, a simple experimental implementation of DRB stations in a small-scale LAN-based DCS testbed was conducted from March through June 1993. This is discussed in Section 3.2.8. We then proceeded to formulate a modular implementation model which can easily incorporate various existing or emerging techniques for network diagnosis and reconfiguration and reliable message communication. This model and partial validation efforts made are discussed in Sections 3.2.9 and 3.2.11.

3.2.6 Principles and Implementation Structures of the ADRB scheme

The ADRB scheme exploits several of the fundamental trade-offs that are found in computing systems in the dimensions of time, equipment and service. The particular instances of those dimensions in ADRB are latency in computation and error recovery, efficiency of resource utilization, and accuracy of computed results.

The ADRB scheme extends the DRB scheme in two major ways. First, a critical real-time task can be executed not only (1) in *the parallel redundant mode*, which is the standard mode used in the basic DRB station, but also (2) in *the sequential backward recovery mode*, which is the execution mode adopted in the original *recovery block scheme* [19, 40], and (3) in *the sequential forward recovery mode*, which has been considered in many previous projects on exception handling. Therefore, an *ADRB station* dynamically switches its operating mode in response to significant changes in the resource and application modes. Secondly, the supervisor station under the DRB scheme is basically responsible for three functions: detection of node crashes, detection of misjudgments by the nodes in DRB stations about the status of their partner nodes, and network reconfiguration including task redistribution. Under the ADRB scheme the supervisory function is not necessarily concentrated in a particular node or computing station. Moreover, the supervisory function station has an additional dimension—that is, changing the set of real-time tasks to be executed. The supervisor function can be executed not only in the centralized mode but also in the decentralized mode. Therefore, again, the system can dynamically switch between the centralized supervisory mode and the decentralized cooperative monitoring and control mode. The algorithms and execution modes for accomplishing the three basic functions can thus be adjusted as significant changes in the resource and application modes occur.

Figure 3.9 depicts the overall ADRB operation. First, the given task set is mapped to the node set by the network configuration management (NCM) server. Each task may be assigned to one or more nodes to form an ADRB station. The execution mode of each

ADRB station is chosen by the NCM server on the basis of the equipment availability and the criticality and recovery time requirement of the task assigned to the ADRB station. As the system resource condition changes and the application proceeds through different phases, the NCM may order each ADRB station to change its execution mode.

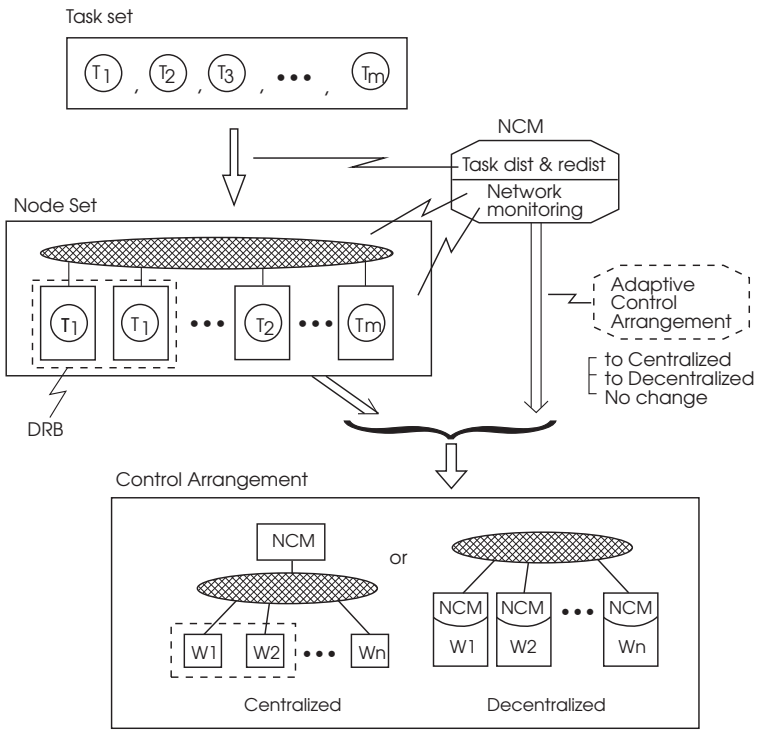


Figure 3.9: Basic operations under the ADRB scheme

Figure 3.9 also depicts the possibility of dynamically switching between the centralized execution mode and the decentralized execution mode for the NCM server. An abstract representation of the logical component that makes decisions on switching of the NCM execution mode is named "adaptive control arrangement". This component can of course be a part of the NCM. In the decentralized execution mode, NCM servers running on distributed nodes need frequent communication among themselves to assure that their views of the network configuration are consistent.

3.2.6.1 Adaptive Changes of the Task Execution and Recovery Mode in an ADRB Station

Three execution modes for real-time tasks A real-time task can be executed in three fundamentally different ways with the cooperation of a task designer willing to provide software redundancy:

- **Sequential backward recovery.** This mode of execution was incorporated in the original RB scheme [19, 40]. It takes a single processing node.

- **Parallel redundant execution.** This mode of execution was incorporated in the DRB scheme and facilitates forward recovery. It takes two or more processing nodes.

- **Sequential forward recovery.** This mode of execution was assumed in many previous projects on exception handling. It takes a single processing node.

These three modes of fault-tolerant execution differ in equipment resource requirements, software design requirements, and recovery times required. Therefore, an ADRB station under the ADRB scheme can take advantage of these three options to maximize the reliable delivery of services under varying conditions of available task execution time and equipment resources.

Figure 3.10 shows an abstract representation of the components in an ADRB station. The main components are as follows.

1. There are a set of input queues, each corresponding to a different input source or a different input type.

2. n different task algorithms are provided. Some of them may be designed for the same or similar computational results. Others may be exception handlers that can be invoked upon failure of an earlier tried algorithm to bring the ADRB station to a safe state (i.e., a state from which the ADRB station has a chance to proceed to a normal state).

3. The acceptance test function is a component provided to check the acceptability of the results of the most recently executed algorithm.

4. Once the acceptance test execution is successful, then the computation result is sent in the form of a message to other ADRB stations and/or saved into a stable storage component that can be accessed by other ADRB stations. The stable storage component may be a storage component exclusively belonging to the ADRB station, a storage component shared by multiple ADRB stations, or a combination of both types.

Figure 3.11 depicts the three different execution modes of an ADRB station. The DB component in this figure corresponds to the stable storage component in Figure 3.10. When the ADRB station is in the sequential backward recovery mode (RB mode) or the sequential forward recovery mode (EH Exception Handler mode), only one processing node is dedicated to the station and the crash of the node dictates functional replacement of the crashed node by a standby node identified by the NCM server. Unless the processing node saved its state vector into a shared database component before the crash, the replacement means a full restart of the task on a new node.

In considering the execution time aspects of each execution mode, we assume that the execution times for the primary and shadow algorithms, respectively, are T_1 and T_2 and the deadline adopted in another node for hearing about the completion of the primary algorithm is T_d. We also assume that the primary algorithm fails to produce an acceptable result because of a software design fault or a hardware fault.

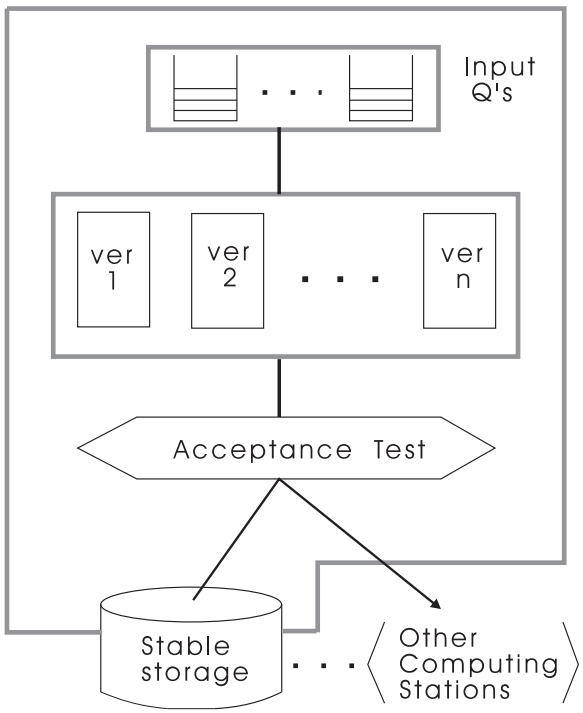


Figure 3.10: Basic Components of an ADRB station

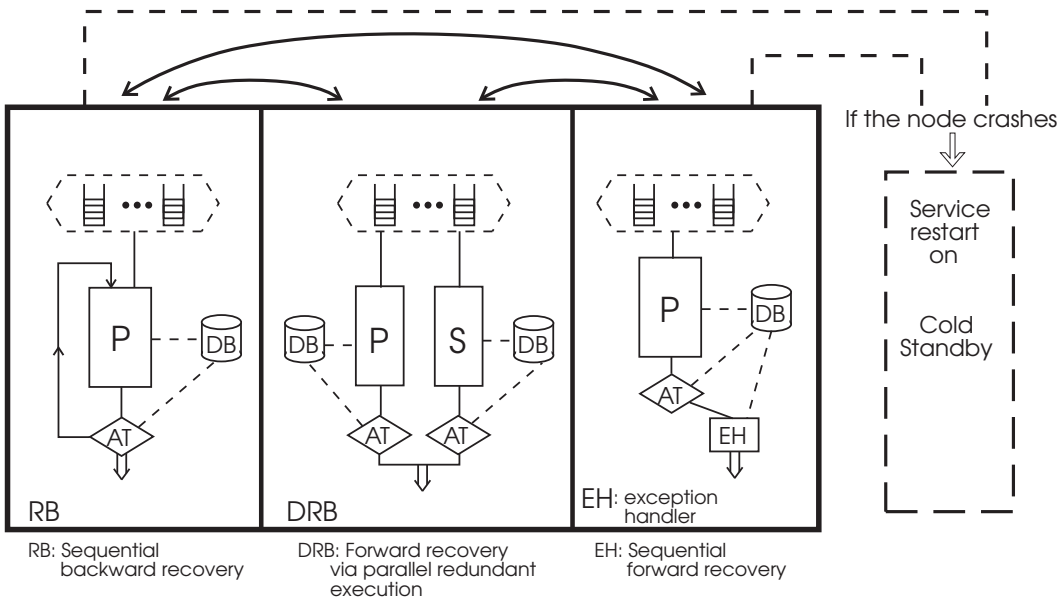


Figure 3.11: Three execution modes of an ADRB station

1. **Sequential backward recovery mode (RB mode).**

   Two different cases, one involving an acceptance test failure and the other involving a node crash, are distinguished.

   (1a) Sequential backward recovery from an acceptance test failure.

   The total execution time in this case is

   $$T.RB.AT = T_1 + T_{sr} + T_2,$$

   where T_{sr} is the time for state restoration, which is needed to prepare for an alternate algorithm. Here $T_{sr} + T_2$ can be greater than T_1.

   (1b) Node crash.

   We assume that the processing node saved its state vector into a shared database component before the crash. The total execution time in this case is

   $$T.RB.CR = T_d + T_{replace} + T_{sr} + T_2,$$

   where $T_{replace}$ is the time spent to identify a new replacement node and T_{sr} and T_2 are the same as in (1a). Here $T_{replace}$ can be greater than T_1.

2. **Parallel redundant execution mode (DRB mode).**

   Again two different cases are distinguished.

   (2a) Primary's failure at the acceptance test.

   The total execution time in this case is

   $$T.DRB.AT = T_1 + T_{sd},$$

   where T_{sd} is the time for a shadow to hear the failure of the primary. Here T_{sd} is typically smaller than T_{sr} or T_2.

   (2b) Node crash.

   The total execution time in this case is

   $$T.DRB.CR = T_d.$$

   Here it is obvious that $T_d > T_1 + T_{sd}$.

3. **Sequential forward recovery mode (EH mode).**

   As before, two different cases are distinguished.

   (3a) Sequential forward recovery from an acceptance test failure.

   The total execution time in this case is

   $$T.EH.AT = T_1 + T_{EH},$$

where T_{EH} is the time for execution of the exception handler. T_{EH} is typically smaller than T_1 or T_2 but larger than T_{sd}.

(3b) Node crash.

In this case, the recovery action is the same as in the case of (1b). Therefore

$$T.EH.CR = T.RB.CR = T_d + T_{replace} + T_{sr} + T_2.$$

The execution times under different execution modes analyzed above can be ordered as shown in Figure 3.12.

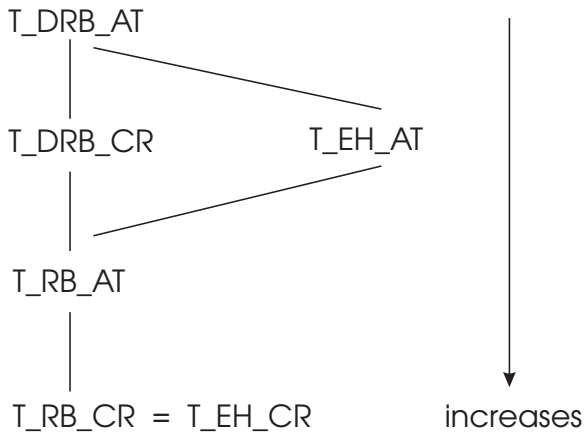


Figure 3.12: Ordering of execution times under different modes

An example adaptation scenario Figure 3.13 depicts one scenario in which an ADRB station adapts to the changing time and equipment resource conditions by changing its mode of execution.

A defense command and control (C^2) situation is assumed in Figure 3.13. During peacetime, both the available time and equipment for task execution are assumed to be in a comfortable, abundant state. The application in such a phase may require execution of a large number of soft-real-time tasks. The sequential backward recovery mode is the prevailing mode of execution for most DRB stations. Once war starts, the set of tasks to be executed changes, and many become hard-real-time tasks. The available time condition becomes tight, although the equipment condition may still be comfortable because not much equipment loss has yet been incurred. Almost all hard-real-time tasks will be executed in the parallel redundant execution mode. After some time passes in the war phase, the equipment condition will become desperately short because of battle damage and other random equipment failures. Therefore, shedding of less critical application tasks as well as executing critical tasks in the less equipment-intensive execution mode, namely, the single forward-recovery mode, will become mandatory.

More exact adaptation possibilities can be described by identifying a desirable execution mode for each point in a two-dimensional resource space—that is, Time $\times$ Equipment space as in Figure 3.16. For example, when the time availability situation is desperate and the

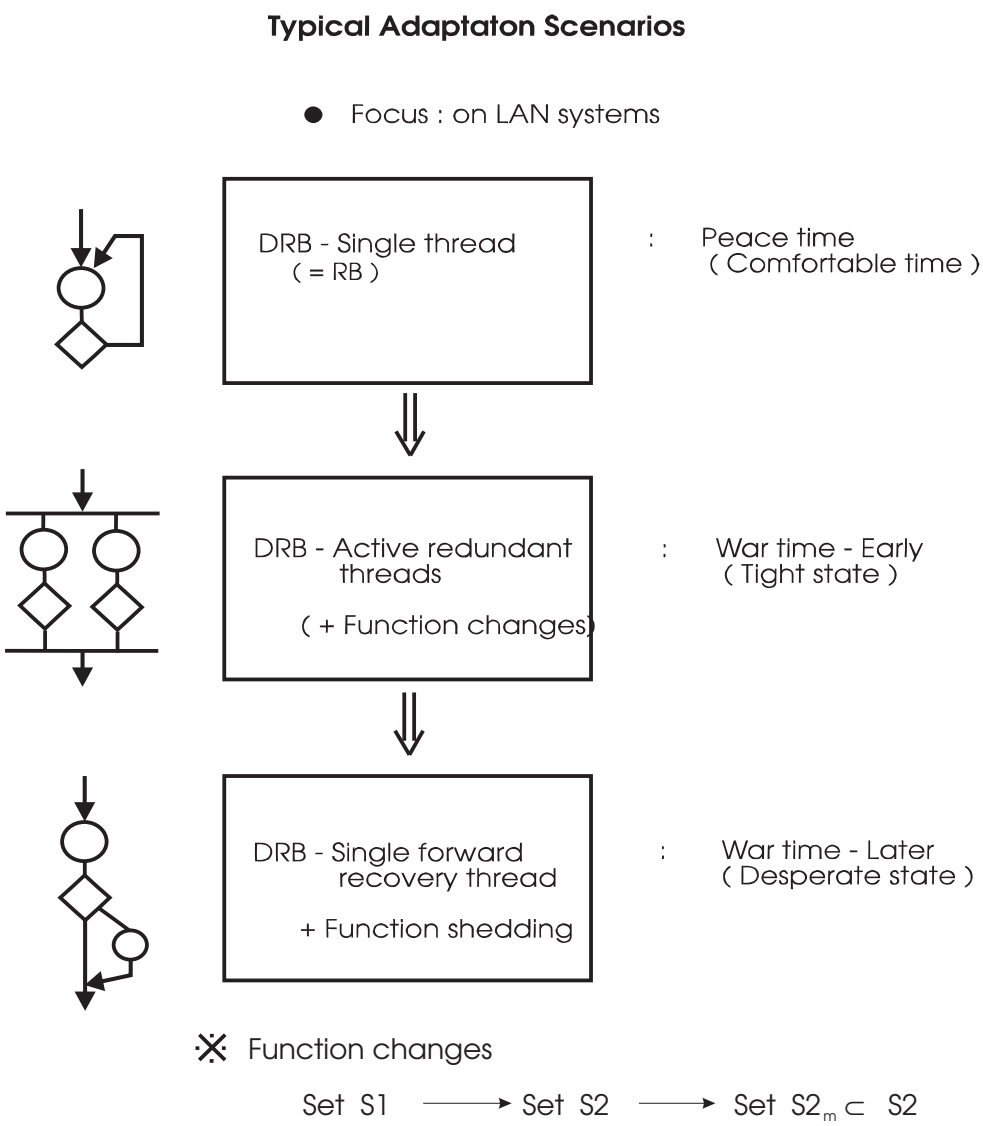


Figure 3.13: Typical adaptation scenarios

equipment availability situation is tight, two possible execution modes are conceivable. Since the time availability is in a desperate condition, every critical task must be executed in the DRB mode or the EH mode. If the equipment availability were in a comfortable condition, then the DRB mode would have been the choice. However, since the equipment availability is in a tight condition, it is not feasible to configure the full set of ADRB stations in the DRB mode. Another option is to configure some less critical tasks in the EH mode, thereby reducing the number of nodes required but at the cost of increasing the risk of those ADRB stations missing their deadlines.

Transition protocols Protocols for the ADRB stations and the NCM server effect switching of ADRB stations among the three execution modes. The facilities used in execution of the protocols are depicted in Figure 3.14.

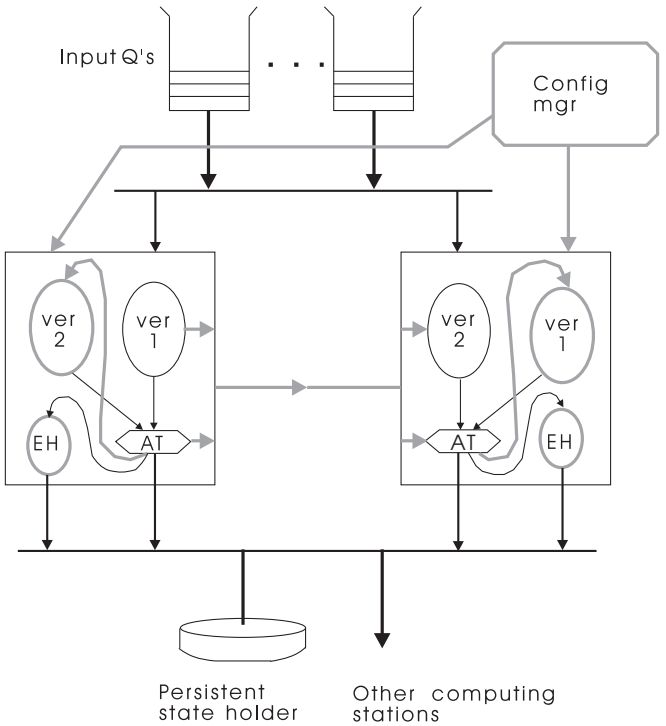


Figure 3.14: A basic configuration of an ADRB station

There are six different possible transitions among the three execution modes. Six transition protocols, each corresponding to one of the six transitions, are presented in Figures 3.15:ab, 3.15:cd, and 3.15:ef.

Among the six transition cases, the case of switching from the RB mode to the DRB mode and the case of switching from the sequential forward recovery (S-FR) mode to the DRB mode require the most complex and time-consuming protocols. This occurs because the primary must provide a copy of its state vector to the newly chosen shadow node.

There are two issues related to preservation of data integrity during execution of transition protocols. One is for the primary node to maintain its data integrity. The basic rule

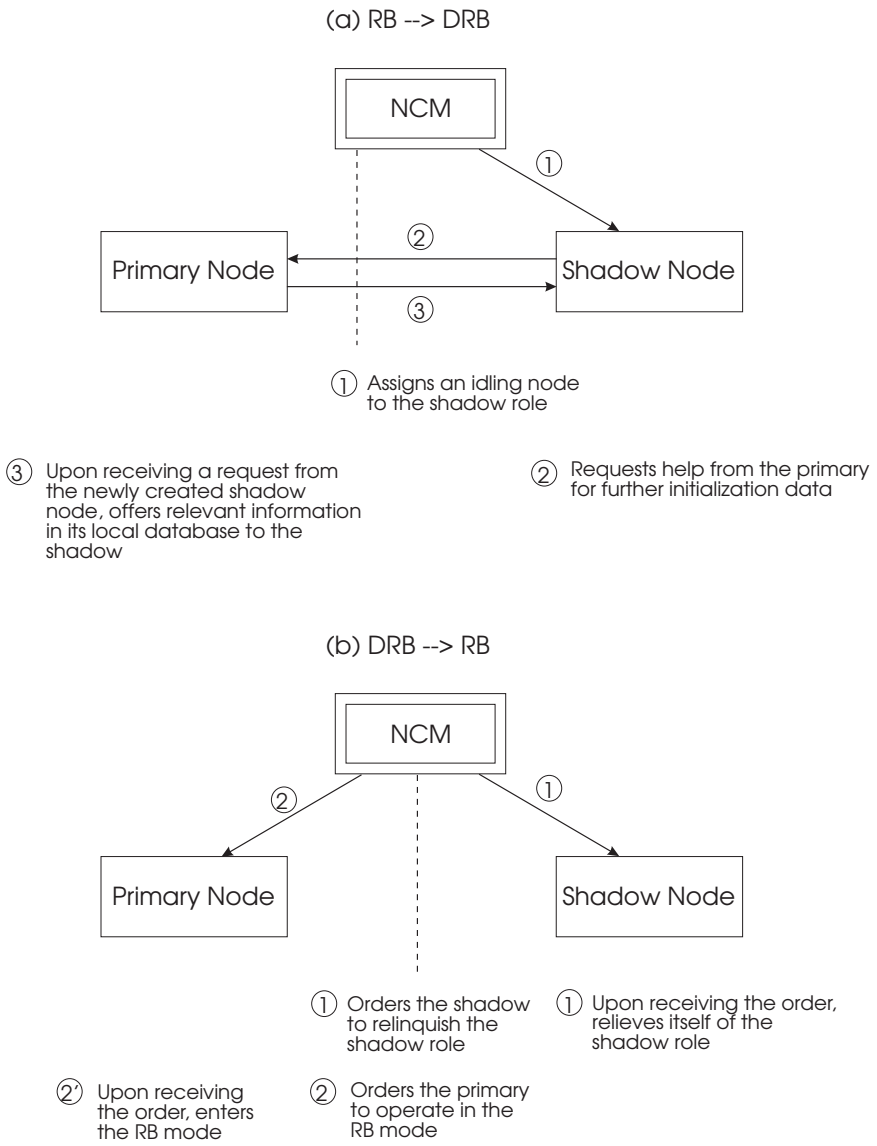
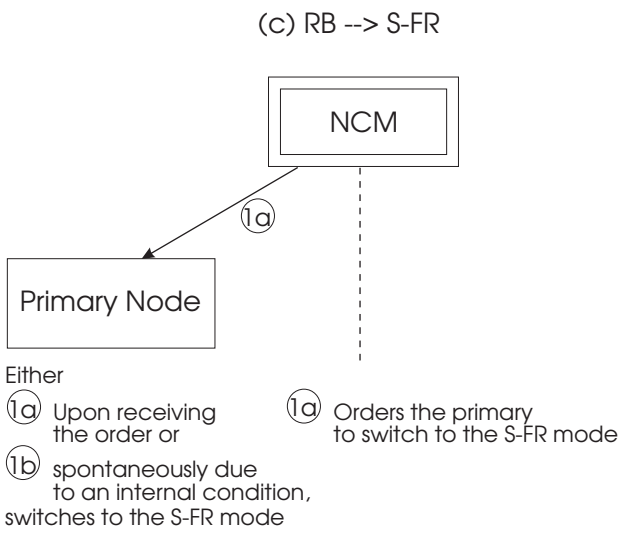


Figure 3.15: ADRB transition protocols I



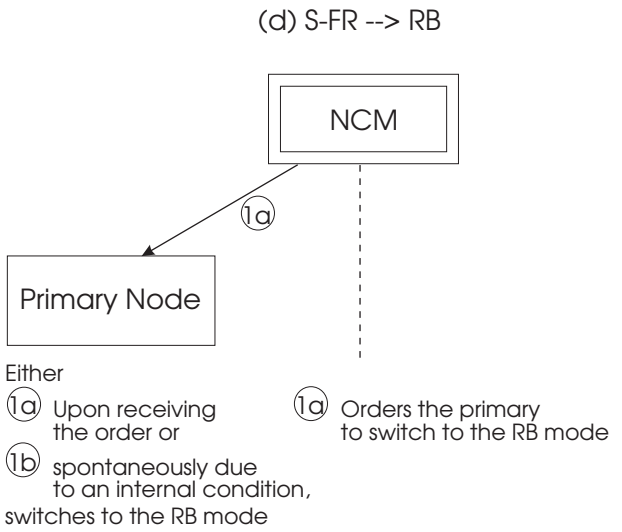


Figure 3.15: ADRB transition protocols II

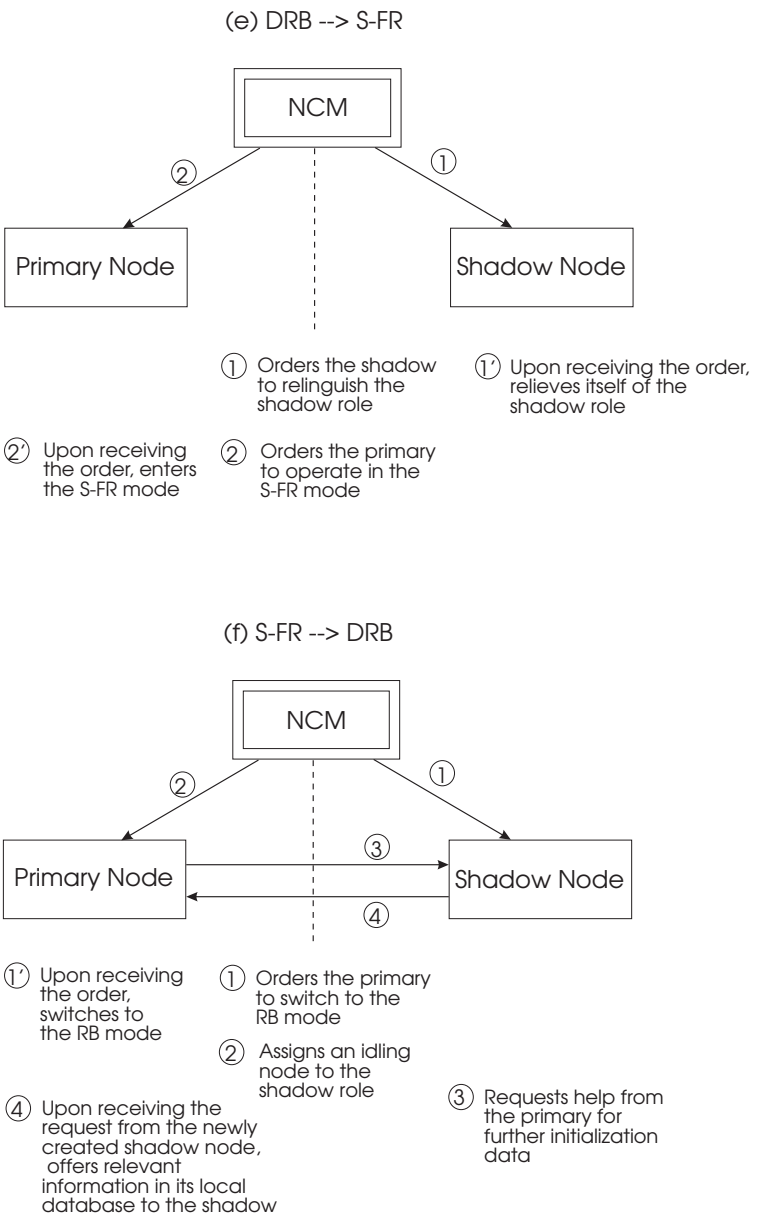


Figure 3.15: ADRB transition protocols III

here is that the primary node must switch its mode of operations only between task execution cycles. If it receives an order to switch the mode in the middle of a task execution cycle, it must defer the switching until the time of completing the current cycle. If the node detects an error—that is, an acceptance test failure, before completing the current task execution cycle, it must make a recovery attempt valid in the current mode. Only after a successful recovery—or an AT failure, the node can switch to the new mode. If the node cannot recover successfuly, then the node must attempt to inform the NCM server of its crippled condition so that the NCM server may make an appropriate reconfiguration decision. This simple operational rule is the most effective in maintaining data integrity within the primary node.

The other issue is for the shadow node to maintain its data integrity. Only two switching cases are relevant here : RB $\longrightarrow$ DRB and DRB $\longrightarrow$ RB. The case of DRB $\longrightarrow$ RB does not present any problem. The shadow node can switch its mode as soon as it receives an order. Even in the other case, RB $\longrightarrow$ DRB, the new shadow node will execute its switching procedure as soon as it receives an order. However, in this case, the switching procedure can be time-consuming since it requires cooperation of the primary node for providing a state vector (i.e., information in the local database of the latter). The primary may choose to send the information in multiple steps. During the switching procedure the shadow must also periodically inform the primary of the sequence of messages it has received so that the primary may determine how much information it must supply from its local database.

Other adaptation parameters Besides the execution mode, there are other adaptation parameters in an ADRB station. Dropping the task is an adaptation parameter used in Figure 3.16 and was set on when the equipment availability condition was tight or desperate. In addition, the degree of redundant execution in an ADRB station is an adaptation parameter. When both the criticality of a task and the fault occurrence rate are high, the ADRB station may execute in the DRB mode with more then two processing nodes as long as a sufficient number of nodes are available in the system. When the equipment availability condition becomes tighter, the ADRB station can reduce the degree of redundant execution.

3.2.6.2 Adaptive Network Surveillance and Reconfiguration

The three major components of network configuration management (NCM) are

- Network surveillance to detect and locate faulty nodes
- Confirmation of message delivery and detection of misjudgments made by the nodes in the DRB stations
- Task redistribution

The first two components are closely coupled. Both involve monitoring messages originating from the subject nodes. On the other hand, message delivery confirmation needs to be executed much more frequently than the recognition of permanent faults.

Equipment

desperate	(DRB +) RB with shedding	(DRB +) single forward recovery + shedding	single forward recovery thread + shedding
tight	RB with possible shedding	DRB with shedding or DRB+RB	DRB with shedding or DRB+single forward recovery thread
comfortable	RB or DRB	DRB	DRB
	comfortable	tight	desperate

Time

Figure 3.16: Adapatation possibilities for various combinations of equipment and time

3.2.6.3 Basic execution modes for NCM

The NCM execution modes are defined as follows:

1. Fully cooperating mode vs. partitioned mode

   When the communication network is not fully connected or communication between some groups of nodes becomes difficult due to a high level of noise interference, it is natural for the system to enter the partitioned mode of operation. In a partitioned mode, a node in a subsystem can have a view of the subsystem configuration but not of the entire system configuration. Basically, the NCM can be forced to operate in the partitioned mode when internode communication becomes highly unreliable. Note that this partitioning is applied to the NCM only; the possibility of application-related communication between a node in one group and another node in a different group is not removed.

2. Centralized mode vs. decentralized mode

   As mentioned earlier, the NCM server can be implemented in either centralized or decentralized forms.

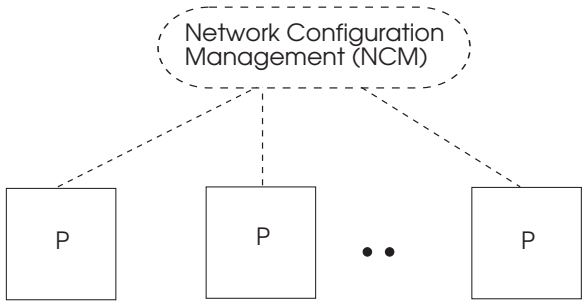


Figure 3.17: Logical NCM function

Figure 3.18 depicts all four different execution modes for the NCM server of which the logical function is shown in Figure 3.17:

(a) Fully cooperating centralized mode

(b) Fully cooperating decentralized mode

(c) Partitioned centralized mode

(d) Partitioned decentralized mode

The basic principles guiding the four execution modes are summarized in Figure 3.19.

The comparisons of the fully cooperating mode and the partitioned mode are made in Figure 3.20. Since under the partitioned mode a node cannot have a global view of the entire system status, the mode sacrifices the possibilities for globally optimal resource sharing and task distribution. On the other hand, if the fully cooperating mode of execution for the NCM is attempted when communication between some groups of nodes is difficult, then the execution efficiency of the NCM will become very low.

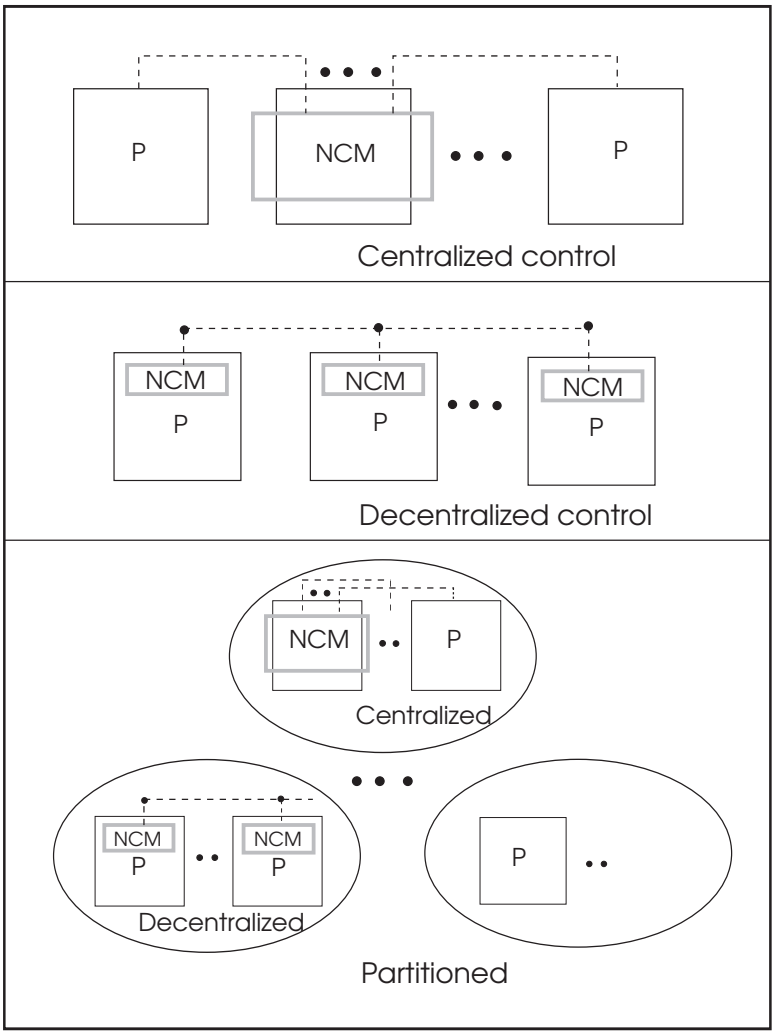


Figure 3.18: Four different execution modes for NCM

<table>
<tr><th colspan="2">NCM Arrangement</th><th colspan="2">Basic Principle</th></tr>
<tr><td colspan="2">Centralized</td><td colspan="2">Select leader</td></tr>
<tr><td colspan="2">Decentralized</td><td colspan="2">Achieve consensus</td></tr>
<tr><td rowspan="2">partitioned</td><td>Centralized</td><td rowspan="2">Manage partitioning during periods of communication difficulties</td><td>Select leader</td></tr>
<tr><td>Decentralized</td><td>Achieve consensus</td></tr>
</table>

Figure 3.19: Basic principles guiding the execution modes

The main performance characteristics of the four different execution modes are also summarized in Figure 3.20. The footnotes associated with Figure 3.20 are the following:

1. Since the transient fault rate is low, the centralized approach (A.1) will yield good coverage in fault detection.

2. In general, the decentralized approach (A.2) has better coverage in fault detection but its relative strength becomes noticeable in situations with high fault rate. However, the decentralized approach requires consumption of more processing power spread over all the nodes.

3. The partitioned mode of operation is entered when high-data-rate communication among groups of nodes becomes difficult. If the transient fault rate is low, then communication difficulties will occur only due to permanent loss of some communication links. Here the centralized approach (B.1) will yield good coverage in fault detection.

4. Because of the same reasons used in both cases 2 and 3 above, the decentralized approach (B.2) will yield good coverage in fault detection.

5. In the situations with high rate of transient faults, the superior capability of the decentralized approach (A.2) over A.1 in terms of the coverage in fault detection should be noticeable.

6. If the transient fault rate is high, then communication difficulties among the largely autonomous node groups can occur not just because of permanent loss of some communication links but also because of a high level of noise interferences in communication links. Within each node group, frequent transient faults may occur in processing nodes or communication links. In such cases, the decentralized approach (B.2) will exhibit superior coverage in fault detection.

	A. Fully Cooperating	B. Partitioned
Observability	high	lower
Resource sharing and Optimality of task dist.	high	lower
Performance in highly noisy environments	low	higher

	A. Fully Cooperating		B. Partitioned	
	A.1 Centralized	A.2 Decentralized	B.1 Centralized	B.2 Decentralized
For situations with low rate of transient faults	good[1]	good[2] but consumes more processing power	good[3]	good[4]
For situations with high rate of transient faults	worse than A.2[5]	good[5]	worse than B.2[6]	good[6]
For situations with low rate of permanent faults	about the same as A.2[7]	about the same as A.1[7]	about the same as B.2[8]	about the same as B.1[8]
For situations with high rate of permanent faults	slightly worse than A.2[9]	slightly better than A.1[9]	about the same as B.2[10]	about the same as B.1[10]
Coverage for fault detection	high[11] (in cases of high transient fault rate)	highest[11] (in cases of high transient fault rate)	lowest[11] (in cases of high transient fault rate)	low[11] (in cases of high transient fault rate)
Msg traffic	lower than A.2[12]	higher than A.1[12]	lower than B.2[12]	higher than B.2[12]

Figure 3.20: Performance characteristics of the four NCM execution modes

7. Permanent faults at low rate are not a major factor influencing the performance of either of the two approaches, centralized (A.1) and decentralized (A.2).

8. Permanent faults at low rate are not a major factor influencing the performance of either of the two approaches, centralized (B.1) and decentralized (B.2).

9. Permanent faults at high rate will cause the differences in performance of the two approaches, A.1 and A.2, to become more noticeable than permanent faults at low rate do but they do not have as much influence as transient faults do. This is partly because usually the partitioned mode of operation is entered after a substantial number of permanent fault occurrences and thus the period during which high rate of permanent faults is exhibited can only be brief. In any case, frequent changes of the ´supervisor˘ node under A.1 due to the supervisor losses will incur more serious performance penalty than node failures of the same frequency under A.2 do.

10. If permanent faults occur at a high rate even after the system enters the partitioned mode of operation, then repeated re-partitioning is likely to occur. In this type of situation, the differences in performance between the two approaches, B.1 and B.2, are not significant.

11. The fully cooperating mode of operation yields high observability whereas the partitioned mode of operation incurs sacrifices of the global view. Therefore, the former yields higher fault detection coverage than the latter does. In addition, between the centralized approaches and the decentralized approaches, the decentralized approaches yield higher coverage in fault detection.

12. In general, the decentralized approach creates a higher message traffic than the centralized approach does. This occurs because the former requires every active node to monitor the health conditions of every other node and participate in decision making regarding the system status and the reconfiguration action to be taken, whereas with the latter, only the supervisor node gets involved in monitoring and decision making in a major way. Between the fully cooperating mode of operation and the partitioned mode of operation, the former creates more message traffic than the latter because in the latter case, frequent message communication occurs only among the nodes belonging to the same group.

Execution modes of each major NCM function Let us now examine the possibilities of executing each of the three major components of NCM in different modes. First of all, since both the recognition of nodes with permanent faults and the confirmation of message deliveries involve monitoring of message communications, it is natural to execute both NCM functions in the same mode. This is why the four execution modes are listed as possible modes for network monitoring in Figure 3.21. Secondly, for task redistribution, the decentralized execution approach is not attractive; it just increases the overhead. The reliability of the node handling task redistribution can be checked by the other two NCM functions (i.e., recognition of permanent faults and confirmation of message deliveries). Therefore, only two execution modes, the fully cooperating centralized mode and the partitioned centralized mode, are listed as possible modes for task redistribution in Figure 3.21.

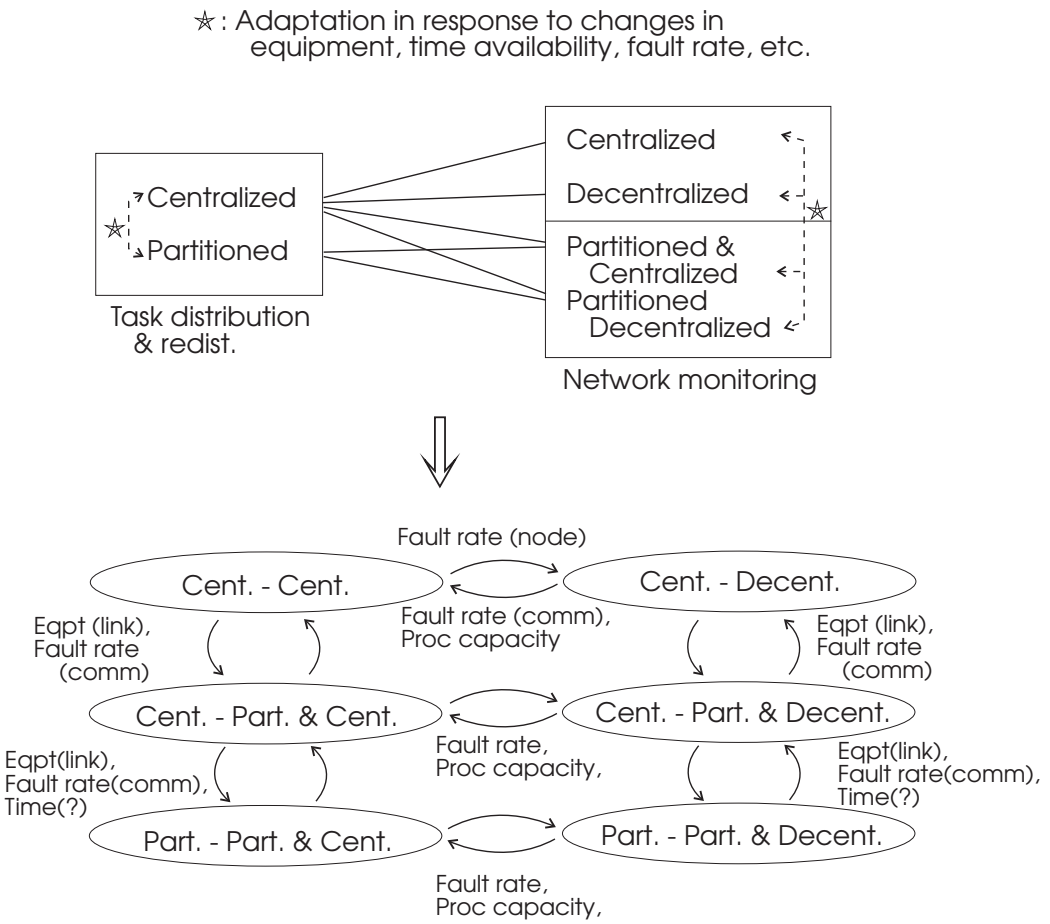


Figure 3.21: A scenario for adaptation of the NCM server

Third, when the network monitoring is executed in the fully cooperating mode, there is no reason for the task redistribution function to be executed in a partitioned mode. On the other hand, the inverse is not true; even if the network monitoring is executed in the partitioned mode, the task redistribution function may be executed in the fully cooperating mode to achieve globally optimal distribution. This is because the task redistribution function is much less frequently executed than the network monitoring is and thus even with the reduced network communication bandwidth caused by the high level of noise interference, execution of the task distribution function in the fully cooperating mode may be feasible.

Therefore, only six possible combinations of the network redistribution execution mode and the network monitoring execution mode need to be considered:

1. (fully cooperating) centralized (mode for task redistribution) and (fully cooperating) centralized (mode for network monitoring)
2. centralized and decentralized
3. centralized and partitioned centralized
4. centralized and partitioned decentralized
5. partitioned centralized and partitioned centralized
6. partitioned centralized and partitioned decentralized

These six combinations are depicted in Figure 3.21.

An example adaptation scenario The reasons for switching of the NCM server among different execution modes are the different performance characteristics of the different execution modes summarized in Figure 3.20. The switching between the fully cooperating mode and the partitioned mode will be dictated primarily by the communication link availability and secondarily by the remaining communication bandwidth available and the execution time available. Similarly, the switching between the centralized mode and the decentralized mode will be dictated primarily by the node fault rate and the communication link fault rate and secondarily by the processing capacity available. Figure 3.21 depicts an adaptation scenario based on these reasonings.

Further classification of the centralized execution mode The centralized execution mode for the network monitoring function can be further classified into the simplex supervisor mode and the active replicated supervisor mode. In the latter case, the network monitoring function is assigned to a DRB station. Again, the primary factors that influence the decisions on switching between the simplex supervisor mode and the active replicated supervisor mode include the time constraints on the network monitoring function, the equipment availability, and the fault rate. The task redistribution function can also be executed in the modes discussed above.

3.2.7 Summary and Remaining ADRB Research Issues

The ADRB scheme we have presented is a concrete instance of the AFT technology that can be put into practice immediately. However, in order to exploit the full potential of the scheme, further research is needed to develop various optimal implementation techniques. Some of the major study topics in this direction are

- Cost-effective integration of the adaptive redundant execution management component and the adaptive NCM component into the ADRB scheme
- Development of an implementation model for the ADRB scheme in object-based distributed systems
- Experimental design of ADRB stations

Successful accomplishment of these studies will contribute significantly to the growth of the AFT technology and its user community.

3.2.8 Partial Validation of a DRB Implementation

To partially validate the DRB implementation structure discussed in Section 3.2.5 and identify detailed implementation issues, a simple experimental implementation of DRB stations in a small-scale LAN-based DCS testbed was conducted from March through June 1993.

The simple freeway-segment monitoring simulation program was first implemented on a LAN of four PCs. As shown in Figure 3.22, the simulated freeway segment has one on-ramp and one off-ramp. One PC is used to simulate the dynamically changing conditions of the freeway segment, including positions of continuously entering and exiting cars. The second PC uses two sensors to keep counts of the cars entering the freeway segment during each checking interval. The third PC uses two sensors to keep counts of the cars exiting the freeway segment during each checking interval. The fourth PC then collects reports from the second and third PCs, calculates the number of cars in the freeway segment, and displays the count.

As shown in Figure 3.22, two DRB stations were configured, and thus six PCs were used. The experiment involved the injection of faults, measurement of the overhead and recovery performance, and subsequent analysis of the system behavior. This experiment was a partial validation of the DRB implementation structure in that the supervisor station was not implemented. The flowcharts in Figures 3.23 and 3.24 show the details of the DRB protocols implemented in the preliminary partial validation experiment.

3.2.9 A Modular Implementation Model for the DRB Scheme with a Configuration Supervisor

The implementation techniques for the DRB scheme, especially those for use in LAN-based systems, are expected to go through continuous refinement and extension in the future. This is partly due to the fact that a higher-fault-coverage real-time fault tolerance scheme can be obtained by combining the DRB scheme with complementary techniques for network diagnosis and reconfiguration and reliable communication. New complementary techniques

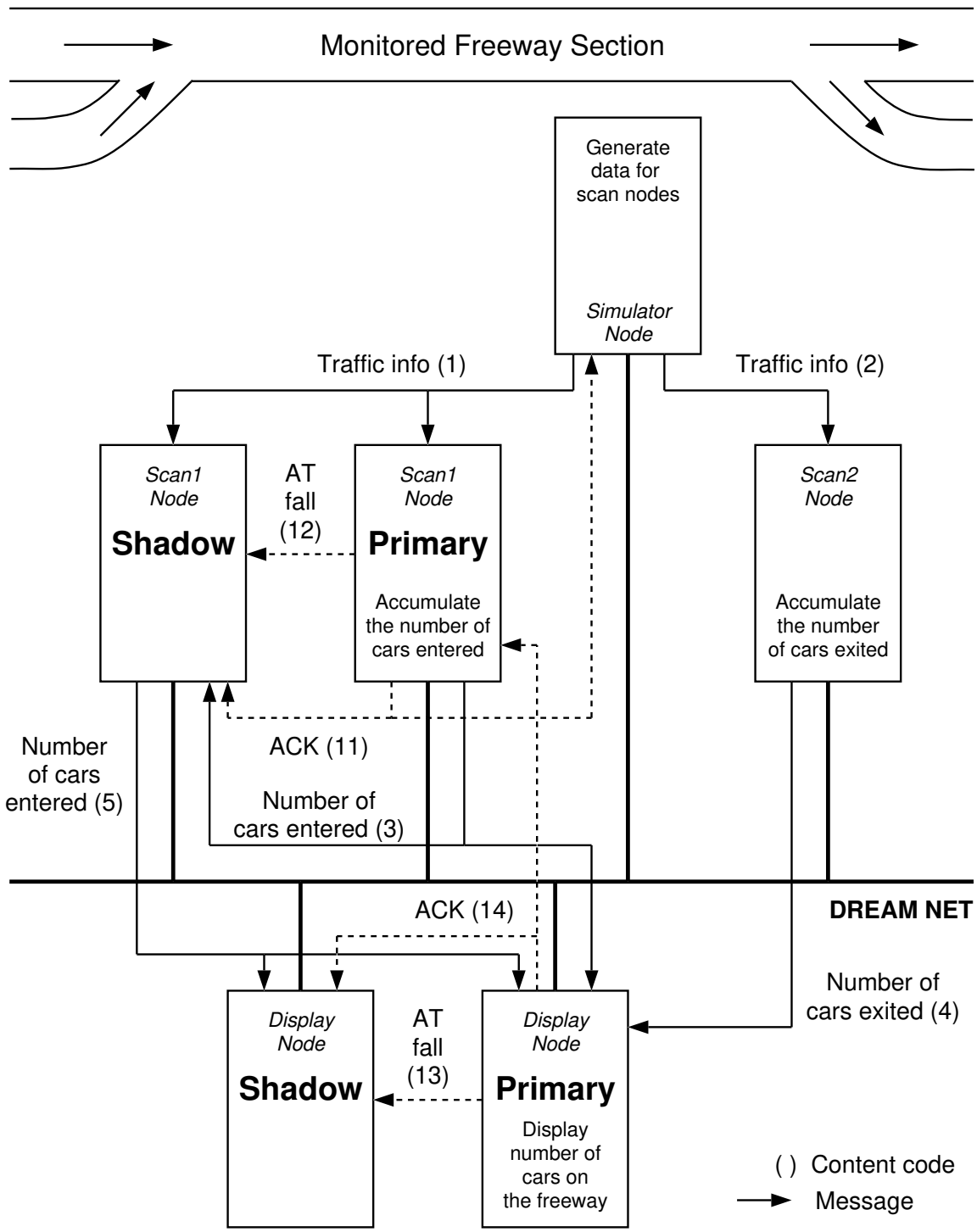


Figure 3.22: Structuring of the freeway monitoring application

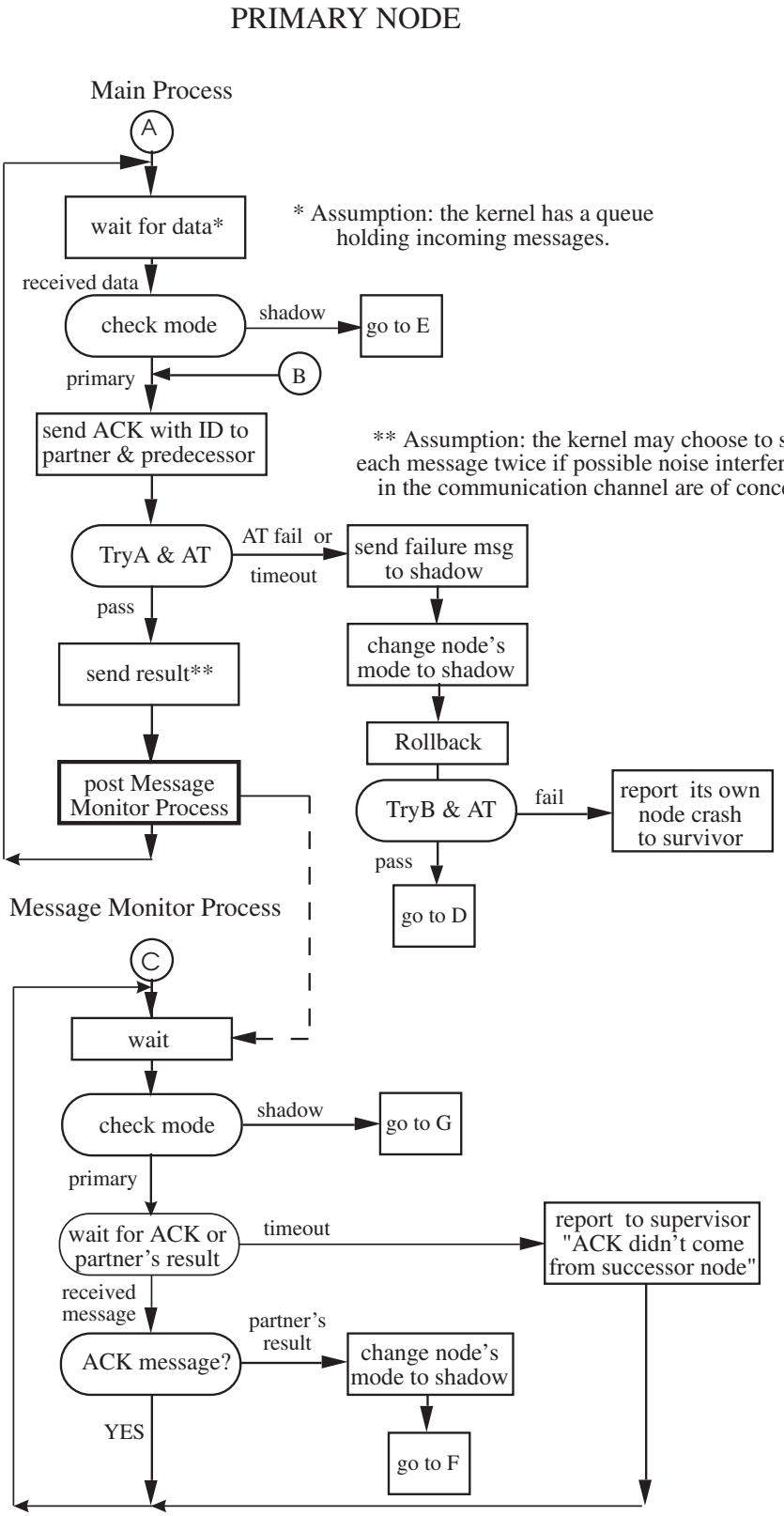


Figure 3.23: Details of the DRB protocols for the primary node from the freeway example

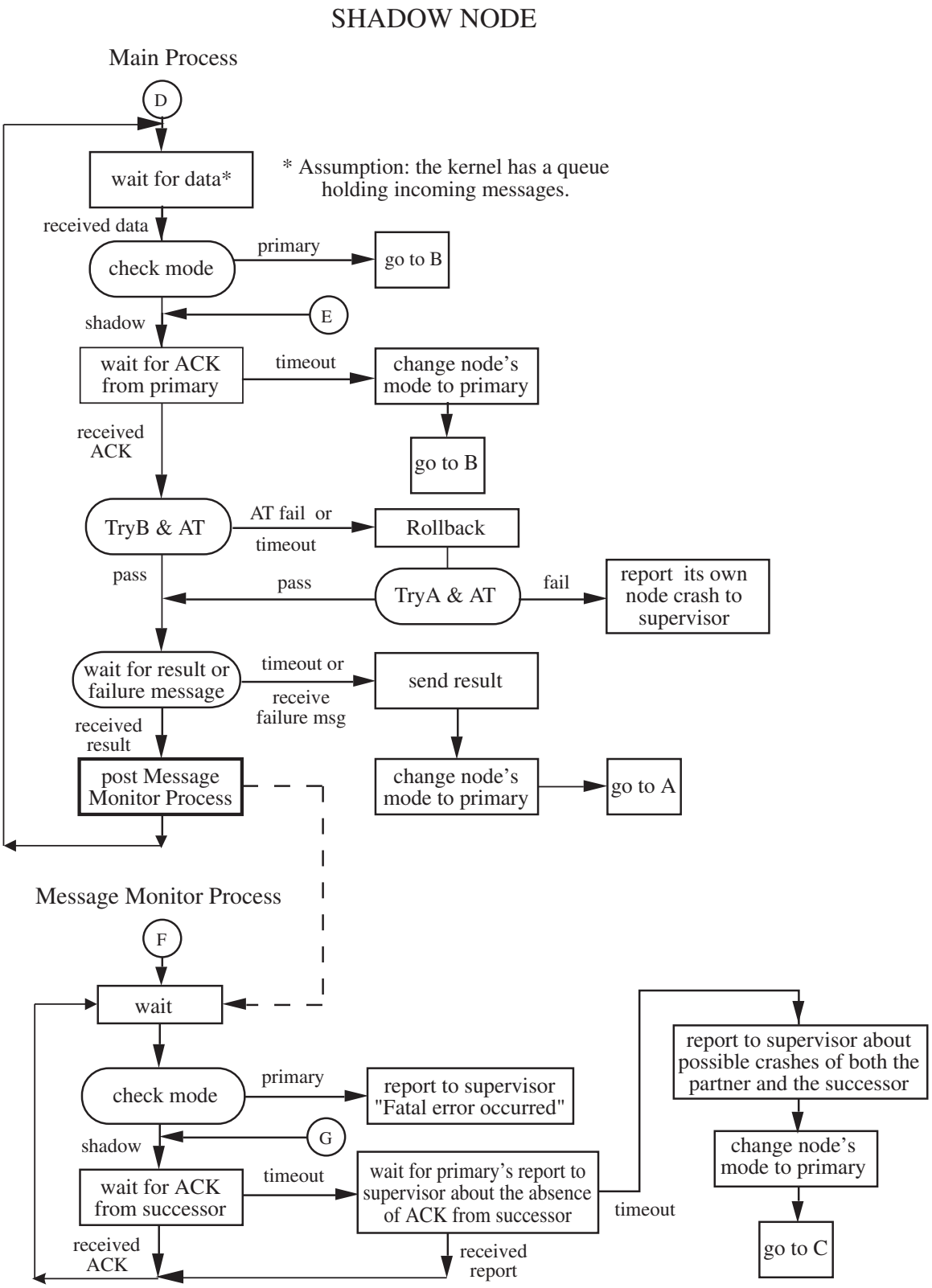


Figure 3.24: Details of the DRB protocols for the shadow node from the freeway example

with such capabilities will continue to emerge at least for the next several years. To facilitate efficient incorporation of new complementary mechanisms, an implementation model that has a modular and easily expandable structure has been devised. The promising nature of the model was confirmed during an experimental implementation conducted from July through September 1993 using both the model and a simple PC-network-based real-time distributed computing testbed.

3.2.9.1 Worker DRB computing stations and a supervisor station

In typical LAN-based real-time fault-tolerant systems using the DRB scheme, real-time data flows among several DRB stations, which are called worker DRB stations. We may also incorporate a supervisor station to make the system highly robust and to extend its lifetime [15, 24]. Figure 3.7 depicts a typical configuration of a system composed of several worker DRB stations and a supervisor station.

The supervisor station is generally responsible for detection of node crashes, detection of misjudgments by the nodes in DRB stations about the status of their partner nodes, and network reconfiguration, including task redistribution. Some of these functions — for example, detection of node crashes — can be decentralized [27]. In fact, new approaches for the detection of node crashes and the loss of messages, and for network reconfiguration have mushroomed in recent years, indicating that these areas are still immature. It is thus advantageous to implement worker DRB stations and the supervisor station such that the new, more cost-effective, complementary approaches and mechanisms emerging in the future can be easily incorporated. This means that it is highly desirable to have a modular implementation model for both the worker DRB station and the supervisor station.

This modular implementation model will facilitate the efficient incorporation of newly emerging approaches for

- Supervisor-worker cooperation
  - Arbitration of the conflicts arising between partners in a worker DRB station
  - Network reconfiguration
- Reliable multicast and crashed node detection.

All promising approaches for detection of crashed nodes and message losses require frequent communication, logging, and analysis of some form of *inquiry messages* and *acknowledgment messages* [15, 27]. Components responsible for such message production, logging, and analysis are thus integral parts of the implementation model. The DF scheme for providing dynamically configured multicast channels is also a part of the implementation model.

3.2.10 An Implementation Model for a Worker DRB Computing Station

A general structure of the implementation model for a DRB computing station is depicted in Figure 3.25. Each node in a worker DRB station uses a slightly different subset of the components, depending on its role (primary or shadow). The shared variable F_{ROLE} in

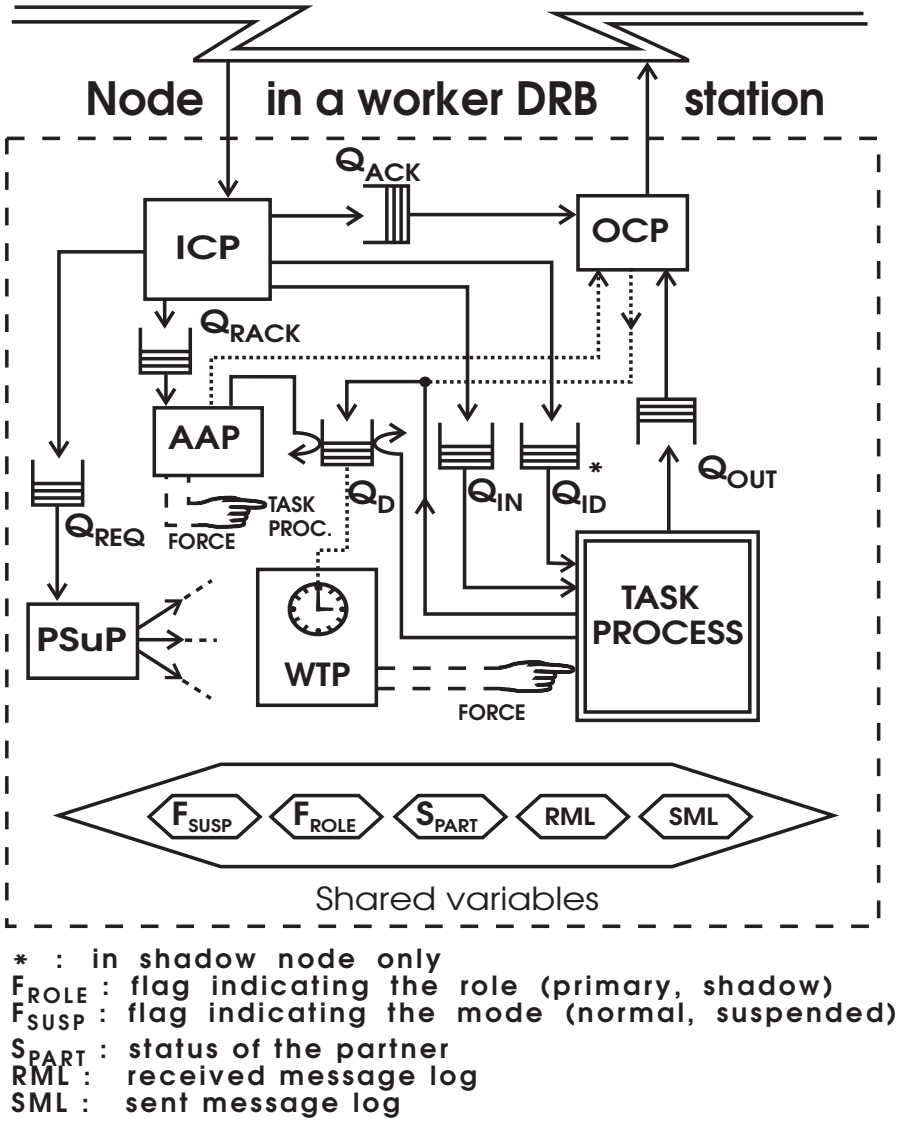


Figure 3.25: Structure of the modular implementation model for a node in a DRB station

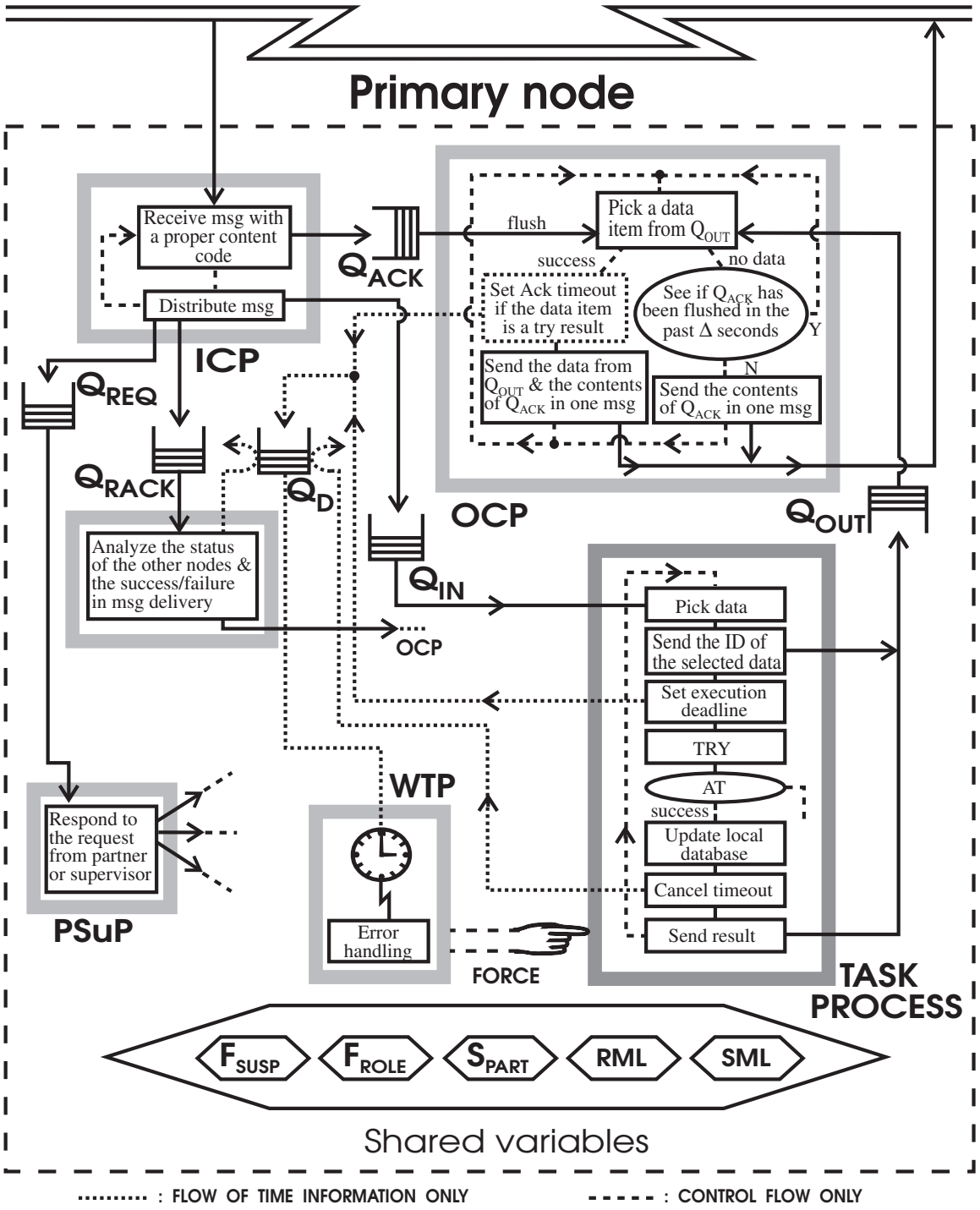


Figure 3.26: Detailed structure of the modular implementation model for a primary node

each node indicates the role of the node. Figure 3.26 provides some more details on the structure of the primary node.

A worker DRB station node has six basic types of processes: (1) Task process, (2) Incoming Communication process (ICP), (3) Outgoing Communication process (OCP), (4) Watchdog Timer process (WTP), (5) Acknowledgment Analyzer process (AAP), and (6) Partnership Support Process (PSuP).

The ICP is responsible for receiving all incoming messages (tagged with appropriate content codes) and distributing them to several data queues, based on their attributes. The data from the predecessor computing stations are stored into the queue Q_{IN} (which may actually represent a set of queues, Q_{IN1}, Q_{IN2}, ..., Q_{INp}, where p is the number of predecessor stations) in both primary and shadow nodes.

At the beginning of each data processing cycle, the primary node's Task process picks a data item from Q_{IN}. To ensure the input data consistency, the primary node's Task process informs its partner (in the shadow node) of the ID of the data item just selected. The ICP on the shadow node receives the ID message and inserts it into Q_{ID}. The shadow's Task process picks a data ID from Q_{ID} at the beginning of each data processing cycle. Using the ID, the shadow node's Task process selects the data item from its Q_{IN}. Whenever possible, both the primary node's Task process and the shadow node's Task process execute different try blocks in each data processing cycle, although they execute the same acceptance test.

The result data produced by the primary node's Task process are given to the OCP through Q_{OUT}. Then, the OCP sends out all such messages onto the multicast channel to which at least the successor stations, the shadow node, and the supervisor station are connected. Throughout the execution of the DRB scheme, a time-out mechanism is engaged for fault detection. The WTP is responsible for reacting to each time-out signal. The time-out values are registered into Q_D by the processes in need of time-out activations. For example, a time-out mechanism is engaged by the Task process before an execution of a try block. If such a time-out occurs, the WTP forces the Task process to start the same sequence of actions that is normally followed upon an acceptance test failure.

To facilitate both node crash detection and message delivery confirmation, all nodes in the system periodically exchange heartbeats in the form of acknowledgment messages. To be more specific, the following cost-effective approach is adopted. As the ICP receives each message from the channel, it produces an acknowledgment (ack) message and inserts it into Q_{ACK}. Normally, as shown in Figure 3.26, the OCP does not send an ack message separately. Instead, when the OCP is ready to send a message picked from Q_{OUT}, the OCP piggybacks the full contents of Q_{ACK} onto the message from Q_{OUT} and then sends the combined message. However, if the OCP notices that Q_{OUT} has remained empty for a period equal to or longer than a preset interval D, then the OCP packs the full contents of Q_{ACK} as a single message and sends it out. This *acknowledgment/heartbeat scheme* enables the AAP to detect node crashes and message delivery failures without much delay. When the crash of the supervisor is detected, the worker stations should elect a new supervisor station.

Once a node encounters an anomalous situation, it enters the *suspended mode* and this entry involves setting the shared variable F_SUSP to indicate the mode. For example, after the primary node sends a task output message but before it receives an acknowledgment from the successor computing station, it may receive a task output message sent by its

partner node. Such a situation may arise because of a fault in the receiver part of the shadow node (now acting as the primary node) or a fault in the sender part of the primary node. The primary node then enters the suspended mode and seeks advice from the supervisor. As another example, if the shadow node discovers that it has missed an application data message from a predecessor station because of a fault in its receiver part, it enters the suspended mode and seeks help from the primary node to obtain a copy of the message. In general, a node is in the suspended mode while it tries to get help from the supervisor and/or its partner node.

The PSuP handles the requests/orders from the supervisor station and the partner node. For example, an order from the supervisor station may be to switch the role from the primary node to the shadow node. A request from the partner may be to provide a copy of an application data message generated by a predecessor station. Or it may be a request from the newly assigned partner node for a copy of the local database. The knowledge a node has about its partner node — for example, the role and the mode of the partner node — is kept in the shared variable S_{PART}.

As the ICP receives an incoming message, it saves a time-stamped copy in the received message log (RML). The RML facilitates an investigation when an acknowledgment message corresponding to an unknown message is received. It also enables the PSuP to honor a request from the partner node for a copy of a message that the latter node missed. As mentioned before, a request for copies of the data messages may also come from a newly assigned partner node.

As the OCP sends out a message, it also records a time-stamped copy in the sent message log (SML). One use of the SML is by the AAP, which, among other things, attempts to confirm the successful delivery of each message sent out.

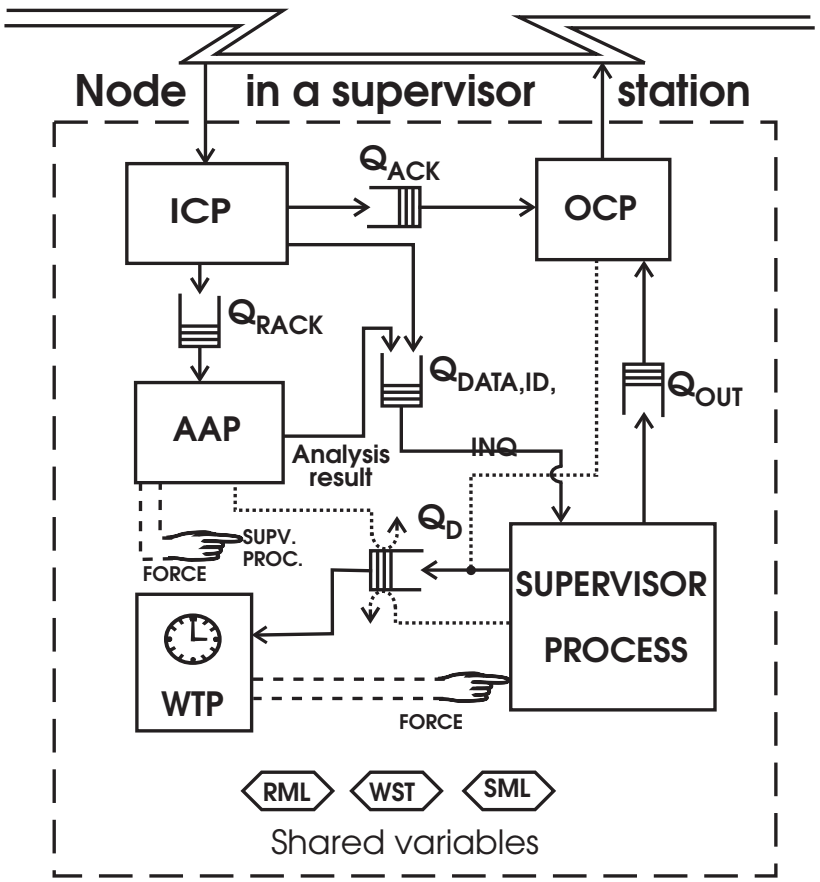
If the primary node's Task process fails to pass the acceptance test, it sends a failure notice to both the partner node and the supervisor station before making an attempt for a rollback-and-retry. Because the local database has not been updated yet and the RML contains the input data item, all the information needed for a rollback is available as long as the main memory is not corrupted. That is, no other preparatory actions are needed to facilitate rollbacks. A node with a memory hardware fault can be treated in the same way a crashed node is treated, even if the memory fault is transient. That is, such a node is taken out of the DRB station for off-line repair and then can be reincorporated as a new partner node.

3.2.10.1 An implementation model for a supervisor station

A general structure of the implementation model for a supervisor station is depicted in Figure 3.27.

The supervisor station is basically responsible for detection of misjudgments by the nodes in DRB computing stations about the status of their partner nodes, detection of node crash and message delivery failure, and network reconfiguration including task redistribution.

A supervisor station controls five types of processes: Supervisor process, ICP, OCP, WTP, and AAP. The Supervisor process monitors the messages communicated between nodes of worker DRB stations and detects misjudgments by the nodes—for example, detection of two primary nodes in a DRB station. It may also receive inquiries from nodes in the



RML: received message log
SML: sent message log.
WST: worker status table
INQ: inquiry from a node in a worker DRB station

Figure 3.27: The structure of the implementation model for a supervisor node

suspended mode regarding possible mistakes by themselves or their partner nodes, such as message transmission failures and message reception failures. When a shadow node decides to become a primary node, it also sends a request to the supervisor node for a posteriori approval.

To support network reconfiguration, the status of worker nodes is preserved in the worker status table, WST. Once a new worker is inserted into the system, the Supervisor process assigns a job to the new mode. For example, the job may be to function as a shadow node in a certain DRB station. As in the case of a worker DRB station, the function of the AAP is to detect node crash and/or message delivery failure by continuously analyzing the ack messages coming in. In the case of a detection, the AAP reports it to the Supervisor process so that the latter may orchestrate a network reconfiguration action. The AAP as well as the WTP may discover that the supervisor node has been faulty, for example, missing a message, or trapping the Supervisor process in an erroneous loop. Upon such discovery, the AAP or WTP can force the Supervisor process to take an appropriate action for conditioning the system for continuous operation, such as ordering other nodes to elect a new supervisor node, or crashing the supervisor node.

3.2.11 An Experimental Validation of the Modular Implementation Model

As a supplement to the logical validation of the implementation model formulated, an experimental implementation based the model discussed in Section 3.2.9 was conducted from July through September 1993. A simple kernel called the ECPM (Extended Concurrent Pascal Machine) was used. The ECPM has evolved in the UCI DREAM (Distributed Real-time Ever Available Microcomputing) Lab over the past ten years and was designed to support real-time processes distributed over a LAN. The kernel was extended to incorporate the DF scheme. The simple case of freeway traffic monitoring used in the earlier experimental implementation discussed in Section 3.2.8 was again implemented on the PC network (PCN) testbed. Two worker DRB stations were implemented and a skeleton version of the supervisor station, which detects misjudgments made by the nodes in worker DRB stations but does not yet possess other capabilities such as network reconfiguration, was established.

The kernel in each node uses dual ready queues and the time-sliced processor allocation approach. The time slices are allocated alternatively to the two ready queues. In addition, a powerful dynamic priority scheme called *priority-bracket scheduling* [22] is applied for allocation of time slices among the processes within each ready queue. This flexible structure is highly useful in scheduling the processor time in each node to support the processes depicted in Figures 3.25 and 3.27. Because the ICP is considered to be the most time-critical process, it was assigned to monopolize one ready queue, whereas other processes were assigned to compete within the other ready queue. This experimental study effort was halted before reaching the full validation stage because of the change in the demonstration plan of the AFR project.

Chapter 4

Adaptive Distributed Recovery Block Demonstration

The results gathered from the Adaptive Distributed Recovery Block (ADRB) demonstration program and lessons learned from designing, implementing, and executing the demonstration are described below.

The ADRB demonstration and the AFRS (Adaptive Fault Resistant Systems) Release of the Alpha Distributed Operating System that supports it are described in a number of other documents[1]. Particularly:

- *Alpha Software, AFRS Release: Computer System Operator's Manual* describes the generation and operation of the AFRS Release of the Alpha Distributed Operating System and its applications
- *Alpha Software, AFRS Release: Software User's Manual* documents the AFRS Release of the Alpha Distributed Operating System and its adaptivity demonstration for potential users of the system
- *Alpha Software, AFRS Release: System/Segment Specification* discusses the Alpha Distributed Computer System Testbed that hosts the AFRS Release and its applications
- *Alpha Software, AFRS Release: Software Design Document* details the design of the AFRS Release and of its adaptivity demonstration

We address three major areas: (1) the behavior of the ADRB facility, including its performance, (2) the structure of the ADRB facility and the application software that employs it, and (3) general lessons learned in applying ADRBs in a system.

The adaptivity demonstration of the AFRS project implemented an Adaptive Distributed Recovery Block facility with a notional plot correlator application function. The ADRB facility is capable of executing in three distinct modes in order to provide a better match between the system requirements and available system resources, each of which can be dynamically changing.

[1] Separately bound

The ADRB facility displays each of these three modes and behaves as expected in each mode. An experimenter can direct the ADRB facility to change ADRB operating mode at any time, and can cause the facility to transition smoothly between any of the three modes. We analyze Specific performance results, although this is not a production system; it is intended to display the ADRB technology and to permit experimenters to study it in a notional situation.

The structure of the ADRB facility and the separation of that facility from the basic application code should allow for an orderly expansion of function in future versions of this software.

Devising, designing, implementing, and exercising the adaptivity demonstration also illustrated a few desirable properties for application functions that are to be encapsulated by an ADRB facility. Most of these properties result from the need to execute two try blocks simultaneously on different nodes in a distributed system. Coordination of their execution is complicated if large databases must be shared or kept consistent between these parallel application functions.

The adaptivity demonstration is operational and permits the ongoing study of ADRBs in an Alpha Distributed Operating System context.

4.1 Adaptive Distributed Recovery Blocks

Adaptive Distributed Recovery Blocks are extensions of Recovery Blocks (RBs) [40] and Distributed Recovery Blocks (DRBs) [24]. ADRBs provide multiple modes of operation in order to provide a better match between system requirements and available system resources, each of which can be dynamically changing.

ADRBs provide application services despite

- Software faults during recovery block execution
- Hardware faults during recovery block execution
- Node failures
- Timing faults (a result that was produced too late)

ADRBs can be configured to utilize or conserve processing resources. The Alpha-provided ADRBs can operate in three distinct modes

- Mode I — Serial recovery blocks. Mode I incorporates two recovery (try) blocks and an acceptance test. For each input datum, the primary try block is executed. If the generated result passes the acceptance test, the results from the primary try block are used. Otherwise, the second try block is executed and its results are passed to the acceptance test. If all results fail the test, an error is reported to a higher level of control.
- Mode II — Concurrent recovery blocks. Mode II is designed to reduce the latency of the recovery block when a fault occurs. Here, too, there are two try blocks and an acceptance test. In Mode II, both try blocks are executed concurrently on separate

nodes, and their results are independently tested by copies of the acceptance test. If the primary try block passes the acceptance test, its results are used. Otherwise, if the secondary try block passes the acceptance test, then its results are used. Unlike Mode I, the secondary try block has already been executed if the primary try block fails the acceptance test. This mode is also known as a distributed recovery block (DRB).

- Mode III — Recovery block with skip. Mode III is designed for the situation where resources are scarce and time is pressing. There is one try block and one acceptance test. The try block is executed. If the generated result passes the acceptance test, the results of the try block are used. Otherwise, the recovery block is ended and the current input is skipped — that is, no result is generated for that input. (This is an acceptable action for some application functions, including the function employed for the adaptation demonstration.)

4.2 ADRB Behavior

Particular behavior of ADRBs is expected in the presence of faults, and in certain situations, a change between ADRB operating modes can be made. ADRB mode behavior is supported by experimental measurements.

4.2.1 Operational ADRB Mode Characteristics

The Alpha-hosted ADRB facility clearly exhibits the three operational modes described in Section 4.1 when executing the ADRB demonstration program. The gross characteristics of each mode are exactly as expected, specifically

1. When a software fault is detected in Mode I, recovery cannot occur until at least another try block has been executed.

2. When a software fault is detected in Mode II, recovery, if it is successful, occurs quickly because the both ADRB stations are working to produce answers at the same time. To obtain such low latency, two try blocks are executed in parallel for each input report.

3. When a software fault is detected in Mode III, recovery takes longer than in Mode II, but is much quicker than in Mode I. However, the quality of the result suffers. In fact, for this demonstration, the default result computed under Mode III indicates that the incoming report could not be correlated with any track (because there was insufficient time to really check). This is, in fact, an acceptable outcome for this application in most cases.

Section 4.2.3 reviews some quantitative experimental results to illustrate these points.

4.2.2 Dynamically Changing ADRB Modes

If ADRBs are to be truly adaptive, it must be possible to dynamically change ADRB operating modes. In fact, ideally, mode transitions would occur instantaneously and "seamlessly." The coordination required to accomplish such transitions can be considerable.

Nonetheless, the Alpha-hosted ADRB facility in the adaptivity demonstration allows the experimenter to change from any ADRB mode to any other ADRB mode at any time. The time required to accomplish the transition is small compared to the anticipated try block execution times of many tens of milliseconds. This is arguably the most noteworthy aspect of the adaptivity demonstration.

One factor that can delay the completion of an ADRB mode transition is the time required to obtain access to critical data structures that describe the current operating mode and configuration. (An *ADRB configuration* is an ADRB operating mode, along with an indication of the ADRB stations involved in the execution and their respective roles.) The current configuration is accessed and recorded along with other critical information for each invocation. The ADRB facility should, in theory, be capable of processing a sequence of application invocations where each individual invocation is carried out in a different ADRB mode.

Transitions into ADRB Mode II from either Mode I or Mode III require additional coordination to assure consistent views of the existing track database.[2] The issue of databases and ADRBs is discussed in some depth in Section 4.4.2.

4.2.3 Experimental Measurements

As stated in Section 4.2.1, the three ADRB modes behaved as expected in the face of faults. However, a number of factors can affect the desirability of one mode over another. For example, if inputs can occasionally be skipped — that is, not processed by the ADRB facility — or can be queued momentarily, Mode I might be far preferable to Mode II when faults are rare and not clustered. This determination might be based on the fact that, on average, Mode II requires twice as many processing cycles (and other computing resources) as Mode I when faults are infrequent. Mode I requires as many total processing cycles as Mode II only when a fault is encountered.

On the other hand, if faults occur frequently, the total number of processing cycles consumed by Mode II compared to Mode I narrows and the reduced recovery latency Mode II provides becomes much more attractive.

To appreciate such aspects of system operation, the ADRB demonstration presents information related to resource consumption, in total and on average, as well as the occurrence of faults and errors and their recovery.

The ADRB demonstration provides the experimenter with a number of parameters to adjust, including

1. ADRB operating mode

2. Try block execution time

[2]Transitions from ADRB Mode II into either of the other modes do not involve this delay because much less state must be propagated in those cases.

3. Input arrival rate

4. Software fault rate

5. Input queueing

These parameters can be adjusted to study each of the ADRB operating modes under various loads and fault environments.

Elevating the software fault rate tends to make the display more interesting because there are more faults to recover from and, in fact, there can be errors due to multiple faults experienced by two try blocks for a single input datum. While executing the demonstration under such conditions is most interesting, it might not be the most relevant, if it does not reflect realistic circumstances.

To address that concern, the ADRB demonstration permits the experimenter to select parameters that are most appropriate for that experimenter's application. If the experimenter wishes to have an even better feel for the relevance of ADRBs for another application function, the plot correlator function is encapsulated in an Alpha object type and could be replaced by another object type that computed another application function. (Section 4.3 discusses the structure of the system in greater depth.)

The following subsections provide a few interesting quantitative results gained through use of the adaptivity demonstration, relative to the use of ADRBs. The table below shows the average values measured for a few critical variables over a large number of runs, featuring a range of input arrival periods, try block execution times, and software fault rates. "TBT" designates try block execution time, and "FTBT" designates the fast try block execution time of the code executed by Mode III when it detects a fault.

Measurement (Avg)	Mode I	Mode II	Mode III
Total Cycles–no fault	TBT + 15 ms	2*TBT + 70 ms	TBT + 15 ms
Recovery Latency	TBT + 6 ms	24 ms	FTBT + 6 ms
Total Cycles–fault(s)	2*TBT + 21 ms	2*TBT + 70 ms	TBT + FTBT + 21 ms

4.2.3.1 Fault Rates and Error Rates

The application experiences an error when the Alpha-hosted application program cannot correctly identify the track with which a given report is associated. There are three potential causes of errors in the adaptivity demonstration:

1. The ADRB-encapsulated plot correlator specifies a track other than the correct track.

2. The ADRB-encapsulated plot correlator experiences a sufficient number of faults that it cannot produce an acceptable result for a given input report.

3. The ADRB-encapsulated plot correlator is too busy to respond to a new input arrival.

The plot correlation function is sufficiently forgiving of faults and errors that there is no evidence that it has ever designated an incorrect track identification for an input report.

Both of the remaining error types have been seen and can be controlled by the experimenter. The rate at which faults occur is under experimenter control.

In ADRB Mode III operation, any fault will result in an error. So the error rate and the fault rate are identical. Under Modes I and II, two faults must occur for a single input in order for an error to occur. So the expected error rate is the square of the fault rate (assuming that software faults occur independently in different try blocks[3]).

The measured rate at which errors occur for a given software fault rate corresponds well with the expected error rate.

The final error class occurs when the application cannot keep up with the input stream. This is often a signal to change ADRB operating modes, although if it occurs in Mode III, there is no more efficient mode to select. In that case, the plot correlator can tolerate some loss of data, although accuracy might often be compromised.

4.2.3.2 ADRB-Related Overhead

ADRB-related overload could mean a number of things. In this discussion, it refers to the amount of time spent processing application invocations and replies in the ADRB facility itself — that is, in the ADRBInvokeWrapper and the ADRBWrappers. (See Section 4.3 for a discussion of these objects.)

The current Alpha DOS does not have a global clock, so it is difficult to measure certain times directly. That includes some portions of the ADRB-related overhead. As a result, the figure is approximated.

Mode I Overhead. Mode I overhead is derived from two measurements:

- The total compute time required to produce a result when there are no faults
- The recovery latency measured when a fault is experienced

Each of these times includes the time required to execute the Application Function's try block, so the overhead in each case is the measured time minus the try block execution time. Over a wide span of try block execution times, input arrival periods, and software fault rates, these numbers vary little.

When there are no faults, Mode I imposes roughly a 15 ms overhead. When one or two faults are encountered an additional 5 to 11 ms is added. (On average, this is an additional 6 ms.)

Mode II Overhead. Mode II overhead is approximated a little differently than Mode I overhead. In this case, the measured compute time reflects the time spent executing both try blocks. Consequently, by removing the actual time spent executing the try blocks, an

[3]The occurrences of faults in this demonstration are determined by drawing random numbers from independent uniform probability distributions. Consequently, they should be, essentially, independent. This might not be the case in a real system, where common programmer errors might provide a correlation between failures of different try blocks.

approximation of the Mode II overhead can be made. Once again, over a large range of try block execution times, input arrival periods, and software fault rates, the overhead figure varies little.

Whether or not there are faults, the ADRB Mode II overhead consistently runs at about 68 to 74 ms.

This higher overhead number, when compared to Mode I, is understandable. First, two full invocations are performed compared to Mode I's single invocation (when no fault is encountered). In addition, these numbers were measured using a two-node testbed configuration. In such a configuration, one node acts as a dedicated ADRB station, while the other acts as an ADRB station and the interface to the application program, the ADRB Manager, and the external world. During executing in Mode I, the ADRB station selected is the one that is local to the ADRB Manager, eliminating the need to incur the additional overhead of remote invocations to access the ADRB station. In Mode II, one of the ADRB stations always requires a remote invocation. By looking at the recovery latency for Mode II, it is possible to estimate the additional time required to access the remote ADRB station — on average 24 ms. This time would be significantly reduced in the processor-pair per node configuration suggested by Kim in Chapter 3.

Mode III Overhead. Mode III overhead is calculated in the same manner as Mode I overhead. In fact, the figures are virtually identical to Mode I — approximately 15 ms of overhead when no faults are encountered, and 5 to 11 additional ms when faults are encountered.

The important difference between Mode I and Mode III is not the amount of ADRB-related overhead, it is the reduced computation time required when a fault is encountered.

4.2.3.3 Minimum Input Arrival Period

Based on the ADRB-related overhead calculations of Subsection 4.2.3.2, a lower bound can be placed on the interarrival time between input reports. So for example, even with an instantaneous try block execution and no faults, Mode I or Mode III could handle at most about 66 inputs/second/ADRB station. The maximum figure for Mode II would be approximately 29 inputs/second/ADRB station pair.

4.3 ADRB and Application Structure

The system structure permits the separation of the code that provides the ADRB facility from the application code, making the ADRB facility largely transparent to the application program. The internal organization of the ADRB facility is is structured to allow future growth, with clear object capability.

4.3.1 Separation of Application and Adaptivity Software

From the start, it was intended that the ADRB software would be separated from the application software to the greatest practical extent. Ideally, an Alpha application program that employed operation invocation to execute application functions could be mechanically

transformed to invoke the ADRB facility, rather than application functions, each time it executed a function. The ADRB facility would then determine how many of what type of application function to invoke to provide the proper level of service to the application program. When these invocations replied, the ADRB facility would coordinate the selection of an acceptable result and return that result to the invoking program.

This ideal has been largely attained in the adaptivity demonstration:

1. When the Application Program object instance logically invokes an Application Function object instance, it actually invokes an ADRBInvokeWrapper object instance.

2. The ADRBInvokeWrapper routes the application invocation, along with other relevant data, to one or more ADRB stations.

3. Another object instance, called an ADRBWrapper, encapsulates each Application Function object instance on an ADRB station.

Consequently, the Application Program invokes a single operation on the ADRBInvokeWrapper and receives a reply, just as if it had invoked the operation directly on the Application Function. The ADRBInvokeWrapper and one or more ADRBWrappers coordinate as needed to route the actual invocation to the proper set of Application Function object instances in the proper order. And each Application Function object instance receives invocations from a single ADRBWrapper, just as it would normally receive them from an Application Program.

Despite the fact that the application and adaptivity software have been largely separated, they have not been totally separated. They affect each other in the following ways:

1. The Application Function object code, although essentially identical to that of a non-ADRB version, is presented as a set of try blocks, rather than as a single function. In addition, the Application Function must provide an acceptance test to check the results it produces. This seems to be an inherent consequence of using ADRBs.

2. The ADRBInvokeWrapper and the ADRBWrapper must know something about the request and reply parameters for the application. At the very least, they must know the size of the parameter blocks. This might not be an inevitable difference; stronger language support than Alpha currently provides might alleviate this problem.

3. In certain cases, including that presented by the adaptivity demonstration, the ADRBInvokeWrapper must understand the syntax and semantics of the application well enough to provide consistent results to the Application Program, despite the fact that the results are produced by two different Application Functions that can develop and maintain different internal state. (This is not a problem when the Application Functions do not have persistent internal state, or when the internal persistent state that they have cannot diverge.) Section 4.4 discusses this point for the adaptivity demonstration in greater depth.

4.3.2 Structure of Adaptivity Software

The adaptation software also has a fair amount of internal structure:

1. The AFR Manager controls all of the fault resistance facilities in the system, determining the allocation of resources to each fault resistance facility.

2. The ADRB Manager controls one of these fault resistance facilities. It organizes its resources as a number of ADRB stations and assigns these stations to perform specific application functions. In the future, more automated managers will select the ADRB operating mode for each application function using the ADRB facility. Today, the experimenter selects the desired ADRB operating mode. The ADRB Manager converts that operating mode into an ADRB configuration — that is, it assigns ADRB stations to support the selected ADRB operating mode. The ADRB Manager communicates the ADRB configuration to the ADRBInvokeWrapper.

3. The ADRBInvokeWrapper acts as the interface for the Application Program to the ADRB facility. It receives ADRB configurations from the ADRB Manager and breaks them down into a set of ADRB roles that are played by individual ADRB stations. Roles include, for instance, being the only ADRB station that is performing traditional, sequential try blocks on a single ADRB station, or being the primary or shadow ADRB station for execution of DRBs in parallel. For each invocation performed by the Application Program, the ADRBInvokeWrapper makes invocations on the participating ADRB stations, telling each the role that it should play for that invocation.

4. Each ADRBWrapper acts as the interface to a single Application Function. Each plays its role, coordinating with its associated Application Function and the ADRBInvokeWrapper as assigned.

This structure seems sufficiently general to permit continued growth of the ADRB and AFR facilities in the future. At the same time, the structure establishes clear guidelines for the responsibility of each of these objects in the adaptivity portion of the system, indicating those resources under each object's control and the locus of control in the overall AFR system.

Although no formal, reflective structure was employed in the adaptivity demonstration, the organization of the AFR and ADRB facilities is similar in spirit to that found in reflective systems, where, informally speaking, each layer of the AFR system observes and controls a lower layer, while interacting with an associated, parallel layer of the application.

4.4 Lessons Learned

Previous sections have already discussed a number of the lessons that the adaptivity demonstration has taught, including the feasibility of implementing an ADRB facility on the Alpha Distributed Operating System. Consequently, we focus here on issues that have not been discussed in depth earlier.

4.4.1 Desirable Application Function Properties

The work in an ADRB-based system is performed by a number of ADRB stations. Each station receives a sequence of inputs and produces a corresponding sequence of outputs. Although the ADRB stations coordinate execution with one another as needed for purposes

of ADRB management — for example, when two stations are executing the same function as parts of a DRB — they do not interact directly when executing their encapsulated application function.

When hosting such a model on an object-oriented operating system like Alpha, ideal application functions would be entirely self-contained; they would not invoke other objects to perform their services. In addition, they would perform their application function in two phases. During the first phase they would read data, and during the second, they would update their internal and external state, after determining that the computed result was acceptable.[4]

The plot correlator application function that was chosen for the adaptivity demonstration possesses these properties. The plot correlator function has been encapsulated into an object that makes no invocations on other objects. In addition, the plot correlator function reads the entries in a track database, searching for the best fit for a new radar report. In the end, when a candidate track is selected — or the plot correlator concludes that the radar report constitutes a new track — an acceptance test is applied to the result and the track database is updated if the result is acceptable.

The plot correlator possesses one other advantageous property for this demonstration: it can tolerate skips. That is, under overload, some input reports can be skipped without seriously affecting the quality of the plot correlation function. Without this property, every input would have to be processed, presumably until an acceptable result was obtained for each. Under stress, this would be a serious constraint on system operation. (Notice that Mode III takes direct advantage of this fact, skipping inputs when the primary try block fails. If this could not be tolerated, Mode III would not be viable, limiting the ADRB facility's options when the system is under stress.

There is one problem with the selection of the plot correlator function: it accesses a potentially large track database. Managing access to this database, particularly in Mode II, can be difficult.

4.4.2 ADRBs and Databases

If an application function is to be encapsulated by ADRBs, then the function must be structured so that it can be executed in Mode II, that is, DRB mode. In that case, two ADRB stations execute the application function in parallel. If the application function accesses a large database, then both ADRB stations must be permitted access to the database in parallel.

There are a number of ways to provide access to the database, and each has its own advantages and disadvantages:

1. The database can be shared by both ADRB stations. If the application function has the two-phase read/write property mentioned in Section 4.4.1, then it seems reasonable to have both ADRB stations read from a single, shared database; only one station

[4]This two-phase execution is not, strictly speaking, a requirement. However, if the application function does not possess this property, then more powerful facilities, like atomic transactions or an ad hoc equivalent, must be provided in order to allow the application function to undo changes that were made based on the assumption that the computation produced an acceptable result.

will actually write the acceptable result to the database. That way, both stations will always share a consistent view of the database.

There are at least two problems with this approach. First, the database is a shared resource and must be able to service twice the normal number of requests when the system is operating in Mode II. More seriously, all of the database references made by these two distributed, database-intensive application functions will be remote references — references to another node in most cases. This is a very expensive method when compared to direct reads from local memory or local files.

Providing multiple, consistent copies of the database for the application functions addresses the latter concern, but requires a powerful fault-tolerant technique to support ADRBs, if it is even practical.

2. A copy of the database can be passed to each ADRB station at the start of each application function execution. At the completion of execution, each station will update its copy. Only one copy will be selected for subsequent use, ensuring consistent views for all executions of the ADRB stations.

   This might be feasible for small databases, but seems prohibitively expensive for large databases.

3. Each ADRB station can create and maintain its own copy of the database. This ensures efficient access to the information, but poses other problems. Since either ADRB station might experience a fault during the execution of the application function, it is possible that their internal databases might diverge — that is, they might not be identical. The ADRB scheme does not permit the stations to exchange results at the conclusion of application function executions. As a result, another party must take the answers offered by the ADRB stations, interpret them, and produce a consistent result to return to the Application Program.

The adaptivity demonstration chose the last approach. It increased the complexity of the ADRBInvokeWrapper object, which was responsible for interpreting the individual results and producing the unified result for the Application Program. In this case, the ADRBInvManager produced an identifier for each unique track that was observed by the Alpha system. Each individual ADRB station's plot correlator would have its own local track identifier for that track. The ADRBInvokeWrapper maintained a set of translation tables to transform the individual plot correlator results into the unique Alpha track identifier.[5]

This choice also complicated transitions into Mode II from the sequential try block modes (Modes I and III). For these ADRB mode transitions, a copy of the track database of the active ADRB station's track table had to be transferred to the partner ADRB station.[6]

[5]Notice that this is not a universal function for such systems. This would be a more attractive option if it were. Performing the specified translations requires knowledge of the plot correlator function, both its syntax and its semantics. Different steps would be required if the application function was something other than plot correlation.

[6]This might make it seem that the databases cannot diverge; but they can. Each time multiple new tracks are detected, there is a chance that a fault will cause the ADRB stations to assign different track identifiers to the new tracks — even if they are running identical plot correlator code.

Despite these difficulties, this seemed to be the most realistic structure for the application at hand. It is difficult to imagine passing multiple copies of real — that is, very large — track databases around the system with the arrival of each new radar report; and it is difficult to imagine making the large number of invocations to read track entries that would be necessary for each plot correlation. If the actual plot correlation were to take place in the shared database object, then the individual results are far less independent then they are under the selected scheme.

Chapter 5

Adaptive Distributed-Thread Integrity

We applied the concepts of adaptive fault tolerance to the kernel-layer abstraction of distributed threads. Our goals were to formulate a high-level design for a portion of an adaptive distributed-thread integrity service and to examine the principles we have developed for adaptive fault tolerance. We focused on the thread maintenance and repair protocol that forms a key portion of this service.

First, we describe the distributed-thread integrity problem in the context of the Alpha programming model and supervisory real-time control systems. We then present the basic components needed to address thread maintenance and repair. To be more concrete, we give an overview of the Alpha TMAR protocol [37], which has been implemented, and the assumptions on which it is based. This protocol is used as the basis for the Thread Polling protocol, one of the protocols that we select for further study.

For adaptive fault tolerance to be viable, there must be at least two alternative implementations of the fault tolerance function, each of which performs better in different regions of the possible operating environment. We examine a variety of alternatives to the Thread Polling protocol, and select one called the Node Alive protocol. It is described in Section 5.6 along with the assumptions on which it is based.

We believe that the Thread Polling protocol and the Node Alive protocol are viable alternatives. The different regions of the operating environment for which they are best suited are presented in Section 5.7. We also discuss how each of these two protocols can be made adaptive by controlling their parameters.

We then discuss the adaptivity functions of monitoring, diagnosis, control, and meta-control in the context of TMAR. Based on our examination of these issues we believe that a practical, adaptive TMAR protocol could be constructed.

Finally, we describe a proof-of-concept simulation system that we implemented to study the TMAR protocols and adaptive fault tolerance. We present some experimental results that illustrate the benefits of using automatic adaptive control to switch between simplified versions of the Thread Polling and Node Alive protocols.

5.1 Alpha Thread Maintenance and Repair

Distributed threads are a programming model abstraction that is used to represent distributed computations. To be concrete, we will consider distributed threads in the context of the Alpha programming model [38, 11], which was designed for real-time supervisory control. Here, the programming model is provided directly by the Alpha distributed real-time operating system kernel, and distributed threads represent real-time distributed computations. When we refer to Alpha, we will usually be referring to the Alpha kernel.

As part of the basic programming model, it is important for the system to provide certain guarantees about the integrity of distributed threads. In Alpha, a distributed thread is a continuous, distributed execution point that transparently and reliably spans physical nodes in the computer system. If a distributed thread is broken, say because of a node failure, the Alpha kernel is responsible for noticing the problem and repairing the thread. Because Alpha is a real-time kernel, this must be done in a timely manner.

This portion of Alpha's distributed-thread integrity support is called *thread maintenance and repair* (TMAR). It is responsible for discovering thread breaks, identifying the continuous portion of the thread that can be recovered, identifying the portions that must be removed, managing removal and clean up, and resuming execution. The portions of a distributed thread that must be removed are called *orphans*. It is desirable to remove orphans as soon as possible to prevent them from consuming resources that could be used by valid computations and to prevent them from performing invalid or contradictory actions.

Providing integrity for distributed threads is similar to providing integrity for other distributed computation abstractions. For example, orphan detection and elimination has been studied in the context of nested transactions [17, 18, 31], remote procedure calls [36, 39, 45], and message logging protocols [2].

A specific TMAR protocol was developed for Alpha, based on assumptions about the expected operating environment [37]. This included assumptions about the workload, the user and application requirements, the system fault model, and the resource base and its configuration. However, this is a rich environment and we believe that other TMAR protocols would perform better under a different set of assumptions. To test this hypothesis, we devised numerous alternative TMAR protocols, and then selected two for more detailed study.

During the AFRS project, we identified three ways to build adaptive fault tolerance into systems. They are changing key algorithm parameters, changing algorithms, and reconfiguring the placement and use of processing and data. Here, we used the first two approaches as the basis for an adaptive TMAR scheme.

5.2 The Alpha Programming Model

The Alpha real-time distributed operating system kernel was designed to support supervisory-control real-time processing in a distributed computing system. The concepts of the Alpha programming model, described here, can be used to develop real-time distributed software.

The Alpha kernel provides the following abstractions for writing real-time distributed software:

- Objects, which are passive abstract data types that contain data and code organized into data-access and control operations
- Distributed threads, which are execution control points that move among objects via operation invocation
- Operation invocation, which is an approach similar to procedure calling by which distributed threads execute object operations

We will refer to distributed threads as *threads* except as necessary for clarity.

5.2.1 Objects

An Alpha *object* is an instance of an abstract data type, and exists entirely on a single node in the computing system. All resources in an Alpha system, such as devices, files, and distributed threads, appear as objects to the programmer.

5.2.2 Distributed Threads

An Alpha *distributed thread* is used to represent a distributed real-time activity or computation. It is a continuous, distributed execution point that transparently and reliably spans physical nodes in the computer system. A thread performs a computation by invoking operations on objects. Each invoked object can be located locally on the node where the thread is currently executing or remotely on another node. After several invocations, a thread can extend across a collection of nodes.

A thread begins executing by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation is called a *thread segment*. Consequently, a thread is composed of a concatenation of thread segments. The initial segment of a thread is called its *root* and the most recent segment of a thread is called its *head*. The head of a thread is the only segment that is active.

When a thread invokes an object operation, we say that the thread enters the object and becomes threaded in the object. A thread becomes unthreaded from (leaves) an object by returning from an invocation. A thread can simultaneously be threaded in a group of objects, and the kernel maintains a history of the objects in which a thread is currently threaded. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node. The first segment in the section results from an invocation from another node, and the last segment in the section performs a remote invocation. At any time, a thread is executing only within the object associated with its head and may directly access only the contents of that object.

A thread maintains its identity, its local state, and its attributes for timeliness, robustness, and so on, as it executes and moves among objects and nodes. These attributes are used by the Alpha kernel at each node to perform resource management on a system-wide basis. This makes it possible for the decisions of individual nodes to be in the best interests of the entire distributed application.

5.2.3 Invocation and Thread Creation

The *invocation* of an object operation is the vehicle for all interactions in Alpha, including operating system calls. Invocation has synchronous request/reply semantics similar to those of remote procedure calls (RPCs). As a result, operation invocations can be nested. If desired, the effect of an asynchronous operation invocation can be obtained through the use of thread creation.

An invocation may fail for various reasons, such as a protection violation, the use of incorrect parameters, a node failure, a machine exception, or the expiration of a time constraint. Notification of an invocation failure is returned to the invoking thread.

When a thread invokes an operation, a new segment is created for the thread. The invoking thread segment provides the initial parameters and capabilities to the new thread segment, which then executes the invoked operation. When that operation invocation subsequently returns, the returning thread segment passes any return values to the invoking thread segment, and is then removed. Invoking an operation of a local object does not involve the scheduler because, in this case, the successive thread segments resemble stack frames supporting nested procedure calls more than they resemble coordinated, independent threads.

Thread creation works similarly to ordinary operation invocation, except that it causes a new thread to be created and invoked on a specified object operation. The actual creation and initiation of the new thread occurs asynchronously from the execution of the creation operation. The new thread runs concurrently with and independently from the thread that creates it, and does not return any values to the thread that creates it when it terminates.

5.2.4 Illustration of Alpha Programming Model Concepts

Figure 5.1 presents a snapshot of a system employing the Alpha programming model. The system contains four nodes, and currently four distributed threads are executing. Each thread root is marked by the thread name, and each thread head is marked by a dot. As an example, thread 1 is rooted on node 1, it is threaded through objects on nodes 1, 2, 3, and then 2 again, and its head is located on node 2. Thread 1 may actually be executing on node 2, but we cannot tell for sure from this snapshot. Each operation execution forms a thread segment, and each maximal contiguous sequence of operation executions on a node forms a thread section. Thread 1 currently has four sections, two of which are located on node 2. The segment structure of section 3 is shown in more detail. Here we see that thread 1 has sequentially invoked operations for objects 1, 2, 3, and then 2 again, creating segments 12 through 15 for the thread.

5.3 Distributed-Thread Integrity

A computer programming model specifies the entities that represent computations, the entities that represent stored application information, and the means by which the computations interact with each other and the information. For example, a common programming model uses processes to represent computation, files to store application information, remote procedure calls to implement interprocess communication, and file reading and writing to enable processes to access files.

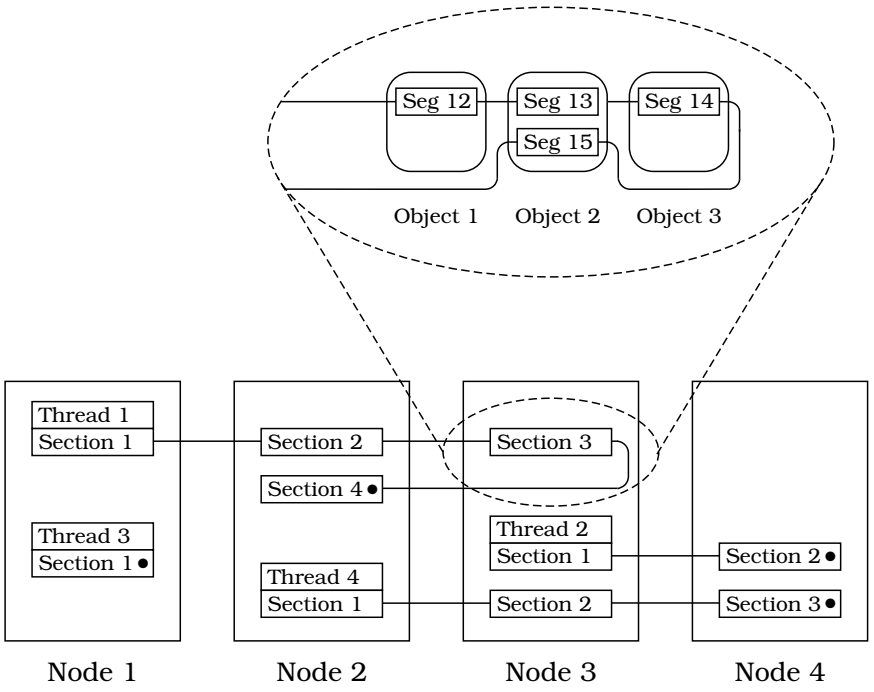


Figure 5.1: Illustration of the Alpha programming model concepts

Programming models like the one provided by Alpha are growing in popularity. In this approach, distributed threads are used to represent computations, and objects with operations are used to represent application information. Threads access application information by invoking object operations, and communicate with each other indirectly through shared object information. Except for timeliness, which is a primary concern in a real-time operating system like Alpha, the same issues must be addressed in any system that employs distributed threads in its programming model.

A variety of active entities called threads have been used in different programming models. A distributed thread (not necessarily an Alpha distributed thread) is unique in several ways. First, it moves from one node to another when it makes a remote operation invocation. Second, it maintains its identity when it makes an operation invocation, even if it is a remote invocation. The identity can include a variety of information, such as the thread's timing requirements and functional importance. Third, it usually does not possess its own address space, but enters and uses the address spaces of the objects it references.

The integrity of a distributed thread is important because it is a basic part of the programming model. Unlike thread abstractions that are bound to a single node, a distributed thread can experience a partial failure when a node fails because a single distributed thread can span several nodes. A node failure can break a distributed thread in one or more of the following ways: it can divide the thread into several pieces, remove the thread's root, or remove the thread's head. In addition, it can simultaneously damage several threads.

The operating system kernel must address the problem of thread breaks to provide the abstraction of a reliable, continuous distributed thread. There are two high-level approaches to providing thread integrity. In the first approach, which could be called optimistic, the

system tries to identify thread breaks. When a break is found, the system trims the thread back to the longest continuous piece from its root, and removes the remaining orphan sections. We will refer to this as the *thread trimming approach*. In the second approach, which could be called pessimistic, the system always maintains enough replicated information to endure thread breaks. When a thread break is encountered while returning from an invocation, the system transparently repairs the break and continues. For this study, we will only consider approaches of the first kind.

In Figure 5.2, we demonstrate the effects of a node failure on the set of operating threads. Assume that node 3 fails immediately after the snapshot shown in Figure 5.1. The failure has broken threads 1, 2, and 4. Threads 1 and 4 have been divided into two pieces, and thread 2 has lost its head. The orphan sections are indicated by dashed outlines. Notice that no thread had its head removed. The orphan sections with heads retain their heads, and can continue to execute until they are stopped by TMAR or until they try to return to an operation on a failed node.

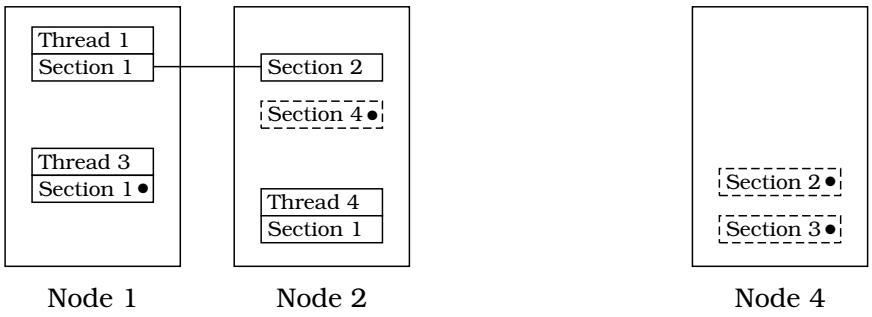


Figure 5.2: Effects of a node failure on executing threads

TMAR is responsible for detecting and recovering from the damaged state shown in Figure 5.2. In Figure 5.3, we show a snapshot of the system after TMAR has repaired the damage. This is only one of many possible states that could result after TMAR has completed its repair work. While TMAR is detecting the failure, analyzing the damage, and repairing the threads, undamaged threads such as thread 3 could continue to execute, and new threads could be created. All of the orphan sections and any further execution they may have performed have been removed. In addition, new heads have been identified and activated for threads 1 and 4.

5.3.1 System Assumptions

Assumptions about the expected operating environment have a strong influence on the efficacy of using adaptation. Fewer degrees of freedom in the operating environment imply that a single approach will be more likely to provide adequate service for the entire range of operating conditions. Consequently, we will make only a few general assumptions here. We consider the operating environment to consist of the hardware resources that are available, the configuration of the hardware resources, the fault model, the characteristics of the application workload, and the objectives of the users and the applications. In addition to

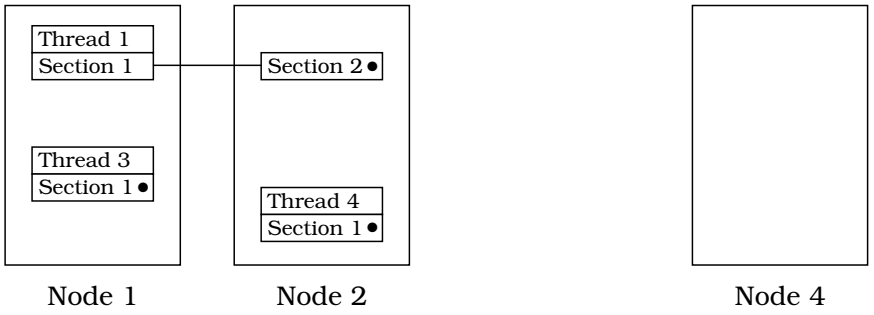


Figure 5.3: Repair actions performed by TMAR

assumptions about the operating environment, we assume that the application may have real-time processing requirements and that the application may interact with the real world.

In general, we assume that the system is asynchronous. We assume that it is organized as a group of processing nodes connected by a communication network. Each node can have one or more independent processors. A node can experience crash, performance, and omission failures. If a node requires several processors to operate and one of the processors fails, the node will undergo a crash failure.

We assume that the communication network can experience performance and omission failures. We make no assumptions about its size or about whether it can perform a hardware broadcast.

Because we are focusing on thread integrity, we will take an abstract view of the application workload. We will consider only the load it places on communication processing and the communication network and the characteristics of distributed threads, which include their structure, lifetime, and placement. We will make no assumptions in these areas. Examples of different thread characteristics include the number of threads rooted on a node, the number of nodes through which a thread is threaded, and the number of invokes during a thread's lifetime.

Similarly, we are concerned only with user and application requirements that affect thread integrity. We make no assumptions in this area except that the thread integrity service will only be expected to handle requirements for which it has provided an interface. For example, an application may be allowed to specify the relative weight that it places on break detection latency and certainty.

5.3.2 Components of the Thread Trimming Approach

The thread trimming approach to TMAR works as follows. First, the system looks for a broken thread. When one is found, the system identifies the longest continuous invocation chain starting from the thread's root. The last segment in this chain will become the head of the trimmed thread. The system then identifies the orphan sections of the thread and manages their removal. Part of orphan removal involves executing application-supplied repair code to undo or compensate for the actions of the orphans. After the orphans have been removed, the thread can resume execution from its new head. If a thread's root is removed, the entire thread essentially becomes an orphan.

We can divide the TMAR protocol into the following tasks. The tasks must be able to simultaneously handle multiple threads and multiple failures.

- *Collect break detection information*
  The system must gather enough information to allow it to determine that a thread break has occurred. It is desirable to detect a thread break early, but speed of detection must be balanced against the overhead that is incurred for normal operations.

- *Analyze break detection information.*
  At some point, the system must examine the information that has been collected to determine whether a thread break has occurred. Once again, the speed of detection must be balanced against the overhead of analysis. In addition, the analysis may be inconclusive and the system may need to gather more information to refine its diagnosis.

- *Declare break.*
  After enough evidence has been obtained, the system must declare that a thread has been broken. Once again, speed of detection must be balanced against certainty of detection. Because the system is asynchronous and can experience omission and performance failures for both processing and communication, TMAR may inadvertently declare a healthy thread broken. While this is unavoidable, it is very undesirable because it causes repair overhead, wastes correct work, and delays completion of the thread.

- *Collect repair information.*
  In addition to declaring a thread break, the system must analyze information that will allow it to repair the break. This information may have been collected previously, say with the break detection information, but if not, it must be collected now. The thread repair information will allow the system to identify the new thread head, the orphan thread sections, and the scheduling constraints for orphan removal.

- *Analyze thread repair information.*
  After collecting the thread repair information, the system must identify the new thread head and identify all of the thread's orphan sections.

- *Manage orphan removal.*
  The system is responsible for managing orphan removal. The old thread head must be prevented from executing, the orphan sections must be placed in their exception handlers, and the exception handlers must be supplied with scheduling constraints to govern their execution.

- *Resume thread execution.*
  After all of the orphans have been removed, the new thread head can be allowed to resume execution.

5.3.3 Alternative Task Designs

The thread integrity tasks can be implemented in a variety of ways, leading to TMAR protocols that emphasize different tradeoffs. When collecting information, whether for

break detection or break repair, the major questions are when, how frequently, how much, what, by whom, and from whom. When analyzing information, whether for break detection or break repair, the major questions are when, what, and by whom. The system must decide when it is certain enough to declare a thread broken, and what criteria it will use to make that judgment. When repairing broken threads the major questions are how, when, and who coordinates, and who participates. The system must also decide when to resume executing a repaired thread.

The design choices that are made establish different tradeoffs in the TMAR mechanisms and policies. These differences can be seen when the TMAR protocols are exposed to different operating environments. The major design properties that are found in the tradeoffs are

- *Break detection latency* — the time between the occurrence of a thread break and its discovery
- *Break repair latency* — the time between the discovery of a thread break and its repair
- *System overhead* — the amount of processing and communication overhead introduced when collecting break detection and break repair information, when analyzing break detection and break repair information, and when repairing thread breaks
- *Break certainty* — the likelihood that a thread is actually broken when the system declares it broken

5.3.4 Interactions with Other System Components

The subsystem that maintains thread integrity does not exist in isolation. It must interact with and be compatible with other major system components such as remote invocation, exception handling, and scheduling. The remote invocation subsystem is responsible for moving threads between nodes when an operation invocation or return occurs. It and TMAR have access to similar information about threads and nodes, and both are concerned with the reliable movement of threads. Remote invocation and TMAR should be designed to support each other without either one duplicating the other's activities.

The exception-handling subsystem provides a mechanism that allows the application to respond to exception conditions in an application-specific way. TMAR is responsible for placing a trimmed thread head in an appropriate exception handler at the right time after a thread has been repaired, and for placing orphan sections in appropriate thread handlers at the right times during thread repair. The application is responsible for providing semantically meaningful exception handler code to allow the application to recover from thread breaks.

The scheduling subsystem is responsible for deciding when computations will execute. In a real-time system, this will probably include the TMAR computations. TMAR may have to accommodate situations where application threads are scheduled ahead of TMAR work, and it will have to be designed to operate in a timely manner. In addition, TMAR will have to specify scheduling constraints for thread repair work, including application exception handlers.

5.4 The Alpha TMAR Protocol

The TMAR protocol that was implemented in Alpha was based on the assumptions described here. Other TMAR approaches that might be more appropriate under a different set of assumptions are discussed in the next section.

5.4.1 Alpha TMAR Assumptions

The term *design considerations*, as originally used for the Alpha TMAR facility, is intended to encompass not only requirements, but also preferences and any metrics that a designer might consult to make decisions. Adaptability, as discussed for AFRS, was not a design consideration.

The TMAR design considerations include the following objectives:

- Failure of the TMAR function should not harm or impede the progress of intact distributed threads.
- TMAR processing should not introduce significant overhead when threads invoke operations on remote nodes. It should be possible to use operations with a short duration that quickly read or write a value. It might be possible to write a better TMAR protocol, that is, one that would be more accurate, detect breaks sooner, recover from breaks sooner, and so forth, if TMAR processing could be executed on every remote invocation and/or return. However, this would be inappropriate if the duration of the TMAR processing swamped the duration of the operation itself.
- It should be possible to manage the resources consumed by the TMAR protocol, including memory, processing time, and communication bandwidth.
- It should be possible to adjust the desired thread break detection latency.

Many of the straightforward functional requirements, such as detecting a thread break and repairing it in a timely manner, have been omitted so that we can focus on those considerations that allow some leeway in design.

Alpha is currently implemented on multiprocessor nodes with three processors, each of which has a dedicated function. The Application Processor (AP) executes application code. The Scheduling Coprocessor executes the scheduling policy for both application and system threads. The Communication Coprocessor executes a group of communication-oriented protocols, including Remote Invoke and TMAR. The Alpha nodes are connected by an Ethernet.

The following assumptions underlie the Alpha Release 1.0 TMAR protocol:

- Alpha nodes are interconnected by a bus, and the bus can broadcast messages to all of the nodes.
- Communication errors are not common, which implies that datagrams are likely to be successfully received.
- Thread integrity is a basic function of the system, and its costs are viewed as overhead.

- Communication is heavily used, which implies that TMAR should be aware of the number and size of messages it uses.
- The number of nodes and distributed threads is small enough to permit thread integrity to be tracked on a per-thread basis.
- TMAR processing is hosted by the Communication Virtual Machine (CVM), which hosts several communication protocols on the Communication Coprocessor.
- TMAR is a decentralized protocol with each node responsible for all distributed threads that are rooted locally.
- Following a thread break, all orphan computations must be eliminated before a repaired distributed thread can resume execution.

5.4.2 The Alpha TMAR Protocol

The Alpha TMAR protocol monitors the integrity of threads and repairs threads when a broken thread is detected. Repair is based on trimming a broken thread back to its most recent viable point and then allowing it to proceed. Before the trimmed thread can be allowed to continue, orphaned remnants of the original distributed thread must be removed, and the state of the internal and external system must be restored or made acceptable. TMAR is also responsible for coordinating the trimming of a thread that has been aborted — for instance, because of an unsatisfied time constraint. It is a decentralized protocol executed by a number of identical peers executing asynchronously on each node in the system.

Responsibility for thread integrity is divided among several components. The primary component, the TMAR protocol, is responsible for detecting broken threads and coordinating their repair. The restoration of internal state and the execution of compensating actions are performed by application code executed by the exception-handling mechanism.

5.4.2.1 Detecting Thread Breaks

Every thread has a root segment, which is located on the node containing the object in which the thread was created. Each node is referred to as the root node for all of the threads that are rooted on that node. Each root node is responsible for maintaining the integrity of the threads rooted on it.

A root node accomplishes this task by periodically executing a three-phase protocol. In the first phase, the root node broadcasts a polling message containing a list of all of the threads rooted at that node that have made remote invocations. In the second phase, each of the nodes that contains sections of the listed threads reports this information to the root node, along with information about the location of the thread's head. This information is examined by the polling node to verify that an unbroken sequence of sections exists between the root and head of each thread in the list. (Some missing section and head reports are tolerated, because communication over an Ethernet is subject to omission failures. However, a consistently missing section or thread head is certainly a problem.) In the third phase, the polling node broadcasts a refresh message that announces the health of its locally rooted

threads to all of the other nodes. If a thread is unbroken, the message indicates that the entire thread should be refreshed; if it is broken, only the continuous portion of the thread that begins at the root and ends at the point of the break is refreshed. The sections beyond the break must be eliminated (aborted) before the thread may resume execution at the point of the break. Upon receipt of a refresh message, the other nodes refresh local thread sections as directed.

If a specified time interval is exceeded since the last refresh message for the sections of a thread on a node, then it is assumed that the root node for that thread has failed. In that case, all of the surviving sections are orphans and must be eliminated. Over time, this three-phase poll-respond-refresh sequence keeps all of the unbroken threads intact, without interrupting the AP.

5.4.2.2 Trimming Broken Threads

The Alpha programming model requires threads to be trimmed according to the proper application-defined scheduling parameters. When scheduling parameters are needed to schedule thread section aborts during a thread repair, TMAR locates the proper parameters and circulates them to the nodes that need them. Until the proper scheduling parameters are received, the abortion of thread sections is scheduled on each node using a default time constraint.

When a thread is being trimmed, each node trims the sections local to it that are beyond the point of the break or abort. When a node trims a thread section, it executes all applicable exception handlers. After all of these sections have been aborted, the root node for the thread is notified. The root node continues to perform its normal polling cycle for the thread, refreshing only that portion preceding the break or abort point. It keeps track of the aborted sections as it is notified of them and determines when the entire thread-trimming operation has been completed — that is, when each section beyond the break or abort point has either been aborted or determined to have failed (presumably because of a node or communication link failure).

Once the entire thread has been trimmed, the thread's root node notifies the node containing the thread's new head that the trimmed thread may resume execution.

5.4.2.3 Responsiveness and Overhead

The use of broadcast and multicast messages allows the amount of information that must be maintained for each thread in the system to be kept small. For instance, a root node may never know exactly where the head of one of its rooted threads is, or how many sections the thread has. It cannot be sure because it only knows what it received in response to its most recent polling message. Because messages may be lost, it cannot be sure that it received a report about all of the existing sections. Furthermore, even if it did receive all of the responses, the thread may have subsequently executed a new invocation that took it to another node, thereby creating a new thread section. Threads can move freely without explicitly reporting their movements to their root node as long as they respond to the root node's polling messages.

Polling messages are sent periodically by each node in the system. In addition, the time limit within which a thread must be refreshed is a programmable, integral number of

polling cycles, as is the number of responses for any thread that must be missed before it is assumed to be broken. Therefore, the polling frequency establishes the responsiveness of the system—that is, the length of time that passes after a thread break occurs until it is conclusively detected. This allows the protocol to trade communications bandwidth for thread break responsiveness: if necessary, a system can be made more responsive by employing sufficient bandwidth to achieve a polling interval short enough to satisfy any responsiveness requirements.

The TMAR protocol can handle successive failures because it continues to perform its normal three-phase refresh function on a thread's root node even while the thread is being trimmed. Subsequent thread breaks or aborts will be handled as soon as they are recognized. This makes the protocol somewhat more complicated than it would have to be if subsequent breaks and aborts could be deferred, but it will improve responsiveness and reduce the time that the trimmed thread must wait before proceeding in several cases, most notably when multiple nested time constraints cover a relatively small interval of time.

Although it is straightforward for the root node to wait for the next polling cycle to take many actions, responsiveness can be enhanced if it acts as soon as possible. Consequently, optimizations have been inserted into the protocol to allow the root node, and in some cases other nodes, to act without waiting for a polling cycle.

5.5 Alternative TMAR Protocols

To gain some understanding for the adaptations that were possible, we examined numerous alternative designs for TMAR. Our goal was to explore the design space and try to identify approaches that were better suited for different assumptions or tradeoffs. Here is a brief description of the idea behind some of the alternative schemes (or portions of schemes) we considered. (The first bullet is a generic description of the Alpha TMAR approach.)

- *Thread Polling.* Each node is responsible for the threads rooted on it. The other nodes are polled for information about these threads. When a broken thread is detected, the new head and orphan sections are identified, and instructions are issued for removing the orphans. All threads are treated equally. One advantage of having a thread's root node be in charge of TMAR is that a thread is usually considered irrevocably broken if its root fails. We will refer to this as the *Thread Polling* protocol.

- *Self Checking.* Each thread is responsible for checking its own health; the system is not responsible. Part of a thread's execution time will be devoted to performing TMAR functions. A disadvantage of this approach is that it is difficult to predict when a thread's TMAR functions will be performed. An advantage is that the TMAR functions are scheduled as part of the thread's activity, and they will not be performed if the thread is not considered beneficial enough to execute.

- *Partner Checking.* Each thread with more than one section (i.e., it has at least one active remote invocation) has a partner thread that performs TMAR for it. The partner is a stub on the original thread's root node that never makes a remote invocation. This is similar to the previous idea except that a thread is not directly blocked by its

own TMAR activity and the stub can execute with different scheduling parameters than its partner.

- *Synchronous Head Movement.* When a thread executes a remote invocation or return, it synchronously reports its movement to its root node. This allows its root node to know where all sections of the thread are located, including the head. If the root node is responsible for checking on the thread, it can direct its polls to this set of nodes. A disadvantage of this approach is the execution delay that a thread experiences after every remote invocation or return. This might be acceptable in systems where many threads rarely make remote invocations or returns.

- *Node Health.* Thread break detection may be based on checking node health instead of thread health. If a failed node is identified, additional checks are then made to determine what threads are broken and where they are damaged. Information about threads that can be used for thread break detection and repair can be gathered in a variety of ways and at several different times relative to the node health checks. This could be an attractive approach if there are many threads with many sections, because of the reduction in overhead during correct operation. A sudden burst of thread repair processing immediately after detection of a failed node could be inappropriate for a real-time system. It is probably less desirable to falsely identify a broken node rather than a broken thread because of the extensive repair that could occur.

- *Selective Section Checking.* TMAR could ignore the health of certain thread sections. This could reduce processing and communication overhead and reduce unnecessary thread repairs. For example, the initial sections of a thread could be ignored if the thread does not need to return to them again. A similar situation is when a section will just be passed through by a return because the remote invocation was the last instruction of the section. To make this approach work, the system will need to properly handle time constraints and exception blocks.

- *Invoked-Node Checking.* Each node could check on the health of the nodes to which it has made a remote invocation on behalf of some thread. A node failure would be independently detected by all of the nodes that were checking its health. While the checking nodes would know which threads were broken, additional information would need to be exchanged at some time to make it possible to determine the new thread heads, the orphan sections, and the proper scheduling parameters for repair.

- *Section History.* Each thread could carry history information about its sections with it when it makes a remote invocation. This would avoid the need to constantly relearn the location of the thread's sections, but it could introduce overhead into remote thread movement. If node health checking is used, this approach could supply the thread break and repair information. This approach could be attractive if threads rarely make remote invocations and returns.

- *Thread Walking.* A thread's health could be checked by sending a pulse down the thread. The thread's root node would send a message to the node with the thread's first section, which would send a message to the node with the thread's second section,

and so on, until the node with the thread's head is reached. If a break is found, repair information would have to be collected. This approach could reduce processing and communication overhead, but it might result in high break detection and break repair latencies. One optimization would be to allow a node with multiple sections for a thread to simultaneously send multiple check messages.

In general, TMAR considers all of the threads in the same way and all of the nodes in the same way. It may be beneficial to divide the threads and nodes into groups that would receive special treatment. For example, threads with a high functional importance or threads that are close to completing might have their health checked more frequently. Similarly, nodes that have been slow to respond or nodes with many threads may have their health checked more frequently. The system could adapt by dynamically forming groups and moving threads or nodes between groups.

Another idea is to create a hybrid TMAR protocol that combines several of the ideas discussed above. Threads or nodes with certain workloads or other characteristics could be protected by suitably matched protocols. For example, a low overhead approach with high break detection and repair latencies could be used for most threads, while a high overhead approach with low break detection and repair latencies could be used for a small number of special threads. The system could adapt by dynamically selecting which protocols would be active and which threads or nodes would use them.

5.6 The Node Alive TMAR Protocol

After considering the various ideas for performing TMAR, we selected several of them to form an alternative TMAR protocol that we call the Node Alive protocol. Our goal was to construct a credible alternative to the existing Alpha TMAR protocol, or more generically the Thread Polling protocol, that would be based on different assumptions and would occupy a different portion of the design space. We would then be able to study an adaptive TMAR protocol that could switch between these two protocols, either on command or as the expected operating environment changes.

The Node Alive protocol is based on the node health and synchronous head movement schemes described in the previous section. The node health approach is used to detect system failures. Instead of directly looking for thread breaks, the protocol will look for node failures, which will then be used to identify problems with threads. Thread break detection and repair information will be collected through the use of synchronous head movement. Every time a thread executes a remote invocation or return, it will reliably inform its root node about its movement. This will allow the root node to always know where all of the thread's sections are located.

5.6.1 Node Alive TMAR Assumptions

To allow a reasonable comparison between the Node Alive and Thread Polling protocols, we have retained most of the Alpha TMAR assumptions for the Node Alive protocol. Only a few key assumptions have been changed.

We assume the same hardware base and architecture. The system is asynchronous and consists of a collection of nodes in a local area network. The nodes are multiprocessors with

three processors, each of which has a dedicated function: application processing, scheduling, and communication. The local area network is an Ethernet with broadcast capability. The nodes can experience crash, omission, and performance failures, and the network can experience omission and performance failures.

The system is assumed to be a real-time system. This means that the system and application computations have time constraints associated with them, and that their execution is scheduled. The scheduling policy will probably not provide fair access to the application processor, and application computations may interact with the real world. Care should be taken so that the TMAR protocol will not cause a sudden overhead when an exception is discovered.

Of the four Alpha TMAR design objectives, we were able to maintain two.:

- The resources consumed by TMAR should be manageable.
- The thread break detection latency should be adjustable.

The third objective is that the failure of TMAR should not harm or impede intact threads. The node health portion of the protocol could delay or block the execution of a good thread if it took a long time or was unable to determine the set of good nodes. The synchronous head movement portion of the protocol could delay or block the execution of a good thread if it took a long time or was unable to respond to a thread's movement notification. However, it should not be difficult to write a relatively reliable agreement protocol to identify the good nodes, and the system should be able to quickly respond to a synchronous head movement interrupt.

The fourth objective is that TMAR should not introduce significant overhead when threads perform remote invocations or returns. Synchronous head movement could introduce significant overhead, including four remote message delays and two interrupt handling delays. This is mitigated by assumptions about the relative size of communication delays, the relative execution duration of remote operations, and the relative frequency of remote thread movement.

We retained all but one of the Alpha thread-polling TMAR assumptions for the Node-alive protocol. Those assumptions are

- The nodes are connected by a bus that can broadcast messages.
- Thread integrity is a basic system function and its cost are overhead.
- Communication errors are not common, which implies that datagrams are likely to be successfully received.
- Communication is heavily used, which implies that the protocol should be aware of the amount of message traffic it generates.
- TMAR processing is hosted by the CVM on the Communication Coprocessor.
- TMAR is a decentralized protocol with each node responsible for all locally rooted threads.
- All orphan sections must be eliminated before a repaired thread can resume execution.

The Thread-polling protocol assumption that we do not retain is that the number of nodes and threads is small enough to permit thread integrity to be tracked on a per-thread basis. We do not make any assumption about the number of nodes and threads, or about the number of sections per thread or the number of threads per node.

For the Node-alive protocol we make the following new assumptions:

- The communication latency is relatively small compared to the execution duration of remote computations.
- It is not common to make frequent remote invocations to computations with short execution durations.

5.6.2 Description of the Node Alive TMAR Protocol

The Node Alive protocol is based on the idea of having a group of good nodes, where all nodes in the group agree on the membership of the group. A good node is a node that is healthy, able to participate in system activities, and known to the other good nodes. The set of good nodes is identified by running a system-wide node group membership (NGM) protocol. When this protocol is run, new healthy nodes can join the group and failed nodes will be removed from the group. The NGM protocol will be based on ideas in group membership protocols that have been proposed for synchronous systems and for asynchronous systems [41].

5.6.2.1 Detecting and Repairing Thread Breaks

When an NGM protocol is run, the good nodes will agree on the group membership. In addition, they will agree on a circular ordering for the nodes. Each node is responsible for monitoring the health of the node that follows it in the circular order. Depending on how the protocol is designed, each node will exchange one or more messages with its neighbor to determine whether its neighbor is still alive. If it becomes suspicious of its neighbor's health, it will request that a new NGM be run.

After the NGM protocol is completed, a new group will be formed. If nodes are missing, then they will be considered to have failed. Any threads with sections on a failed node will have to be repaired. If a broken thread was rooted on a failed node, then other nodes with sections of that thread will independently recover from the work performed by those sections and remove them. A node recovers by executing an exception handler containing application-specific code provided for a section (if one exists).

The repair of other broken threads will be coordinated by the root nodes for those threads. As in the Thread Polling protocol, the root node will identify the new thread head, the orphan sections, and the scheduling parameters for the repair work. It will notify the nodes with the orphan sections that they need to perform thread repair, and will supply them with the appropriate scheduling information. When all of the nodes with orphans notify the root node that they have completed their repair work, the root node will notify the node with the new thread head that it can resume execution of the thread.

5.6.2.2 Collecting Thread-Break Detection and Repair Information

Thread break detection and repair information is collected by synchronous head movement. When a thread executes a remote invocation or return, it reports its movement to its root node. This allows the root node to have accurate information about the location of all of the sections of all of its rooted threads, including the location of their heads. When a node failure is identified, each node can immediately check to see whether any of its rooted threads are broken. It will also have all the information it needs to manage their repair.

Synchronous head movement involves the following steps:

- A thread's current node notifies the thread's root node that the thread is going to move.
- After some delay, the root node handles the notification.
- The root node sends an acknowledgment to the current node.
- The thread is moved to its destination.
- The thread's destination node notifies the thread's root node of the thread's arrival.
- After some delay, the root node handles the notification.
- The root node sends an acknowledgment to the destination node.
- The thread is added to the destination node's ready queue and scheduled for execution.

A thread will be blocked if its root node is unresponsive.

Notice that synchronous head movement might cause a thread to experience a significant execution delay due to the four communication delays and the two notification-handling delays. Because, in general, little work is needed to handle movement notification they probably will cause little execution delay. However, a node might be busy with a more beneficial computation. A smaller overhead will result if the root node is the same as either the current node or the destination node.

A root node needs scheduling information for its rooted threads so that it can send appropriate scheduling parameters during thread repair. This scheduling information, which is the same as the scheduling information needed by the Thread Polling protocol, could be collected during synchronous head movement, or the root node could request it after a node failure is detected.

5.6.2.3 False Break Detection

Because of the asynchronous nature of the system and the possibility of omission and performance failures in the nodes or the network, it is possible for a node to become suspicious of its neighbor even though its neighbor is still healthy. In fact, the node that called for the NGM may be the node that is having problems. For example, it may have failed to receive proper response messages from its neighbor, or it may not be participating in the check protocol in a timely way.

Identifying the group of good nodes is a two-step protocol. First, a node checks on the health of its neighbor. If it is suspicious, it starts an NGM protocol where all of the

good nodes come to agreement about the set of good nodes. If a node is mistaken about its neighbor having failed, this will most likely be caught by the NGM protocol, which will then keep the node in the group. A node can expend more effort to increase the likelihood that its diagnosis is correct, but this will increase node failure detection latency and hence thread break detection and repair latencies. Thus, a balance must be struck between diagnostic risk and speed of detection.

There must also be a balance between the amount of work performed by the initial neighbor check and the following NGM protocol. A node should not casually call for an NGM protocol because it is potentially expensive. All good nodes must participate in the protocol, and a number of rounds of message exchanges may be needed.

Because the system is asynchronous, an NGM protocol can also incorrectly identify a good node as failed. Unlike a true distributed consensus protocol, which has the properties that

- all good nodes are in the group
- only good nodes are in the group
- all good nodes agree on the group

an NGM protocol has the properties that

- only good nodes are in the group
- all nodes in the group agree on the group

This means that a good node may accidentally be left out of the group. While such a node can later rejoin the group, the cost of a mistake in a TMAR protocol is high because the system must incorrectly endure a great deal of repair work. An NGM protocol can expend more effort to increase the likelihood that the group membership is correct, but once again this will increase detection and repair latencies for true failures. The Node Alive TMAR protocol will probably be designed so that the NGM protocol is more likely to be correct than the neighbor checking protocol.

5.7 Adaptive TMAR

Adaptive TMAR protocols can be constructed from the Thread Polling and Node Alive TMAR protocols we have described. Here, we discuss changing parameters associated with the two protocols, and switching between the two protocols. In the next section, we consider the adaptive functions needed to carry out these adaptive changes.

5.7.1 An Adaptive Thread Polling Protocol

The Thread Polling protocol has several parameters that can be adjusted. They are the polling frequency, the poll response time-out period, and the number of polling rounds needed to declare a thread break. In addition, the protocol does not have to treat all threads identically. Adjusting all of these factors dynamically will allow the Thread Polling protocol to be tuned to a wide range of operating environments.

The polling frequency controls a tradeoff between overhead and break detection latency. Each poll requires a root node to prepare a poll packet, broadcast the poll packet to all other nodes, and analyze the poll responses. The other nodes must prepare and send a response packet. This could constitute a significant amount of processing and communication overhead. On the other hand, the more frequently this information is collected, the sooner a broken thread can be detected. The setting of this parameter would depend on factors such as the load on the network, the load on the nodes, the number of threads, the average number of sections per threads, the number of nodes, the application's need for quick break detection, and so forth.

The poll response time-out period would govern a tradeoff between break detection certainty and break detection latency. By waiting for a longer time period before analyzing poll response packets, a root node would be more likely to have received all of the poll responses, in spite of network and node performance failures. Thus, it would be less likely to mistakenly declare a thread broken. On the other hand, it would take longer before a truly broken thread could be detected. This parameter could be affected by the load on the nodes, the load on the network, historical information about performance failures, the application's need for quick break detection, and so forth.

The number of polling rounds needed to declare a thread break also affects the tradeoff between break detection certainty and break detection latency. Because of the asynchronous nature of the system and the protocol, each polling round can produce incomplete or contradictory information. For example, a thread section may be missing because a poll response was lost, and two nodes might report that they have a certain thread's head. Examining the responses from several polling rounds would help to clear up mistakes and identify true problems. It is desirable to avoid mistakenly declaring a thread broken because of the significant repair work involved and the detrimental effect on the thread. On the other hand, it is desirable to detect a truly broken thread as soon as possible. This parameter would be affected by the same factors as the poll response timeout period.

Currently, the Thread Polling protocol treats all threads identically. Let's say that quick break detection is needed for some thread. This would require the protocol to provide frequent polling for all threads, which could result in high overhead, or a small number of polling rounds for break detection for all threads, which could result in numerous false thread break repairs, and so forth. The protocol could dynamically select threads for special treatment according to the current situation. For example, polling information could be collected more frequently for the most important threads, and threads that had executed for a long period of time and were close to completing could require a problem to persist for a larger number of polling rounds before they could be declared broken. The number of threads chosen for special treatment and the specific parameters selected for them would be governed by factors such as those listed above, as well as the thread's functional importance, the amount of work the thread had already done, the thread's closeness to completing, and so forth.

5.7.2 An Adaptive Node Alive Protocol

Several parameters can also be adjusted in the Node Alive protocol to allow it to adapt to the current operating environment. They are the strength of the neighbor-checking protocol,

the strength of the NGM protocol, and the frequency of neighbor checking. By strength, we mean the effort expended by the protocol to overcome omission and performance failures.

The strength of the neighbor-checking protocol affects the tradeoff between the certainty with which a node failure (and hence a thread failure) is detected and the node failure detection latency. By using a larger time out period to wait for late messages and by using retries to overcome lost messages, this protocol can reduce the likelihood that it will incorrectly deduce that a node has failed. However, this will increase the amount of time before a true node failure can be addressed. This protocol also affects the tradeoff between system overhead and failure detection latency. If a node suspects that its neighbor has failed, it will start an NGM protocol, which requires the participation of all nodes and an exchange of several rounds of messages. This overhead can be reduced by decreasing the likelihood that a node will start an NGM protocol, at the cost of increasing detection latency for a truly failed node.

The strength of the NGM protocol has a similar affect on the tradeoff between the certainty and latency of failure detection, and on the tradeoff between system overhead and the latency of failure detection. Increasing the strength of the NGM protocol will increase system overhead, will reduce the likelihood of incorrectly deducing that a node has failed, and will increase failure detection latency. A mistaken node failure detection by an NGM protocol can result in a significant amount of thread repair work and wasted application execution, as well as the ripple effect of an increased likelihood that application threads will experience timing constraint failures.

The frequency with which nodes check the health of their neighbors affects the tradeoff between system overhead and the speed with which node failures are detected. More frequent checks result in more network traffic and more processing interruptions, but unless these checks tend to result in relatively frequent unnecessary NGM protocol executions, this overhead is relatively modest.

5.7.3 Switching between Thread Polling and Node Alive Protocols

Another approach is to use algorithmic adaptation and dynamically switch between the Thread Polling and Node Alive protocols. The idea would be to identify operating environments that would favor one of the protocols over the other, to monitor the state of the operating environment, and to activate the protocol that was best suited for the current environment. The adaptation could be caused by the user or application announcing that the system was entering a particular operating environment, or it could be accomplished by the system deducing the expected operating environment.

The adaptation will center around tradeoffs among system overhead, break detection latency, break repair latency, and the certainty with which a break is detected. Although the protocols can affect these tradeoffs in a variety of ways, two key factors will be the overhead caused by thread polling and the overhead caused by synchronous head movement. Thread polling will place a burden on the communication network and the Communication Coprocessor, while synchronous head movement will delay thread execution.

Let us consider some other factors that can affect the choice between the two protocols. It appears that the Node Alive protocol is more likely than the Thread Polling protocol to experience bursts of overhead and repair when exceptions occur. For example, the

NGM protocol can cause widespread delay as all nodes participate in the protocol to reach agreement about the group of good nodes. In addition, the declaration of a node failure by the NGM protocol can cause widespread thread repair work. These effects, which are undesirable in a real-time system because of their adverse affect on system predictability, would be more harmful in certain operating environments than in others.

Both of these protocols can mistakenly declare that a failure has occurred. It appears that it is more costly for the Node Alive protocol to do so. If the Node Alive protocol makes a mistake, all good nodes will declare that another node has failed. This can be costly because it will require the system to repair all threads with sections on that node. If a mistake is made with the Thread Polling protocol, some node will declare broken those of its threads with sections on some other specific node. (It will come to this conclusion because it will have missed one or more response packets from that other node.) This will probably involve significantly less repair work.

On the other hand, it appears that it is more likely for the Thread Polling protocol to make this kind of mistake for a given amount of break detection and repair latency. One reason for this is that the Thread Polling protocol would require multiple polling rounds over a relatively long period of time to increase its certainty. Another is that the Thread Polling protocol does not expend a lot of effort on any individual polling round to avoid missing and inconsistent information. The Node Alive protocol, on the other hand, makes a concerted effort over a short period of time to obtain an accurate census of the good nodes.

In order for adaptation between the two protocols to be warranted, there will have to be distinct real operating environments that favor each of the protocols. Looking at some extreme cases, we conjecture that the Node Alive protocol would be favored in an environment with many threads, many nodes, many sections per thread, and small communication delay, while the Thread Polling protocol would be favored in an environment with few threads, few nodes, few sections per node, and a large communication delay. In the first case, there would be a large overhead for producing poll packets and poll responses, and for analyzing poll responses, while there would be a small delay for synchronous head movement. The situation would be reversed in the second case. Less extreme situations would be more balanced, but we expect that there will be numerous operating environments favoring each of the protocols.

5.8 Adaptivity Functions

To provide an autonomous adaptive TMAR protocol, the system will have to supply the adaptivity functions of monitoring, diagnosis, control, and metacontrol, as discussed in Chapter 2. These functions can be implemented in a variety of ways in a concrete system architecture. They can be organized into one or more components and can be placed in one or more of several system layers. The control functions could be specialized for TMAR, or could serve a larger group of fault tolerance needs.

TMAR is located in the operating system kernel. At this low layer of the system, it would probably be inappropriate to implement extensive versions of the adaptivity functions. Simplified versions could probably coexist with the TMAR protocols, but more elaborate versions would probably have to be located at a higher system layer. This should not be a

problem, however, because adaptive changes would probably take place at a relatively slow pace, based on longer-term changes in the operating environment.

We assume that the TMAR protocols that are to be controlled will have been analyzed before the adaptivity functions are designed. This analysis will indicate what information needs to be collected to guide the control decisions, how well each version of the TMAR protocols performs with respect to different operating environments, how different changes will affect the performance of the protocols, and so forth. The analysis could be performed analytically, by simulation, or by observing a running system.

Interestingly, TMAR itself could be considered to be a simple adaptive fault tolerance technique based on the principle of reconfiguration. TMAR monitors the state of the system looking for broken threads. When it diagnoses that a thread is broken, it issues control commands to clean up orphans and to resume the thread. Adaptive adjustment of a TMAR protocol's parameters and adaptive switching between two simple adaptive TMAR protocols can be viewed as a higher, reflective layer of adaptive control. Higher layers of adaptivity will be discussed in Section 5.8.4.

5.8.1 Monitoring

The monitoring function is responsible for obtaining the information that will be used to decide whether a change will be made to the current TMAR protocol, and what that change will be. From the point of view of TMAR, the goal is to collect enough information to characterize the operating environment for TMAR, not the complete environment. After analyzing the TMAR protocols, it may turn out that only a small amount of information is necessary to guide the adaptation decision. For example, it may be sufficient to know, for some recent time period, the average number of thread sections in the system and the average communication delay. Some of the information that is needed could be extracted from the information already collected by TMAR. Some additional information could be obtained by piggybacking it on TMAR messages that are already exchanged. If more elaborate information about the state of the system is needed, it should probably be collected outside the kernel.

5.8.2 Diagnosis

The information gathered by the monitoring function is analyzed by the diagnosis function. Its goal is to determine whether the current instantiation of the TMAR protocol sufficiently matches the expected operating environment. Things to look for might be a change in the communication load, the frequency of message omission failures, the number of threads in the system, the execution patterns of threads, and so forth. Note that problems experienced by the current TMAR instantiation, such as mistaken declarations of breaks and high communication overhead, may not indicate that the TMAR protocol needs to be changed. Similarly, an instantiation of the protocol that is not currently experiencing any problems may need to be changed because it is not the best match for the expected operating environment. If a mismatch is found, this fact is conveyed to the control function.

5.8.3 Control

The control function is responsible for modifying the current instantiation of the TMAR protocol to bring it in line with the expected operating environment. The diagnosis function will notify the control function if it discovers a mismatch. The control function uses this information plus the information gathered by the monitor function to decide whether a change should be made, what change to make, and how to make the change. The control function may decide not to make a change even though a mismatch was identified because it is an inappropriate time or because there is nothing better to do.

With the two TMAR protocols we have been considering, the control function could decide to adjust a protocol parameter or to switch between protocols. A variety of strategies could be used. For example, if the Thread Polling protocol was currently running, the control function could decide to change the polling frequency based on the information it had available. This could be done incrementally, with the control function observing the effect on the system. (Recall that the information available to diagnosis and control will inevitably, to some extent, be incomplete, inconsistent, and out of date.)

If the control function decided to switch between the two protocols, it would be desirable to do so quickly without delaying the execution of application threads. We believe that it would be possible to do so, although while the switch occurred, some threads might not be covered and some extra overhead might be incurred. (Failures that occurred during the switch would eventually be discovered, although after a somewhat longer delay than usual.)

An example of a control function would be to instantiate the Thread Polling protocol or the Node Alive protocol, depending on the value of a key metric such as the average communication delay. The monitoring function would measure the average communication delay over a recent time period. If the value rose above a certain threshold, the system could switch to the Thread Polling protocol. If it fell below a certain threshold, the system could switch to the Node Alive protocol. (Recall that synchronous head movement, which is used by the Node Alive protocol, is adversely affected by a high level of communication delay.) A hysteresis gap could be used to prevent the system from thrashing. This is shown in Figure 5.4.

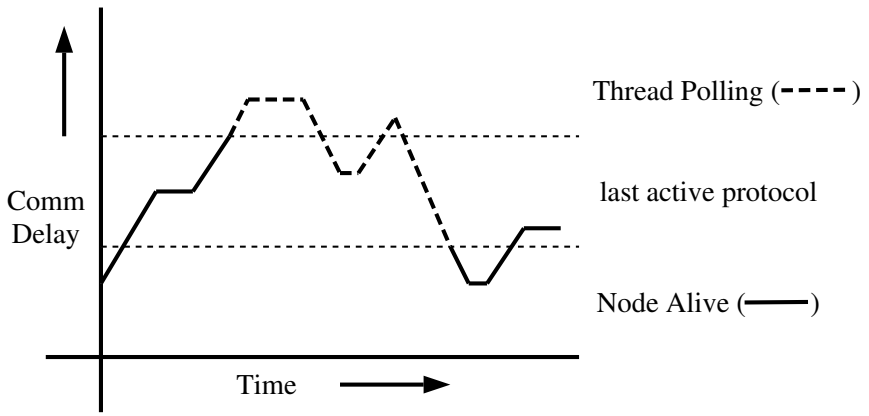


Figure 5.4: Example control function using hyst.pseresis

5.8.4 Metacontrol

The metacontrol function adjusts how the control function operates, based on information gathered about the state of the system and the past performance of adaptive control. In the case of TMAR, it could tell the control function how certain it should be about a mismatch before a change is made, how big a mismatch is needed before a change should be made, how large a hysteresis gap to use, or how quickly to adjust a protocol parameter, such as the number of polling rounds needed to detect a thread break. It could even change the control function's preference for addressing a mismatch by changing a parameter or switching protocols.

5.9 Simulation

To validate the idea of adaptive TMAR, we developed a proof-of-concept simulation system with two alternative protocols, Thread Polling and Node Alive, and a threshold-based control strategy. The simulation enabled us to improve our understanding of how these alternative TMAR protocols are influenced by the operating environment. It also allowed us to demonstrate that use of an adaptive control strategy could improve system behavior in the area of distributed-thread integrity.

5.9.1 Simulation System

The simulation system was written using the CSIM[1] simulation package, which was developed at Microelectronics and Computer Technology Company (MCC) [43, 44]. CSIM supports the writing of process-oriented discrete-event simulations with the C programming language. The abstractions provided by the package include simulated time, processes, events, facilities, storage, mailboxes, histograms, tables, queues, reports, and random-number distributions.

In this approach, the activities being modeled are represented by processes written in C. The processes can cause events to occur, wait for events to occur, and wait for a period of simulated time to pass. Simulated time advances only when processes are blocked.

The computing system being simulated contains a collection of nodes connected by a network. Each node contains a single processor that must execute the application code and the active TMAR protocol.

Our simulation is organized around two major types of processes called GEN and NODE. During startup, configuration information is read from a file. A GEN process is started for each active workload. It periodically generates distributed threads according to the workload parameters, and randomly assigns them to nodes.

A NODE process is started for each node in the simulated network. Each NODE process selects a distributed thread from its ready queue and simulates a burst of execution for that thread. Threads are selected, first-come first-served. After a burst of execution, NODE decides whether the thread will execute a remote invocation or return. If an invocation occurs, the destination node is selected randomly.

[1] All product names mentioned in this document are the trademarks of their respective holders.

Each NODE process is also responsible for executing a TMAR protocol. Only one of the two TMAR protocols is active at a time, and all of the nodes execute the same protocol. When an adaptation occurs, all nodes switch to the new protocol at the same time.

CSIM processes are also used to cause events to happen in the future, after specific periods of simulated time have passed. One type of process is used to simulate communication delay between nodes, while another is used to post interrupts to nodes. Node execution is interruptible, and many kinds of interrupts can occur. More specifically, an interrupt can preempt a distributed thread executing application code, but an interrupt handler cannot be preempted.

Nodes may crash without warning. When a crash occurs, the TMAR protocols are responsible for repairing the system. They must detect the crash, identify the broken threads, identify the new thread heads, identify the orphan thread sections, remove the orphans, and resume execution of broken threads.

The system can adapt between the two TMAR protocols. Adaptation can occur in two ways. First, the system can automatically adapt when the adaptation metric passes a threshold. The adaptation metric used is the number of distributed-thread sections that exist in the network. Our hypothesis is that it will be better for the system to use the Node Alive protocol as the number of thread sections increases. Second, the system can also adapt by switching TMAR protocols when a specified simulated time occurs.

The overheads associated with the control function are not modeled in the current simulation. As a result, the automated control function does not have to be associated with the nodes, and automated control is implemented with a separate control process.

5.9.2 Assumptions

A variety of assumptions were made for the current version of the simulation. Some reflected the choice of a particular style of distributed system, while others reduced the complexity that had to be addressed in the first implementation. Another group abstracted away details that were not relevant at this time. Future versions of the simulation should address other environments and, when appropriate, more closely model certain system features.

Only the final group of assumptions about adaptation control affect the experiment that was performed. While more realistic modeling of adaptation control would be desirable, we believe that the basic outcome of the experiment would not change. Adding latency and overhead to control should just add some overhead to the nodes, delay adaptation switches, and reduce certainty about the current value of the metric. These effects should not have a big impact if adaptation is not too fine grained.

The current version of the simulation is based on the following assumptions:

- The nodes are assumed to be homogeneous with the same hardware features. In addition, all nodes run the same protocols and operate in the same way.
- The network is assumed to be homogeneous with all pairs of nodes connected in the same way. All pairs of nodes are assumed to experience the same communication delay.
- Processing nodes are assumed to be uniprocessors. Other choices would be to have dedicated-function multiprocessors like those in the current Alpha implementation,

or symmetric multiprocessors. Employing uniprocessors means that the application code, the TMAR code, and the control code must share the same processor.

- The only kind of failure that can occur is a node crash. Omission and performance failures are not simulated, either for nodes or the communication network, and communication link failures are not simulated. This has the effect of ruling out false failure detections of broken threads or failed nodes.[2]

- Node failures are permanent.

- Simultaneous node failures are not allowed. Another node cannot fail until the effect of the previous failure on threads and on the system has been completely repaired.

- The only thread-scheduling policy available is first-come first-served. Specifically, Best Effort thread scheduling is not available.

- In general, an interrupt is not allowed to preempt an interrupt handler. This is important because the processing for some interrupts can be lengthy. However, a few special events with short execution times are allowed to occur while an interrupt handler is executing, such as notifying a node that one of its rooted threads is performing a remote invocation or return in the Node Alive protocol.

- Adaptation control is assumed to operate without overhead. This includes the time needed to monitor the system, to determine that adaptation is needed, and to effect adaptation. It is also assumed that when the time arrives to make an adaptation, the change occurs instantly throughout the system.

5.9.3 The TMAR Protocols

We simulated simplified versions of the two TMAR protocols that were developed, the Thread-Polling protocol and the Node-Alive protocol. Most of the simplifications resulted from the assumptions mentioned above, such as the fact that the only kind of failure was a node crash and that only one could occur at a time. Also, in some cases we simulated the effect of an event and did not try to directly model the way it would be implemented in a real system. For example, in the Thread Polling protocol, a node actually looks at global data structures, not the information in response packets, to determine if a thread break has occurred. These abstractions were reasonable for the activities we planned to examine; further detail would be overkill.

5.9.3.1 Thread Polling Protocol

The bulk of the Thread Polling protocol is performed by the NODE process. Each node is periodically interrupted to check the health of the threads rooted on that node, and the

[2] A false failure detection occurs when a node erroneously assumes that a missing message was caused by a failed node, when it was actually caused by a different kind of failure. This cannot happen in a system where the only kind of failure is a node crash. This type of system is known as synchronous and has the property that a message will arrive within a known period of time unless the sending node has crashed. [35]

health of threads rooted on other nodes that have sections on that node. The time between health checks is configurable.

A poll packet is composed and sent to the other nodes. When a node receives a poll packet, it composes and returns a response packet. After response packets have been received from all nodes or a timeout has occurred, the polling node checks to see if any of its rooted threads have been broken because of a node failure. Because node crashes are the only kind of failure that can occur, a node always makes an accurate diagnosis. If one of its rooted threads had a section on a failed node, it notices that no response packet has accounted for the section and correctly deduces that the thread is broken.

When a node detects that one of its rooted threads is broken, it repairs the thread. If the thread is executing, the root node causes the thread to be preempted. It then identifies the new thread head and removes all of the thread's orphan sections. Next, it notifies nodes where orphan sections were removed so they can simulate the orphan removal work. After a delay, it allows the thread to resume execution at its new thread head.

During its health check, a node also determines whether any of the thread sections executing on the node is an orphan because its root node has died. A node is always able to detect that it has not received a poll packet from another node within a reasonable time, and correctly deduce that the other node has failed. The node immediately removes any thread section executing on it that was orphaned by the failure of the section's root node.

5.9.3.2 Node Alive Protocol

The bulk of the Node Alive protocol is also performed by the NODE process. Each node is periodically interrupted to check the health of its neighbor node. The healthy nodes are arranged in a ring, with node 1 watching node 2, node 2 watching node 3, and so on. The time between health checks is configurable.

A node is immediately able to determine if its neighbor has failed. When a node failure is detected, all healthy nodes execute a node group management protocol. This allows them to reach consensus about the group of healthy nodes. After the NGM protocol is completed, all nodes know about the failed node and can identify the broken threads. Each node then follows the same repair procedure that was described for the Thread Polling protocol to repair its broken threads and to remove the failed node's orphaned sections.

5.9.4 Adaptation Control Strategy

The simulation uses a threshold-based adaptation control strategy with a simple adaptation metric, as described in Section 5.8.3. Two thresholds are used with hysteresis to reduce the system's potential for thrashing. The metric that was selected is the total number of thread sections that exist. We believe this metric is both relatively easy to gather and representative enough to guide adaptation.

In the simulation, The number of existing thread sections is always globally known.[3] The control process periodically compares the current number of thread sections against the thresholds to see if an adaptation is warranted. Because this check is made periodically,

[3] In a real system, nodes would have to exchange information to determine the number of existing thread sections. Because of communication and processing delays, the calculated value would vary somewhat from the real value.

it is possible for the system to momentarily cross a significant threshold without this fact being detected. The checking period is configurable.

When the control process detects that a significant threshold has been crossed, it causes all nodes to instantly switch their TMAR protocols. An interesting interaction is that a node failure can cause an adaptation because it can suddenly cause many thread sections to be removed.

5.9.5 User Interface

A user can provide directives to the simulation program through a configuration file. Every time the simulation program is run, it reads the configuration file to obtain its controlling parameters. When the simulation run is complete, the program produces an output file with the results. The simulation program always produces the same output when started with a particular configuration file.

The user can assign values to several groups of configuration parameters to set up a simulation experiment. These parameters control the general system, two thread workloads, the TMAR protocols, adaptation, and node death. A sample configuration file is shown in Figure 5.5. The configuration parameters do not have to be specified in this order, but the last line of the configuration file must contain the word "quit".

5.9.5.1 General System Parameters

The general system parameters set up the simulation and the network architecture:

- Master random seed (seed) — the seed that is used to set the seeds for all of the independent random number sequences that control the behavior of the simulation program. By varying just the seed and keeping all of the other configuration parameters constant, the user can generate independent trials for a given simulation context.

- Number of nodes (num_nodes) — the initial number of nodes in the computer network. Some of the nodes can fail during the simulation, which results in a smaller number of active nodes. The nodes are identified with numbers from 1 to num_nodes.

- Communication Delay (comm_delay) — the number of basic time units required to send a message from one node to another. Currently, it is constant for any message between any pair of nodes.

5.9.5.2 Thread Workloads

A *thread workload* is a prototypical pattern of thread activity in the network. Some characteristics of a thread workload are the frequency with which threads are created, the average number of sections per thread, and the average lifetime of a thread.

The user can specify two thread workloads, which can be run sequentially or simultaneously. Each workload allows the user to specify information about thread creation and execution. These parameters indirectly determine more general characteristics for the group of threads. For example, specifying when threads are created, the average amount of time

```
          seed  7
      num_nodes  8
     comm_delay  5

    max_threads  50
    thread_iatm  500
     min_exec_t  5
     max_exec_t  50
     sec_bound   6
     ir_ratio_r  80
      ir_ratio   50
     ir_ratio_b  20

        start_2  10000
  max_threads_2  10
  thread_iatm_2  300
   min_exec_t_2  5
   max_exec_t_2  100
   sec_bound_2   8
   ir_ratio_r_2  80
    ir_ratio_2   50
   ir_ratio_b_2  20

      node_alg   0
     thread_alg  1
  na_health_wait 200
  tp_health_wait 200

   control_wait  250
       adapt_t   0
     auto_adapt  1
     sec_TH_hi   45
     sec_TH_lo   35
         death   2
             6   12000
             4   16000
          quit
```

Figure 5.5: Sample simulation program configuration file

needed for a thread section to execute, and the likelihood of threads performing remote invocations or returns indirectly determines the average number of existing thread sections in the system and the average number of threads per node.

The primary thread workload starts when the simulation begins. The general parameters for the primary workload are

- Maximum number of threads (max_threads) — the number of threads that will be generated by the primary workload.
- Mean Thread Interarrival Time (thread_iatm) — the mean of the thread interarrival time distribution. Thread arrival is assumed to follow an exponential distribution.

We simulate the execution of a thread section with a burst of local execution on a node. During a burst, local segments can be added and/or removed from the thread, or the thread may execute within a single object operation. A burst ends when the thread performs a remote invocation or a remote return to an object on another node. The amount of time for a particular section execution is randomly selected from a bounded range:

- Minimum Section Execution Time (min_exec_t) — the minimum time needed to perform a section execution.
- Maximum Section Execution Time (max_exec_t) — the maximum time needed to perform a section execution.

During its lifetime, a thread adds and removes sections as it executes remote invocations and returns. The following parameters control the way in which a thread randomly grows and shrinks, and indirectly determine a thread's lifetime and number of sections. Greater values for the ratio parameters lead to longer thread lifetimes. A larger value for the section bound leads to greater numbers of sections per thread and longer lifetimes.

- Section Bound (sec_bound) — used to limit thread growth. After a thread has grown to sec_bound sections, its probability of further growth is reduced.
- Invoke/Return Ratio for the Root (ir_ratio_r) — the probability that a thread will perform a remote invocation when execution of the thread's root section has just been simulated. As an example, a value of 80 indicates that 80% of the time the thread will perform a remote execution and 20% of the time it will terminate.
- Invoke/Return Ratio for the Middle (ir_ratio) — the probability that a thread will perform a remote invocation when execution of a section between the root and section sec_bound has just been simulated.
- Invoke/Return Ratio for the Bound Section (ir_ratio_b) — the probability with which a thread will perform a remote invocation when execution of a section at or beyond section sec_bound has just been simulated.

The secondary thread workload has the same parameters as the primary workload. A user can also specify when the secondary workload will begin:

- Starting Time (start_2) — the simulation time when the simulation program will begin to generate threads from the second workload. A value of 0 indicates that the secondary workload does not exist.

5.9.5.3 TMAR Protocols

The simulation can run with the Thread Polling protocol active, the Node Alive protocol active, or no TMAR protocol active. Longer time periods between health checks reduce overhead, but they also increase the time needed to detect and hence repair broken threads. The following parameters can be specified about the TMAR protocols:

- Node Alive Protocol (node_alg) — 1 if the Node Alive protocol is active, and 0 if it is not active.
- Thread Polling Protocol (thread_alg) — 1 if the Thread Polling protocol is active, and 0 if it is not active.
- Time Between Node Alive Health Checks (na_health_wait) — the time period between health checks for the Node Alive protocol.
- Time Between Thread Polling Health Checks (tp_health_wait) — the time period between health checks for the Thread Polling protocol.

5.9.5.4 Adaptation Control

Two kinds of adaptation are available. First, the user can specify a simulation time when the simulation program will switch between the TMAR protocols. Second, the user can indicate that the simulation program should automatically switch between the protocols. At most one of these adaptation modes can be active. Switching between protocols is assumed to occur instantly throughout the system.

Automatic adaptation is controlled by a threshold-based scheme with hysteresis and a single adaptation metric. The adaptation metric is the number of existing thread sections. When the number of existing thread sections is greater than the upper threshold, the Node Alive protocol is active. When the number of sections is less than the lower threshold, the Thread Polling protocol is active.

The user can specify the following adaptation parameters:

- Time Between Adaptation Checks (control_wait) — the time period between adaptation checks by the adaptation controller. When the time period is over, the controller wakes up and checks the current value of the metric to determine whether an adaptation should occur. A longer time period may delay a valid adaptation, and may cause the controller to miss a momentary metric value that would have triggered an adaptation.
- Time-Based Adaptation Time (adapt_t) — the simulation time when a time-based adaptation will occur. If adapt_t is 0, time-based adaptation is not active.
- Automatic Adaptation (auto_adapt) — 1 if automatic adaptation is active, and 0 if it is not active.
- Upper Threshold (sec_TH_hi) — the number of existing thread sections used for the upper adaptation threshold.

- Lower Threshold (sec_TH_lo) — the number of existing thread sections used for the lower adaptation threshold.

5.9.5.5 Node Death Specification

The user can specify when node deaths will occur and which nodes will die. Because nodes cannot be repaired and new nodes cannot be created, no more than $num_nodes - 2$ node deaths can be specified, and the nodes that die must be unique. The node death parameter is

- Node Death Control (death) — indicates the number of node deaths that will occur. Exactly that number of death indications must follow the death parameter. A death indication consists of a node identifier and a simulation time. For each death indication, the simulation program causes the specified node to die at the specified time. No node deaths will occur if death has a value of 0.

5.9.6 Adaptive Control Experiment

We performed an experiment to examine the benefits of adapting between the TMAR protocols. Our goal was to demonstrate that a simple adaptation controller would be able to distinguish between different operating regimes and dynamically select the appropriate TMAR protocol for the current operating regime. In this experiment, we focused on the overhead caused by the two TMAR protocols during normal system operation. Other experiments could study differences in thread break detection latency, thread repair latency, or a combination of these properties.

We used the automatic adaptation feature supported by the simulation program. The metric used for adaptation was the number of thread sections in the network at a given moment. The adaptation thresholds and hysteresis determined the two operating regimes. The Thread Polling protocol would be active if the number of thread sections was below the lower threshold, and the Node Alive protocol would be active if the number of thread sections was above the higher threshold. The active protocol would not change if the number of thread sections entered the region between the two thresholds.

5.9.6.1 Conjectures

We thought that these regions would distinguish between the two protocols for the following reasons. The overhead required for the Thread Polling protocol to perform a health check is related to characteristics such as the number of existing threads, the number of sections per thread, and the number of nodes spanned by a thread. We believed that the number of thread sections would serve as a sufficiently good representation of these characteristics to allow us to say that the Thread Polling overhead would rise as the number of thread sections increased. The time spent performing Thread Polling overhead reduces the amount of time available to execute threads and delays their completion.

On the other hand, the overhead for the Node Alive protocol is much less related to these factors. The major overhead for threads executing under the Node Alive protocol is caused by synchronous head movement, and is closely related to the communication delay. This is not an important problem for the Thread Polling protocol.

Thus, we conjectured that throughput would be higher if the Thread Polling protocol was active in the regime when there were relatively few thread sections, and if the Node Alive protocol was active in the regime when there were relatively many thread sections. Stated another way, we conjectured that a thread whose lifetime bridges periods with few and many thread sections will complete sooner if it operates under both protocols for a sufficient period of time than if it operates exclusively under either protocol. In addition, a thread whose execution is all or mostly under the protocol selected by the adaptation controller should complete sooner than if it executes under the opposite protocol. On the other hand, a thread whose execution is mostly under a single protocol during an adaptive run should, on average, finish at about the same time as it would in a nonadaptive run where that was the only protocol employed. The success of this approach depends on achieving a good match between the thresholds and the thread workload for the existing network configuration.

5.9.6.2 The Experiment

For our experiment, we used the configuration parameters shown in Figure 5.5, with a few exceptions. We ran the simulation program for five trials, varying only the seed. Each trial consisted of three runs, one with only the Thread Polling protocol active and no adaptation, one with only the Node Alive protocol active and no adaptation, and one with automatic adaptation active. The adaptive runs began with the Thread Polling protocol active because initially no thread sections exist. Note that the parameter death was set to 0 in the configuration files used for the experiment because the experiment did not generate any node deaths.

Threads were generated with both thread workloads. The primary workload generated a large number of relatively short threads over a relatively long period of time. During this background workload, the secondary workload generated a few relatively long threads over a relatively short period of time. Together with the other system parameters, these workloads were expected to produce two exploitable operating regimes. A thread was considered to have executed under a protocol for a significant amount of time if it spent 10% of its lifetime under that protocol.

The results are summarized in Figure 5.6. The extent to which both protocols were used during the adaptive runs is indicated by the second through fourth columns, which display the total simulation time for the run, the amount of time spent under the Node Alive protocol as a result of adaptation, and the percentage of time spent under the Node Alive protocol. As can be seen from these columns, the results varied widely, from a run where adaptation almost didn't occur, to a run where the Node Alive protocol was executed 27% of the time. The trials have been ordered by the amount of time spent under the Node Alive protocol.

The Dual Protocol columns compare the lifetimes of threads that executed under both protocols for a significant portion of time during the adaptive run to their lifetimes under the nonadaptive single-protocol runs. <=TN refers to dual-protocol threads that executed at least as fast as in either of the single-protocol runs, <=T refers to dual-protocol threads that executed at least as fast as in the Thread Polling run, <=N refers to dual-protocol threads that executed at least as fast as in the Node Alive run, and >TN refers to dual-protocol

Trial	Total Time	Node Alive Time	Percent Node Alive	Dual Protocol				Same Protocol		Opposite Protocol		Initial Opposite	
				<=TN	<=T	<=N	>TN	<=	>	<=	>	<=	>
a	40033	11000	27	17	3	1	2	10	4	12	2	15	8
b	34910	7250	21	8	5	3	1	5	7	8	4	21	10
c	33281	6250	14	7	6	1	4	12	12	12	12	13	5
d	33282	4000	12	2	6	3	2	11	18	24	5	13	5
e	31700	1000	3	3	2	0	3	16	13	18	11	16	7
avg	34641	5900	17	7.4	4.4	1.6	2.4	10.8	10.8	12.8	6.8	15.6	7
e′	31630	11000	35	8	1	5	2	11	13	15	9	15	5

Figure 5.6: Results of the adaptive control experiment

threads that were beaten by both of the single-protocol runs.

The Same Protocol, Opposite Protocol, and Initial Opposite columns compare threads that executed mostly under one protocol in the adaptive runs against their performance in the single-protocol runs. Because the adaptive runs always start with the Thread Polling protocol active, we divide these threads into an initial group whose entire lifetimes precede the first adaptation and a remainder group. The Initial Opposite columns compare the performance of the initial threads in the adaptive run and the Node Alive run. The Same Protocol and Opposite Protocol columns compare the performance of the remainder threads in the adaptation run and the appropriate single-protocol run. For example, assume that most of a thread's lifetime was spent under the Thread Polling protocol in the adaptation run. Then its performance in the adaptation run would be compared against its performance in the Thread Polling run for the Same Protocol columns, and against its performance in the Node Alive run for the Opposite Protocol columns. <= refers to threads that executed at least as fast, and > refers to threads that executed more slowly. Because the Same Protocol and Opposite Protocol columns refer to the same group of threads, the thread total across a row will be greater than the number of threads in the experiment, in this case sixty.[4]

5.9.6.3 Analysis

The top three trials appear to have been successful when comparing the adaptive runs to the single-protocol runs. A good percentage of the threads executed for a significant portion of time under both protocols, meaning that they had the opportunity to perform better than under either of the protocols alone. The net number of these dual-protocol threads that ended up improved over both of the protocols alone was encouraging. In addition, the initial threads showed improvement, and the threads that mostly executed under a single protocol tended to do better than under the opposite protocol.

The other two trials were neutral with respect to the dual-protocol threads. They did show good improvement, though, when the performance of single-protocol threads was compared with their performance in the opposite single-protocol runs.

On average, automatic adaptation was successful and helped improve performance. 26% of the sixty threads operated for a significant portion of time in both regimes, and of

[4]The sixty threads specified by the configuration file in Figure 5.5 consist of fifty threads in the primary workload (max_threads) and ten threads in the secondary workload (max_threads_2).

these threads, 47% showed clearly better performance. For all threads, 12% showed clearly better performance as a result of executing in both regimes. As expected, threads that operated mostly under a single protocol did not show any change when compared with their performance in the single-protocol run using the same protocol. However, they did improve when compared against the single-protocol run using the opposite protocol; 69% of the initial threads and 65% of the remainder threads finished at least as soon.

Trial e appeared to perform poorly because of a mismatch between the settings for the thresholds and the number of thread sections that were generated. To see whether automatic adaptation would work for trial e at all, we reran the trial with different threshold settings. A dramatic improvement was found for the threshold settings of 35 and 25, and we have recorded these results as trial e'. Now, in trial e', 27% of the threads operated in both regimes for a significant portion of their time, 50% of them showed clearly better performance, and 13% of all threads showed clearly better performance. In addition, single-protocol threads continued to show significant improvement when compared with the opposite protocol, and essentially broke even when compared with the same protocol.

5.9.6.4 Conclusions

This experiment illustrates the potential benefits of adaptive fault tolerance. Even with a simple adaptation control strategy and a crude metric, we were able to show a significant improvement in performance. The improvement for trial e when the new thresholds were used demonstrates the need to match the adaptation control parameters to the situation, and shows the potential benefit of dynamically adjusting the adaptation control strategy itself. In general, it will not be possible to make micro adjustments for every situation, and adaptation control will have to be tuned for an average situation.

5.9.7 Further Development of the Simulation System

The simulation system could be significantly improved in several directions. This includes improvements to the simulated TMAR protocols, the control strategy, and the simulation system itself.

The existing simulation system contains basic versions of the Thread Polling and Node Alive protocols. While they were adequate for this case study, their sophistication and realism should be improved. More detail should be added to the way the system models execution and communication overhead, especially in the area of thread repair. The fault model should be enhanced in several ways. First, simultaneous node failures should be permitted so that a node failure can occur during repair. Second, omission and performance failures should be added so that false failure detections can be modeled. It would also be beneficial to simulate additional TMAR protocols, such as a hybrid protocol that simultaneously uses Thread Polling and Node Alive for different portions of the network.

Modeling of the control strategy should be improved so that its effects on the simulated system can be observed. Execution and communication overhead should be added for the monitoring and analysis needed to perform control, and adaptation latency should be modeled for the switch between TMAR protocols. New adaptation metrics should be implemented, and it should be possible to base the adaptation decision on multiple metrics. History and filtering could also be added to control monitoring. Presently, adaptation is

based on the current value of the metric. History will allow adaptation to be based on the additional information provided by a sequence of values, and a filter will allow an appropriate portion of the history to be selected. For example, adaptation could be based on the average value of the metric over a window of time or over the metric's last n values. The controller might also be allowed to dynamically set its thresholds based on its experience. The effect on control of incomplete, inconsistent, and out-of-date monitoring information should also be considered.

The simulation system would benefit from an improved user interface, an important part of which would be better ways to generate thread workloads. More statistics should be gathered to help with analysis of the TMAR protocols and the adaptation control strategy. Additional improvements include better modeling of the communication subsystem, the ability to repair nodes, and the ability to add new nodes.

5.10 Conclusions and Recommendations

We believe that this effort has helped to validate the adaptive fault tolerance ideas developed during the AFRS project. Using these ideas, we designed alternative TMAR protocols with different operating characteristics. We then identified aspects of the operating environment that would cause the protocols to be better suited for different regions of the operating environment. We also identified aspects of the operating environment that would cause different parameter settings to be preferred for the protocols. Finally, we identified ways in which the adaptivity functions could be implemented for TMAR. Our simulation experiment allowed us to demonstrate these ideas, and to show the potential benefit of adaptive fault tolerance.

Further study of this important fault tolerance service could proceed in several directions. First, a more detailed design should be developed for the Node Alive protocol. Second, it would be desirable to develop a better understanding of how these protocols are influenced by factors in the operating environment. This could be accomplished through further development of the simulation system. Third, the effects and effectiveness of different control strategies should be studied, possibly through simulation. Finally, a great deal could be learned from a proof-of-concept prototype.

Chapter 6

Implementing Adaptive Fault-Tolerant Services for Hybrid Faults

6.1 Introduction

In building fault tolerance services in a distributed system, there are two major approaches, namely, the Primary-Backup approach (PB) (e.g., [1, 10]) and the State-Machine approach (SM) (e.g., [50, 42]). Each approach has its distinctive advantages. To tolerate simple faults such as crash and omission, PB protocols are generally significantly cheaper than SM protocols in terms of the numbers of processors, messages, and rounds (which directly affects the service response time). PB protocols are also much simpler than SM protocols, and thus the efforts of debugging or formal verification of PB protocols are also easier. On the other hand, in choosing to run a PB protocol instead of a SM protocol, one risks providing incorrect service functions or values, which may cause the overall system to fail, in the face of more serious *types* of faults such as arbitrary (Byzantine) faults[1]. Therefore, it is common practice for critical applications to run a SM protocol, possibly using Byzantine agreement [28]. The high cost of running such a protocol is compensated by the belief that all possible faults (up to a certain number) are adequately tolerated.

Instead of being forced to make a design choice between using SM or PB, thus either incurring a high running cost or risking system failure when unexpected faults occur, we advocate an approach of adaptive fault tolerance [14]. Given that in many situations Byzantine or other nontrivial faults occur only relatively infrequently, we develop intelligent adaptive algorithms, using PB and SM protocols as building blocks, that runs typically at a cost close to that of a PB protocol and switches to a more expensive SM protocol only as complicated faults (which cannot be tolerated by a PB protocol) occur. This adaptive approach thus has the potential to retain the best of both worlds. In addition, our adaptive approach is modular in that any PB or SM protocol can be used.

[1] Any given system configuration can tolerate only up to a certain *number* of faults. The emphasis here is on the distinction that a PB protocol cannot tolerate Byzantine faults.

For noncritical applications, our approach may be seen as a way of extending PB to cover more complex faults at low additional cost. For critical applications, our approach may be seen as a way of allowing some of the processing resources required for SM to be used for other services when full SM functionality is not needed. For example, when only manifest faults occur, an adaptive algorithm runs in the PB mode and can thus tolerate a maximum number of such benign faults. The adaptation can also be viewed as between an optimistic algorithm and a pessimistic one where the former is the default mode of operation and the latter is invoked only when necessary.

In the rest of this paper[2], we first outline a general strategy of adaptation for handling hybrid faults. We then present two adaptive fault tolerance algorithms, analyze their correctness and complexity, and compare them with nonadaptive approaches. We conclude with a summary and suggestions for future research.

6.2 An Adaptation Strategy

System functions can be concentrated in some central location or distributed around a network, and the software for these functions can consist of modules on separate processors or can be more closely integrated. Conceptually, however, a fault-tolerant service generally contains some or all of the following functions: processing of requests, fault forecast, fault detection, fault masking, fault diagnosis, fault removal, repair, and reintegration of repaired components.

To explain our general adaptation strategy, suppose that in the course of operations, faults of two types, A and B, may occur. Also suppose that type A faults occur more frequently and are less expensive to tolerate than type B faults. If both types of faults must be tolerated, the traditional approach has been to assume the worst and constantly run an (expensive) algorithm that can handle both types of faults.

We observe that detecting a fault is in general less expensive than tolerating it. Based on this premise, our strategy is to run, as a default, an algorithm that can tolerate type A faults and can also reliably detect the occurrences of type B faults. When type B faults occur, the default algorithm switches to a more expensive one that can tolerate both type A and type B faults. Some decision procedure then decides when to switch back to the default algorithm. For example, when the occurrences of type B faults are bursty according to a fault forecast, it may be wise to continue running the expensive algorithm for some period of time.

If the difference in the cost of tolerating the two types of faults is significant, such as in the case of simple crash faults versus Byzantine faults, then an adaptive strategy gains a great deal by reducing the average running cost. The strategy is at its best if (1) the cost of adding the extra fault detection mechanism to an algorithm that tolerates type A faults is negligible (so that the service efficiency is near optimal when type B faults do not occur) and (2) the default algorithm or the fault detection algorithm forms the initial segment of the more expensive algorithm (so that nothing is lost when type B faults do occur). The next section gives adaptive algorithms that exhibit such desirable behaviours. The strategy can

[2]This chapter has appeared as a separate technical report under the same title, by Li Gong and Jack Goldberg, SRI-CSL-94-03, March 22, 1994, revised September 30, 1994.

be extended to handle faults of multiple types, in this case a more elaborate fault diagnosis mechanism (especially the online variety) is needed to determine the exact types of fault in order to direct the adaptation.

6.3 Two Adaptive Fault Tolerance Algorithms

Two algorithms are described that adapt between the primary-backup approach and the state-machine approach. But first, we need to explain the assumptions we make about the execution environment of ourr adaptive algorithms.

6.3.1 System Model

The environment we assume is the following. Clients send their requests to the servers who process the requests and respond, all by exchanging messages. For simplicity, we assume that the communication channel between a client and any server is reliable and FIFO, and we aim to tolerate faulty servers but not faulty clients. We also assume that the servers are deterministic – because in the state-machine approach it is usually undesirable to allow nondeterministic behaviours in the (correct) servers. Moreover, we assume that the system is synchronous, and thus we can use a model of computation based on rounds. The reason for this limitation is that it is impossible to guarantee both safety and liveness in asynchronous systems [7, p.19].

Following the literature, we classify faults into three categories [30]:

- Manifest fault – one that produces detectably missing values (e.g., crash and omission faults) or that produces a value that all nonfaulty recipients can detect as bad (e.g., it fails checksum or format or typing tests).
- Symmetric fault – one that delivers the same wrong value to every nonfaulty receiver.[3]
- Asymmetric fault – an arbitrary fault with no constraints, also known as Byzantine fault.

We assume that the reader is familiar with both primary-backup and state-machine approaches, and omit non-essential details of the algorithms. Briefly, in PB, one and at most one server is designated as the primary at any time. A client sends a request to the primary, who processes it and then broadcasts the necessary state change to all backup servers. In a nonblocking PB protocol, the primary server responds to the client before receiving acknowledgements to its broadcast whereas in a blocking protocol, the primary blocks until all backups have acknowledged or after a timeout period. The schema for a server consists of three modules for: (1) deciding whether it is a primary or a backup, (2) processing requests, and (3) fault detection and recovery [7, p.56]. It is apparent that the PB approach can tolerate only manifest faults. For example, an incorrect primary can broadcast an incorrect state change and backup servers cannot detect this fact because they do not know the client's service request.

[3] It will become clear later that we can weaken this definition to that all nonfaulty recipients receive some wrong values, although they may not receive the same wrong value.

In a SM protocol, a client broadcasts its request to all servers, and then takes a vote on the responses it receives. Therefore, the client will decide on the correct response if a majority of the servers are nonfaulty. For correctness, all nonfaulty servers must process requests (possibly from multiple clients) in the same order. This requirement is called replica coordination [42] and is not necessary in a PB protocol. Satisfying this coordination requirement is quite expensive – for example, a Byzantine agreement protocol is a typical solution. With this heavy cost in resources and performance, the SM approach gains the ability to tolerate symmetric as well as asymmetric faults, in addition to manifest faults.

In our adaptive algorithms given below, the high-cost Byzantine agreement machinery is used only when needed, thus we call these algorithms Byzantine-On-Demand or BOD.

6.3.2 Manifest versus Symmetric Faults

Our first adaptive algorithm BOD-1 is to tolerate both manifest and symmetric faults, but not asymmetric faults. For the moment, we assume that links connecting servers are nonfaulty.

Given any PB protocol (blocking or nonblocking), we need only make a few simple changes to make it adaptive. We assume the servers have implemented a Byzantine agreement (BA) protocol – for agreeing on the next client request for processing, or replica coordination – that can handle symmetric faults. Strictly speaking, any protocol that can mask symmetric faults is sufficient for BOD-1, but for convenient discussion, we always refer to a BA protocol.

The basic idea is to let the backup servers participate in the service passively as in the PB protocol, except that they now also receive the original request from the client and watch the primary for any inconsistency (compared with themselves). If they detect an inconsistency, they report an error to the client (who will then wait for further action on the part of the servers) and initiate a Byzantine agreement protocol among the servers to mask the error.

Notice that since we are using a primary-backup approach as default, it is important, from the viewpoint of providing a correct service to the client, to detect non-manifest faults only in the primary, from whom the client takes a response. Nonmanifest faults in backups will lower the overall degree of fault tolerance (for additional faults) but can be safely ignored for the meantime because once such a faulty backup becomes a primary, its erratic behaviour will be immediately detected. It is this property that makes the adaptive algorithm so cost-effective.

Nevertheless, it may not be desirable to allow a significant proportion of backups to become faulty because the chance of detecting faults in the primary and the ability to replace it will be reduced. Faulty backup servers can be dealt with by an additional fault diagnosis and removal mechanism. For example, if only one backup disagrees with the primary, then this backup must be faulty (on the assumption that the majority is nonfaulty) and can immediately be removed and repaired.

The outline of the BOD-1 algorithm is in Table 6.1. We use r to denote a request, a(r) for the response, and s-c for information regarding state change.

A few points need to be clarified about the algorithm. For simplicity, we have omitted some details of the PB protocol, especially its failure detector and the handling of manifest

Round 0.	*Client*:	Broadcast request r to all servers.
	Primary:	Wait for request from client.
	Backup:	Wait for request from client.
Round 1.	*Client*:	Wait for response from primary.
	Primary:	Broadcast (r, a(r), s-c) to all backups. Respond a(r) to client.
	Backup:	Queue r. Wait for message from primary.
Round 2.	*Client*:	Wait for error report from backup.
	Primary:	Wait for error report from backup.
	Backup:	Verify the correctness of a(r). If error, broadcast ERROR to client and all servers, and start BA protocol (to agree on which client request to process). Wait for error report from other backups.
Round 3.	*Client*:	If receive ERROR from a majority of servers, wait for the BA protocol to complete; then vote on the responses. Otherwise, accept a(r).
	Primary:	If receive ERROR from a majority of servers, switch to the BA protocol, process request and respond to client. Return to **Round 1**.
	Backup:	If receive ERROR from a majority of servers, switch to or continue with the BA protocol, process request and respond to client. Otherwise, terminate BA protocol it started earlier, if any. Return to **Round 1**.

Table 6.1: Byzantine-On-Demand Algorithm BOD-1

faults. The mechanism for detecting (and masking) symmetric fault in BOD-1 is *in addition to* the failure detector of the PB protocol. In theory, either detection mechanism can take precedence over the other. For example, if manifest and symmetric faults occur concurrently, we can deal with the manifest faults first (e.g., arranging a new primary) and the symmetric faults later. Or we can mask the symmetric faults first and handle the manifest faults later. The latter scheme has the advantage that the client receives responses earlier than in the former scheme. Variations are possible. For example, if both fault detection mechanisms detect faults related to the primary, a symmetric fault can be assumed and the BA protocol initiated.

We have used a nonblocking PB protocol in BOD-1 such that the primary responds to the client without waiting for acknowledgement from the backups. In such a case, it is important that the primary's response to the client and its broadcast to the backups must be in the same round because otherwise, a fault may occur when the primary is responding to the client so that the client receives an incorrect response while all backups receive the correct response (and thus do not complain). Obviously, any blocking PB protocols will also work in this framework and our algorithm BOD-1 needs only minor modifications (which we do not go into here).

In addition, we have assumed that the client always expects to get the response a(r) from the primary. Some PB protocols are "pass-the-buck" in that the response always comes from a different server [7, p.100]. Our framework can also accommodate these protocols.

If the primary has failed manifestly, the identity of the new primary is decided according to the PB protocol and is conveyed to the client. When the identity of the primary is in dispute, a non-manifest fault has occurred and a BA protocol can be used to reach an agreement. It is not difficult to see how this additional BA protocol can be added, so we will not discuss this issue in more detail.

A backup server checks for two new errors in **Round 2**. One is that the primary processes a client's request not according to a FIFO order. The other is that the primary's response is wrong. A backup server can check the first error by looking at its local queue of requests, and can check the second by taking the request the primary broadcast, processing it, and comparing its own result with the one sent by the primary. A discrepancy signifies that a symmetric fault has occurred. Note that when a server broadcasts an error message, it can use the *same* message to carry out the next step of the BA protocol, instead of waiting till **Round 3**. We do not discuss such optimizations.

In BOD-1, a server does a majority voting on error messages before deciding to switch to the BA protocol. This is because of the assumption that the links between the servers are nonfaulty, thus a symmetric error will be detected and reported by all nonfaulty servers. Without this assumption, a server may need to switch to the BA protocol even when receiving just one error report.

A backup server that has detected an error in **Round 2** immediately starts the BA protocol. This is the earliest possible time to convert to running the BA protocol. Under the assumption that links connecting the servers are nonfaulty, if the primary is faulty, then all nonfaulty backups will have started running the BA protocol in **Round 2**. Thus it may appear to contribute little for the primary and the faulty backups to start catching up to running the BA protocol in **Round 3**. However, a server experiencing a transient fault in **Round 2** may have recovered by **Round 3** and thus could be quite useful in the vote. If

the primary is not faulty, then a backup server will receive error report from only a minority of servers in **Round 3**, and those who have started running the BA protocol earlier should terminate it. Without this assumption of nonfaulty links, a server need not vote on error report and must continue to complete the BA protocol.

In BOD-1, at the end of the protocol, servers return to start from the beginning, in the default PB mode. This can be changed easily. For example, if the occurrence of non-manifest faults has been frequent for a period of time, the algorithm can stay in the SM mode for a while before returning to the PB mode. Here, fault forecast and heuristic methods can be useful.

Finally, the adaptive algorithm imposes no ordering among the processing of requests from multiple clients. The primary is free to choose the next request to process, as long as the order among requests from the same client is FIFO. The backups simply follow the primary's lead. This arrangement satisfies the Replica Coordination requirement [42], and is crucial for keeping the cost down. This is in the same spirit of the coordinator-cohort scheme [4, 6]. Any additional ordering can be enforced with other methods, which are beyond the scope of this paper.

Proof of correctness. We give a proof outline by enumerating all cases. (1) If there is no fault, then the protocol terminates essentially as the PB protocol. (2) If the primary and backups exhibit only manifest fault, then the PB protocol's fault detection and recovery mechanism handles these faults. (3) If the primary exhibits a symmetric fault, then some nonfaulty backups (or all nonfaulty backups, if there are no link faults) will detect the fault (by the additional fault detection mechanism in BOD-1) and report to the client. A subsequent BA protocol will mask this fault and the client can obtain the correct response by simple majority voting [42]. (4) If the primary is nonfaulty, then at most a minority of backup servers will report error (note that no protocol can tolerate a majority of servers being non-manifestly faulty), and these error messages are false alarms and rightly ignored. □

Analysis of complexity. Compared with the PB approach, when there are no faults, BOD-1 requires m extra messages in the client's initial broadcast, m being the number of backup servers. BOD-1 also uses one more round than a nonblocking PB protocol, the same number of round as a "pass-the-buck" PB protocol, but one fewer round than a blocking PB protocol. The primary's broadcast message in Round 2 is slightly longer (it contains both the request and response), and the backups will all have to process the request individually (thus consume more CPU cycles). Therefore, BOD-1 is slightly more expensive than a typical PB protocol, and this is quite reasonably compensated by the fact that symmetric faults can now be detected and masked.

If there are only manifest faults, then a PB protocol requires more rounds to recover. Thus the overall response time for BOD-1 is no worse than the PB protocol it uses as default, and the only additional expense in BOD-1 is the client's initial broadcast.

When symmetric faults occur, all nonfaulty servers in BOD-1 convert to running a BA protocol (or whatever algorithm that can tolerate this type of fault) in **Round 2**, assuming no link faults, but with some overhead. Compared with the state-machine approach, BOD-1 uses one more round because in SM the BA protocol can start in **Round 1**. However, the client's broadcast and that of the primary in the first two rounds of BOD-1 are not wasted – they can be used as part of the early rounds of the BA protocol – and the only extra

messages are those reporting errors to the client. On the other hand, if the state-machine approach is used as default, then even when no fault or only manifest faults occur, the running cost is significantly higher than that of BOD-1. □

In Table 6.2 below, we compare the algorithm complexity of blocking PB (denoted as bPB, and including "pass-the-buck" protocols), State Machine (SM), and BOD-1. We give only the difference between the complexity of BOD-1 and that of other algorithms because the absolute complexity varies depending on the PB or SM protocol we use as building blocks. For brevity, we do not include nonblocking PB protocols because their running cost differs from "pass-the-buck" protocols only in that nonblocking protocols use one fewer round. Suppose there are a total of m backup servers.

	bPB		BOD-1		SM	
	msgs	rounds	msgs	rounds	msgs	rounds
fault-free	msg(bPB)	r(bPB)	msg(bPB)+m	r(bPB)	msg(SM)	r(SM)
manifest	msg(bPB)	r(bPB)	msg(bPB)+m	r(bPB)	msg(SM)	r(SM)
symmetric			msg(SM)+m	r(SM)+1	msg(SM)	r(SM)

Table 6.2: Complexity Comparison

We can clearly see that when no fault occurs, BOD-1 uses one more broadcast (from client to all the servers) and one more round than the cheapest nonblocking PB protocol. However, nonblocking protocols need additional mechanisms for error recovery, such as checkpointing, and cannot handle send-omission and general-omission faults. Also, in a fault-free run, some blocking protocols (such as "pass-the-buck" protocols) use just one more round than nonblocking protocols, while other blocking protocols use more rounds. When only manifest faults occur, BOD-1 uses one more broadcast than either nonblocking or blocking Primary-Backup, but no more rounds. When symmetric faults occur, BOD-1 is as slightly more expensive than the SM protocol.

Since SM is much more expensive than PB, but non-manifest faults occur relatively rarely, the adaptive algorithm BOD-1 is superior than the PB approach in that non-manifest faults can now be tolerated and also superior than the SM approach in that the average running cost is greatly reduced. It should be pointed out that in our discussion we use Byzantine agreement protocol to tolerate symmetric faults, so the comparison in cost in Table 6.2 may be a little unfair because a cheaper protocol may also tolerate such faults. However, it is intuitive that any method that can tolerate symmetric faults will likely be significantly more expensive than the PB approach, and thus adaptive algorithms similar to ours will likely be beneficial.

6.3.3 Manifest versus Asymmetric Faults

BOD-1 cannot handle asymmetric faults. For example, the primary can send correct responses to all backup servers but send a wrong one to the client. Therefore, the client cannot rely on the primary's response as before, and it is not enough for backup servers to watch the primary.

Our second algorithm BOD-2 is a simple modification of BOD-1 and can tolerate manifest and asymmetric faults. As before, we assume the availability of a PB protocol (blocking or nonblocking) and a Byzantine agreement (BA) protocol.

As can be seen in Table 6.3, there are only two major differences between BOD-1 and BOD-2. First, when a backup server is satisfied with the primary's response, instead of remaining silent, it sends the response back to the client as well. Second, the client votes on all the responses and reports an error (to initiate the BA protocol) if the primary's response is not the majority vote of all responses from the servers. Note also that it is no longer meaningful to vote on error reports since faults can be arbitrary.

Round 0. *Client*: Broadcast request r to all servers.
Primary: Wait for request from client.
Backup: Wait for request from client.

Round 1. *Client*: Wait for response from primary.
Primary: Broadcast (r, a(r), s-c) to all backups. Respond a(r) to client.
Backup: Queue r. Wait for message from primary.

Round 2. *Client*: Wait for a(r) or error report from backup.
Primary: Wait for error report from backup.
Backup: Verify the correctness of a(r). If correct, respond a(r) to client.
If error, broadcast ERROR to client and all servers,
and start BA protocol (to agree on which client request to process).
Wait for error report from other backups.

Round 3. *Client*: If receive an error report,
wait for the BA protocol to complete; then vote on the responses.
If no error is reported but primary's a(r) is not the majority vote
of the a(r)'s, broadcast to all servers to initiate BA protocol.
Otherwise, accept a(r).
Primary: If receive an error report, switch to the BA protocol, process request,
respond to client, and return to **Round 1**. Otherwise, go to **Round 4**.
Backup: If receive an error report, switch to or continue with the BA protocol,
process request, respond to client, and return to **Round 1**.
Otherwise, go to **Round 4**.

Round 4. *Primary*: Wait to see if client report error. If error, switch to BA protocol.
Otherwise, return to **Round 1**.
Backup: Wait to see if client report error. If error, switch to BA protocol.
Otherwise, return to **Round 1**.

Table 6.3: Byzantine-On-Demand Algorithm BOD-2

The correctness argument for BOD-2 is similar to that of BOD-1 – any non-manifest

fault is detected and a Byzantine agreement protocol is initiated to mask it.

The complexity of BOD-2 is higher than BOD-1. In a fault-free run, an extra m messages will be needed in Round 2 (for the "backup" servers to respond to the clients). When only manifest faults occur, it is still much cheaper than a full-fledged state-machine approach in that backup servers simply follow the lead by the primary in deciding the next request to process. This arrangement eliminates the need for extra effort to satisfy the Replica Coordination requirement [42]. When asymmetric faults occur, BOD-2 may use one more round than BOD-1, for example, when the backup servers have to wait till **Round 4** to decide whether to switch to the BA protocol. However, if the client does not report error in **Round 3**, it can complete the protocol after **Round 3**, earlier than the servers.

6.4 Related Work

Our work is undertaken within the general framework outlined in [14] and can be viewed as a realization of some of the principles of adaptive fault tolerance. We have made heavy use of materials on primary-backup protocols [10, 9, 7] and the state-machine approach [42]. In particular, our adaptive algorithms use those protocols as building blocks, in a modular fashion.

Previous work on handling hybrid faults appears to focus on extending protocols for Byzantine agreement so that they can tolerate a higher number of benign or hybrid faults (e.g., [30, 32, 46]) than a standard Byzantine agreement protocol. These algorithms typically can tolerate as many Byzantine faults as possible (bounded by one third of the number of processors [28]). However, when other non-manifest faults do not occur, the algorithm by Thambidurai and Park [46] cannot tolerate many manifest faults whereas our algorithms can tolerate a maximum number of manifest faults because they will be running a Primary-Backup protocol. The algorithm by Lincoln and Rushby [30] can tolerate a maximum number off manifest faults but, like the algorithm by Thambidurai and Park [46], it is non-adaptive in that the number of rounds of each execution of the protocol is decided in advance so the complexity of the protocol does not decrease when no or fewer faults occur. Moreover, the complexity of such algorithms is typically comparable to that of Byzantine agreement because they aim to tolerate arbitrary faults, including symmetric and asymmetric faults, all the time. In contrast, our adaptive algorithms are much less expensive because for most of the time they merely attempt to detect arbitrary faults, and activate the heavy machinery to tolerate arbitrary faults only *as they occur*.

Our adaptation strategy is in flavour similar to early-stopping protocols (e.g., [12, 8, 3]). The complexity (i.e., numbers of messages and rounds) of these protocols is proportional to the number of actual faults occurring instead of the maximum number of faults that can be tolerated. In other words, the protocols terminate earlier if fewer faults occur. This line of work has largely been focused on Byzantine-agreement-type problems, so the protocols are adaptive only to the extent of the number of actual faults, not distinguishing the types of faults. Therefore, they are usually much more expensive than our algorithms, which take advantage of the common observation that in most applications Byzantine faults occur only infrequently. Nevertheless, our algorithms can use early-stopping Byzantine agreement protocols or those above for hybrid faults to further increase efficiency.

Garay and Perry [13] recently proposed a continuum of failure models with crash-only faults and Byzantine faults at the extremes. This can be taken as a combination of early stopping and dealing with hybrid faults. Although their model is flexible in that a design does not have to choose one of the two extremes, their method again concentrates on solving agreement-type problems. Our algorithms, on the other hand, are aimed at building general fault-tolerant services. In particular, the algorithms adapt between two different approaches, namely primary-backup and state-machine, and utilize to the maximum the efficiency of a primary-backup protocol.

There have been efforts to evaluate the relative merits of various fault tolerance techniques for different applications (e.g., [49]), especially following the recent precise formulation and analysis of the widely used primary-backup approach [10, 9, 7]. Our work provides some insight in that one can adaptively use these different approaches and retain (almost) the best of both worlds.

Finally, since our adaptation is modular in that it uses existing primary-backup and Byzantine agreement protocols as building blocks, our algorithms are conceptually simple. An important benefit is that the correctness proof is much simpler than that for earlier protocols for handling hybrid faults. We need only show that the adaptation correctly detects and adapts to the occurrence of certain types of faults. The potential reduction in the effort of formal verification is then significant.

6.5 Summary and Future Work

We have shown how to apply the principles of adaptive fault tolerance to handle hybrid faults. We have presented algorithms that intelligently adapt between the Primary-Backup (PB) approach and the State-Machine (SM) approach. Such an adaptive algorithm runs a default PB protocol but also attempts to detect the occurrence of non-manifest faults. When these fault occur, a Byzantine agreement type protocol is activated to mask the faults. Given that in practice manifest faults (e.g., crash and omission faults) are the most common ones, our adaptive approach is more cost-effective than the traditional "one or the other" approach because it retains (almost) the best of both worlds. Our approach is modular in that any specific PB or SM protocol can be plugged in. This keeps our algorithms conceptually simple, and it also significantly reduces the complexity of the correctness argument and the effort of formal verification.

There are many directions for future research. One is to investigate optimizations of our algorithms. For example, given an assumption of the number of possible faults in a period of time (such as predicted by the fault forecast component), some messages in BOD-1 may not need to be broadcast to all backup servers. This is because non-manifest faults are symmetric, so only one nonfaulty server is needed to detect a fault, depending on the assumptions of link faults. Some simulation may also provide fresh insights.

Another is to examine other possible adaptations, such as between symmetric and asymmetric faults, and between various manifest faults. As we already mentioned, other mechanisms for fault diagnosis and fault removal can be integrated. They can run parallel to our algorithms. To reintegrate a repaired component, if only symmetric faults are possible, algorithms can be developed so that the repaired server obtains state information from a small number of existing servers. If asymmetric faults are possible, then those methods

mentioned in [42] can be used. Other aspects of adaptation, such as fault transparency to clients and the cost to repair damaged servers, may also be worth investigating.

Our adaptive algorithms not only offer some theoretical insight into the relationship between the primary-backup and state-machine approaches to implementing fault-tolerant services, they also appear to be very practical. For example, given the existing support for process groups and virtual synchrony in the ISIS/Horus system [5], it should not be difficult to add an adaptation facility so that non-manifest faults can be tolerated as needed.

Chapter 7

Conclusions

The challenge of new environments The research on adaptive systems has been motivated by the needs of new, highly dynamic computing environments. The new kind of architecture strongly departs from the assumptions of past computer system design, in that many of the decisions about the use of computing resources that are normally reserved for the design time will, in adaptive systems, be made during operation. This new mode of operation requires qualitatively new design methods — which were the target of our research.

The need for new system design methodology Our research results represent a theoretical and methodological approach to the design of adaptive fault-resistant computers. We have illustrated a formalisms for specifying adaptive fault resistant behavior and for building models for predicting performance and failure behavior. We think that such formalisms are essential to making adaptive design a standard design process that can meet demands for unambiguous specification and predictable behavior.

Managing service trade-offs and design alternatives We have examined the trade-off relations in system service–such as timeliness, precision and accuracy, that are induced by adaptability, and have suggested ways to use these relations in making adaptation decisions that serve the needs of users in particular operating situations. We have also observed and categorized the rich range of choices available for modifying system configuration–from algorithms to parameters.

The criticality of meta-control It is clear that adaptivity requires a kind of higher-level of control intelligence to make proper decisions about system configuration. We have found inspiration in the concepts of modern control theory, whose central issues–model-based control, environmental and system state estimation and prediction, are vitally concerned with accuracy and stability. We have found these concepts to be directly mappable to adaptive control of digital systems, where system state is the configuration of resources and the algorithms that employ them. A direct example of this mapping of concepts is the use of multiple diagnostic models to improve the speed and accuracy of system characterization. In implementing this model, we have found the scheme of reflective architecture to be attractive for its logical power and generality. We also note the usefulness of classical

(pre-modern) control ideas, such as filtered, hysteresis-based, reactive control, for simple adaptive components.

New requirements for distributed and layered adaptation We also explored propagation effects in layered and distributed systems, noting the possible danger of endless propagation of adaptation, and the need to provide economical system support for the various operating modes.

Results from case studies We have conducted several case studies to explore the practical potential and design problems of adaptive designs. The major case study was Adaptive Distributed Recovery Blocks (ADRB). This scheme provides several modes of fault tolerance, including serial recovery, parallel recovery, and reduced-accuracy recovery. We developed a general architecture for ADRB and built a demonstration of multiple-mode behavior in a highly simplified C^2 application. The demonstration runs on the Alpha environments at Rome and Concurrent Computers. The alpha computing model is particularly well suited to adaptive, distributed systems, because of its support for a wide range of resource-utilization algorithms, and for its uncompromised control of distributed computations.

A second case study focused on the Alpha programming model. We analyzed opportunities for adaptation in the vital problem of fault recovery — in this case, the isolation and termination of unrecoverable sections of broken distributed threads. We described a wide range of protocols that are suitable for various operating conditions. We also simulated two of these protocols, and demonstrated the benefits of adaptive control with a hysteresis-based control scheme.

Using the adaptive model, we also invented a new algorithm for tolerating hybrid faults — faults that range from simple crashes to complex patterns of inconsistency among redundant processors. Adaptation actually allowed a significant economy in processing overhead — compared to existing solutions, for systems that must tolerate occasional complex faults.

Future issues We have only started on the path of developing and demonstrating a general and practical design methodology for adaptive systems. The formal models of adaptation need to be strengthened to assure users that adaptive systems can be well-specified and behave predictably. More examples of practical design are needed to determine the most robust design approaches. Techniques that we have outlined — real-time diagnosis, reflective architecture, and distributed adaptation management, should be elaborated in practical design situations.

We are pleased at the interest of some academic researchers. We hope that this will encourage other researchers and practitioners to advance the art that we have tried to define.

Bibliography

[1] P.A. Alsberg and J.D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, October 1976.

[2] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 145–154, Toulouse, France, June 1993.

[3] P. Berman, J.A. Garay, and K.J. Perry. Optimal Early Stopping in Distributed Consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237, Haifa, Israel, November 1992. Springer-Verlag.

[4] K.P. Birman. Replication and Availability in the ISIS System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, volume 19(5) of *ACM Operating Systems Review*, pages 79–86, December 1985.

[5] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53/103, December 1993.

[6] K.P. Birman, T.A. Joseph, T. Raeuchle, and A. El Abadi. Implementing Fault-tolerant Distributed Objects. *IEEE Transactions on Software Engineering*, 6(11):502–508, June 1985.

[7] N. Budhiraja. *The Primary-Backup Approach: Lower and Upper Bounds.* Ph.d. dissertation, Cornell University, Ithaca, New York, June 1993. Available as Technical Report 93-1353.

[8] N. Budhiraja, A. Gopal, and S. Toueg. Early Stopping Distributed Bidding and Applications. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, volume 486 of *Lecture Notes in Computer Science*, pages 304–320, Haifa, Israel, September 1990. Springer-Verlag.

[9] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Optimal Primary-Backup Protocols. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378, Haifa, Israel, November 1992. Springer-Verlag.

[10] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–196, Sicily, Italy, September 1992.

[11] Raymond K. Clark, E. Douglas Jensen, and Franklin D. Reynolds. An Architectural Overview of the Alpha Real-Time Distributed Kernel. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, Seattle, WA, April 1992.

[12] D. Dolev, R. Reischuk, and H.R. Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[13] J.A. Garay and K.J. Perry. A Continuum of Failure Models for Distributed Computing. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 153–165, Haifa, Israel, November 1992. Springer-Verlag.

[14] J. Goldberg, I. Greenberg, and T.F. Lawrence. Adaptive Fault Tolerance. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132, Princeton, New Jersey, October 1993.

[15] M. Hecht, J. Agron, H. Hecht, and K.H. Kim. A Distributed Fault Tolerant Architecture for Nuclear Reactor and Other Critical Process Control Applications. In *Proceedings of the Twenty-First International Symposium on Fault-Tolerant Computing*, pages 462–469, Montreal, June 1991. IEEE Computer Society.

[16] M. Hecht, J. Agron, and S. Hochhauser. A Distributed Fault Tolerant Architecture for Nuclear Reactor Control and Safety Functions. In *Proceedings of the Real-Time Systems Symposium*, pages 214–221. IEEE Computer Society, December 1989.

[17] Maurice Herlihy, Nancy Lynch, Michael Merritt, and William Weihl. On the Correctness of Orphan Elimination Algorithms. In *Seventeenth International Symposium on Fault-Tolerant Computing*, pages 8–13, Pittsburgh, PA, July 1987.

[18] Maurice P. Herlihy and Martin S. McKendry. Timestamp-Based Orphan Elimination. *IEEE Transactions on Software Engineering*, 15(7):825–831, July 1989.

[19] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randall. A Program Structure for Error Detection and Recovery. In *Lecture Notes in Computer Science*, volume 16, pages 171–187. Springer-Verlag, New York, NY, 1974.

[20] Editor J.C. Laprie. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, Wien, New York, 1991.

[21] K. H. Kim and T. F. Lawrence. Adaptive Fault Tolerance in Complex Real-Time Distributed Computer System Applications. *Computer Communications*, 15(4):243–251, May 1992.

[22] K.H. Kim. Process Scheduling and Prevention of Communication Deadlocks in an Experimental Microcomputer Network. In *Proceedings of the Real-Time Systems Symposium*, pages 124–132. IEEE Computer Society, December 1982.

[23] K.H. Kim. Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults. In *Proceedings of the Fourth International Conference on Distributed Computing Systems*, pages 526–532. IEEE Computer society, May 1984.

[24] K.H. Kim. Structuring DRB Computing Stations in Highly Decentralized LAN Systems. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pages 305–314, Kawasaki, Japan, March 1993. IEEE Computer Society.

[25] K.H. Kim and B.J. Min. Approaches to Implementation of Multiple DRB Stations in Tightly Coupled Computer Networks and an Experimental Validation. In *Proceedings of the Fifteenth International Computer Software and Applications Conference (COMPSAC 91)*, pages 550–557, Tokyo, Japan, September 1991. IEEE Computer Society.

[26] K.H. Kim and H.O. Welch. Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computers*, pages 626–636, May 1989.

[27] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, pages 167–174, Santa Barbara, CA, August 1989. IFIP WG 10.4.

[28] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[29] J.R. Leigh. *Applied Digital Control, 2nd Edition.* Prentice-Hall, New York, NY, 1992.

[30] P. Lincoln and J. Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 402–411, Toulouse, France, June 1993.

[31] B. H. Liskov, R. Scheifler, E. Walker, and W. E. Weihl. Orphan Detection. In *Seventeenth International Symposium on Fault-Tolerant Computing*, pages 2–7, Pittsburgh, PA, July 1987.

[32] F.J. Meyer and D.K. Pradhan. Consensus with Dual Failure Modes. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):214–222, April 1991.

[33] K. Mori. Autonomous Decentralized Software Structure and its Application. In *Proceedings of the Fall Joint Computer Conference*, pages 1056–1063, Dallas, TX, November 1986.

[34] K. Mori and H. Ihara. Autonomous Decentralized Loop Network. In *Proceedings of the Spring COMPCON*, 1982.

[35] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, 2 edition, 1993.

[36] B. Nelson. Remote Procedure Call. Technical Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA, 1981.

[37] J. Duane Northcutt and Raymond K. Clark. The Alpha Operating System: Kernel Internals. Archons Project Technical Report TR #88051, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1988.

[38] J. Duane Northcutt and Raymond K. Clark. The Alpha Operating System: Programming Model. Archons Project Technical Report TR #88021, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1988.

[39] F. Panzieri and S. K. Shrivastava. Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, January 1988.

[40] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, pages 220–232, June 1975.

[41] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–351, August 1991.

[42] F.B. Schneider. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[43] Herb Schwetman. CSIM: A C-Based, Process-Oriented Simulation Language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, December 1986.

[44] Herb Schwetman. *CSIM Reference Manual (Revision 16)*. MCC Technical Report ACT-ST-252-87, Austin, Texas, May 1992.

[45] S. K. Shrivastava. On the Treatment of Orphans in Distributed Systems. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, FL, October 1983. y.

[46] P. Thambidurai and Y.K. Park. Interactive Consistency with Multiple Failure Modes. In *Proceedings of 7th IEEE Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, Ohio, October 1988.

[47] W.N. Toy. Fault-Tolerant Computing. In *Advances in Computers*, volume 27, pages 201–279. Academic Press, 1987.

[48] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Cmputer Science Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[49] A. Waterworth, P.D. Ezhilchelvan, and S.K. Shrivastava. Understanding the Cost of Replication in Distributed Systems. Technical report, Computing Laboratory, University of Newcastle upon Tyne, U.K., January 1993.

[50] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[51] D. Wilson. The STRATUS Computer System. In T. Anderson, editor, *Resilient Computing Systems*, volume I, chapter 12, pages 45–67. John Wiley and Sons, Inc., 1985.

Appendix A

Adaptive Fault Tolerance