

https://dspace.org/

An error recovery technique for real-time distributed computer systems

Bloch, Gerald

2016-08-12
http://hdl.handle.net/10539/20853

Downloaded from a DSpace repository. https://dspace.org/

An ERROR RECOVERY TECHNIQUE for REAL-TIME DISTRIBUTED COMPUTER SYSTEMS

Gerald Bloch

A Thesis Submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg in fulfilment of the requirements for the Degree of Doctor of Philosophy.

Johannesburg 1990.

DECLARATION

I hereby declare that this thesis is my own work and that no part or substance of it has previously been submitted for a degree at any other University.



G. Bloch

This 3rd *day of* June *1990*

ABSTRACT

This thesis studies fault tolerant strategies for real-time distributed computer control systems so as to propose an error recovery technique that renders individual processors on the network resistant to soft failures. The technique is effective for soft failures which have as certain maximum duration, and does not require the use of specialised hardware. Attention is focused on achieving resistance to soft failures in environments which have demanding time constraints such as those found in computer systems for process control, materials handling and automated manufacturing.

These systems are usually termed real-time systems, and the thesis examines the impact of this notion on the error recovery process. An analysis is made of the key issues that determine the temporal constraints on error recovery in real-time computer systems. From this a definition of Class I and a Class II system is derived. The Class II characteristic plus two other key factors, namely the statefullness of the controller and the existence of event-based inputs, are consolidated into a definition of the error recovery problem to be solved. The problem under consideration is Complex, time critical error recovery for transient errors and a study of the existing error recovery strategies shows that a suitable solution to this problem does not exist.

The technique proposed in this study is Real-time Error Recovery (RER). The process operates on a number of loosely coupled processors interconnected by a high speed network. The basic concept of RER is that a node on the network can store in real-time its history state in a stable storage node in such a way that during a recovery, this history state data can be timeously retrieved. The process consists of three phases. The first phase is the setup phase in which a processor selects another processor as its stable storage node. The second phase is the save phase in which the processor saves its state variables to the stable storage node in real-time during normal operation. The third phase occurs after the processor has failed and is then performing a recovery. The state variables are retrieved from the stable storage node. The processor can then read its inputs and, as its history state has been restored by RER recovery, it can resume normal controlling operations.

There are a number of important timing considerations in RER that are studied in

detail. An expression is derived to determine the update rate and validity time of the state variables. The problem of imperfect synchronisation between the processors on the network is examined to determine its effect on the validity of the restored state variables. The difficult question of fault latency is also examined to determine its effects on the RER process.

The RER process is examined under a number of different network architectures and the advantages and disadvantages of each are examined.

Finally, a comparison is made between the reliability models of a processor operating without RER and a processor operating in an RER environment. We find that a reliability improvement of at least 10^5 can theoretically be achieved.

PREFACE

Over the past 15 years the author has been directly involved in a variety of industrial applications of embedded real-time computer applications ranging from safety critical systems controlling train movement to embedded process controllers operating in harsh environments. In many of these applications there was a requirement to provide some form of fault tolerance particularly to overcome transient problems. This was typically implemented by a hot standby configuration. The effectiveness of these systems was usually limited by cost, processor loading constraints as well as the reliability of the switchover mechanism.

Since 1986 the author has been involved in the development of embedded avionics. In this environment fault tolerance is specifically designed into the hardware and budgeted for both in terms of cost and processor power. However it has become clear that the solutions to fault tolerance practices in the aerospace industry, although effective, are not easily transferred to the industrial distributed process control domain due to costs and the relatively tightly coupled and integrated nature of the avionics architectures.

Another common fallacy in some existing fault tolerance proposals is the homogeneity of the processors involved. In many real-time applications it is not possible for one processor to take on functions of another at runtime as proposed in some remote procedure based solutions.

These experiences have prompted this study which tries to develop a fault tolerance strategy suitable for the distributed process control environment where development is often incremental over a period of time. The objective is to utilise existing resources in the computer control system and to be transparent to the main control functions performed by these processors. Some of the issues raised in this study were presented in a paper by the author at the 8th IFAC workshop on Distributed Computer Control Systems at Vitznau, in 1988.

ACKNOWLEDGEMENTS

The author wishes to sincerely thank the many people who have contributed in various ways to the work reported in this thesis. In particular

- My supervisor, Professor I.M. MacLeod, professor of con l engineering in the Department of Electrical Engineering, University of the Witwatersrand for his suggestions, guidance and support.

The author also wishes to acknowledge Teklogic for the provision of facilities, particularly the library staff for the help in locating the required background material.

TABLE OF CONTENTS

DECLARATION i
ABSTRACT ii
PREFACE iv
ACKNOWLEDGEMENTS v
ABBREVIATIONS xi

1. INTRODUCTION 1
    1.1 SCOPE OF RESEARCH 1
    1.2 SUMMARY OF THESIS 5

2. TEMPORAL ISSUES IN REAL-TIME ERROR RECOVERY 7
    2.1 INTRODUCTION 7
    2.2 DEFINITION OF TERMS 8
    2.3 CLASSES OF SYSTEM 10
    2.4 REQUIREMENTS FOR RECOVERY OF CLASS II SYSTEMS 13
    2.5 THE EFFECT OF STATE VARIABLES ON RECOVERABILITY 14
    2.6 EFFECT OF I/O DYNAMICS ON RECOVERABILITY 16
    2.7 TIMING CRITICALITY AND COMPLEXITY OF ERROR RECOVERY 18
    2.8 CONCLUSION 20

3. STATE OF THE ART ERROR RECOVERY STRATEGIES 21
    3.1 INTRODUCTION 21
    3.2 FAULT TOLERANCE PRINCIPLES 22
        3.2.1 General 22
        3.2.2 Error Recovery 25
        3.2.3 Correctness Issues 26
        3.2.4 Fault Mechanisms, Types and Taxonomy 28
        3.2.5 Permanent and Transient Faults 31

3.3 FAULT TOLERANCE STRATEGIES 34
3.3.1 General 34
3.3.2 Masking Redundancy 35
3.3.3 Replacement and/or Reconfiguration 38
3.3.4 State Restoration 42
3.4 CONCLUSION 44

4. REAL-TIME ERROR RECOVERY (RER) 46
4.1 INTRODUCTION 46
4.2 PRINCIPLES OF RER 47
4.3 RER ARCHITECTURE 49
4.3.1 Generic RER Architecture 49
4.3.2 Tightly versus Loosely Coupled Architecture 51
4.3.3 Other Architectural Considerations 52
4.3.4 Physical and Logical RER Architecture 52
4.4 THE RER PROCESS 55
4.4.1 General 55
4.4.2 Local Node State Machine 56
4.4.3 Remote Node State Machine 58
4.4.4 Detailed Description of the RER Process 61
4.5 VALIDATION OF STATE VARIABLES IN THE RSVS 82
4.5.1 RER Sampling Rate 82
4.5.2 Expiry Time Calculation 86
4.6 TIMING CONSIDERATIONS FOR RECOVERY 88
4.6.1 General Timing Analysis 88
4.6.2 Local and Remote Node Synchronisation Issues 91
4.7 FAULT DURATION: IMPACT ON RER 92
4.8 CONCLUSION 95

5. REAL-TIME ERROR RECOVERY ARCHITECTURES 97
5.1 INTRODUCTION 97
5.2 RSVS TOPOLOGY 98
5.2.1 Single-point vs Broadcast 98
5.2.1.1 Single-point Remote Storage 99
5.2.1.2 Broadcast Remote Storage 102
5.3 DYNAMIC RSVS ALLOCATION 103
5.4 NESTED RECOVERY PATHS 107
5.5 CONCLUSION 110

6. EVALUATION of REAL-TIME ERROR RECOVERY 111
6.1 FAULT DETECTION LATENCY 111
6.1.1 General 111
6.1.2 Factors Determining the Latency Period 112
6.1.3 Impact of Fault Latency on RER 113
6.2 CPU AND COMMUNICATION MEDIUM LOADING 116
6.2.1 General 116
6.2.2 CPU Loading 116
6.2.3 Communication Medium Loading 122
6.3 RELIABILITY ANALYSIS 124
6.3.1 General 124
6.3.2 Reliability Models 125
6.3.3 Methods for Reducing TTR 130
6.3.4 Conclusion 131
6.4 A-CORRECT OPERATION 132
6.5 DEALING WITH PERMANENT FAULTS USING RER 134

7. CONCLUSION 136

A. RER Timing Model 139

REFERENCES 148

LIST OF FIGURES

Figure 2-1: The Components of a SYSTEM 8
Figure 2-2: CLASS I System: Relationship between Environment and Controller 11
Figure 2-3: CLASS II System: Relationship between Environment and Controller 11
Figure 2-4: Examples of Class I and Class II Data Base System 13
Figure 2-5: The Components of a CONTROLLER 15
Figure 3-1: Fault Management Hierarchy. 23
Figure 3-2: Fault Management and System Life-cycle. 24
Figure 3-3: Fault Failure State Model 30
Figure 3-4: NMR Architecture 36
Figure 3-5: Recovery Block Functional Flow Diagram 41
Figure 3-6: Hot Standby Functional Flow Diagram 41
Figure 4-1: Generic RER Architecture 50
Figure 4-2: Physical RER Architecture 54
Figure 4-3: Logical RER Architecture 54
Figure 4-4: Local Node State Machine 57
Figure 4-5: Remote Node State Machine 59
Figure 4-6: The Components of RER Nodes 60
Figure 4-7: Components of Validity Time 84
Figure 4-8: Tc_1: The RER Sampling Interval 85
Figure 4-9: Example of Synchronisation Timing 90
Figure 4-10: Transient Fault Filter Analogy 94
Figure 4-11: Transient Error Filter Spectrum 95
Figure 5-1: Basic RER Architecture 97
Figure 5-2: Mutual Single-point Topology 99
Figure 5-3: Sequential Single-point Topology 100
Figure 5-4: Random Single-point Topology 101
Figure 5-5: Star Single-point Topology 102
Figure 6-1: Components of Validity Time with Latency 114
Figure 6-2: Double Buffered RSVS to Cater for Latency 115
Figure 6-3: Relationship Between MTBS and MTTS 118
Figure 6-4: Example Local Node 119

Figure 6-5: Relationship Between MNBS and B (total bandwidth) 123
Figure 6-6: Timing Diagram for Simple Redundancy Architecture 127
Figure 6-7: Timing Diagram for Redundancy-with-Repair Architecture .. 127
Figure 6-8: RER used with Standby Redundancy Architecture 135

LIST OF TABLES

Table I: Error Recovery Modes 19

ABBREVIATIONS

ASR	Authoritative System Reference
ATM	Automatic Teller Machine
BIT	Built-in Test
DCCS	Distributed Computer Control System
EMC	Electromagnetic Coupling
EMI	Electromagnetic Interference
LAN	Local Area Network
LSVS	Local State Variable Store
MNBS	Mean Number of Bits per Save
MTBF	Mean Time Between Failures
MTBS	Mean Time Between Saves
MTTS	Mean Time to Save
NMR	N - Modular Redundancy
NVP	N Version Programming
PLC	Programmable Logic Controller
PRL	Preferred Remote List
QA	Quality Assurance
RER	Real-time Error Recovery
RF	Radio Frequency
RSVS	Remote State Variable Store
TMR	Triple Modular Redundancy
TSC	Totally Self Checking
TTR	Time to Repair
VLSI	Very Large Scale Integration
VLSIC	VLSI Circuit

b_i	The number of bits in the ith state variable including the expiry time
C_f	The state of the controller at time of controller failure
C_r	The state of the controller at time of controller recovery
dT_{j-k}	The difference in time between the local clocks in nodes j and k
dT_s	The time difference due to imperfect synchronisation

E	The difference in global clock time between actual expiry time and expiry time held in the node after recovery
E_f	The state of the environment at time of controller failure
E_r	The state of the environment at time of controller recovery
ϵ	A small time interval
f_i	Sampling rate of input variable i in order to meet fidelity requirements
f_{RER}	Creation rate of state variables taking RER process into account
g_i	Gradient, or first derivative of input variable i
λ	The failure rate of a component
nl^k	Duration of one clock tick at node k
O_i	Offset in bits to the ith state variable in the RSVS
s_i	The ith state variable
T_{BIT}	The time taken to perform the BIT cycle
tc	Sampling interval of input variables in order to meet fidelity requirements
Tc	Creation interval of state variables taking RER process into account
TE_i	Expiry time of state variable i
T_f	The time at which a failure is detected in the controller
T_{FD}	Fault Duration
T_g	The global clock time
T_{LCPU}	The CPU time used in the local node to perform a SAVE transaction
T^k_l	The local clock time in node k
T_{RCPU}	The CPU time used in the remote node to perform a SAVE transaction
T_{SAVE}	The time taken to implement the SAVE transaction
T_{RST}	The time taken to implement the RESTORE transaction
T_r	The time at which the controller has recovered from a failure
Tv_i	Validity time period for state variable i
V_i	The ith input variable
a-correct	Acceptably correct
l-correct	Loosely correct
s-correct	Strictly correct
h-state	History state
i-state	Initial state

CHAPTER ONE

1. INTRODUCTION

1.1 SCOPE OF RESEARCH

Of the many developments taking place in the field of computer systems, there are two threads of activity of particular significance to this study. The first is that the growing sophistication and criticality of systems controlled by computers has resulted in a very high reliability being demanded from these computer systems. For example, military aircraft are required to demonstrate a failure rate of 1×10^{-7} failures per hour for components such as the flight control computer in a fly-by-wire system, and civilian requirements are some two orders greater than this [SHE82]. Another area requiring high reliability is the nuclear power industry. British requirements for reactor safety systems mandate that no single fault shall cause a reactor trip and that there be a 10^{-7} average probability, over 5000 hours, of failure to meet a demand to trip [LEV86].

The second significant thread is the trend towards distribution of computing resources. This is certainly not a new phenomenon but, if anything, there is an accelerating trend in many computer control architectures away from a centralised resource to a distributed network of nodes interconnected by means of a high-speed communication medium. The reasons for this trend are well documented and include such factors as reliability, flexibility, incremental implementation, cost, maintainability and many others [MAC83] and [STA84].

An important driving factor behind the growth of distributed systems and the increasing reliance on computer systems (and hence demand for reliability), is the materials technology which allows the semiconductor industry to scale down feature sizes in VLSI digital circuits so as to achieve greater packaging densities. However, this reduction in the size of circuits has, in turn, made these circuits more susceptible to *soft* or transient errors, especially from ionizing radiation and electromagnetic interference. The extent of these problems is well described by

[SAV85] and [HESS88].

Based on all of this we see that there is potential for an increase in the number of distributed computer systems from which very high reliability is demanded, but which are more and more prone to soft or transient failures. In spite of the large amount of effort invested in fault tolerant designs, most designs are heavily biased towards replication of hardware and software resources. The reason for this being that until recently, soft errors were not identified as a major source of faults and most effort went into the designs that were tolerant to permanent failures. The drawback of replication strategies is that they are expensive in many ways such as mass, space (important parameters in aerospace applications), development time and of course, financially. If the main source of unreliability becomes transient failures (we define this concept more precisely in chapter 3), it seems reasonable to require that whatever solution is found, it too should make a transient demand on resources, both physical and computational. Thus, high levels of replication of hardware and software and the associated development costs such as in Triple Modular Redundancy (TMR) may be unnecessary for a major source of faults, transient faults. From this it is clear that it is highly desirable to be able to utilise existing resources in a computer control system to achieve tolerance to soft errors.

The purpose of this thesis is to study fault tolerant strategies so as to propose a technique which utilises an error recovery process operating in a distributed computer environment to make individual processors in a network resistant to soft failures. The thesis looks specifically at achieving resistance to soft failures in environments which have demanding hard time constraints such as those found in the computer systems for process control, materials handling and automated manufacturing. These systems are often called real-time systems. The thesis looks at this notion, especially as it relates to the issues of fault tolerance and error recovery.

In order to provide some insight into the direction we have taken in this thesis to achieve the goals stated above, some background issues are discussed here.

The requirement for very high reliability in computer systems is the underlying motivation for our study. These requirements have lead to a dual approach consisting of the two mutually supportive techniques of fault avoidance and fault tolerance [JOH89]. Fault avoidance as a strategy has driven the development of a broad range of techniques and tools such as structured design, quality assurance

procedures, validated compilers and other good software and system engineering practices. Fault avoidance technology reaches its pinnacle in the still elusive field of software proving [PAR85]. However, due to the constraints of cost and technology, this fault avoidance technique of designing out faults in the first place has proven insufficient on its own for computer systems to achieve the required reliability in many critical applications [PAR85]. There is data available to indicate that on average only 49 percent of the faults inherent in a program are discovered prior to entry of that software into service [SHE82]. It is therefore clear that fault tolerant techniques are also required to extend the system reliability beyond that achievable with fault avoidance alone.

Fault tolerance is a systems approach to increasing the reliability by adding redundant physical and/or informational components. The model for fault tolerance in computer systems is typically presented as a sequence of steps beginning with error detection and ending with error recovery. Various intermediate stages are also commonly proposed for such activities as fault diagnosis, damage assessment and system reconfiguration [KOP80], [RAN78a], [AND81]. Several practical techniques for error recovery have evolved [RAN78b]. The aim of these techniques is to restore the system to a valid and useful state after error recovery.

The error recovery design adopted in a fault tolerant computer system varies in accordance with the class of system. For example, a commercial data processing system may require recovery of large amounts of data but with relatively lenient time constraints. An embedded flight controller will have different constraints. However it is clear, as defined more fully in chapter 2, that timeliness is a critical factor especially in real-time systems. Much debate has taken place over what constitutes a real-time computer system. In chapter 2, we look closely at the issue of the time relatedness of systems and how this has a fundamental bearing on how error recovery can be accomplished.

On examination of the existing literature covering error recovery strategies for soft errors we find that there have been no adequate error recovery techniques proposed for systems that contain a history-state and that must perform the error recovery process within time constraints imposed by the environment (Class II systems). [SAV86] proposes a soft error filter technique which is applicable to VLSI circuits but which provides a useful analogy for the type of system in which we are interested. [HESS88] recognises in broad terms the need for state restoration for

Class II systems but does not develop the idea fully. Other dynamic error recovery techniques such as those proposed by [LOQ86] and [AGR88] focus on error recovery by replicating functionality in a remote processor. These approaches use significant additional hardware and software resources compared to the non-fault tolerant system and therefore contradict our requirement for minimal additional resources. This gap in the spectrum of error recovery techniques is where we focus our attention in this study.

There is a class of real-time computer control systems for which recovery is very efficient and simple. These are systems that contain no history of the processes in which they are embedded. An example is the controller of a modern motor vehicle engine governing fuel flow and other engine parameters. This controller determines what controls to assert (its outputs) from the current state of the engine (its inputs) and a fixed algorithm. If such a system were to suffer a transient failure, simply resetting the system to some initial state after the transient fault has disappeared would effect a complete recovery. Recovery for these systems is therefore equivalent to a restart. The problem lies in that the majority of applications that would require some sort of fault tolerance such as critical industrial real-time control systems and most aerospace applications do in fact contain history states. Failures due to typical area disturbances such as power brownouts and lightening strikes would normally cause the history state to be lost as it is usually part of a real-time data base residing in volatile RAM. In the case where the data resides in non-volatile RAM such as NOVRAM or battery backed-up RAM, the history state would still be susceptible to transient faults arising from ionising radiation. Keeping the history state on offline storage such as discs is not normally feasible due to access time restrictions and due to the fact that many embedded systems do not have discs.

The scope of this thesis is to explore the concept of implementing recovery by performing a restart using only information available from the environment. This is done within the framework of a distributed computer control system and the aim is to derive a solution to the complex problem caused by the fact that many real computer controllers do have history states that must be restored after a failure. The objective is to explore an architecture whereby a node in a distributed environment can appear to be "history-stateless" on recovery. This then means that the normal complex state restoration process required after a transient failure can be replaced by a simple restart.

The desirable features are that the solution should not necessarily be dependant on custom-designed hardware but should be part of the "system design". Furthermore, in order that the technique we propose is not simply a replication scheme as in some other proposals [AGR88] and [LOQ86], the resources demanded from the node for fault tolerance should not be a significant increase over those required for their normal controlling function. The objective of the thesis is to propose an architecture and related algorithms that will, under a definable set of conditions, result in reducing the recovery problem with all its generalities and complications to a single startup procedure where the controller acquires all the necessary startup state information from its environment. In the thesis a conceptual design of the strategy, algorithms and protocols is presented and a theoretical analysis is done of the performance of an example system that is representative of the type of system to which the technique is applicable.

1.2 SUMMARY OF THESIS

Because we study error recovery in real-time systems, the timing relationships that exist between the computer system and the "environment system" in which the computer operates and over which it exercises control, is an important parameter. The thesis commences in chapter 2 by analyzing this time relatedness of systems and how this affects the recovery process. A computer controller and the environment in which it operates are both systems the behaviour of which is closely related to each other. From this we aim to elucidate the parameters determining real-time issues in the error recovery process.

In the first part of chapter 3 we examine a number of fundamental issues referred to in the rest of the thesis. These include issues such as fault latency, the definitions of hard, soft and plastic faults and the general terminology and taxonomy of fault tolerance. A survey of existing fault tolerant strategies is then given in the second part of chapter 3 in order to demonstrate that existing procedures do not explicitly deal with the real-time nature of the recovery process as defined in chapter 2.

Chapter 4 is then the crux of the thesis and introduces a proposed new error recovery strategy that aims to accomplish the defined goals. The strategy is to define a technique to achieve error recovery in systems requiring complex error recovery in real-time, where complex error recovery is a concept developed in chapter 2. Chapter 4 is a step by step examination of the algorithms involved. These

include the operations demanded of the nodes, as well as the important timing and validation considerations.

In chapter 5 we look at the way the technique can be applied to different network architectures and how the topologies affect the recovery process. We examine interconnection strategies, the question of nesting of stable storage nodes to provide fault tolerance of the stable storage node itself, and broadcast versus point-to-point communications between the processors in the distributed control system.

Finally, in chapter 6 we look at the performance factors introduced by the proposed recovery technique. The impact of fault latency on the basic recovery model is examined. The effect on reliability, as well as resource loading, is also examined to determine whether a significant increase in reliability is achieved at a relatively low resource utilisation overhead. The analysis of CPU and communication medium loading is a theoretical analysis based on an example of the type of system for which the proposed recovery technique would be applicable.

CHAPTER TWO

2. TEMPORAL ISSUES IN REAL-TIME ERROR RECOVERY

2.1 INTRODUCTION

In this chapter we perform an analysis to define the fundamental factors underlying the requirements of error recovery in real-time systems. We do this in order to clarify the nature of the problem of error recovery in real-time computer control systems. We also wish to define more precisely the type of problem that we will address in the main body of this thesis. In order to do this, it is necessary to introduce the new concepts of Class I and II systems and to show how they play a part in the recovery process. We also introduce the concept of "statefullness" in the computer systems under consideration and examine the role of the dynamics of the inputs to and outputs from the processor.

Error recovery is a state restoration process in which alignment is achieved between the state of the computer and the state of the process which it controls. The computer is a system which progresses through a number of internal states and similarly the plant is also a system progressing through a number of internal states. A symbiotic relationship exists between these two systems to form a greater system. The internal state of the computer controller is a function both of its internal model of the plant (i.e., its internal algorithms) and the current state of the plant as perceived though measured outputs of the plant. The internal state of the plant is a function of both its internal processes and the control outputs of the computer controller. Thus there are two systems moving through different internal states that are dependent on the internal states of each other.

Either of these two systems can fail and we are interested in analyzing the effects of failure and the recovery process. In this study we are not concerned with the effect of plant failure on the computer and we assume that this does not cause computer failure. In other words the plant's failure modes are anticipated and specified and the controller behaves in accordance with this specification.

The effect of controller failure on the system is important because it highlights two different classes of process. One process type in effect "halts" (in the computer sense) when the controller fails. The other keeps running, although on a different path. This distinction is important because it underpins the difference in strategy for recovery of a controller for the two different types of system.

2.2 DEFINITION OF TERMS

The domain of systems that are of interest to us in this study consists typically of one or more computer system controlling some process such as a part of an industrial plant, a vehicle, a peripheral device or even a logical entity such as a data base. Figure 2-1 shows the major components of such a system. Both the computer, which is sometimes termed the controller and the process under its control, which we term the environment, are systems that undergo state transitions that are some complex function of one another. This symbiotic combination of the controller, environment and inputs and outputs we term the "system" as shown in Figure 2-1. The synergy that exists between the controller and its environment is considered by the author to be one of the defining conditions for real-time systems and in this thesis is the basis for real-time error recovery. This view is supported by [SHI87] who also considers the relationship between controller and environment to be one

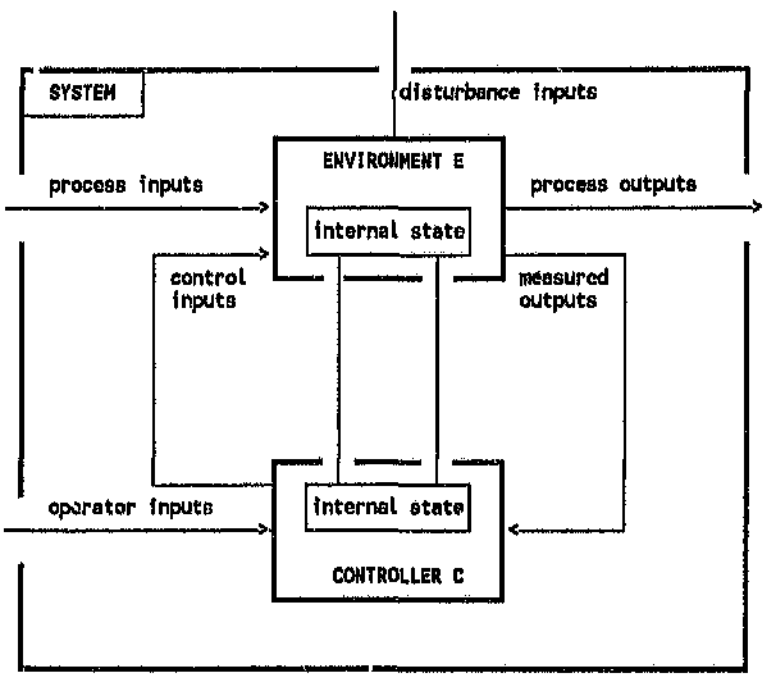


Figure 2-1: The Components of a SYSTEM

of the three legs used to define real-time systems. In this thesis the interaction between the controller and its environment is defined for a more general set of computer systems than is covered by [SHI87]. In order to minimise confusion arising from the overloading of certain terms, the following definitions will be used in this thesis:

CONTROLLER
The computer subsystem performing some control function, which may include control of a industrial plant or the control of a computer peripheral.

ENVIRONMENT
That subsystem that falls under the control of the controller.

PROCESS INPUTS / OUTPUTS
These are the inputs and outputs to and from the plant that falls within the scope of the controller. These are physical quantities such as materials or applied forces flowing into or out of the plant.

CONTROL INPUTS
These are inputs to the plant originating from the controller. The inputs will in fact be drives to effectors or actuators that physically influence the plant.

DISTURBANCE INPUTS
These are uncontrolled inputs to the plant that affect the operation of the plant.

OPERATOR INPUTS
These are inputs, usually into the controller, made by the plant operator to establish setpoints or targets.

MEASURED OUTPUTS
These arise from physical parameters that can be measured directly and become inputs to the controller.

INTERNAL STATE
The set of attribute values of parameters that define the state of the system.

SYSTEM
The controller and its environment.

E_f	The state of the environment at time of controller failure.
E_r	The state of the environment at time of controller recovery.
C_f	The state of the controller at time of controller failure.
C_r	The state of the controller at time of controller recovery.
T_f	Time f = time of controller failure.
T_r	Time r = time of controller recovery.

2.3 CLASSES OF SYSTEM

We are particularly interested in the time dynamics of the relationship between the controller and the environment. There is some function relating the states of the controller to the states of the environment. In this analysis, we are not particularly interested in the exact nature of this function. Our interest lies in the way the controller and environment states relate to each other just before, during and just after a failure of the controller. Based on the analysis of this relationship we have identified two types of system, which we propose to call Class I and Class II systems as shown in figures 2-2 and 2-3 respectively.

Class I System: The environment cannot change state without inputs from the controller. We call such a system a "control driven" system as the control inputs are a necessary condition for internal state transitions in the environment. It is assumed for the purpose of this analysis that disturbance inputs to the environment do not cause state changes in the environment during the time the controller has failed. Referring to figure 2-2, at times T_f and T_r the state of the environment has not changed.

Class II System: The environment is capable of changing its state without inputs from the controller. We may call such a system a "process driven" system as the process inputs continue to cause the environment to change state even in the absence of control inputs. The changes of state of the environment that occur without inputs from the controller may be undesirable. Referring to figure 2-3, at times T_f and T_r the states of the environment are different.

This classification of systems into Class I and Class II has not been discussed before. [SHI87] refers to such relationships in broad terms while defining real-time systems, but does not consider the implication to error recovery processes.

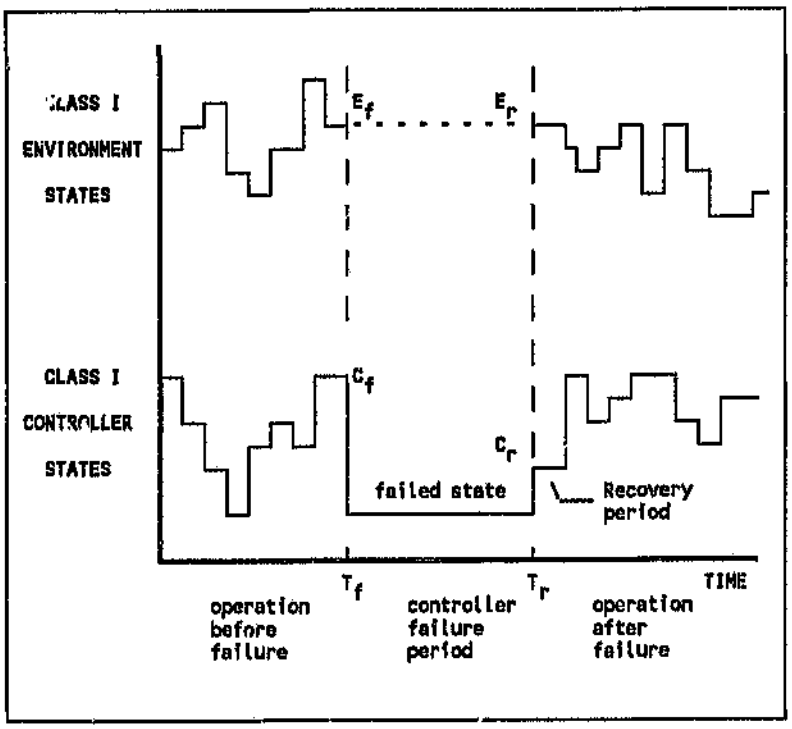


Figure 2-2: CLASS I System: Relationship between Environment and Controller

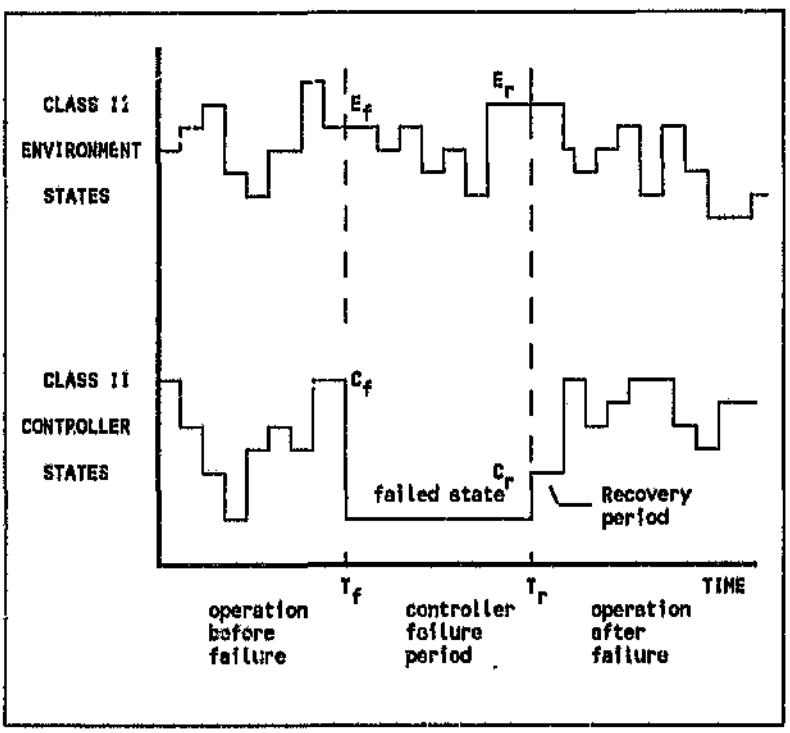


Figure 2-3: CLASS II System: Relationship between Environment and Controller

The implications of these classes of system on the error recovery process can best be described by way of examples:

Consider an on-line booking system for air travel. The controller here is the computer system that gives the human operator access to a data base, while the environment in this case is the data base itself which is controlled and managed by the computer system. The operator provides inputs to the controller.

In the event of controller failure, no further updates or changes can be made to the data base (assuming that it is a single user system) as it is entirely dependent on inputs from the controller in order to change from one state to another. During the period of controller failure, no changes of state can occur in the environment and therefore nothing is lost. It follows that when the controller returns to service after recovery, operations can easily be resumed from where it left off prior to the failure.

For this sort of behaviour to be exhibited two assumptions are made. The first is that the transactions with the data base are atomic and so partially completed transactions will not have occurred. The second assumption concerns the error detection capability of the system, for it must be assumed that errors are detected immediately causing the controller to enter a failed state. In these examples, failure is deemed to mean cessation of the controlling or monitoring function and excludes the failure mode where operation continues but in an erroneous manner. These assumptions are necessary so that the process can resume correctly once the controller has recovered. If the environment has been corrupted by incorrect operation of the controller prior to failure, then a rollback is required to undo the incorrect actions.

In contrast to the above example, consider the failure of an aircraft flight control computer during flight. The aircraft could, and most probably would, go out of control and crash if backup procedures (perhaps manual) were not quickly introduced. During the period of failure the environment would continue to evolve in terms of aircraft attitudes and position and on recovery of the controller, even after a relatively short period, it would not be useful to attempt to pick up where it had left off as the environment has evolved to a new state.

At first glance it may appear that the first example is only applicable to Data Processing type environments, but this is not so. Consider the case of a robot arm

controlled by an embedded microprocessor. If the controller were to fail, the arm would cease to move. On recovery, the controller would be able (in principle) to continue controlling the arm from where it had left off as the arm has no life of its own without the controller.

A final example to illustrate the difference between class I and II systems is shown in figures 2-4a and 2-4b. Figure 2-4a is a single user data base system and figure 2-4b is a multi-user data base system. Figure 2-4a is a Class I system because when the controller fails the state of the data base cannot alter. Figure 2-4b shows a Class II system because if controller 1 fails the state of the data base can be altered by transactions performed from controller 2.

The relevance of distinguishing these classes of system to the fault tolerance process lies in the viability of techniques that can be applied for error recovery.

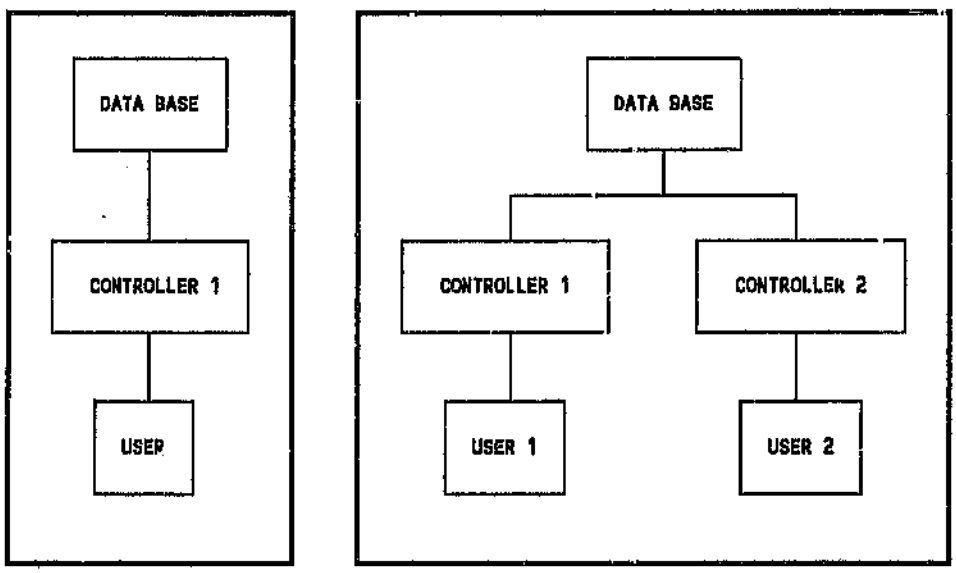


Figure 2-4: Examples of Class I and Class II Data Base System

2.4 REQUIREMENTS FOR RECOVERY OF CLASS II SYSTEMS

It can be seen that class II systems are typically systems with hard real-time constraints such as Distributed Computer Control Systems or embedded controllers. It therefore seems appropriate to call a recovery technique tailored for such systems "Real-time Error Recovery". The identifying feature of Real-time Error Recovery (RER) is that it is not compromised by the behaviour of Class II systems. Evolution

of the environment during failure is not of consequence to the effectiveness of the recovery process. The principle of real-time error recovery can be stated as follows:

> **In Real-time Error Recovery, the controller uses the information available to it from its environment at the time of recovery to determine a state that is valid to resume control of that environment.**

Clearly this places design constraints on the system to provide the measured parameters necessary to deduce a valid recovery state for the controller.

A typical example of a system that by default uses real-time error recovery is a controller for the electronic controls for the fuel and timing in the engine of a modern motor vehicle. Such a controller always derives its state from th state of the engine via measured parameters such as throttle setting, engine speed, etc. Recovery is treated in much the same way as an initial startup with the controller reading input parameters and using these to determine the output parameters such as ignition timing based on algorithms that are not dependent on the short term history of the input parameters. Another example of where real-time error recovery is used is PLCs where again the outputs are a function of the current state of the inputs only and not dependent on their history.

2.5 THE EFFECT OF STATE VARIABLES ON RECOVERABILITY

In a system containing the interaction of a plant and a controller as defined in figure 2-1, it can be seen that the ability of the internal state of the controller to model the internal state of the environment is dependent on the nature of the measured environment outputs. The measured outputs provide the controller with a window into the internal state of the environment. If these outputs can all be measured as state inputs then the controller becomes a stateless component (if there are no event inputs as well) and real-time error recovery becomes simple to implement. If, however, all the plant's state variables are not directly measurable, they can be created by the observation over time of the measurable output variables. This results in the existence in the controller of a set of internal state variables that represent the historical accumulation of the plant's state as follows:

$$S = (S_1, S_2, \ldots, S_n)$$

The state variables are a set of variables that define the absolute attribute values of parameters from the internal state of the plant that are not directly measurable. This set S is built up over time in the controller.

This introduces the concept of "Statefullness" in controllers; i.e., containing the set S or not containing the set S. The class of controller that lends itself to real-time error recovery is the "stateless component". This is a component whose output is not a function of its previous inputs or outputs. This can be compared to the "functional component" defined in [KOP82], i.e., a component whose h-state (history state) is always empty. The h-state comprises all relevant information which has been accumulated in the history (i.e., the sequence of computations since the initialisation) of the component. The h-state is dynamic and changes during operation of the controller. This is shown figure 2-5. So, a stateless component performs a fixed operation on an input and is unaffected by previous operation cycles; e.g.: the component implementing the mathematical function y = sin(x).

Stateless components are not the only class of components and [KOP82] defines a "state component" in which the h-state of the component is non-empty. Reconstruction of a valid internal state for a controller that is a stateless component is considerably easier than for controllers that contain state components.

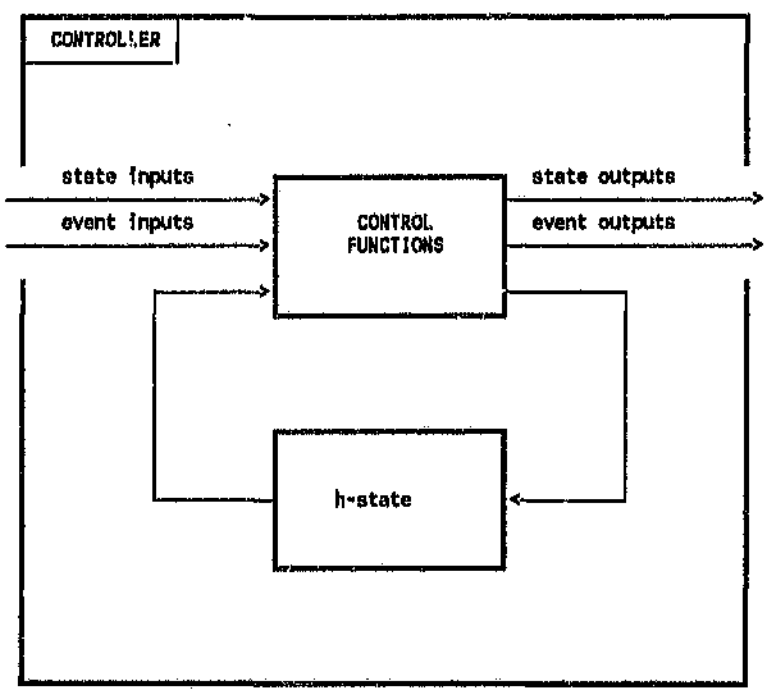


Figure 2-5: The Components of a CONTROLLER

2.6 EFFECT OF I/O DYNAMICS ON RECOVERABILITY

In the previous section, the concept of "statefullness" of the controller was introduced. In such a controller the h-state arises due to the absence of measured outputs of the plant for all the plant variables. It is not only the absence of certain measured outputs that creates a need for state variables in the controller but also the form of the measured output variable. In this section we examine the inputs from and outputs to the environment for their effect on the real-time recoverability of the controller. The inputs and outputs are divided into two types: state and event, and are defined as follows:

Outputs:

State outputs: Asserting a state output puts the environment into a known state. This implies a fixed relationship between the output variable from the controller and some plant internal attribute value after a fixed time interval. This is in fact open loop control. For example, a pipe with a fixed head of pressure, where the control valve position (output variable) gives a known flow rate (environment state). Another example is the end effector of a typical robot arm. The end effector position is given by commanding joint angles. These joint angles commands are state variable outputs of the controller (not to be confused with the closed loop control occurring at each joint).

Event outputs: Asserting an event output causes the environment to evolve or increment to its next state. To assert a valid event output requires a knowledge of the current state of the environment. This corresponds to closed loop control because the output only sets the position of an actuator, thus causing the environment to move in a direction, but does not set the environment into a known state. By monitoring the plant movement over time and stopping the control output at the appropriate time as determined by a control algorithm, the desired environment state is achieved. This is closed loop control.

Inputs:

State inputs: Absolute attribute values of objects which are directly usable by a controller.

Event inputs: Periodic events: events that occur with a certain bounded periodicity. The period may not be constant but could change during the operation, for example the pulses from a counter in a flow meter. The pulse rate has a determinable upper and lower bound given by the maximum and minimum flow rates.

Aperiodic events: events that do not have a bounded periodicity within the mission time of the controller. Such an event may not occur during a mission or may occur in a high rate burst. For example operator input events that can occur at any time.

The recoverability of a controller is related to the above input and output types as follows:

a) **State Input and Outputs:**

For a stateless controller the recovery will be a matter of reading the inputs and generating the state outputs. This implies a class I system, as the state outputs will configure the environment to a known state.

In a controller that is a state component, the h-state must be re-established before control outputs can be generated.

b) **Event Outputs:**

As there are event outputs this implies the controller has a model of the plant which must be restored before the control can continue. This implies a state component controller.

c) **Event Inputs:**

Event inputs require that the controller have an internal model of the plant

i.e., it is a state component. This must be restored before control can continue. If the event inputs are periodic, there is a known limit on the restoration time which is the shortest period of the input. If the input is aperiodic, it is not possible to define a recovery time. This means that the event input could always be lost in a recovery. Therefore systems must be designed so that aperiodic event inputs such as operator inputs can detect a failure/recovery so that the input can be reasserted.

2.7 TIMING CRITICALITY AND COMPLEXITY OF ERROR RECOVERY

From the analyses in this chapter it can be seen that there are three important interlocking concepts that determine the type of error recovery mechanism in a particular system. These are:

1) Class I or Class II System
2) State or Stateless Controller
3) State or Event I/O

The way these three factors combine to influence error recovery requirements is through two parameters. These are the Time Criticality requirements and the Complexity requirements of the recovery. Table I shows the relationship that exists for these factors. Firstly the concept of Error Recovery Complexity must be defined.

Simple Error Recovery is the case in which a controller can resume control of its environment by reading the measured outputs of the plant, executing a fixed algorithm and immediately generating control outputs. The controller requires no special setup procedures before this control can resume. In other words, the internal state of the plant as represented in the controller can be derived directly from the measured outputs. This is the stateless controller without event inputs.

The Complex Recovery process occurs when the controller is not able to generate control outputs immediately upon reading measured plant outputs and executing a fixed algorithm. This is due to the fact that the plant's state is unknown. This occurs when the system is a Class II system and the controller is a state component (it has an h-state which was lost in the failure) and/or there are event inputs among the measured plant outputs.

Secondly, the Time Criticality of the recovery process refers to the duration of the recovery process in relation to the important time constants of the controller which are in turn governed by the time constants of the environment (plant). In the "Time Critical" system, recovery must be completed within a defined period that depends on such factors as the rate of change of measured variables. In a "Non Time Critical" system, the recoverability does not deteriorate with the amount of time taken to perform the recovery process.

From this it can be seen that the difficulty in performing error recovery in a system increases from sector I to sector IV. Sector IV is the real challenge, requiring state variable restoration within strict time limits. The basis of this thesis is a recovery strategy to deal with this. The strategy, termed Real-time Error recovery, is described in chapter 4.

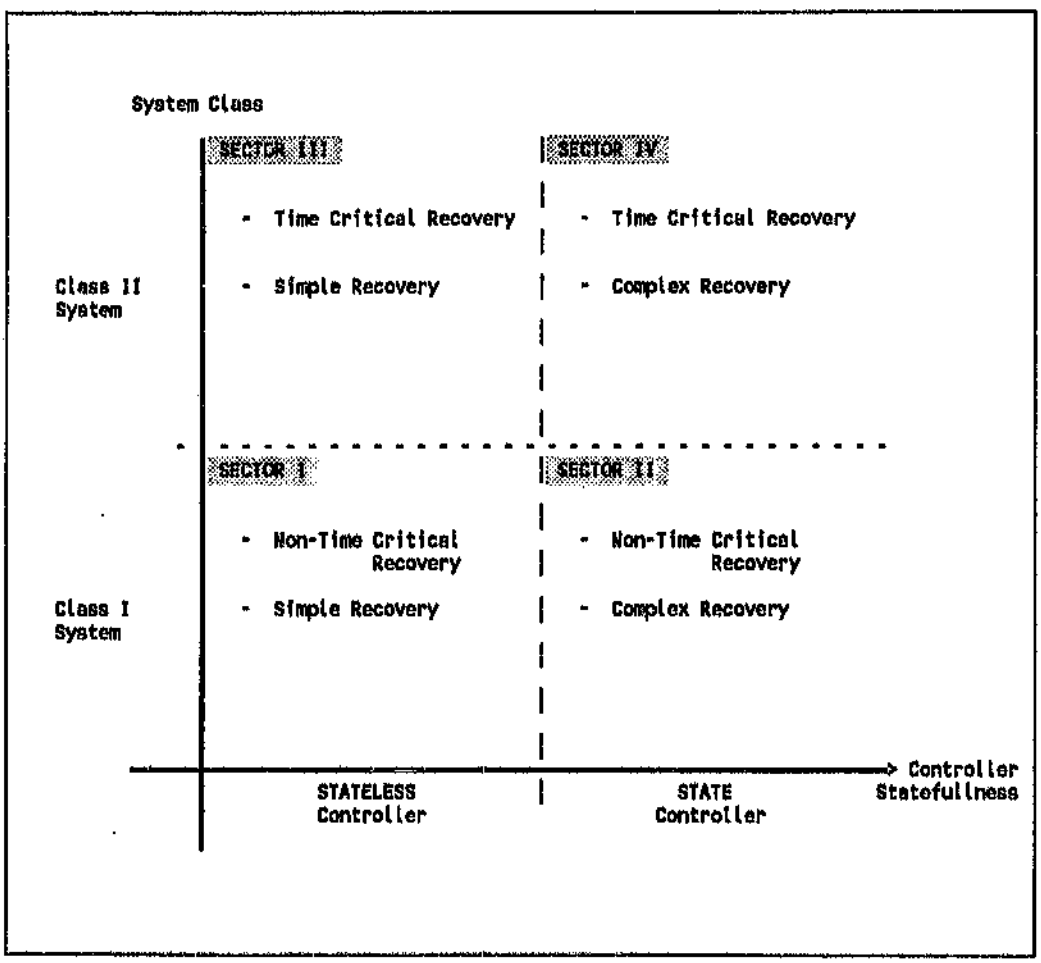


Table I: Error Recovery Modes

2.8 CONCLUSION

In this chapter we have analyzed the fundamental factors that influence error recover in real-time computer systems. The purpose of this analysis is to identify and pinpoint the key factors that need attention when designing an error recover strategy and also to clearly define the type of error recovery scenario that we are interested in addressing. We have shown that there are three factors which influence the performance of error recovery.

- Firstly we introduced the new concept of Class I and Class II systems. This concept and the definition of these two classes of system raises the importance of the time dynamics that exist between a computer controller and the environment that falls under the control of the computer. We show that an environment that can operate independently of the controlling computer presents a significant challenge to the error recovery of that computer.

- The second key factor determining error recovery strategy that we discussed is the presence or absence of a history state in the controller. This we termed the "statefullness" of the controller and we have shown why a statefull controller is the more complex to perform error recovery on than a stateless controller.

- The third factor is the time dynamics of the inputs to and outputs from the controller. These determine amongst other things the statefullness of the controller and the responsiveness of the controller. A controller with a requirement to perform error recovery in a very short time we have termed a time critical system.

In a computer operating as a controller in a real-time environment these three factors interact with each other in a complex way. This interaction has been encapsulated in Table I where we define that a controller can be in any one of four sectors that describe the nature of the error recovery strategy. Of particular interest to us in this thesis is Sector IV which is time critical, complex error recovery. In this thesis we propose an error recovery strategy that satisfies the requirements of Sector IV.

CHAPTER THREE

3. STATE OF THE ART ERROR RECOVERY STRATEGIES

3.1 INTRODUCTION

In chapter 2 we identified three factors which influence the recoverability of a computer system after failure. These factors lead to the identification of sector IV in the error recovery modes table (Table I) as being the problem area. In this sector we must deal with complex, time critical error recovery. In this chapter we wish to relate this requirement for complex/time critical error recovery to the more general problem of fault tolerance. We look at fault tolerance strategies and determine how these cope with the sector IV error recovery problem. By doing this, we identify a particular need for a recovery strategy not provided in existing systems.

A vast array of solutions to the problem of fault tolerance in computer systems has been proposed and/or implemented. Many solutions are hybrids of the theoretical classes of fault tolerant techniques. For example the STAR (Self Testing And Repair) computer which uses reconfiguration and triplication with voting [JOH89]. This is for very practical reasons but the need to group fault tolerance techniques into classes is still a valid exercise in order to understand the relevance of any new scheme that is proposed. This chapter therefore attempts to structure existing fault tolerance strategies into different technique groupings and then examine each to establish how it deals with the problem of complex error recovery as defined in chapter 2. Before commencing this analysis, some of the relevant fault tolerance principles are discussed as they help with the analysis that takes place later in this thesis.

3.2 FAULT TOLERANCE PRINCIPLES

Computer applications have evolved to the point where there is a level of dependence that precludes the simple option of reverting to manual backup in failure situations because either the high level of complexity performed by the processor is beyond human capability, or simply because of great inconvenience. This has been a great driver to improve reliability in computer systems. Advanced military aircraft developed today are designed to be unstable and rely entirely on flight control computers to make them flyable. At the other extreme of the application spectrum, the nature of banking systems has changed to where distributed computer controlled automatic tellers (ATMs) have become the norm. It is a "chicken and egg" problem whether the application demand has driven reliability or whether more reliable computer technology has opened up new applications that require higher reliability. Nevertheless reliability in computer systems has become an issue of central importance.

Reliability in a computing system is a time-based concept of quality and hence is concerned with failures in the time domain. As computing systems are complex and contain many interacting subsystems, the reliability of the total system becomes a probabilistic quantity. From the above can be derived the usual engineering definition of reliability as:

> **"the probability that the computer system will perform a required function without failure under stated conditions for a given period of time".**

The required function of a computer system is usually contained in a specification that defines the desired behaviour of the system. How the requirements or correct behaviour of a system is specified is a complex issue that is dealt with in section 3.2.3 below.

3.2.1 General

In order to achieve failure free operation of a computer system, a Fault Management activity is required which consists of two complementary approaches. These are Fault Prevention and Fault Tolerance. Figure 3-1 shows this fault management hierarchy. Fault prevention can be further divided into Fault Avoid-

ance and Fault Removal. The difference in these two approaches lies in the phase of the life-cycle of the system to which they apply. Fault prevention is an activity that occurs in t' .. 'evelopment phase of the system and includes activities such as good engineering design, QA procedures, component selection, software engineering methodologies, etc. It is essentially an attempt to produce a product with as few faults as possible to begin with. Fault removal is a screening or testing process that is undertaken after development of a system but before full-scale operation occurs.

Due to the complexity and requirements demanded of modern computer systems in terms of performance and cost, experience has shown that fault prevention can never entirely remove the possibility of failure of a system during the operational phase. Therefore, in order to achieve the reliability demanded, fault tolerance is required. Fault tolerance is a capability of a system that is exercised during the operational phase of the life-cycle of the system. The relationship of the fault management hierarchy to the life-cycle phases of a system is shown in figure 3-2. In a typical system, the operational and maintenance phases are cyclically repeated. The purpose of the maintenance phase is to remove permanent faults. These would be either broken components or hardware or software design faults that manifest themselves during the operation of the system.

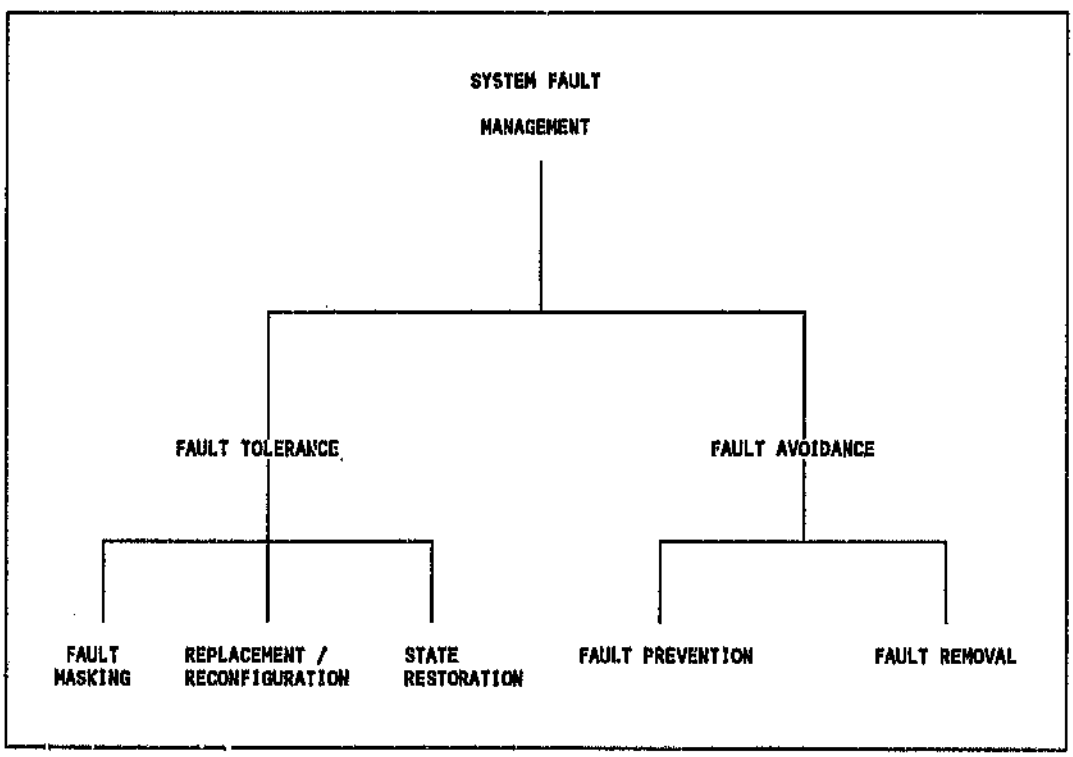


Figure 3-1: Fault Management Hierarchy.

	SYSTEM LIFE-CYCLE PHASE		
	DESIGN AND DEVELOPMENT	OPERATIONAL	MAINTENANCE
FAILURE MANAGEMENT ACTIVITY	Fault Avoidance Fault Prevention + Fault Removal	Fault Tolerance	Fault Removal (Correction of permanent faults)

Figure 3-2: Fault Management and System Life-cycle.

Fault tolerance is distinct from fault prevention in that it is a runtime activity that is designed to deal with faults as they occur during the operation of the computing system. Fault tolerance has traditionally been a hardware issue providing what can be termed [AND81] static or masking fault tolerance. The canonical example of this is Triple Modular Redundancy (TMR) which provides tolerance of hardware module failures. Classical examples are the OAO (Orbiting Astronomical Observatory) satellite, and the guidance computer for the Saturn IB and V launch vehicles for the manned Apollo and Skylab space exploration projects [AVI78]. The failure handling is largely independent of software and is transparent to the system. A failure in one of the TMR modules is masked from the system.

Dynamic fault tolerance on the other hand involves a process driven by the software to implement fault tolerance. This is necessary as the cost of static fault tolerance becomes prohibitively expensive, and even impossible as the size and complexity of the systems increase [AND81].

Dynamic fault tolerance is a sequence of events that occur once a fault has occurred in a system and consists of two stages. A malfunction in the system such as a memory cell failure will precipitate certain results in the computer system that will manifest as an error. The error or errors resulting from this fault must be detected. Thus the first stage of fault tolerance process is error detection.

Once the error has been detected the computing system switches from its normal operation into some form of error recovery or exception handling process. This is the second stage of fault tolerance. A deeper analysis shows that error detection and error recovery can be further subdivided into sub-processes such as damage isolation and containment and fault treatment.

3.2.2 Error Recovery

Error recovery begins once an error has been detected and the extent of damage has been determined. The damage assessment could be implicit in the recovery procedures or explicitly investigated after error detection. Thus after an error has been detected and the damage assessed to determine the extent to which the system state is erroneous, the error recovery process must attempt to eliminate those errors from the system state. Error recovery is thus very closely allied to state restoration. The details of the state restoration vary widely and are discussed in detail in later sections. Fundamentally state restoration requires the presence of redundant information which must be involved in order to restore the system to an error free state. The difficulty in achieving this depends on the extent to which the error recovery process must deal with anticipated or unanticipated errors, and also timeliness of the recovery process which is important in real-time systems.

The recovery process can vary enormously from manual procedures such as replacing a computer board to fully automatic procedures. The automatic recovery procedures typically fall into one of two categories, forward or backward error recovery. While each of these two techniques has strengths and weaknesses, neither explicitly takes account of what the computer systems's environment is doing during the recovery process. This issue has been identified in chapter 2 and is discussed in detail in later chapters.

Whenever the recovery process is invoked, certain assumptions are always implicitly made. It is assumed that the damage has been confined to some degree and that the error recovery process itself is intact. As an extension to this it is assumed that the redundant information that will be used is intact. From this it can be seen that systems capable of early detection of faults and confinement of damage will be far more recoverable than otherwise. The concept of fault latency is discussed below and its impact on the recovery technique proposed in this thesis is discussed in chapter 6.

3.2.3 Correctness Issues

During its operation a computer system proceeds through a sequence of states according to its specification. While it is generally acknowledged that failures in a system must be judged against a specification, it is also recognised that in fact a more complex authority process is called into play in order to determine whether or not a proposed interpretation of the specification, or the specification itself is in error. In their studies [LEE82] and [ROB82] introduce the important concept of an Authoritative System Requirement (ASR). They point out that real system specifications are often incomplete, imprecise, inconsistent, and subject to varying interpretations of their meaning, all of which detract from the usefulness of the specification for the judgement of the system behaviour. For example, it may be argued that a disagreement between a system and its specification is a result of an incorrect specification, rather than a problem with the design, implementation, or usage of the system. To resolve such conflicts it is necessary to have some higher authority to provide the definitive judgement on the system and its behaviour. This they define as the ASR for the system:

> **The Authoritative System Reference for a system can be envisaged as a model of correct system behaviour against which the actual behaviour of the system can be compared. The ASR is assumed to be definitive, and its validity is never questioned.**

In practice the ASR is a complex concept and is the sum of the requirements held by all parties with the power to affect the prescribed behaviour of the system. It is usually captured in written specifications, original proposals, contractual, written and verbal agreements and related reference material.

This definition provides a minimum criterion for adjudicating the correctness of a system in that if a system operates in accordance with its ASR, it is definitely correct. There are, however, other important concepts of correctness that must be considered by the designer of fault tolerant systems.

The terminology we wish to use to define the grades of correctness of a system are an extension to that proposed by [MIL85]. The definitions used in this paper, while mathematically rigorous, do not cover all the concepts we wish to cover here.

We can define three levels of correctness in the state of a system:

1) s-correct (as per [MIL85])
2) l-correct (as per [MIL85])
3) a-correct (an extension to [MIL85])

S-correct: Strictly correct. Such a system will produce the correct outputs in accordance with the ASR and will achieve these outputs by traversing the correct internal states in accordance with the ASR.

L-correct: Loosely Correct. Such a system will produce the correct outputs in accordance with the ASR but will achieve this via states not considered correct in the ASR.

A-correct: Acceptably Correct. Such a system will not produce the correct outputs as defined in the ASR but these will be close enough to correct that it is better to have the system operational than not operational. This is the degraded mode of operation.

The system can be in one of these states of correctness as a result of a failure. Prior to any failure a system is always in an s-correct state. After the failure has occurred the computer undergoes a recovery process which can leave it in any one of the above states of correctness. The relationship between the recovery process and these states are defined as follows:

S-correct: If recovery is successfully achieved through backward error recovery (state restoration) such that the state after recovery is exactly the state prior to failure (from figure 2-2 $C_f = C_r$) then the new state is s-correct. The system will continue executing through correct states in accordance with the ASR and will produce the correct outputs.

L-correct: A system proceeds into an l-correct state after any successful recovery process other that backward error recovery, for example fault masking or component replacement. The state of the computer system after recovery is not the same as its state prior to failure (from figure 2-3, $C_f <> C_r$). The computer continues operation from the new recovery state and will then achieve correct outputs in accordance with the ASR but by using different components and

internal states.

A-correct: If error recovery in a system is not successful, there are two ways of dealing with this system. Either it is shut down and removed from operation or it enters a degraded mode of operation. In this mode the performance of the system is acceptably correct and is preferable to having no controller.

From the above it can be seen that it is the goal of fault tolerant design to achieve an s-correct or l-correct state after recovery. It also behoves the designer to consider degraded or a-correct modes of operation in the event of unsuccessful recovery. In chapter 6.4 we look at the possible a-correct modes for the recovery scheme proposed in this thesis.

3.2.4 Fault Mechanisms, Types and Taxonomy

The definition and description of commonly used terms and concepts in fault tolerant systems are not universally accepted although many common concepts are to be found. Various authors have proposed standard taxonomies or at least explained their view in a number of papers [LAP82], [KOP82], [AND82], [LEE82], [AVI82], [SWE87]. A concept that we identify in this thesis is the necessity to differentiate between error states and error events.

Each of the states that are defined in the ASR are correct states, so under normal operation the computer system proceeds from one correct state to another. The events that cause these state transitions can be viewed at different levels of abstraction or granularity [KOP82] and are for example signals at the inputs from the environment, execution of the computer program or at the lower levels gates switching or clock transitions. In his paper [KOP82] proposes a model in which a failure causes a fault causes a failure and so on, with each cycle increasing in level of abstraction, from a gate failure to a computation fault.

For this thesis we would prefer a model that shows more clearly the relationship between faulty states, correct states and the states traversed between these two. We therefore move away from proposals made by [KOP82], [AND82] and [AVI82]. We see a failure as an event that causes a transition to a state that is not correct. In other words the erroneous state is not defined in the ASR. This erroneous state is

called a fault. Once the system is in an erroneous or fault state it can go one of two ways. It can keep making erroneous transitions to other fault states until it terminates with a system "crash". These transitions would be caused by a sequence of failure events. The second possibility is that it makes a transition back to a correct state. This transition would be an error recovery event. The model we propose, based on our concept of states and events, is shown in figure 3-3.

Events that cause a transition -

i) from a correct state to an erroneous state are primary failure events.
ii) from an erroneous state to another erroneous state are secondary failure events.
iii) from an erroneous state to a correct state are error recovery events or fault repair events depending on whether the fault was transient or permanent respectively. Permanent and transient faults are discussed further below.

A fault (erroneous state of the system) in a component of the system (hardware or software) does not cause a problem or failure until the component is used by the system. Err· fault detection is the process of using the component so as to generate a state transition and then determining if the transition is correct or a failure. If the new state is incorrect, there was a fault in the previous state (assuming no faults in the error detection component).

This introduces the concept of a latent fault. A latent fault is either:

- an erroneous state in a component of the system that is not currently being used or
- a fault in a component that even if it were used at the present time, would not cause a failure transition to occur, although it may do so later under different conditions.

The second part of the definition is necessary to cover substandard hardware components that may fail at some later time even if their ASR is not violated. An example of this is a poor physical connection between a VLSI chip and the pin-outs which may fail under certain temperature and shock conditions that are within the specification for the component.

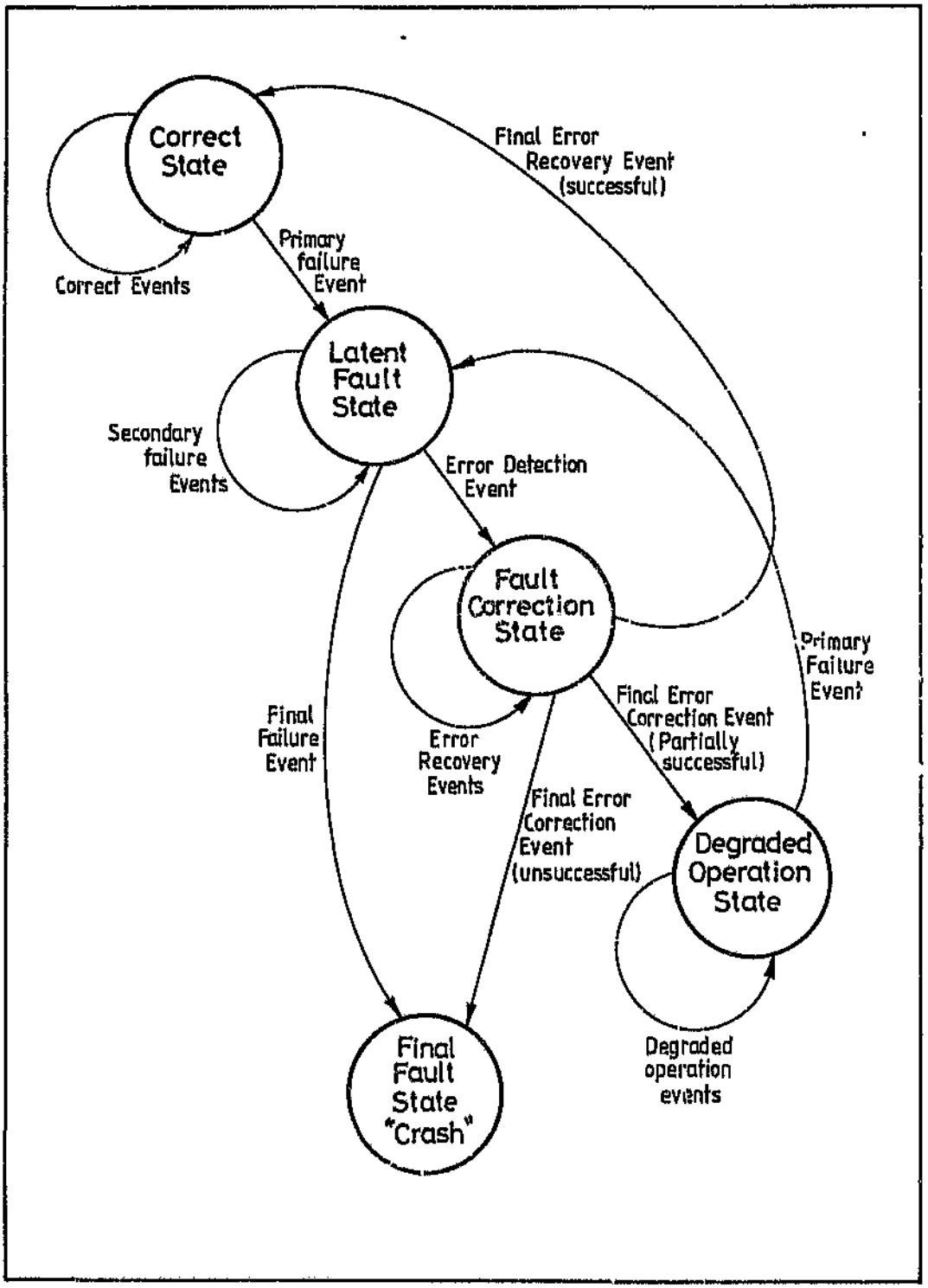


Figure 3-3: Fault Failure State Model

Due to this concept of latent faults it is possible for a system to be commissioned and used for an extensive period without it ever having met its ASR. [SWE87] call this type of fault an indistinguishable fault. In seeking to define the primary failure event, it is important not to consider the failure event giving rise to these indistinguishable faults as the primary failure event particulary for design and manufacturing faults. Clearly the failure event of these faults lie outside and prior to the scope of operation of the system. Primary failure events are therefore defined as the first event that causes the system to deviate from its ASR since the start of the current mission or session. The secondary failures are the events causing transitions to subsequent fault states.

3.2.5 Permanent and Transient Faults

Faults can be categorised according to their ability to persist and resist correction into Hard, Soft and Plastic faults. Hard and Soft faults are more properly termed Permanent and Transient faults respectively, while Plastic faults are usually called intermittent faults. Various error recovery strategies have greater or lesser effectiveness depending on the class of fault precipitating the recovery process.

a) Hard Faults

A permanent fault usually refers to a hardware failure in which a component is irrevocably damaged, such as a burnt out component. Recovery can only take place by physically removing the broken component and replacing it with a good one. This sort of repair cannot usually be done online or during the mission and always requires shutting down the affected part of the system. Permanent faults can also be found in the software in the form of serious specification or coding errors but are usually picked up in the development cycle or early field testing. Masking and Replacement/Reconfiguration fault tolerant strategies were developed with the primary purpose of dealing with permanent faults. The broken component is masked out, ignored or bypassed and the remaining redundant components are used. These strategies do not require any online recovery to occur but with each new permanent fault the level of redundancy is reduced until no more faults can be tolerated.

b) Soft Faults.

Transient faults, also referred to in the digital computer community as "digital circuit upsets", is a fault that can be corrected by the system in which it occurs without outside (i.e., manual) intervention. The correction is performed by the error recovery subsystem and takes the system back to a correct or a degraded operational state. Transient faults are a well-known phenomenon in digital circuit-based components and occur despite the substantial design measures (timing margins, transmission line interconnects, ground and power planes, clock enables of digital circuits, etc.) to achieve a relatively high degree of integrity in digital processor operation. Progress in the form of miniaturisation and increased clock speeds has also played a part. In older digital systems using discrete transistor devices, the transient energy necessary to cause an upset is of the order of 10^{-5} Joules. With the advent of VHSIC/VLSIC, technology digital system upset occurs at only 10^{-9} Joules [HES88] or less. This means that advanced technology systems will be four orders of magnitude more sensitive to upset. In addition, these digital devices can now be upset by transient disturbances that last less than 2 nanoseconds; previously, the disturbance had to last a few microseconds to cause upset.

There is a wide variety of causes of transient faults such as high energy electromagnetic interference (lightening and RF), cosmic radiation and alpha particles, and power line surges and brownouts. One of the most serious sources of soft errors is ionizing radiation. This can be separated further into alpha particles from the decay of Uranium and thorium impurities and cosmic rays. Uranium and thorium are present as traces in a number of materials used to fabricate integrated circuits. Cosmic rays contain penetrating particles which tend to generate cascades of secondary particles as they propagate through matter. Based on a number of reasonable assumptions [SAV86], the soft error rate of a large VLSI chip (1 cm^2) could be of the order of 10^{-4}/hour. Thus as the failure rates due to fabrication problems decrease, the soft error rate will become the dominant factor in determining reliability with the scaling down of technology [SAV86].

Masking or Replacement/Reconfiguration techniques can deal with transient faults if the fault occurs only in one channel or lane or component. However, often the cause of transient failures has a high common mode potential such as lightening strikes or power brownouts which will reduce the effectiveness of these techniques. State restoration techniques both for typical Class I systems or for Class II systems as proposed by this thesis are well suited to deal with transient faults. The loosely

coupled and distributed nature of real-time error recovery techniques defined later in this study make the system relatively immune to common mode faults.

c) Plastic Faults

One further form of fault that is sometimes overlooked in the literature, needs to be discussed. In this thesis it is termed a "plastic fault" as it lies somewhere between a hard fault and a soft fault. Some authors have recognised this class of fault, such as [AVI82], and have termed them pseudo transient, intermittent or pattern sensitive faults.

These faults are caused by an interaction of a latent permanent component fault with a rarely occurring combination of conditions. This is often exhibited as a transient software fault and occurs when an unusual set of conditions arises that, together with some latent software fault, causes the system to fail. When the system is restarted it will perform correctly again as the specific set of conditions has gone even though the latent software fault remains. An example is the case where an input variable goes beyond the range for which it is specified or designed in the software. Although the software may be correct in terms of its written specification, the physical parameter (which does not know or care about the spec cation) may behave temporarily in an unexpected way. This occurrence, together with the fact that the software does not cater for the unexpected value, can cause a failure in the system. Another example is when an unusual combination of variables in a computation yields a divide by zero that is not explicitly catered for. These faults are not the typical soft fault as the failure will always occur given the same set of variables or input conditions. It is also not a hard fault as the next computation cycle, containing new variables and/or input condition, will not yield the fault again.

Plastic faults relate not only to the software domain as described above but occur similarly in hardware such as pattern sensitive faults in semiconductor memories, marginal timing situations and race conditions in logic designs.

The recovery technique we propose in this thesis is well suited to handle this class of fault as shown in later chapters. Masking techniques that use identical hardware and software in the different lanes would not detect such faults at all. Replacement /Reconfiguration techniques such as Recovery Blocks may correct for such a fault if the different recovery blocks used do not all suffer the same weakness.

3.3 FAULT TOLERANCE STRATEGIES

3.3.1 General

A large number of strategies have been devised which have the goal of minimising the impact of a failure in a subsystem on the behaviour of the larger system. Most fault tolerance strategies aim to mask the effect of a fault by containing the effect to some level in the hierarchy of assemblies and components that make up the system and preventing this fault from impacting the operation of other components. These strategies can be divided into three major categories. These categories are descriptive and not exact as many real fault tolerance strategies are combinations of all three. Historically fault tolerance strategies were related to hardware fault tolerance only, but as the criticality of computer systems increased the need arose to consider software fault tolerance techniques. The categorisation given below encompasses all the above techniques and the analysis that follows investigates the applicability of each strategy to Class II systems. The three categories of fault tolerant strategies are:

a) Static Masking
b) Replacement/Reconfiguration
c) State Restoration

All the above techniques for fault tolerance depend on the effective deployment and utilization of redundancy, that is, extra elements in the system which are redundant in the sense that they would not be required in a system which could be guaranteed to be free from faults. A more abstract approach leads to another view of redundancy [AND81]; given that any system contains a set of components which interact at a number of interfaces, then redundancy within that system may be identified in three forms. Firstly, a component (and therefore all its interactions) may be redundant; secondly, an interface between two essential components may be redundant; thirdly, a component may be used with temporal redundancy, in that repeated use (which is redundant use in the absence of faults) may be made of that component. In all the above cases additional components (hardware and/or software) may be necessary to effect the use of the redundant components.

The recovery of the controller after a fault has been detected and isolated usually falls to one of two techniques: Backward Error Recovery or Forward Error

Recovery. Both approaches utilise redundant information of various types in order to achieve recovery of the internal state of the controller. In the case of Backward error recovery, this involves restoration to a previously existing state. In the case of Forward Error Recovery, this involves restoration to a new correct or acceptably correct state. Neither of these two models takes into account explicitly the ..... dynamics of the environment in which they operate. This has different implications for Class I and Class II systems and these implications are examined below.

3.3.2 Masking Redundancy

Masking redundancy was one of the first techniques used to achieve fault tolerance. Masking redundancy is often equated to a form of redundancy which is termed static redundancy because the redundant components that it involves remain in use, and in the same fixed relationship, whether or not any errors are detected. This fault tolerant strategy is often termed a "replication and voting" system. The canonical example of this is Triple Modular Redundancy (TMR) as described in [RAN78] and [IHA78]. The more generalised form of TMR is N - Modular Redundancy (NMR) in which N copies of the hardware component are duplicated with the outputs going through a voting system to mask the faulty outputs. See figure 3-1. In this technique faults are masked at whatever level the redundancy is designed to be. It could be at the gate level or as in the case of the space shuttle [LOM81] at the processor level. In this case there are five processors (IBM AP-101 computers) that synchronise through a voting circuit. This arrangement allows for up to two failures before the redundancy is exceeded. Further failures will be detected and in this case the strategy changes to a reconfiguration strategy (see below) and only one processor is utilised (the backup processor) with the others switched out. NMR is sometimes combined with Replacement/Reconfiguration to achieve "tolerance renewal" [HOP78] such as in the proposed FTMP system. In this case failed units in the TMR triad are replaced from a pool of online spares. Thus the integrity of the TMR system is maintained as long as there are spares available.

NMR is a very powerful fault tolerant technique and is used successfully with many class II system such as the space shuttle. As long as the redundancy is not exceeded, the fault is effectively masked from the controller and hence from figure 2-2, $C_r = C_f$. In fact the controller never enters the faulty state due to the masking. The problems with NMR are its cost, the risk of single point failures, and the inability to cope with design faults. A good example of a design fault foiling a

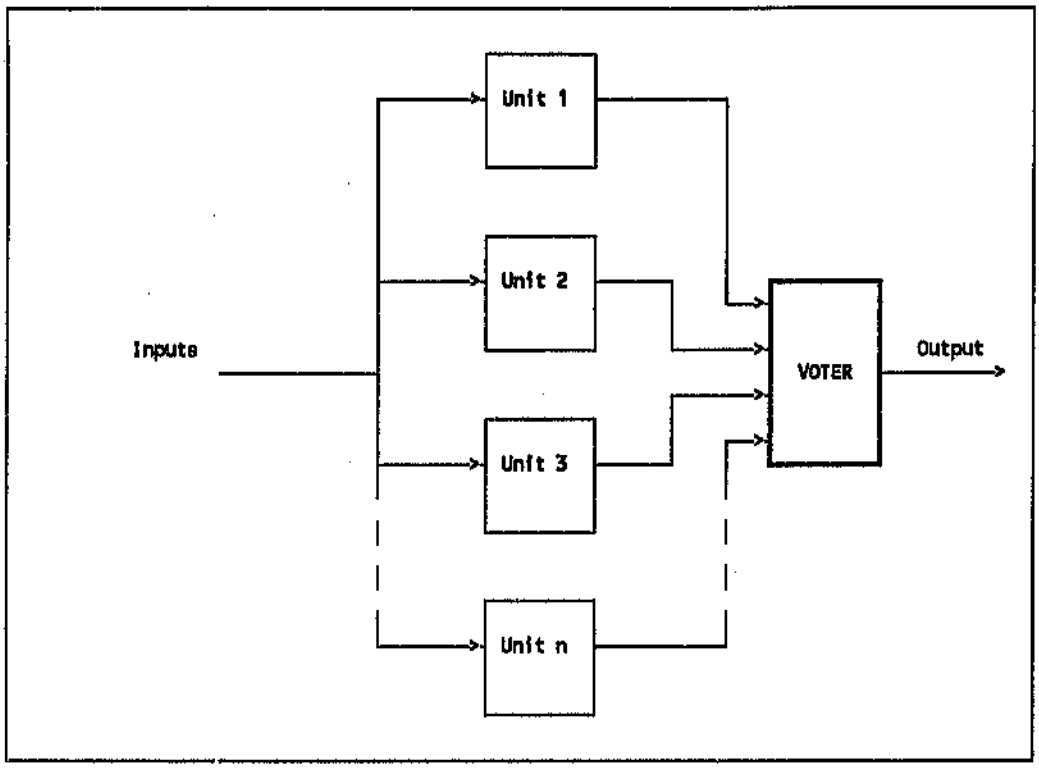


Figure 3-4: NMR Architecture

complex NMR system is the fault that delayed the first Space Shuttle flight in 1981 [GAR81]. NMR is expensive in terms of complexity, space and mass and of course financially as the components are fully replicated and voting and synchronisation system are required. The voting circuit is a single point where failure can occur and so is required to check itself as well as the rest of the system. Designs for totally self-checking triple modular redundancy (TSC-TMR) have been proposed [GAI88]. This additional circuitry reduces the reliability of the system and, as pointed out in [MAC88], in a sufficiently long mission TMR actually exhibits a lower reliability than a simplex system. In the example in [MAC88] the duration is approximately 2.2×10^4 hours. In general when λt approaches unity (where λ is the failure rate) the advantages of NMR become disadvantages because the redundant hardware increases the probability of failure [HOP80]. Thus NMR is most suited to short high reliability missions. The final drawback with NMR is that if all the replicated units have the same fault designed into them, this fault will not be detected. This is true for any common mode faults that are exhibited at the same time by all the replicated units.

NMR was developed as a hardware fault masking technique. A similar technique

of replication and voting for software was proposed as early as 1834 [AVI86] and became known in about 1975 as N version programming (NVP). While in NMR systems the tendency has been the identical replication of both hardware and software, in NVP the trend has been towards diversification [AVI88]. In NVP the replicated units are not be the same but different software modules designed from the same requirements, often coded in different languages [HIL88]. The outputs of the different modules are then put through voting software in a manner analogous to NMR. The purpose of this design diversity is to reduce the change of common mode failures. The success of NMR lies in the fact that hardware failures are statistically independent. This fact is the very weakness of NVP as it has be shown [AVI88], [KNI86], [LIT87], [LIT89] and [TSO87] to be very difficult to write different versions of software from the same requirements specification that have independent failures.

In principle however NVP as a fault tolerant strategy would work with class II systems for the same reasons that NMR does. The faults are masked and the processor never enters the failed state.

Variations on NVP have been proposed such as [MOR87] and [SHE82] that combine NVP with temporal redundancy also known as temporal separation. The concept of temporal separation is essentially based upon the fact that while spatial separation prevents the proliferation of hardware faults, software faults are time dependent. Thus, to ensure that a software fault does not affect all channels simultaneously, it is necessary to ensure that each channel does not process identical data. The method of ensuring this depends upon the fact that for a real-time system such as flight control a high iteration rate is used. The method then is to use inputs from different time periods for each of the channels. This alleviates the problem in NVP of creating multiple versions that have independent failure modes. A technique such as this, as it is also a fault masking process, will in principle also work with class II systems.

In summary then, fault masking by replication and voting is a powerful technique for fault tolerance in class I or class II systems. It covers the problem of Complex Error Recovery as defined in chapter 2 in the sense that in a pure masking system no recovery process occurs. Its major drawbacks are cost and an inability to deal with common mode faults. Fault masking is applicable to architectures that are tightly coupled due to the large amount of sharing of data and sensor and effector parameters. Class II systems such as flight controllers lend themselves to this as well

as justifying the enormous cost [HIL88], [DZW88] and [YOU88]. A widely distributed system as is often found in the process control environment, lends itself more to a fault tolerance approach that uses a loosely coupled architecture. This is the approach used in Real-time Error Recovery discussed in detail in this thesis.

3.3.3 Replacement and/or Reconfiguration

Replacement and/or reconfiguration strategies are commonly termed Forward Error Recovery especially in the software domain. Forward Error Recovery techniques attempt to repair the faulty state, which may be an internal state of the controller or the state of the controlled process such as self-repairing flight control systems defined in [RUB83]. Forward recovery may return the system to a correct state or, if that is not possible, contain or minimise the effects of the failure. An attempt is made to reach a new state that provides for strictly correct operation, or, if this is not possible, loosely correct operation as described by [MIL85]. Examples of forward error recovery techniques include using robust data structures [TAY80], dynamically altering the flow of control, ignoring single cycle errors that will be corrected on the next iteration and changing to reduced functions or fail-safe modes [LEV86]. Forward Error Recovery differs from Backward Error Recovery (described in the next section) in that no attempt is made to restore the controller to a state that existed previously. In this approach there is one primary active component (hardware and/or software) and a number of redundant standby components. The redundant components may be hardware modules or software modules or a combination of both. The redundant modules are "switched in" after detection of the error and may be left in place for the duration of the mission as is typical for hardware modules, or may revert to standby status again once the fault has be cleared, as is the case typically for software modules. Fault detection is critically important as faults are not automatically masked as in the NMR or NVP techniques. [RAN83] gives a summary of a number of fault detection techniques used in replacement/reconfiguration strategies. The reliability will not be improved unless the faults are detected so that the standby modules can be utilised. Separate and additional fault finding hardware and software are necessary to detect the faults and thus trigger the reconfiguration to the standby modules.

Recovery Blocks and Programmed Exception Handling are two examples as defined by [LEV83], [RAN78] and [AND81] that fall into the category of software module replacement in order to achieve forward error recovery. Hardware reconfiguration

strategies are defined by [MAC88] and [HOP78]. A difference that exists between the hardware and software implementations of Replacement/Reconfiguration lies in the way the faults are detected. In the Recovery Block scheme an acceptance test is done after the module has executed. In hardware implementations the Built-in Test (BIT) hardware runs in parallel to the application and triggers an interrupt when a fault is detected. Figures 3-5 and 3-6 show the functional block diagrams for hardware and software (recovery block) implementations of Replacement/Reconfiguration.

A more flexible scheme for error recovery based on reconfiguration and especially geared to distributed systems is described by [LOQ86]. This scheme moves away from the tightly coupled architectures normally defined in terms of software Replacement/Reconfiguration such as in [HEC76] and [AND81]. Aimed at providing distributed computer control systems with a flexible means of fault tolerance, [LOQ86] proposes a system in which standby nodes can implement remote procedure calls. State information is passed regularly at checkpoints to the standby nodes from the active node so that if a module in the standby node must be activated it will have current state information. [LOQ86] does not define how this rather critical function is achieved. He also does not address the problem of processor loading in standby nodes. If a node fails it is assumed that other nodes will have the processing power available to absorb the failed node's functions. [LOQ86]'s approach of using a remote node in the system as a stable storage medium is similar to the RER approach proposed in this thesis. However after this the two approaches diverge. [LOQ86] tries to use the distributed mechanism in a replacement/reconfiguration manner while RER uses it for state restoration. The problems of using a distributed system for replacement/reconfiguration are the practical issues of processor loading. If sufficient power is available in all processors to cover functions that can migrate to them under failure conditions then the processors will have to be significantly over designed for normal operation. It is our contention that the additional cost and complexity would be better utilised by having hot standby replacement modules available for each processor. If, on the other hand [LOQ86] does not intend to over design the processors then he is implicitly defining a system in which a failed mode is automatically a degraded mode of operation. RER on the other hand places very limited additional processor loading on the remote node in the form of RSVS Management as defined in chapter 6.

A similar argument can be levelled against the RAFT system proposed by [AGR88].

This system uses a distributed processor network to achieve fault tolerance but requires any one of the processors to execute any function in the system, under the control of a centralised scheduler. This implies an integrated design with serious problems in time critical applications. This proposal of [AGR88] is very much a replication design and therefore requires significant additional hardware and software resources.

In evaluating the ability of Replacement/Reconfiguration fault tolerance techniques to handle Class II systems and Complex Error Recovery a number of problems emerge. Replacement/Reconfiguration strategies rely on redundant modules being switched in at the time of failure to take over the operation of the failed module. In a system containing state information it is essential that the standby modules are kept up to date with the current state variables during normal operation. In hardware standby systems this is done by continuously updating the "hot standby" module with the relevant state data. In software techniques such as Recovery Blocks it is assumed that the new software modules can be switched in quickly enough that the state data will be valid when the standby is activated. In most of the literature where these techniques of fault tolerance are discussed it is assumed that this is feasible such as in [LOQ86]. It is our contention that this problem is a central issue especially in the presence of event inputs. While Recovery /Reconfiguration will work in principle for complex error recovery if the standby modules are fully up to date with the current state variables, we believe this issue is not sufficiently addressed.

The advantage of Replacement/Reconfiguration strategy is that it can deal with permanent and transient faults and does not contain any single points of failure. On the negative side it is just as susceptible as fault masking to common mode faults and also has the additional cost of redundant modules.

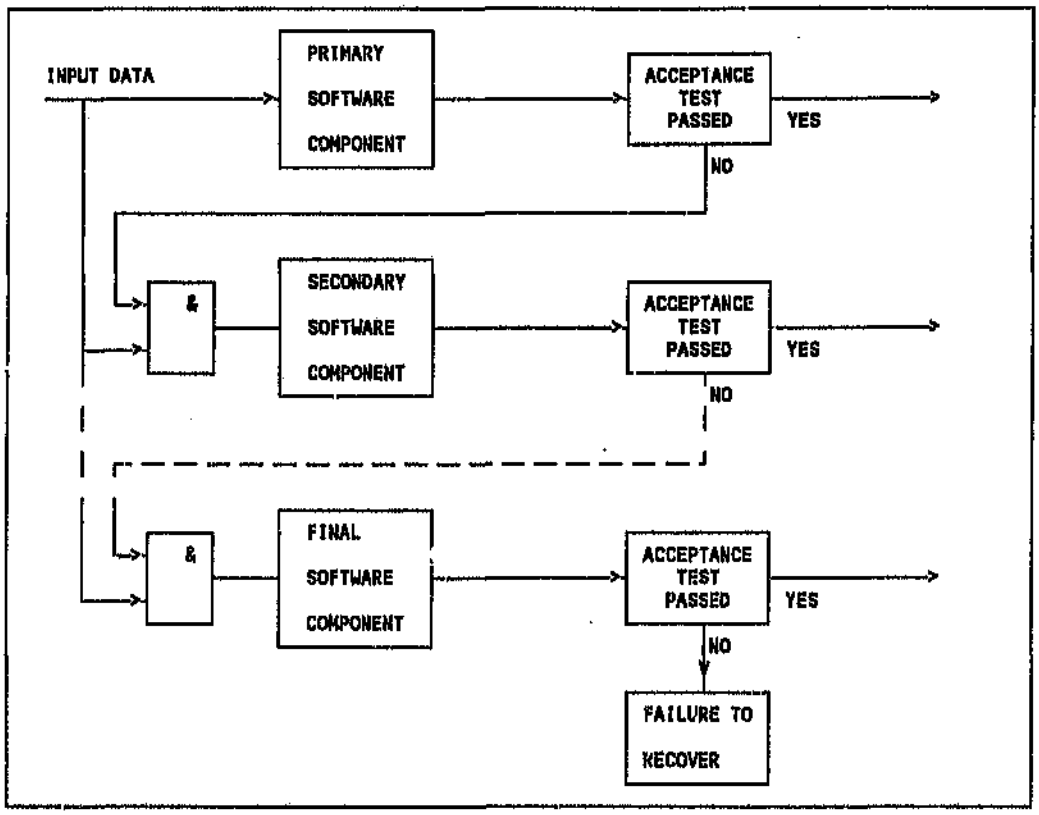


Figure 3-5: Recovery Block Functional Flow Diagram

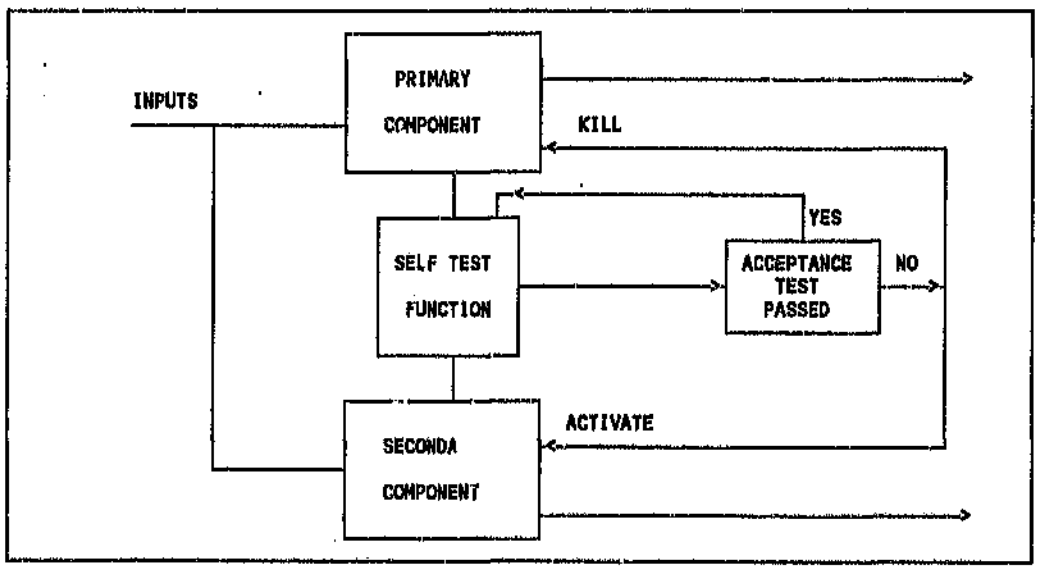


Figure 3-6: Hot Standby Functional Flow Diagram

3.3.4 State Restoration

State restoration (or Backward Error Recovery) as a fault tolerant strategy involves transforming the currently incorrect state in a controller to a correct state that existed previously in the controller. Backward Error Recovery is a primary example of such a state restoration process in which the controller is restored to a previous valid state, no matter how long ago that state existed. This process has been described in detail in a number of texts [RAN78a], [AND81], [RAN78b], [VER78] and has been designed to be utilised with database systems as defined in [KOH81] and [VER78]. It is well suited to these types of systems but not to Class II distributed computer control systems. The reason will become clear from a description of the process. In outline, the process consists of periodically saving the internal state of the controller, usually the data files onto some form of stable storage. This action is known as checkpointing. As the internal state may be large, overheads may be reduced by reducing the frequency of checkpointing the entire internal state and for the in-between periods, recording only those elements that have changed in what is known as an audit trail. Recovery consists of reloading the last checkpoint and replaying the audit trail to restore the internal state.

Referring to figure 2-2 we see that Backward Error Recovery consists of translating C_r back to C_p. In other words an interval of time, T_r to T_p is spliced out as if it never existed. This section of time is the period during which the controller was faulty.

For Backward Error Recovery to work all the assumptions of a Class I system must hold. The states $E_r = E_f$ must hold at time T_r. The state of the environment must not change while the controller is out of action. The rollback procedures used in Backward Error Recovery return the controller to the state C_f at time T_r. The controller and the environment are then in the state just prior to failure. Valid recovery can now be achieved by the controller simply resuming operation.

As can be seen from figure 2-2, for Class II systems $E_r <> E_f$ and so restoring the internal state of the controller to some previous state (E_f) will not give a valid recovery. By its very "backward" nature Backward Error Recovery is not a suitable recovery mechanism for class II systems which by definition will continue to evolve through new states while the controller is inoperative. A good example of this difficulty is the "domino effect" [RAN78] that is experienced in distributed systems attempting to use backward error recovery as described above. Although techniques

have been devised such as [KUH86] to permit recovery in distributed systems, these techniques are impractical in Class II systems as they require other non-failed controllers to roll back their states in order to produce a consistent recovery line. These distributed systems are Class II systems when seen from the viewpoint of the failed controller. In other words the state of the plant (including other controllers) has altered during the time the controller was in a failed state to the extent that the previously valid states that have been checkpointed are no longer applicable.

Part of the problem with Backward Error Recovery is the time overhead associated with the checkpointing and rollback activity. In general, control actions which depend upon the incremental state of the system such as torquing a gyro or a stepping motor, cannot be recovered by checkpointing and rollback [ROS82]. While studies such as [YOU74] and [GEL79] have investigated minimising the impact of backward error recovery, these studies concern themselves only with the optimum checkpoint interval. Reduction of this parameter reduces the average load imposed by the strategy during normal operation when the checkpointing is being done, but does not address the instantaneous delays caused during a recovery. Reducing the time spent in the checkpointing process increases the average availability of the system. The key parameter for recovery of a class II system is the instantaneous delay that occurs at the point after failure when the recovery process is instituted.

A technique has been proposed to circumvent the delays in reloading the system state after failure by using a hardware cache [HUN87]. The controller executes out of the cache under normal conditions and the updated contents of the cache and the CPU registers are saved into the main memory when:

- the cache is full
- an interrupt occurs
- an I/O instruction is executed.

This process of saving the cache and the CPU registers is termed the creation of a rollback point. The rollback data is stored in the main memory. After a failure and on recovery the CPU restores its registers from the main memory, clears the cache and executes from the main memory which presumably contains valid state data. One of the major problems with this technique is that the main memory of the controller is assumed to have independent failure modes to the CPU for transient faults such as power brownouts. This is an improbable situation in most embedded and distributed computer control systems. While the concept of saving the controller

state on the basis of state altering events is a good one, the implementation suggested by [HUN87] appears to be impractical for most situations.

In summary, the state restoration technique for fault tolerance as exemplified by backward error recovery is a good general purpose technique suited for class I systems such as data bases. For class II systems and in dealing with Complex Error Recovery the technique fails as it does not take into account the staleness of the data saved at the last checkpoint. Further, for real-time systems with time constraints the overheads of restoring a checkpoint is prohibitive. In practice this method is not used in embedded real-time systems. Some authors such as [HES88] have recognised the need for state restoration of Class II systems for catering for transient faults in avionic systems, but the idea has not been fully developed.

This then is where Real-time Error Recovery (RER) as defined in this thesis fits into the picture of fault tolerance techniques. It is a state restoration technique aimed at Class II systems and Complex Error Recovery and therefore explicitly looks at the time constraints imposed by the changing environment on the recovery process of the computer system.

3.4 CONCLUSION

This chapter has two objectives. Firstly, we look at some of the fundamental principles of the fault tolerance process such as the:

- Role of error recovery in the fault tolerance process.
- Issue of the degrees of correctness that may be found in a recovered computer system.
- Fault mechanism and the multiplicity of definitions and terminology.
- Important issue of fault classes such as hard, soft and plastic faults.

These issues have all been introduced and discussed as they have a bearing on the error recovery technique proposed in the rest of this thesis.

Secondly, we have attempted to test a broad spectrum of existing fault tolerance architectures and techniques for their ability to deal with complex, time critical error recovery. This is the key problem we wish to address (see Table I of chapter 2).

We have examined three fault tolerance categories :

- Fault masking
- Replacement / Reconfiguration
- State restoration

which covers the full spectrum of existing strategies discussed in the literature. In one way or another none of the above techniques manages to deal with the full set of recovery requirements we have set as the problem in chapters 1 and 2. Either the technique is too expensive and specialised for general use such as NMR and NVP or too slow for time critical systems such as Backward Error Recovery. In other cases such as Forward Error Recovery, the technique is not sufficiently general to cope with unanticipated faults. In general for the non-static masking strategies the critical issue of the time penalty of the recovery process is not adequately discussed in the literature.

From this examination of existing fault tolerance strategies we see that there is still a need to define a strategy that covers the issues of complex, time critical recovery especially for transient faults that can be applied to distributed computer control system. In this thesis we propose such a suitable technique as a solution to this problem.

CHAPTER FOUR

4. REAL-TIME ERROR RECOVERY (RER)

4.1 INTRODUCTION

In the previous chapters we introduced the concept of the Complex/Time-critical Error Recovery requirement and have argued that many existing error recovery strategies do not deal effectively with this requirement. Complex Error Recovery captures the requirements for recovery of typical Real-time systems, that is, systems that contain the three factors: Class II characteristics, State variables and Event inputs. Fault tolerant strategies do exist for such real-time systems, the most obvious being the fault masking technique of TMR. As covered in our earlier discussion we have seen that TMR or any of the fault masking techniques do not actually contain an error recovery phase. TMR is only found in avionic or spaceborne applications and is not all that well suited to industrial applications which are characterised by wide physical distribution, incremental growth and relatively restrictive budgets. Other fault tolerant strategies that do have an error recovery phase exhibit their own set of shortcomings. Forward error recovery techniques such as Recovery Blocks suffer from the problem that recovery can only occur for anticipated faults. That is for faults for which a recovery block has been written. The backward error recovery, or state restoration approach is more general than forward error recovery and seems the most promising. However, our examination of this technique in chapter 3 shows that this approach has not been well developed for Class II systems or for systems with event inputs, in other words for real-time systems.

In chapter 2 we defined the modes of error recovery in a computer system in Table I. From this we see that depending on the system Class and the Statefullness of the controller we have four types of error recovery. These are described in sector I to sector IV and cover the cases of time critical, non time critical, Complex or Simple. The preferred sector to be operating in from an error recovery point of view is sector I where recovery is non time critical and simple. This involves a Class I system and a stateless controller. The Class of a system is governed by the nature

of the application in which the controller exists. This is normally a given and is not a design parameter that can be selected when developing a fault tolerant system. As stated in the previous chapters the objective is to cater for Class II systems. From this argument we see that we are required to operate in sectors III and IV. The factor that the designer does have control over is the "Statefullness" of the controller. If a state controller can be reduced to a stateless controller then error recovery moves from being complex to being simple. This means we move from sector IV to sector III. This is the underlying principle of the error recovery technique we wish to introduce in this thesis.

In this chapter we introduce Real-time Error Recovery (RER) as a technique that addresses this problem of complex error recovery operating under time critical constraints. We first examine the basic concepts and then at the type of architecture in which such a system can be realised. As we have identified above, certain applications lend themselves to specific computer architectures. In this study we look specifically at the loosely coupled architectures found in many industrial process control or materials handling environments. We then "walk through" the process in order to highlight the various phases of its operation. Being a technique aimed at real-time applications, timing considerations are, of course, very important. In this chapter we look at the timing issues and explore some of the limitations and driving factors for the implementer of such a system.

4.2 PRINCIPLES OF RER

Real-time Error Recovery is a fault tolerance strategy that implements state restoration in real-time. The aim of Real-time Error Recovery is to reduce the error recovery process to that of a startup procedure for a stateless system where the initial state of the processor is derived entirely from the inputs to the processor, i.e., the measured outputs of the environment. The recovering processor determines its appropriate internal state in order to be synchronised with its environment by obtaining data from the environment itself. This approach is simple for a controller which does not contain a history of the environment within its internal state. At this point we need to consider more fully the nature of internal states of a controller in a system.

Referring to figure 2-1 we see that an environment or a plant has an internal state. This internal state is the set of values of physical parameters which at any point in

time describes the state of the plant. These state parameters vary in time according to the design and construction of the plant, the inputs to the plant (e.g., process inputs, control inputs, disturbing inputs) and the history of the plant. As many of the physical parameters in the plant are not directly measurable, they must be deduced by the effects they have on the other measurable physical parameters. This requires the monitoring over time of the measured values of the plant in order to build up an internal state representation of the plant. These state parameters and any other data built up over time that are necessary for the control of the plant are the history-state or h-state of the controller (see figure 2-5).

In order to control the plant, the controller requires a model of the plant's behaviour as well as an internal representation of the plant's physical parameter values. The model of the plant's behaviour is not usually time varying and so the algorithm describing the plant behaviour in the controller does not change in time. It is always at the initial state and is referred to by [KOP80] as the Initial-state or I-state

The recovery strategy of a controller depends largely on the extent to which the controller contains an h-state. A controller with an empty h-state (a stateless component) has a Simple recovery strategy. Such a controller may, for recovery, read its inputs and directly generate outputs. Thus restarting the controller after failure will always generate correct results. The restart always begins at the same initial state which will be the same as the startup process. For stateless class I systems the recovery strategy is simple and is equivalent to a restart. Regardless of when the failure occurred, nothing is lost as outputs are generated cyclically for each input.

For those components which do contain an internal history state (h-state) the approach proposed in Real-time Error Recovery is to export the set of state variables (S) to a stable storage node on a real-time basis during operation such that on recovery from a failure, the state variables can be restored. These restored state variables, together with the current inputs from the environment, then constitute the necessary and sufficient information to return the controller to a valid and correct state to continue operation. If one views the stable storage node as part of the processor's environment, we have achieved a situation in which the state variables appear as measured variables that can be read directly from the environment. Of course when the state variables are read back from the "environment" they must be valid and ensuring this is the essential characteristic

of RER. The state component has thus been reduced to a stateless component and the recovery problem has been simplified to a startup operation.

The state variables are exported or saved to a stable storage node which is an essential ingredient of the Real-time Error Recovery architecture. This is a component connected to the processor requiring to implement real-time error recovery. The stable storage node must have independent failure modes such that failures in the processor will not occur simultaneously with failures in the stable storage node. The stable storage node can in principle be realised physically by a number of different devices, but as Real-time Error Recovery requires the state variables to be available with delay times less than the important plant and controller time constants, the responsiveness of the stable storage node is an important parameter that governs RER performance.

Based on these two requirements of independence of failure mode and timeliness of response, we consider in the following section a suitable architecture.

4.3 RER ARCHITECTURE

4.3.1 Generic RER Architecture

From the above section it is clear that the fundamental principle of RER involves securing the controller's state variables in a stable storage node. This occurs during normal operation of the controller. When recovery is required after a failure of the controller, the state variables are retrieved from the stable storage node. From this sequence of events we derive the generic architecture of RER which is shown in figure 4-1. There are three components that make up the generic RER architecture. These are:

- Controller
- Communication medium
- Stable storage node

The broken vertical line in figure 4-1 is the line of failure mode independence. This means that the failures of the stable storage node are independent of the failures of the controller and the communication medium. Thus a failure or malfunction in either of these two subsystems will not affect the integrity of the stable storage

node. The reverse is also true in that a malfunction of the stable storage node will not affect the operation of the controller or communication medium. All three subsystems may in fact be susceptible to the same type of disturbance inputs such as power brownouts or electromagnetic interference from lightening strike, but they must be sufficiently isolated from each other that they are not affected by these disturbance inputs at the same time. Thus an important component of the reliability of RER is that the stable storage node will not fail at the same time as either the controller or communication medium.

The communication medium is used to transfer the state variables from the controller to the stable storage node and vice vera. As RER is intended for use in real-time systems a constraint of a certain minimum bandwidth is imposed on the communication medium. The bandwidth of the communication medium must be sufficiently large that the time it takes to store and/or retrieve state variables to/from the stable storage node is small compared to the rate at which the state variables are created in the controller. A more detailed analysis of the bandwidth requirements of the communication medium is discussed in chapter 6.

The controller itself is seen as the seat of control for the RER process as well being the controller of the application program. Thus in the controller there must be functions, over and above the application programs, to perform, manage and monitor the RER process.

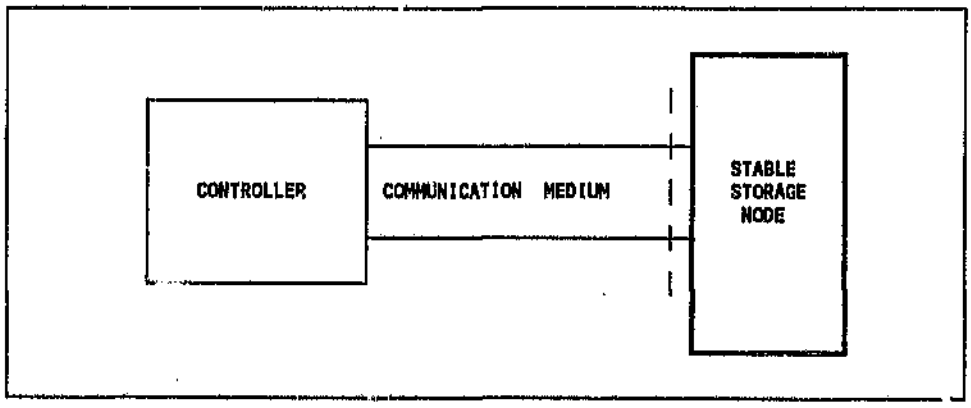


Figure 4-1: Generic RER Architecture

4.3.2 Tightly versus Loosely Coupled Architecture

RER can be implemented by either a tightly coupled or loosely coupled architecture. In the tightly coupled architecture the controller and stable storage node may be coupled together by say the backplane of the computer with the stable storage node being hardened memory protected against the envisaged disturbance input or threat. The advantage of the tightly couple architecture would be the bandwidth achievable in such a dedicated, highly parallel communication medium. The major disadvantage of the tightly coupled architecture lies in the fact that the controller and stable storage node are physically collocated. As the majority of disturbance inputs against which one wishes to protect the system are area disturbances such as power brownouts, EMI/EMC, etc., this collocation presents a significant obstacle to achieving the required independence between the controller and stable storage node to these disturbance inputs.

A second disadvantage of the tightly coupled architecture is that it implies the development of dedicated hardware to the communication medium and stable storage node. This drives up the cost as well as making it difficult to implement RER on an incremental basis.

In this thesis the architecture that has been adopted is the loosely coupled architecture based on the concept of a distributed computer control system (DCCS) architecture such as defined in [MAC83], [IHA84] and [STA84]. Such systems are found frequently in industrial process control and materials handling applications. We will focus our attention on this architecture as it offers a number of significant advantages over the tightly coupled architecture. These are:

- The distributed and loosely coupled nature means that common mode failures are less of a risk than in a centralised or tightly coupled system.

- Data transfer rates are high enough to enable transfer of state variables.

- Such systems are often employed in situations requiring incremental implementation.

For the above reasons, in this study only the implementation of Real-time Error

Recovery with a loosely coupled architecture is considered.

4.3.3 Other Architectural Considerations

In a typical plant there may be a number of controllers covering different aspects of the plant. In exceptional situations all the controllers would have been introduced into the plant at the same time. More commonly an initial set of controllers would be implemented at one time and the rest would be introduced over a period of time as the plant expands and new areas of the plant are brought under computer control. Because of the extended period over which this incremental expansion occurs, an RER architecture that relies on dedicated hardware will suffer from problems of maintainability and obsolescence. This occurs because new controllers are often of a different make or even different technology. This problem of the non-homogeneity of processors in designs of distributed systems is noted in [ROD88] and is a weakness of the fault tolerant designs proposed by [LOQ86] and [AGR88]. These designs have been discussed in chapter 3.

One of the goals of RER is to introduce fault tolerance with as little impact on hardware design and with as much transparency to application software as possible.

4.3.4 Physical and Logical RER Architecture

Based on the generic architecture and the above requirements for RER we propose a physical architecture as shown in figure 4-2. The terminology that will be used in this thesis is as follows:

LOCAL NODE i
This is the ith controller that is required to save its state variables in a stable storage node.

REMOTE NODE j
This is the jth controller that acts as a stable storage node for some local node(s)

LOCAL STATE VARIABLE STORE (LSVS)
This is the set of state variables in the ith local node that must be saved in the stable storage node for the local node to be able to effect recovery after a failure.

REMOTE STATE VARIABLE STORE ($RSVS_i$)
This is the image of the $LSVS_i$ residing in the remote node.

In figure 4-2 we can see the physical architecture for RER consisting of a number of nodes, each being a controller in the system. These nodes are interconnected by a high-speed local area network.

In each node i is a Local State Variable Store for that node called $LSVS_i$. These state variables are saved in node j which is the remote node (stable storage node) for node i. This image of the state variables in node j is called the Remote State Variable Store for node i ($RSVS_i$). At the same time that node j is a remote node for node i, it too has its own local state variable store called $LSVS_j$ which in the example of figure 4-2 is stored in $RSVS_j$ which is located in node k. This symbiotic way in which controllers can operate as remote nodes (stable storage nodes) for each other is explored more fully in chapter 5. The basic concept of mutual storage of state variables is shown in figure 4-3 and is the logical (as opposed to physical) architecture of RER.

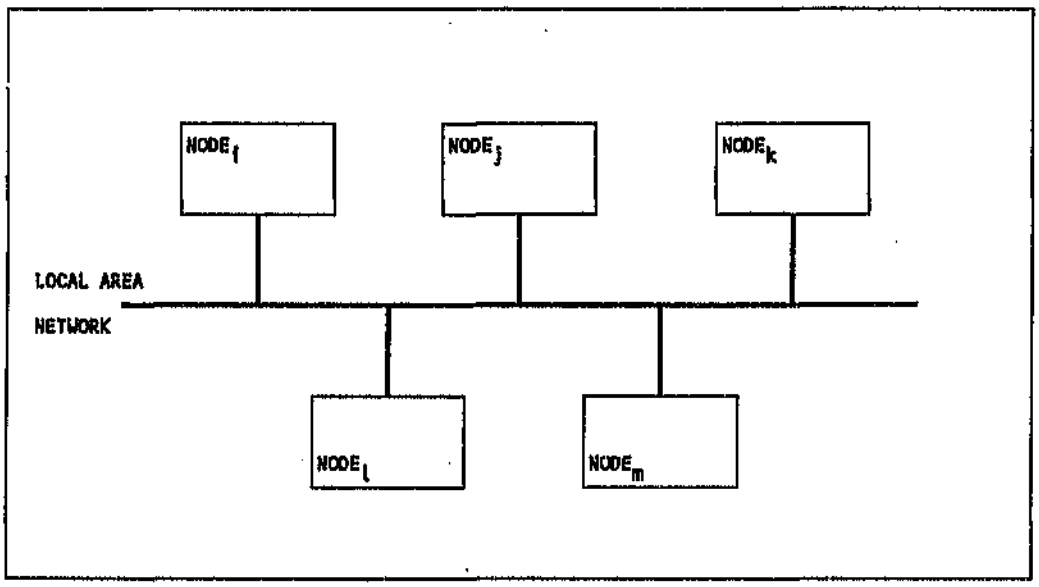


Figure 4-2: Physical RER Architecture

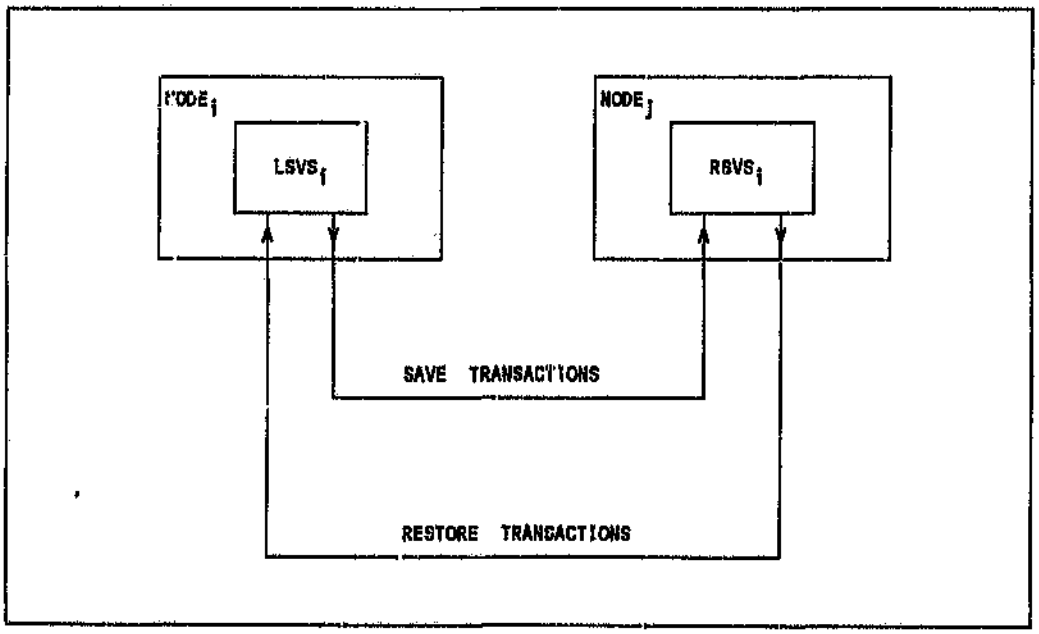


Figure 4-3: Logical RER Architecture

4.4 THE RER PROCESS

4.4.1 General

In this section we describe the RER process. This description is done in conjunction with figures 4-4, 4-5 and 4-6. The first two figures are state diagrams for the local and remote nodes respectively. During the description of the process these two state machines should be seen as running in parallel to each other.

Figure 4-6 is a functional block diagram for the local and remote nodes. The essential functional elements of these two nodes are shown in this diagram. The two nodes used in this description are nodes i and j. Node i is the local node and node j is the remote node (stable storage node) for node i. The functional block diagram for node j shows only the functions that pertain to its activity as a remote node. It should be noted that in practice node j could be a local node with its own controller functions and LSVs. These additional functions are not shown in the diagram. Similarly the local node, node i, could be a remote node for some other local node. These functions are also not shown in this diagram. This aspect of RER is discussed more fully in chapter 5.

The description of the RER process consists of two stages each with an increasing level of detail. In the first stage the state machines of figures 4-4 and 4-5 are walked through. In the second stage only those states which are unique to RER are expanded into greater detail. These three RER phases are the:

- INITIALISATION phase
- SAVE phase
- RESTORE phase

In the expanded detail description the activities in each of these three phases are described in the form of a table. The left-hand column of the table contains the activities that take place in the local node. The right-hand column contains the corresponding activities taking place in the remote node. The centre column called "Notes" contains pointers to notes at the end of the table in which the descriptions contained in the table are again expanded in detail. A timing model of the activities taking place in the SAVE and RESTORE phases is given in appendix A and should be read in conjunction with this section.

4.4.2 Local Node State Machine

Refer to figure 4-4. The local node powers up from the OFF state and enters the LOCAL NODE INITIALISATION state in which basic hardware and program initialisation occurs. This process is the normal cold startup operation for the node. Once this is complete the first of the RER states is entered.

This is the RER INITIALISATION phase which is used to configure a remote node as the stable storage device to be used in the RER process. There are a number of algorithms that can be used during this phase for the processor to determine which remote node will operate as the stable storage device. These algorithms are discussed in more detail below. This phase ends when the remote node has been successfully configured as a stable storage node with an RSVS.

The next state to be entered is the RER SAVE phase. This state operates cyclically with the state NORMAL NODE OPERATION as follows. During NORMAL NODE OPERATION the local node generates a set (S) of state variables. The update of this set acts as a trigger to enter the SAVE state in which the state variables are transmitted to the remote node for storage in the RSVS. When this transaction is complete the NORMAL NODE OPERATION state is again entered and the local node carries on with its controller functions. This process continues indefinitely until the local node detects a failure. This detection can occur in either of these two states and is shown in figure 4-4 as the arc "error detection event". This arc takes the local node back to the LOCAL NODE INITIALISATION state where the program and hardware are again initialised. This state is exited when initialisation is complete and is a local node hot start.

The state to be entered after a hot startup is the RESTORE phase in which the local node initiates the transfer of the state variables stored in the RSVS in the remote node to the LSVS. In this state, validation of the state variables also occurs as described later in the chapter.

If the RER process is successful the local node again enters the NORMAL NODE OPERATION state and continues its controller functions as well as its SAVE functions. If the RER process is unsuccessful, the local node either fails permanently or enters a degraded mode of operation. The state machine in figure 4-4 does not show the degraded mode but this would be the same transition as the successful RER operation.

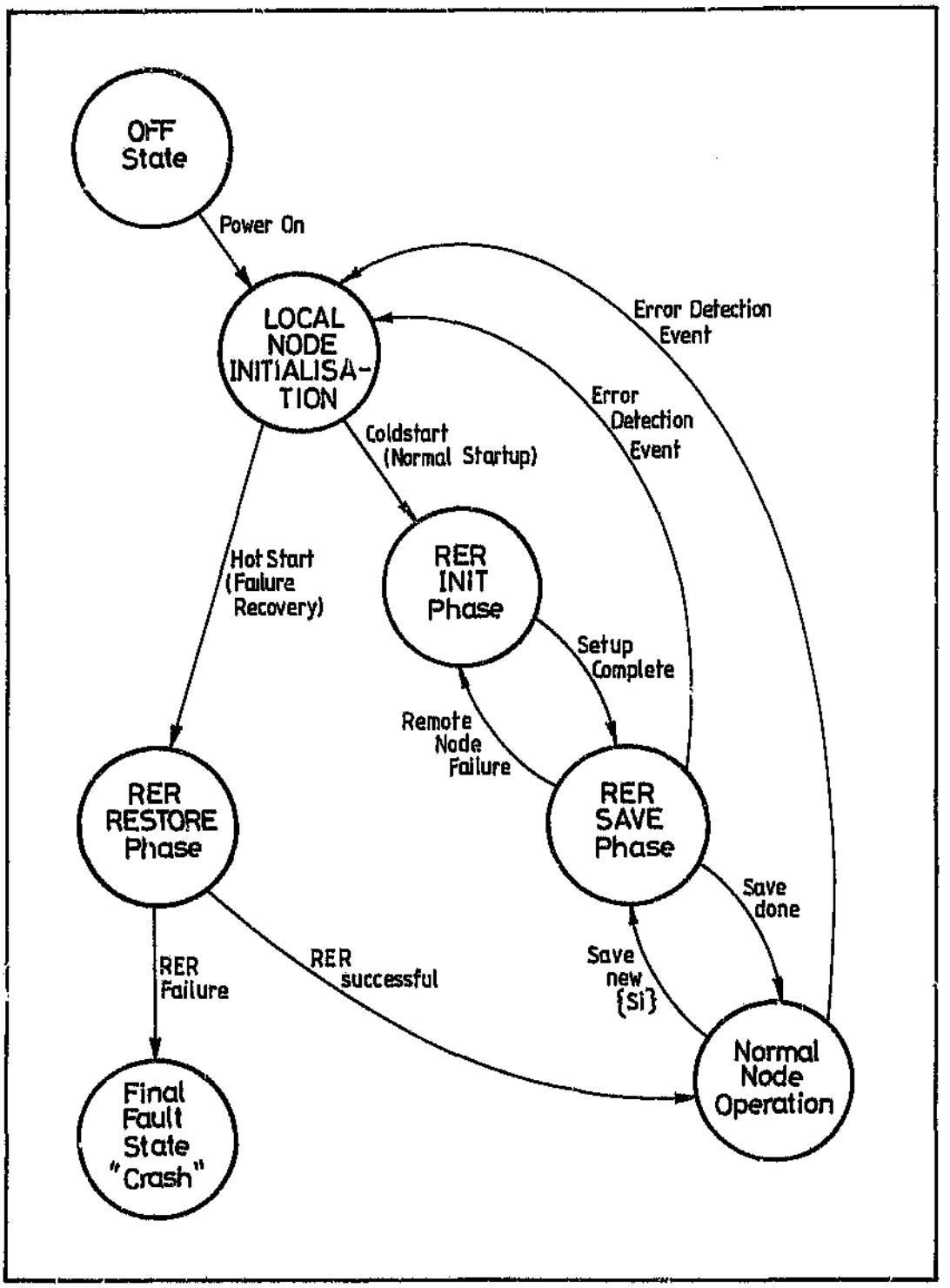


Figure 4-4: Local Node State Machine

4.4.3 Remote Node State Machine

Refer to figure 4-5.

The remote node powers up from the OFF state and enters the NORMAL NODE OPERATION state. In this state it will operate as a controller for some subsystem of the plant. If the remote node were in fact to be a dedicated remote node with no controller function, the NORMAL NODE OPERATION state would simply reduce to a waiting state for a RER transaction from the local node.

The first transaction to occur is the setup request message from the local node. This causes the remote node to transition to the RSVS SETUP state where it sets up the RSVS in accordance to the parameters of the request message. When this is done, a setup response message is issued to the local node and the remote node makes a transition back to the NORMAL NODE OPERATION state.

The remote node will then cycle continually between the NORMAL NODE OPERATION state and the REMOTE PUT state. This corresponds to the save phase in the local node. The REMOTE PUT state is entered when a save request message arrives from the local node and is exited when the state variables have been stored in the RSVS and a save response message returned to the local node.

The remote nodes makes a transition from the NORMAL NODE OPERATION state to the REMOTE GET state when a restore request message is received from the local node. This corresponds to the restore phase in the local node. In this state the remote node sends the state variables in the RSVS to the local node. On completion of this, the restore response message returns the remote node to the NORMAL NODE OPERATION state.

From the above descriptions it can be seen that the RSVS SETUP and the REMOTE GET states are once-off states, while the REMOTE PUT state cycles continually with the NORMAL NODE OPERATION state.

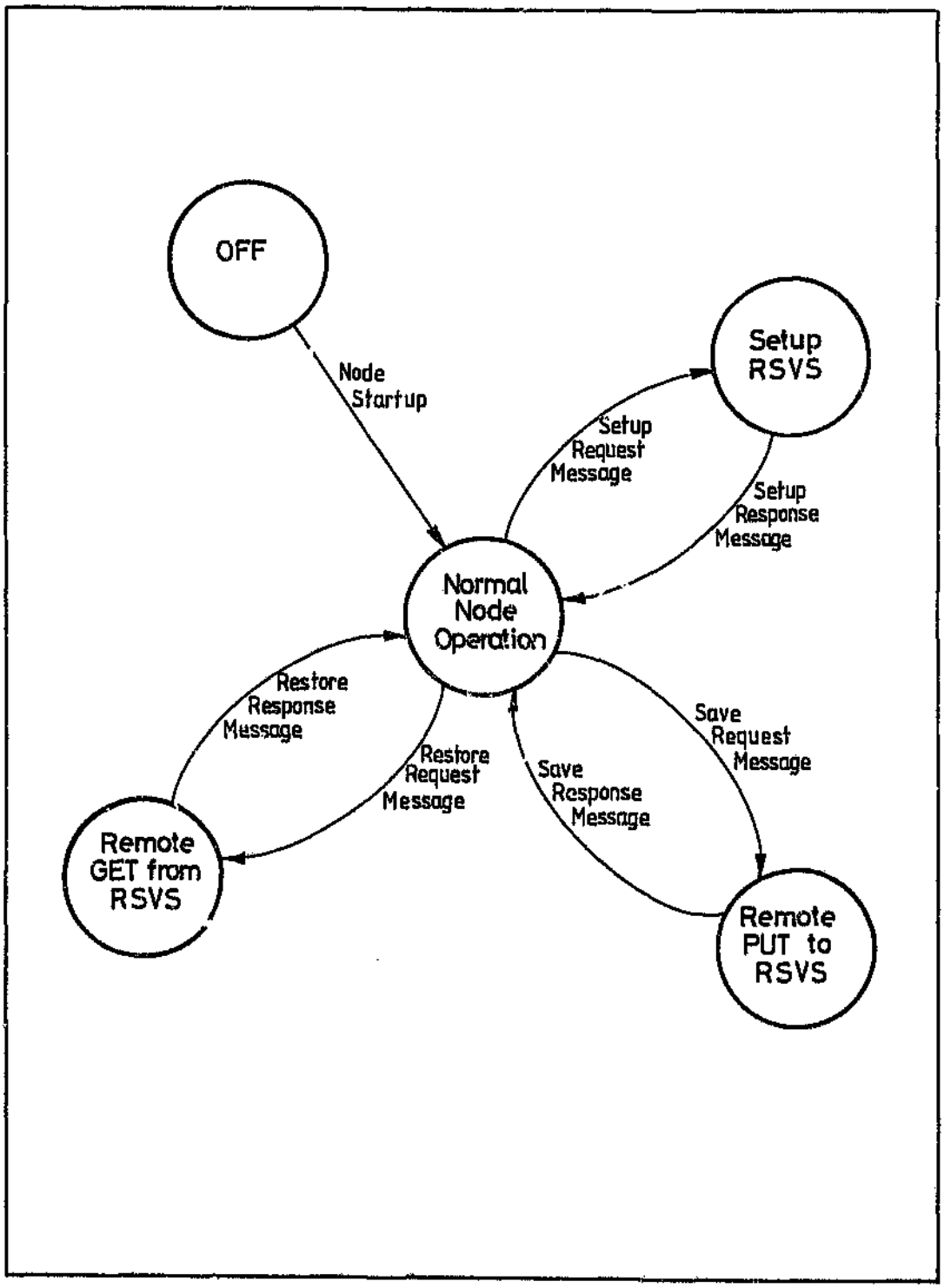


Figure 4-5: Remote Node State Machine

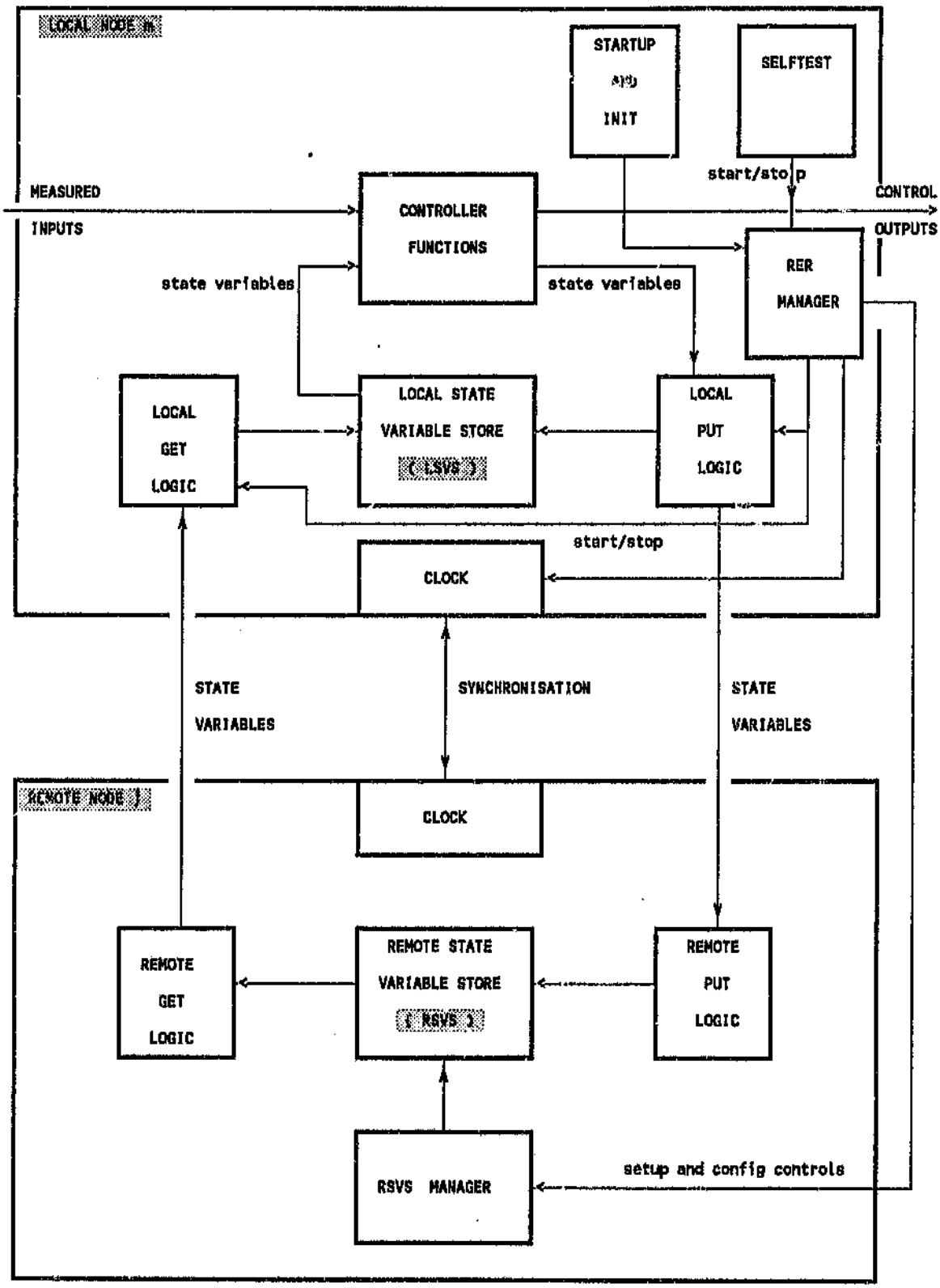


Figure 4-6: The Components of RER Nodes

4.4.4 Detailed Description of the RER Process

The following table contains a description of the operation of each of the three RER phases. These are:

- INITIALISATION phase
- SAVE phase
- RESTORE phase

The table should be read with time running vertically downwards.

INITIALISATION PHASE		
In this phase of the RER process the local node configures the selected remote node as stable storage node containing the RSVS.		
LOCAL NODE ACTIONS	NOTES	REMOTE NODE ACTIONS
1: Select a Remote Node.	1	1: Wait for SETUP Request Message from local node.
2: Send SETUP Request Message to remote node.	2	2: Receive SETUP Request Message
3: Wait for response from remote node.	3	3: Determine if the node is viable as a stable storage device for the requesting local node.

3: (cont.) Wait for response from remote node.	3	3: IF insufficient memory to perform local node function or if node is faulty or unavailable then set status = NOTOK.
3: (cont.) Wait for response from remote node.		4: ELSE IF this node is viable as a remote node then allocate space for the RSVS in accordance with the size parameter in the request message.
3: (cont.)	4	5: Synchronise clock in accordance with parameter in Request message.
3: (cont.)	2	6: Send SETUP Response message to local node with status = OK.
4: Receive status in SETUP Response Message. IF status in Response message is NOTOK or if no response then select next node in the PRL and goto step 2.		

ELSE End of INITIALISATION phase.		End of RSVS SETUP.

SAVE PHASE		
In this phase of the RER process the local node sends state variables to the remote node for storage in the RSVS. These state variables are transferred in a SAVE transaction.		
LOCAL NODE ACTIONS	NOTES	REMOTE NODE ACTIONS
1: For each state variable s_i in the set (S) do: - Save the state variable in the LSVS - Attach the header information to the state variable.	5	1: Wait for SAVE Request Messages from the local node.
2: Send a SAVE Request Message to the remote node.	6	2: Receive the SAVE Request Message from the local node.
3: Wait for response from remote node.		3: For each s_i received save in the RSVS as follows: - Strip off O_i and b_i from the message as well as other comms information. This leaves only the data and expiry time. - Using O_i, compute the address into the RSVS. - Write b_i bits from address.

3: (cont.) Wait for response from remote node.		4: If no problems then send SAVE Response Message to the local with an OK status. If problems then send a NOTOK status.
4: Receive SAVE Response Message from the remote node. IF: status is OK then go to step 1.	6	5: Go to step 1.
5: ELSIF status is NOTOK or if no response then the remote node has failed. Go to the INITIALISATION phase.	7	6: If no SAVE request messages arrive then timeout and abort the RSVS.
Note: if the local node fails then it will go to the RESTORE phase after restart.		

RESTORE PHASE		
In this phase of the RER process the local node implements a recovery of the state variables saved in the RSVS of the remote node.		
LOCAL NODE ACTIONS	NOTES	REMOTE NODE ACTIONS
1: The first step in the restore phase is to determine which node, if any, contains the RSVS. It is assumed this information was lost in the failure / restart process. There are two options: a) PRL b) Broadcast	1	1: Wait for RESTORE Request Message from local node.
2: PRL: Each node in turn in the PRL is polled with a Restore Request Message until one responds as the remote. If no valid remote is found then the RSVS has been lost. Go to INITIALISATION Phase.	8	2: Receive RESTORE Request Message

3: Broadcast: The Restore Request Message is broadcast. If no response is received then go to INITIALISATION Phase.	8	3: Receive broadcast RESTORE Request Message.
4: Wait for response from remote node.	8	4: Retrieve state variables from the RSVS in accordance with the RESTORE Request message and send to the local node in a RESTORE Response Message.
5: Once the remote node containing the RSVS has been found then the following steps are performed. - Read entire RSVS using Restore Request Transaction. - Using TE_i and local clock time determine if S_i has expired. (see section 4.5).	9	

5: (cont.) - If all S_i are still valid then restore them to the LSVS and continue else determine if degraded operation is possible and then continue, else redundancy has been exceeded and the failure is now permanent.	9	
6: If RESTORE Response indicates a WAIT status this means that remote node has failed and is busy with its own RER recovery. Local node then waits for status to change to either OK or to NOTOK.		5: If remote node is performing RER recovery from its own remote node then set WAIT status in the RESTORE Response Message status and send to local node.
Note: If the remote node has its RSVS stored in the local node (see Mutual Architecture in chapter 5) then both nodes will have lost their respective RSVS's. The local node must then return a status of NOTOK to the RESTORE Request from the remote node. This will then cause the remote to return NOTOK to the local node's RESTORE Request. If this is not done then deadlock will occur, as both nodes will wait indefinitely for each other.		

NOTE 1:

SELECTING A REMOTE NODE

We consider the local node to be operating in a network of computers, some or all of which are suitable to operate as a remote node. There are two options for selecting the remote node, one being static, the other dynamic. The two options are based on:

- A Preferred Remote List (PRL).
- Dynamic polling at runtime.

The PRL is a list of suitable nodes that exists in a static form in the local node. The local node simply tries each node in the list in turn until one of them accepts the role as remote node. The list can be prioritised.

The dynamic approach is to broadcast a message which solicits the services of a remote node from all nodes on the network.

The advantages and disadvantages of the two techniques are as follows:

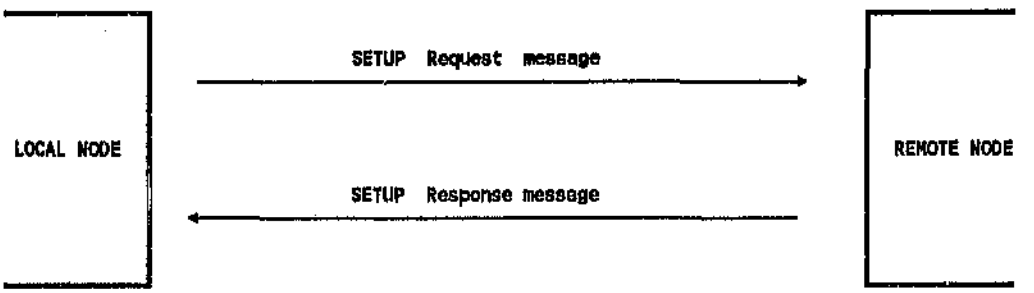
- The PRL approach has the advantage that the system designer can determine which nodes will be remote nodes. This is important from the point of view of loading in the remote nodes.
- The dynamic approach has the advantage of greater flexibility over the life-cycle of the computer network. As new nodes are added they can in principle become remote node candidates.
- A disadvantage with the dynamic approach is that intelligence is required in each node so that it can determine whether it is viable as a remote node.
- Another disadvantage with the dynamic approach is that one node may become the remote node for more than one local. This will affect the loading of this node and also affect the system reliability should this node fail.

The action of selecting a remote node is performed by the RER Manager function in the local node and the RSVS Manager function in the remote node. See figure 4-6.

NOTE 2:

SETUP TRANSACTION.

As part of the initialisation of the RER process the local node must establish communications with the appropriate remote node. This is done by means of a transaction consisting of two messages:



SETUP Request Message	
Param #	Parameter Description
1	Local Node ID
2	Setup Request ID
3	Synchronisation Time
4	RSVS Size
5	Checksum

SETUP Response Message	
Param #	Parameter Description
1	Remote Node ID
2	Setup Response ID
3	Status (OK / NOTOK)
4	Checksum

The SETUP Request Message solicits the recciving node to set up a RSVS in accordance with the size parameter provided in the message.

The response to this request is sent back to the local node in the SETUP Response Message. If this is possible then the status is set to OK else it is set to NOTOK.

The SETUP Request message causes the receiving node to determine if it has sufficient memory to operate as a remote node. This it determines from the size information (given in bytes) in the Request message.

The status is set to OK if the receiving node is available for use as a remote. If this condition is not met then the status is set NOTOK.

These function are performed by the RER Manager in the local node and the RSVS Manager in the remote node. See figure 4-6.

NOTE 3:

ESTABLISHING VIABILITY OF THE REMOTE NODE AS STABLE STORAGE OF RSVS.

Establishing the viability as a remote consists of a test performed by the remote upon receipt of a SETUP Request message. The test establishes if there is sufficient primary memory space available in the node to accommodate the requirements of the RSVS. The sizing parameter for the RSVS are provided in the SETUP Request Message, parameter number 4.

The size parameter is calculated as follows:

bits required:

$$B = F + \sum_{i=1}^{n} b_i$$

Where:

F	=	Overhead bits for Synchronisation time
n	=	number of state variables to be located in the RSVS
b_i	=	number of bits in the ith state variable including the expiry time tag.
	=	$<d_i> + <TE_i>$

Where:

d_i	=	data for state variable i
TE_i	=	Time tag for state variable i
$<TE_i>$	=	number of bits in TE_i
$<d_i>$	=	number of bits in d_i

This calculation is done in the design stage of the controller and is a constant value for a given design. It is transmitted as parameter 4 in the SETUP message.

NOTE 4:

CLOCK SYNCHRONISATION

The synchronisation of the clocks in the local and remote nodes must occur for the validation of restored state variables to be possible. There are a number of ways in which clocks may be synchronised in a distributed system. In order to this without error in a distributed system, in the face of malicious errors, is a complex process and is discussed in [LAM82], [WEN78] and [MAC83].

In the RER process we propose a simple approach in which the remote node includes in the RESTORE response messages the local time in the remote. The remote node updates this parameter in the Response message as soon as its internal clock is updated thereby ensuring that the local node will synchronise with the same time if the transport delay is less than the granularity of the remote node clock. The issues of clock granularity are well covered in [KOP87].

This issue is discussed more fully in section 4.6.

NOTE 5:

LOCAL/REMOTE PUT/GET LOGIC

Refer to figure 4-6.

The principal functions in the implementation of RER are LOCAL PUT and GET functions that reside in the local node and the REMOTE PUT and GET functions that reside in the remote node.

In the local node, state variables are generated by the controller function. These state variables are received by the LOCAL PUT logic and stored in the LSVS. This function then tags the state variables with an expiry time calculated as shown in section 4.5 and then transfers them to the remote node in a SAVE Request message.

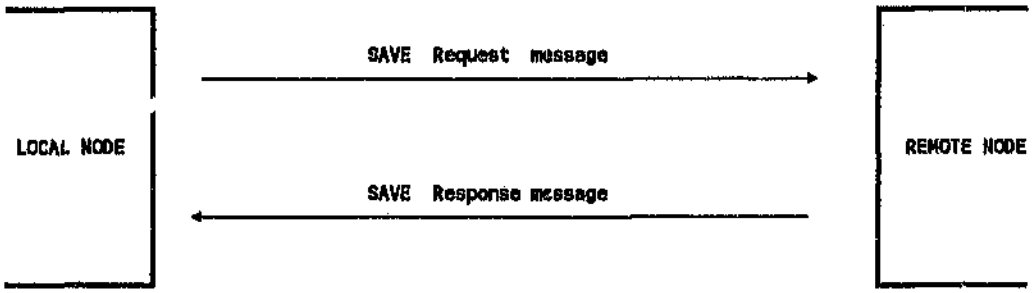
The LOCAL GET logic operates during the RESTORE phase. Its function, under the direction of the RER Manager, is to implement the transfer of state variables from the RSVS using the RESTORE transaction. It then validates the state variables as defined in section 4.5 and restores them to the LSVS.

In the remote node the REMOTE PUT logic operates in the SAVE phase and implements SAVE Request messages, storing state variables in the RSVS.

The REMOTE GET logic operates in the RESTORE phase and implements the RESTORE Request messages by retrieving state variables from the RSVS.

NOTE 6:

SAVE TRANSACTION



SAVE Request Message	
Param #	Parameter Description
1	Local Node ID
2	Save Request ID
3	O_i , b_i
4	State variable i
5	TE_i
6	Checksum

SAVE Response Message	
Param #	Parameter Description
1	Remote Node ID
2	Local Node Id
3	Save Request ID
4	Status (OK / NOTOK)
5	Checksum

For each s_i the following must be put into the SAVE Request Message:

TE_i = expiry time, calculated as shown in section 4.5

O_i = offset in bits to the start of s_i in the RSVS.

d_i = number of data bits in s_i as defined in note 3.

A table containing this information for each state variable is required in the local node. This table is the LOCAL STATE VARIABLE CONFIGURATION FILE and contains the following information:

State variable i	Validity time Tv	Offset O	data bits d
1	Tv_1	O_1	d_1
2	Tv_2	O_2	d_2
:	:	:	:
:	:	:	:
n	Tv_n	O_n	d_n

NOTE 7:

IDENTIFYING REMOTE NODE FAILURE

It is important that the local node be aware if the remote node has failed, for the obvious reason that there is no longer a secure RSVS from which the local node can recover. There are two cases to consider:

a) The communication link to the remote is bidirectional with some sort of PAR protocol for the save request.

b) The communication link is unidirectional with no explicit feedback to a save request.

In case (a) the PAR protocol can be used to give feedback about the health of the remote node. In fact, failed communications will not differentiate if the remote node itself or the link has failed. This distinction is not required, as good end communications are required to save the RSVS.

In case (b) a remote node status monitor function would be required in the local node. This periodically obtains the status of the remote node either by a conversational test (such as sending a test pattern and receiving it back for comparison) or by communication to a health monitor node in the system that knows the status of all nodes. The latter approach is a problem if that link fails as the local node then does not then know the status of the remote node.

A further reason to have regular periodic flow of transactions between the local and remote node is to enable the remote node to time-out as an RSVS for the local. Consider the case where the physical communication link between local and remote is temporarily out of order. During this time the local will establish an alternative RSVS node. If the original remote node does not realise it has been dropped as a RSVS then in the restore phase it will still respond as the RSVS node. The local node may then restore from an invalid RSVS. Therefore regular periodic communications are required so that the remote can time-out as an RSVS once it has not received a valid transaction for a given time-out period. This time-out period must clearly be greater than the time it takes for the local to fail, recover and attempt to restore from the RSVS. If not, the remote node would time-out just when

it is needed. This establishes the lower bound of the time-out delay period:

$T_{delay} > T_{RST}$ (Assumes almost zero time in failed state. See 4.7 for non zero failure durations.)

T_{RST} is defined in appendix A.

The upper bound is determined by the maximum validity time for an s_i (max TE_i)

$T_{delay} < maxTE_i$

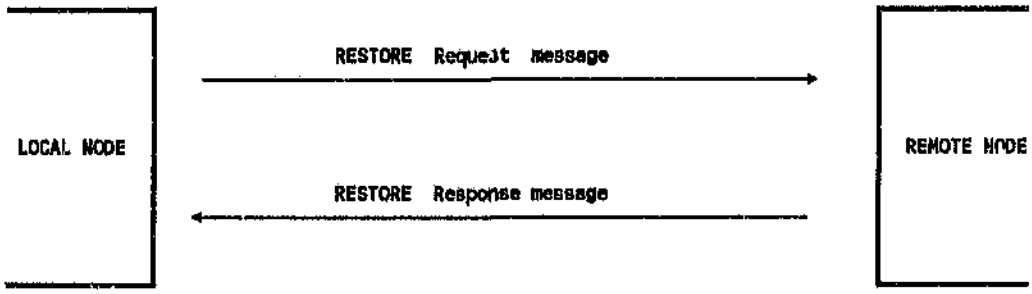
Therefore:

$T_{RST} < T_{delay} < maxTE_i$

In this thesis we consider only the case where the SAVE transaction is bidirectional, as it is essential to have feedback of the health of the remote node. Case (b) above is not considered further.

NOTE 8:

RESTORE TRANSACTION



RESTORE Request Message	
Param #	Parameter Description
1	Local Node ID
2	RESTORE Request ID
3	RSVS size
4	Sequence Number
5	Checksum

RESTORE Response Message	
Param #	Parameter Description
1	Remote Node ID
2	Local Node Id
3	RESTORE Response ID
4	Status (OK / NOTOK)
5	Current Time in Remote node
6	S_i and TE_i i = 1 to n
7	Checksum

If status is NOTOK, i.e., the remote being polled does not contain the RSVS (or the RSVS it did contain is no longer valid), the data stream is omitted.

If it does contain the RSVS, the status becomes a sequence number, so that the RSVS can be broken into a number of PAR transactions each with the number of bits depending on how long the bit stream is.

If the remote node itself is undergoing a RER recovery at the time that the local node tries to activate a RESTORE, the status is returned as WAIT until the remote node has fully recovered.

The restore transaction is implemented by the LOCAL and REMOTE GET logic under the direction of the RER and RSVS Manager in the local and remote nodes respectively. See figure 4-6.

NOTE 9:

MUTUAL STATE VARIABLE DEPENDENCY

The processes involved in restoring data to the LSVS are:

- Receive Restore Response message.
- Synchronise local clock with remote using parameter 5 in the Response message.
- Evaluate TE_i for variables as defined in 4.5.
- Strip off TE_i information.
- Save state variables in LSVS.

These processes may proceed in parallel with intervening queues, but the sequencing of the processes is influenced by the mutual dependency of the state variables.

Mutual dependency here means that the controller is required to use groups of state variables at a time and the group or set must be consistent in itself. For example state variables S_i and S_j define the internal state at a particular interval and for any computation using them they must both belong to the same sample interval.

Consider the case where each state variable is independent of the others. During the restore phase, as each S_i is received from the RSVS and its TE_i validated, it can be restored to the LSVS and used by the processor.

In the case of dependant state variables the entire LSVS must be restored (hence all the expiry times for the state variables must be valid) before the processor can proceed to use the LSVS.

These restoring functions are performed by the LOCAL GET function. See figure 4-6.

4.5 VALIDATION OF STATE VARIABLES IN THE RSVS

In order to determine the validity of the state variables in the RSVS we first need to define how to establish the validity time for any one state variable. In section 4.5.1 we look at the normal derivation for sampling of input variables in a system and then look at how this develops towards a new sampling rate required to implement RER. In section 4.5.2 we look at how the validity time associated with each state variable is used in the RER Recovery process.

4.5.1 RER Sampling Rate

The validity and usefulness of state variables recovered from the RSVS depend on whether the values of these state variables are current when the reconfiguration process is complete. State variables in the LSVS of the controller have a validity time associated with them [MAC83]. The validity time of these variables is usually defined as being a function of the rate of change or gradient of the input variables from which these state variables are computed [MAC83] and [KOP88]. Consider input variable V_i:

$$\text{Maximum rate of change} = \frac{d}{dt}(V_i) = g_i \qquad (4.1)$$

For V_i a maximum deviation between the observed value and the actual value is allowed and is called dV_i where:

$$dV_i = g_i \times dt \qquad (4.2)$$

dt is the slowest acceptable update interval for V_i and is the period during which the sample is considered valid, i.e.:

$$dV_i \leq g_i \times dt$$

This in effect gives the sampling frequency for V_i as follows:

$$f_i = 1/dt \ \text{Hz} \qquad (4.3)$$

The relationship between input variables and state variables can be explained as follows: In the processor state variables are created by combining input variables with existing state variables in a cyclic computation thus creating a new state variable. In general not all the input variables are sampled at the same rate but the state variable computation will be performed at the rate of the variable(s) with the highest sampling rate. Call this input variable V_i. In other words V_i is the input variable, whose sampling rate determines the creation rate of the state variable S_i.

We now consider state variable S_i to be tied by a cyclic computation to the input variable V_i. The creation rate of S_i is then directly related to dt.

Creation interval for state variable S_i is:

$$tc_i = dt \tag{4.4}$$

$$f_i = 1/tc_i \tag{4.5}$$

Once the state variable S_i has been created (computed) in the local node, it is stored in the LSVS and then transferred to the RSVS in the remote node. We define this time from end of creation to successful storage in the RSVS as the SAVE transaction time. This is the time taken for one cycle of the SAVE transaction (see appendix A).

$$\text{SAVE Transaction time} = T_{SAVE}$$

For this analysis it is assumed that the SAVE time is constant for all S_i.

We now define the time taken for recovery of the state variable from the RSVS to the LSVS. This time period begins when a fault in the local node is detected. This then triggers a RESTORE phase which ends when the validated state variables are in the LSVS and the local node is ready to continue execution of application programs (see appendix A). Call this time the RESTORE Time:

$$\text{RESTORE Transaction time} = T_{RST}$$

The relationships of the times defined above are shown in the timing diagram figure 4-7.

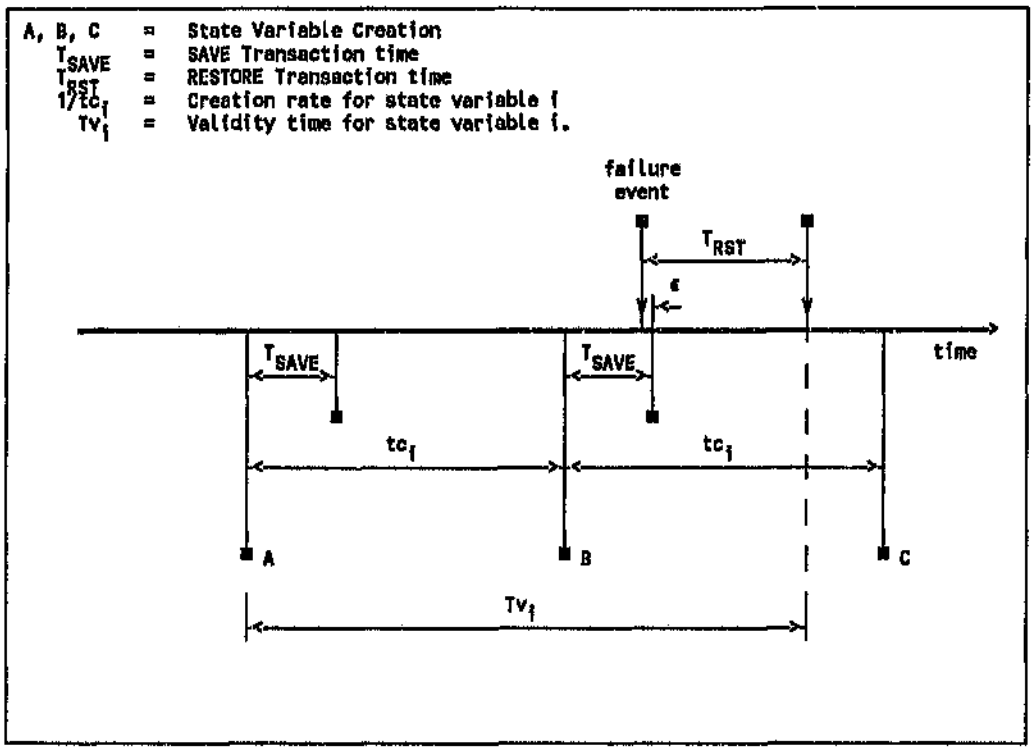


Figure 4-7: Components of Validity Time

From the timing diagram it can be seen that the worst case time occurs when the failure event is detected just before (early by the small time interval ϵ) the end of the transfer time to the RSVS. Two assumptions are made for this analysis:

a) It is assumed that there is no fault latency. This means that the fault is detected as soon as it occurs. This is not generally true and the more complex analysis taking fault latency into account is given in section 6.1.3.

b) It is assumed that when the fault is detected, the transfer process to the RSVS will be aborted and that the RSVS will remain intact with the previous value for S_i.

The failure event occurs a very short time before the end of the transfer time. Call this time interval ϵ. As soon as the failure event has occurred the recovery process begins and lasts for T_{RST}.

From the above it can be seen that the minimum validity time for the state variable S_i is:

$$Tv_i = tc_i + T_{SAVE} + T_{RST} - \epsilon \quad (4.6)$$

Tv_i is the validity time for input variable V_i. Also Tv_i is greater than tc_i. However from equation (4.4) tc_i is derived from the characteristics of the input variable and cannot be altered. The validity time cannot therefore be greater than tc_i without exceeding the deviation dV_i. In order to achieve the required sampling accuracy and take into account the delays introduced by the RER process, V_i needs to be sampled at a rate greater than calculated in (4.5) above. This relationship between Tc_i and tc_i is shown in the figure below:

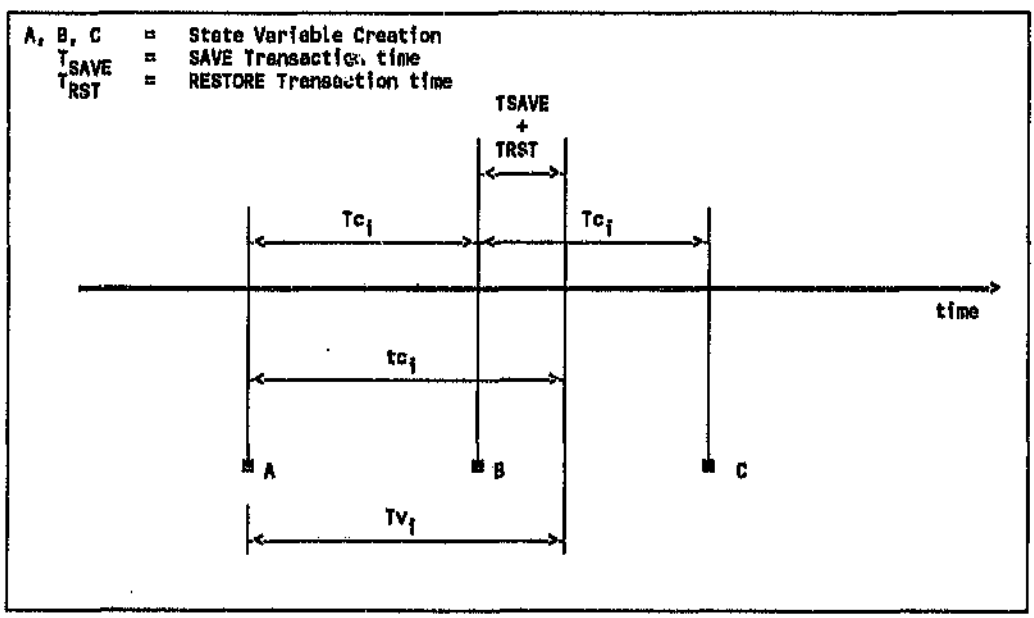


Figure 4-8: Tc_i: The RER Sampling Interval

We define the creation interval of state variables taking into account the RER process as follows:

$$Tc_i = tc_i - (T_{SAVE} + T_{RST}) \quad (4.7)$$

This gives the new higher sampling frequency as :

$$f_{RER} = 1 / Tc_i \quad (4.8)$$

Tc_i now replaces tc_i as the creation rate for S_i and therefore the sampling rate for V_i. This gives the validity time we are looking for:

$$Tv_i = Tc_i + T_{SAVE} + T_{RST} - e \quad (4.9)$$

From equation (4.7) we can see that two cases exist:

$$1) \quad tci > T_{SAVE} + T_{RST} \quad (4.10a)$$

$$2) \quad tci \leq T_{SAVE} + T_{RST} \quad (4.10b)$$

If (4.10a) holds true then it is possible to create the state variable at such a rate that its validity will be assured after RER recovery even if failure occurs at the worst possible instant i.e., e time interval before the end of TSAVE.

If (4.10b) holds true then it is possible for a failure to occur such that it is not possible to restore the state variables before they have expired. In this case degraded operation may be possible (as discussed in section 6.4).

This clearly indicates the need to minimise (T_{SAVE} + T_{RST}) in any system implementing RER. If the input variable that makes up the state variable changes at a rate that exceeds the bandwidth imposed by the communication medium and the RER operations in the local and remote nodes, then RER is not feasible in that system. However, there are many applications of embedded and dedicated real-time controllers of the type discussed in 6.2.2 (d) where the plant processes are relatively slow compared to the computers and communication medium.

4.5.2 Expiry Time Calculation

Tv_i as calculated in equation (4.9) is a time interval. What is required for validation of the restored state variables is an absolute time. This is known as the expiry time and is calculated as follows:

For node k the expiry time of state variable i = TE_i where:

$$TE_i = T_i^k + Tv_i \quad (4.11)$$

T_i^k is the local clock time in node k.

Based on the above definitions, the process operates as follows: When each state variable is computed, its expiry time is calculated and tagged to the variable. This is sent with the state variable to the RSVS. On recovery the expiry time of the state variable is compared to the current local time T_l^k.

- If $TE_i < T_l^k$ (4.12)

  then the data has expired and cannot be used, or may only be used in a degraded mode.

- If $TE_i \geq T_l^k$ (4.13)

  then the data is still valid and may be restored to the LSVS.

4.6 TIMING CONSIDERATIONS FOR RECOVERY

4.6.1 General Timing Analysis

As described above, the local node (K) calculates the expiry time for variable V_I by adding a validity time to the local clock time of node K. In the scenario considered here, the local node fails temporarily at some arbitrary time and as part of the real-time error recovery process it must again synchronise its local clock to the global clock time. As discussed in [KOP87] global clock time in a distributed system is an abstract quantity as there is in fact no one global clock. However, if all the local clocks are synchronised to some accuracy and with the same granularity then it is useful to introduce the concept of a global time such as the average duration between two ticks, and the tick number. In the following analysis we look at the impact that synchronising the local clock to global clock time has on the validity calculation of the restored variable. In this analysis only phase differences between two clocks are considered and not the frequency differences. It is assumed that the clocks are "good" clocks and that deviation from a standard rate is small enough to be ignored.

The following terminology is used in the analysis [KOP87]:

Tg	=	global clock time
T_I^k	=	local time of clock in node K
T_I^J	=	local time of clock in node J
nl^k	=	duration of one clock tick at node K

Consider input variable V_i that has a validity time of Tv_i. The expiry time for this variable has been shown in (4.11) above to be:

$$TEi = T_I^k + Tv_i$$

At this point we wish to calculate what the difference is between the local time in node K and the local time in node J, from which the local node K will restore its clock and state variables. This is derived as follows:

Each local clock in J and K deviate from global clock time by an amount δT.

$$T_l^j + \delta T_j = T_g$$
$$T_l^k + \delta T_j = T_g$$

therefore: $$T_l^j - T_l^k = \delta T_k - \delta T_j$$
$$= \delta T_{j-k} \quad (4.14)$$

Thus, at the time when node K calculated the expiry time from variable V_i, the local time in node J was:

$$T_l^j = T_l^k + \delta T_{j-k} \quad (4.15)$$

At some time later node K fails and as part of the recovery process synchronises its clock with node J. At this point in time:

$$T_l^k = T_l^j + \delta T_s \quad (4.16)$$

Where δT_s is the synchronisation time.

The synchronisation time is always a positive duration during which the node K will establish a synchronised value for its local clock. Of course, due to message transfer times, the local clock will only set to the new synchronisation value some time after this value was achieved by the clock in node J. However, from equation (4.15) we saw that node J's time was different from node K by an amount δT_{j-k}. This must be added to the synchronisation delay to give the difference in time between the original expiry time and the new expiry time in terms of global clock time. Assume that the clock for node K before failure corresponds exactly to global clock time. The difference in global clock time between actual expiry time for V_i and the expiry time now held by node K's clock is:

$$E = \delta T_{j-k} + \delta T_s \quad (4.17)$$

This is shown in the timing diagram example in figure 4-9.

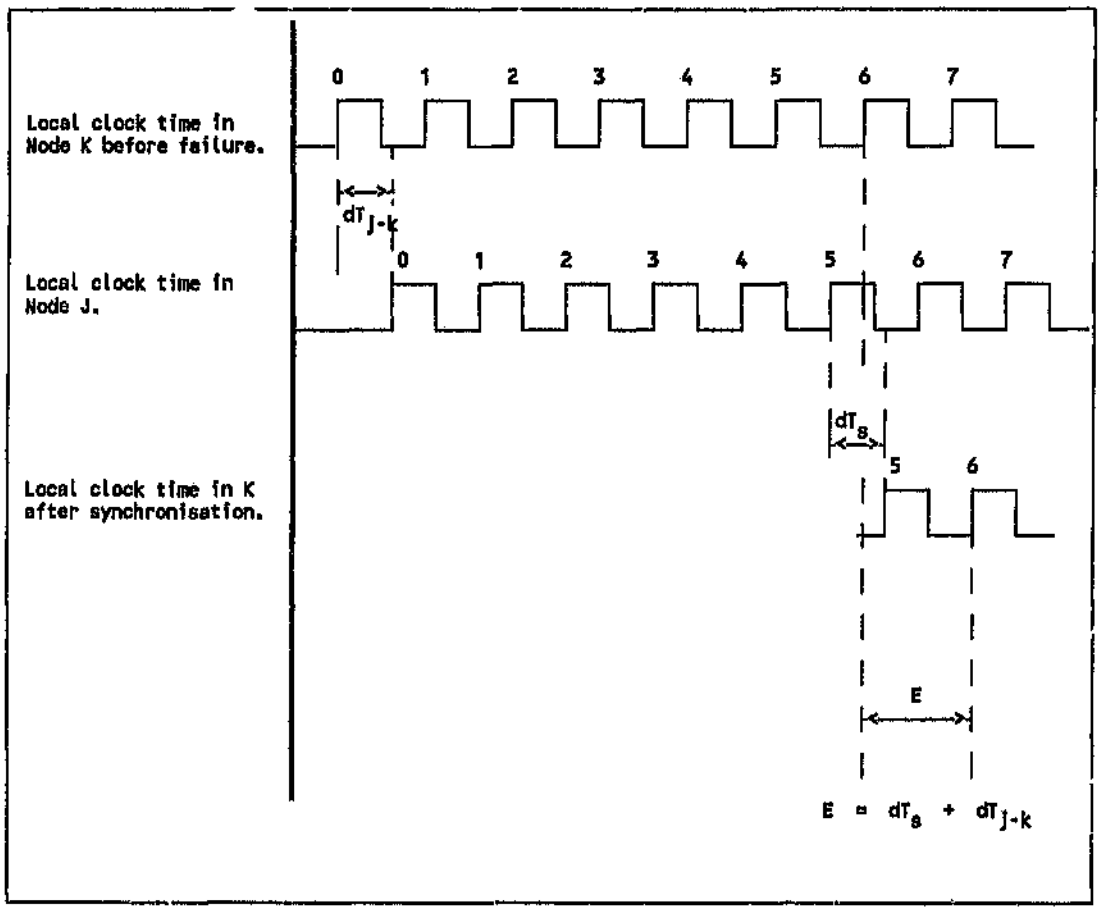


Figure 4-9: Example of Synchronisation Timing

The range of values for E depends on:

δT_s : which will always be positive.

$$0 < \delta T_s < n i^k$$

dT_{j-k} : which depends on the maximum drift between the two clocks.

$$-\text{Maxdrift} \leq dT_{j-k} \leq +\text{Maxdrift}$$

It is possible therefore to end up with either an optimistic or a pessimistic evaluation of whether the restored variable is still valid. Consider the implications:

Case 1: $E > 0$

This means that the new value for TE_i is greater than the old value for TE_i.

$$newTE_i = oldTE_i + E$$

In this case data that has expired will be seen to be valid and will be used incorrectly.

Case 2: $E < 0$

This means that the new value for TE_i is less than the old value for TE_i.

$$newTE_i = oldTE_i - E$$

In this case data that is valid will be discarded unnecessarily.

From this it is possible to derive a relaxed or conservative condition for the calculation of TF_i depending on whether it is better to go into a degraded mode of operation or to operate only with strictly correct variables.

Relaxed: $$TE_i = T_l^k + Tv_i + nl^k \quad (4.18)$$

Conservative: $$TE_i = T_l^k + Tv_i - nl^k \quad (4.19)$$

4.6.2 Local and Remote Node Synchronisation Issues

During the INITIALISATION phase of RER the local node attempts to synchronise its clock with the remote node's local clock. It was identified in equation (4.14) that a difference may exist between these two clocks. The manner in which this difference is managed is as follows:

Consider firstly the case where there is only one local and one remote node. If the difference identified in (4.14) is less than one clock tick in the remote node, i.e.:

$$dT_{j-k} < nl^j$$

the remote node will not detect that the local node time is different to its own and hence no corrective action can be taken. There therefore remains an uncertainty in the clock synchronisation in the order of nl^j.

If the clocks are sufficiently different such that:

$$dT_{j-k} > nl^j$$

the remote node can synchronise its clock to the local node and will only be slow by an amount of T_s as shown in equation (4.16).

Consider the case where the remote node is operating as a remote node for more than one local node thus containing more than one RSVS. This is possible with the star topology defined in chapter 5. In this situation it is not possible for the remote node to synchronise its clock directly with any one of the many local nodes unless the remote node maintains multiple clocks. Synchronisation is therefore achieved by establishing a difference time for each local node K as in equation (4.16). This difference is saved in the remote node and used when each local node is restored.

During the recovery process this difference is added to or subtracted from the remote's local time. This new time is the one inserted as a parameter in the RESTORE Response message and is used by the local node to synchronise its clock.

In this way the remote node can maintain state variables with expiry times synchronised to multiple clocks.

4.7 FAULT DURATION: IMPACT ON RER

In the foregoing analyses of RER we have only considered the case where faults precipitating the RER process were transient faults with zero duration. In this section we wish to analyze the case in which the fault duration is greater than zero. We are still considering soft or transient faults. Typically, a physical realisation of a transient fault with non zero duration would be a power interruption. Transient faults caused by ionising radiation or electromagnetic interference would normally have durations approximating zero.

In section 4.4.2 we described the state transitions that occur in the local node once

a fault has been detected (see figure 4-4). The first process implemented after fault detection is LOCAL NODE INITIALISATION. This description was based on the assumption of zero fault duration. If the fault duration is non zero, the process of LOCAL NODE INITIALISATION is delayed by the duration of the fault. The impact of non zero fault duration on the timing model for the RESTORE phase is shown in appendix A and is identified as :

$$\text{Fault Duration} = T_{FD}$$

The effect of a non zero fault duration ($T_{FD} > 0$) is that it erodes the time remaining to the expiry time of the state variable in the RSVS. This can be expressed as follows:

Let the time at which a fault is detected be:

$$\text{Time of detection} = T_f$$

Based in equation (4.13) we then have:

$$TE > T_f + T_{RST} + T_{FD}$$

This expression tells us that from the time the fault is detected (T_f) there is a time period before the state variables in the RSVS expire. Part of this time must be used for the restore process (T_{RST}) and the remaining time can be absorbed by the local node being in a faulty state (T_{FD}).

We wish to establish the range of values that T_{FD} can take on. The lower bound of T_{FD} is zero as described above for transient faults caused by such events as ionising radiation or electromagnetic interference. The upper bound that T_{FD} can take while still having valid RER recovery is derived from the following expression:

$$T_{FD} \leq TE - T_f - T_{RST}$$

T_f is the only variable and defines the actual time in the SAVE phase when the fault occurs and is detected (we assume zero latency here).

Referring to figure 4-7 we defined the worst case time for a fault to occur as being a short time ϵ before the end of the SAVE transaction. Based on a similar argument

the best time for a fault to occur (maximum time to expiry of state variables) is a short time ϵ after the SAVE transaction is completed. This gives the maximum duration for T_{FD} as:

$$T_{FD} \leq Tv - T_{RST} - T_{SAVE} - \epsilon$$

substituting Tv from equation (4.9) we get :

$$T_{FD} \leq Tc - 2\epsilon$$

Thus the bounds for T_{FD} are:

$$0 \leq T_{FD} \leq Tc - 2\epsilon \qquad (4.20)$$

The above analysis gives rise to an interesting analogy between the RER process and a low pass filter for transient errors as shown in figure 4-10. We have shown that the RER process is effective against transient errors less than Tc - 2ϵ in duration. This is the cutoff point of the low pass filter. Longer duration faults (i.e., faults with duration greater than Tc - 2ϵ) are not stopped by the RER process. This would correspond to permanent faults that exceed the expiry time of the state variables in the RSVS. The RSVS is analogous to a capacitor which stores the information for a period of time. The capacitor is a leaky one as the "charge" decays over time (the expiry time). This is shown in figure 4-11. The concept of a "Soft Error Filter" has been described by [SAV86] in connection with fault tolerance in VLSI circuits. The system defined by this author uses actual filters for transient signal. RER provides a filter for transient disturbances at the system level.

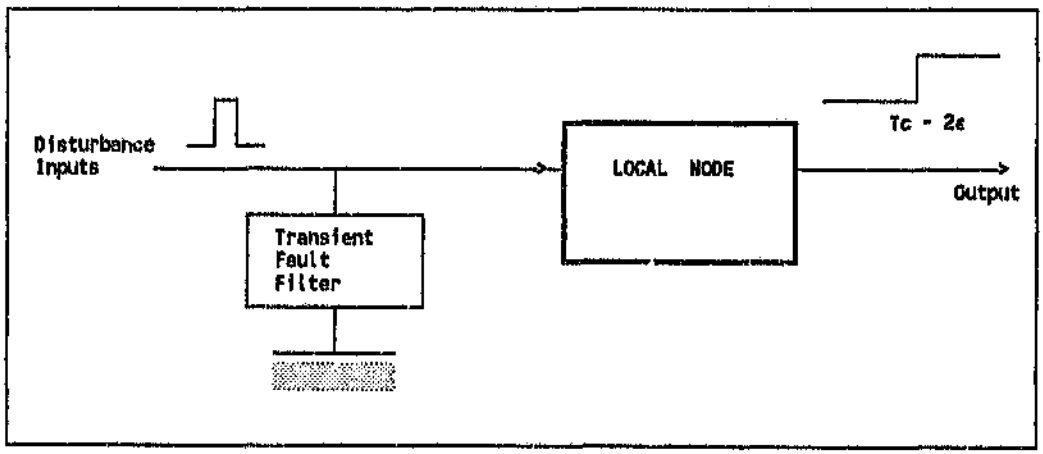


Figure 4-10: Transient Fault Filter Analogy

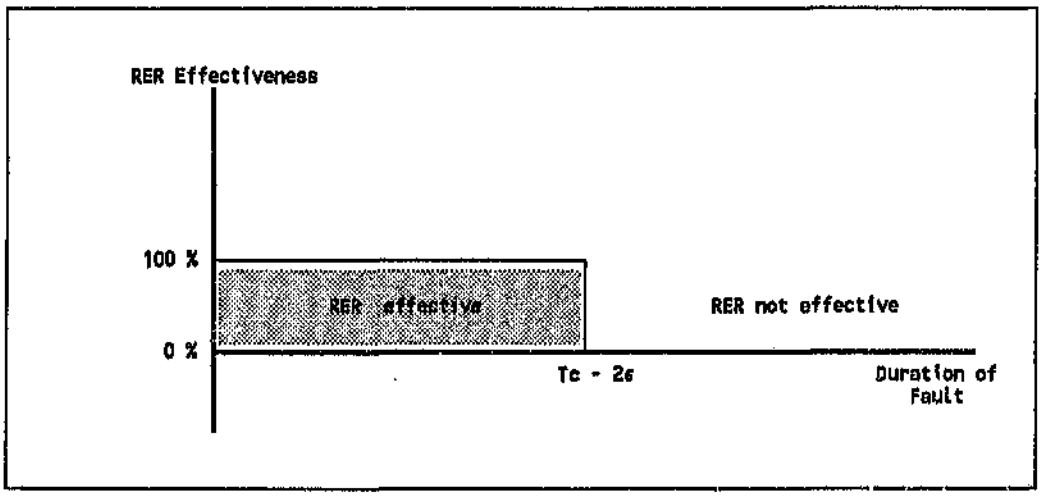


Figure 4-11: Transient Error Filter Spectrum

4.8 CONCLUSION

In this chapter we have proposed a solution to the problem of error recovery posed in chapters 1 and 2 for complex, time critical error recovery. The solution is fundamentally a state restoration technique that circumvents the shortcomings of backward error recovery described in chapter 3. In normal backward error recovery techniques the problem lies in the time taken to restore the computer's state. This is usually a sufficiently lengthy process so as to make it unacceptable for real-time systems and Class II systems in particular. As the process we propose does restore the system state in real-time under defined conditions, we call it Real-time Error Recover and have adopted the abbreviation RER.

We have shown that RER has a generic architecture consisting of a controller, a communication medium and a stable storage node. We show that this can easily be realised in practice in a real-time distributed computer control system consisting of a number of autonomous controllers interconnected by a high-speed local area network where each controller acts as the stable storage node for another controller. By implementing it in this architecture we derive the benefits of physical distribution and also make the technique available to a wide range of industrial applications.

RER consists of a three-phase process. In this chapter we have produced an operational description of these three phases which are:

- Setup Phase
- Save Phase
- Restore Phase

We have shown that it is viable to export state variables from a controller at runtime (Save phase) and save these variables in a stable storage medium for later restoration. The state variables are tagged with an expiry time before being stored in the stable storage node. This expiry time is an important concept in RER and is used to validate the state variable in the RESTORE phase.

The restoration process takes place in the RESTORE phase in which the state variables are retrieved from the stable storage node and checked for validity before being restored to the local processors state variable store. As RER is a real-time error recovery process, validity is very much a temporal issue and the algorithm for determining the expiry time for a state variable is described in this chapter. Other timing issues that are discussed are the problems that arise when attempting to synchronise two remote nodes on a network. This synchronisation discrepancy has a bearing on the expiry time. This is analyzed, showing that it is possible to have a relaxed or conservative approach to the expiry time calculation.

Finally we come to the interesting analogy that can be drawn for RER. It can be compared to a filter existing at the disturbance inputs to the controller that filters transient faults up to a given duration. This duration in effect is the cutoff point for the "low pass disturbance filter" and we derive this cutoff duration in this chapter. Transient faults with a duration less that this cutoff time are filtered by the controller in real-time and are thus transparent to its normal controller function. Faults that are longer in duration that the cutoff are either permanent faults which require repair or transient faults that cannot be dealt with validly given the constraints of the process under control. In other words, because we are dealing with a Class II system, the time allocation for recovery is determined by the environment.

CHAPTER FIVE

5. REAL-TIME ERROR RECOVERY ARCHITECTURES

5.1 INTRODUCTION

In chapter 4 RER has been introduced as a strategy utilising a local node and a remote node. The local node contains a local state variable store (LSVS) and the remote node contains a remote state variable store (RSVS). During operation local state variables are saved in the RSVS, and after a failure, these state variables are restored to the LSVS subject to timing validity criteria. This architecture is shown in figure 5-1.

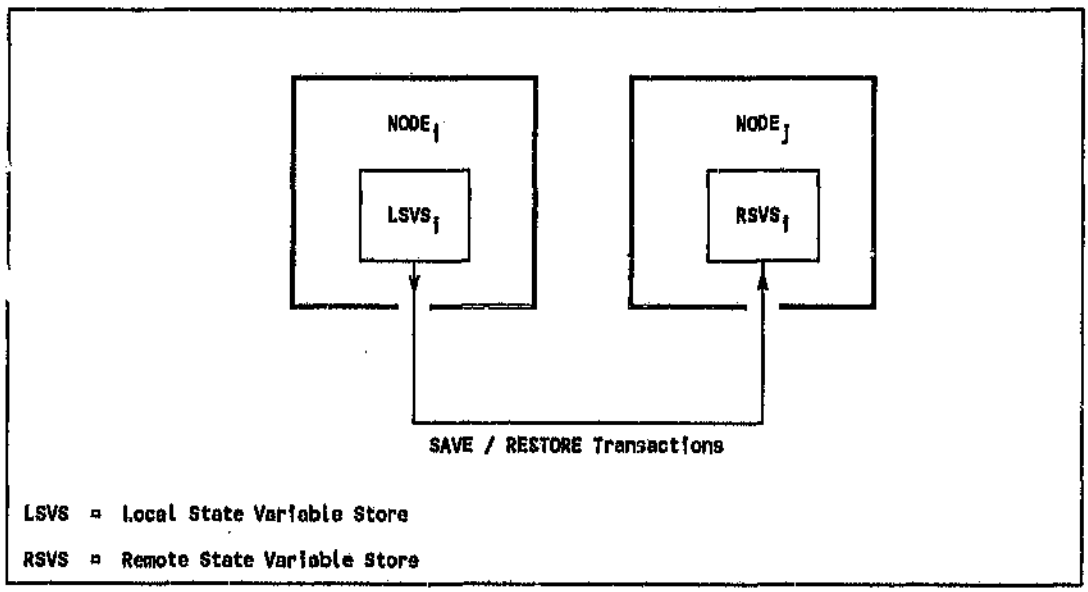


Figure 5-1: Basic RER Architecture

In this study we consider the case where real-time error recovery is used in a distributed computer control system consisting of a number of fairly autonomous and loosely coupled nodes. Such an architecture lends itself to RER because the

nodes have independent failure modes due to the fact that they are geographically separated. In this case each node or controller requires to be able to save its local state variables. This requires mutual service rendered by each node to another.

When the number of nodes is greater than 2 a number of issues arise that require analysis. These are:

a) RSVS Topology:
   Single-point or Broadcast.
   Location of RSVS.
b) Fixed or dynamic RSVS allocation.
c) Nesting of RSVSs.

In this chapter we discuss these issues as they affect the way in which nodes in a network can cooperate to achieve an RER capability.

5.2 RSVS TOPOLOGY

5.2.1 Single-point vs Broadcast

There are two alternative approaches for deciding how to locate the RSVS for a given local node. In the first case the local node is allocated a single remote node in which to store its RSVS. In the alternative approach the RSVS for a local node resides on all other remote nodes capable of providing stable storage of the RSVS in question.

In the first case a logical point-to-point communication link is required between the local and remote node. In the latter case, as the data must be stored on all possible nodes, a logical broadcast link is required for the saving of state variable data. The point-to-point or broadcast links mentioned here are logical operations of the communication system and do not refer to the capabilities of the physical link.

There are pros and cons associated with both these approaches and the best approach to adopt will be determined by system level performance factors such as required reliability, bus bandwidth and others. These two architectures are discussed below.

5.2.1.1 Single-point Remote Storage

In single-point remote storage a single remote node is selected as the holder of the RSVS for each particular local node. The pairing philosophy can be either mutual, sequential or random.

a) Mutual Single-point Topology

See figure 5-2.
In the mutual single-point topology nodes are paired off, each one holding the RSVS for its partner.

Conceptually this mode of operation is very simple. Pairs can be selected on the basis of adequate storage for the partner's RSVS and for a high degree of independence in failure modes. The limitation of this approach is the need for an even number of nodes.

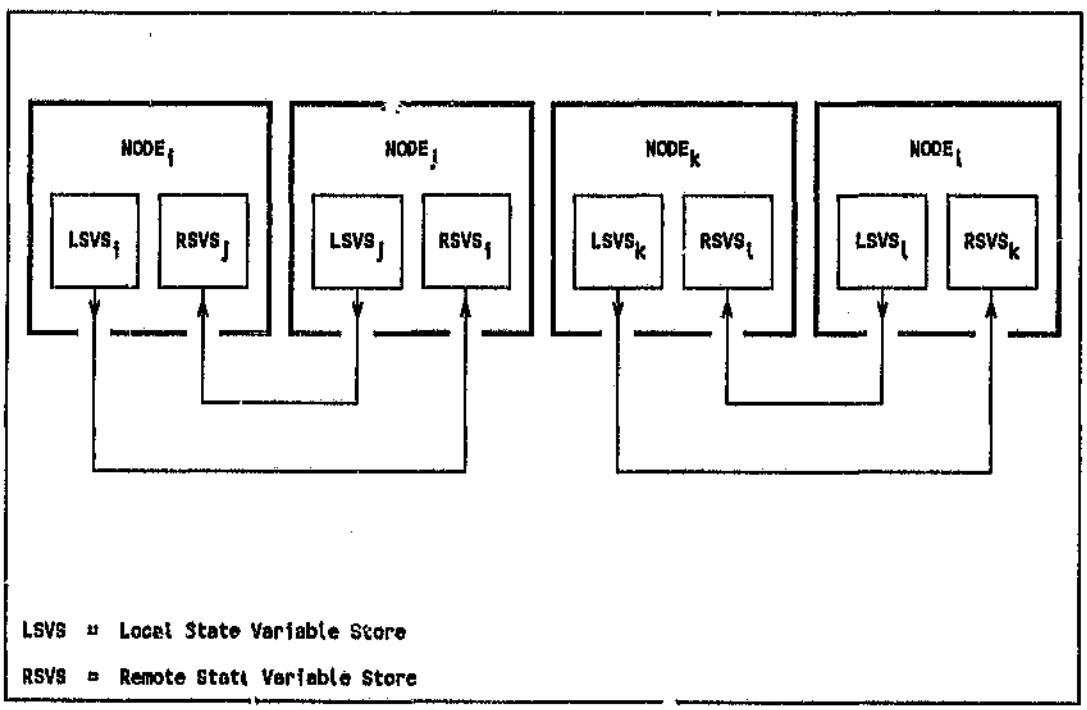


Figure 5-2: Mutual Single-point Topology

b) Sequential Single-point Topology

See figure 5-3.

In the sequential mode of operation each node has an RER link to its logical neighbour. New nodes added to the system require the ring to be broken and re-established with the new node. This will involve a configuration change of the preceding node. The new node will be configured to link it to the following node. Odd or even numbers of nodes can be handled. This architecture is applicable where there is a high degree of similarity of nodes so that a more complex method of choosing the remote node is not warranted. This architecture also protects against failures of adjacent nodes if nested recovery has been implemented (see section 5.4).

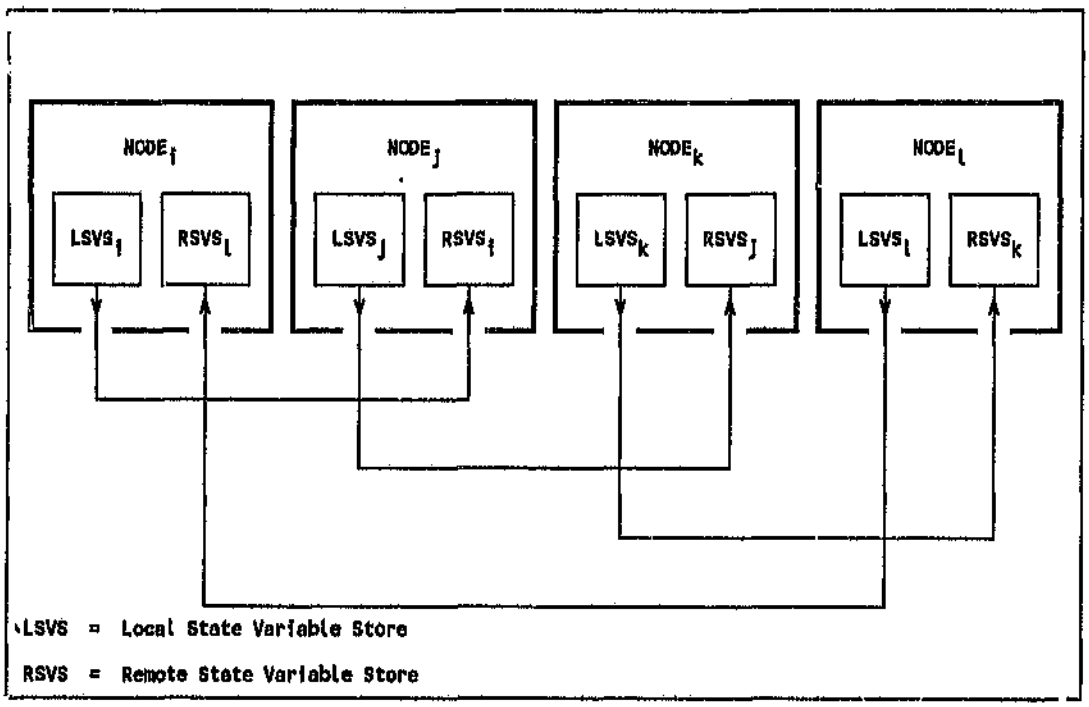


Figure 5-3: Sequential Single-point Topology

c) Random Single-point Topology

See figure 5-4.

In random single-point topology the logical links are random in the sense that they are neither sequential nor paired. This topology will arise in practice when nodes are paired off with each other, not on the basis of physical location on the network,

but on the basis of some performance parameter such as limitations in spare memory and CPU capacity required to handle the RSVS or independence of specific area threats.

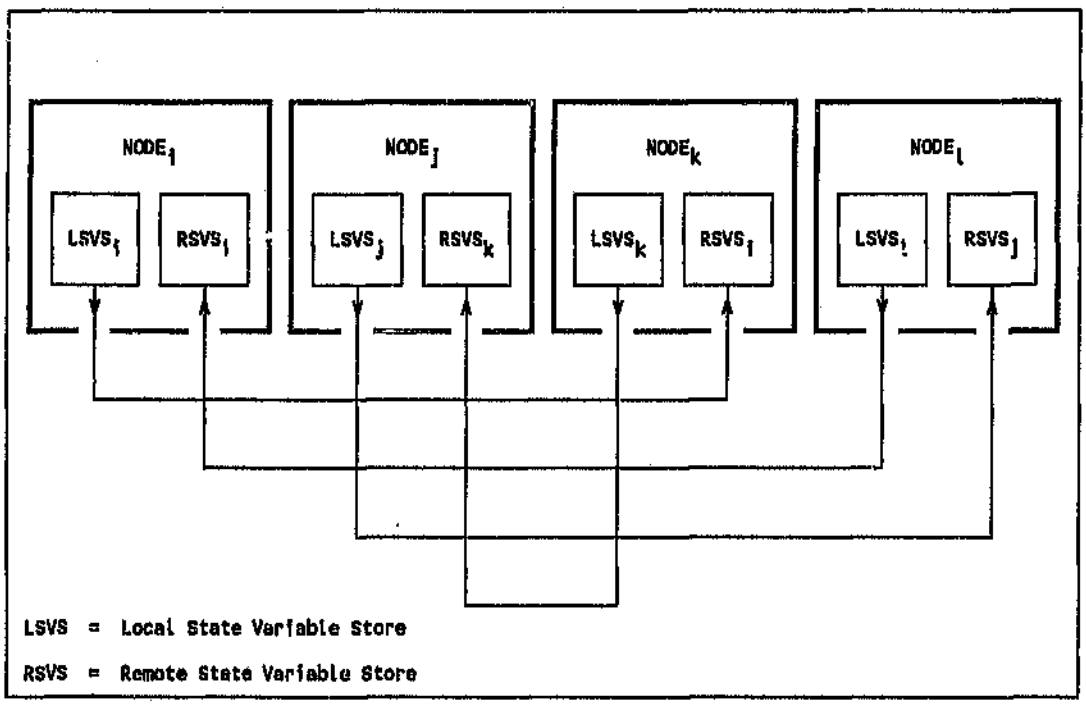


Figure 5-4: Random Single-point Topology

d) Star Single-point Topology

See figure 5-5.

In all the above topologies it has been assumed that a single RSVS resides in a single node. It is possible to locate multiple RSVSs in a single node and in the limiting case to locate all RSVSs in a single node creating a star configuration shown in figure 5-5. Justification for a topology such as this would depend on the relative failure rates of the various nodes. If node m's failure rate is very low relative to node i, j, k and l, this arrangement may yield the highest system reliability. Of course the potential for a single-point failure is unlikely to see such an architecture being selected.

5.2.1.2 Broadcast Remote Storage

Broadcast remote state variable storage derives its name from the fact that local state variable data from each node is broadcast on the network to all other nodes capable of acting as stable storage.

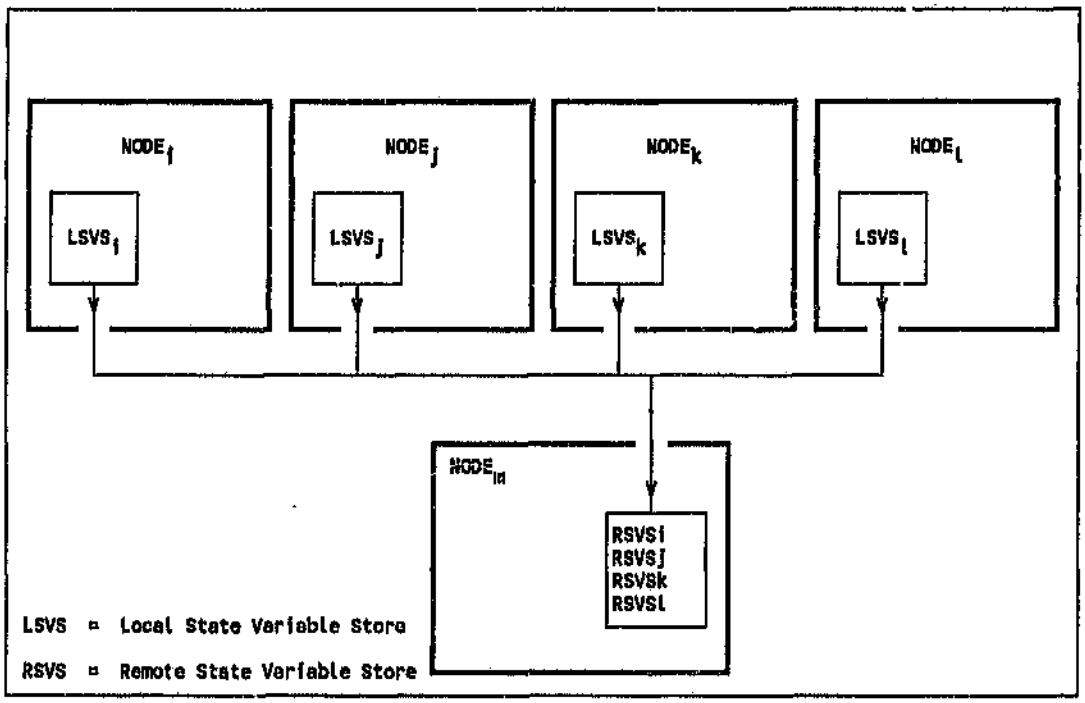


Figure 5-5: Star Single-point Topology

Multiple copies of the RSVS for each node will therefore be established in all other nodes. This has the potential of improving the reliability of the system by increasing the diversification but at the expense of CPU and communication system loading.

The reliability improvements can be considerable especially if there is a high probability of common mode failures. For example if subsections of the plant could suffer power brownouts then the likelihood of a simultaneous failure of a local and its remote becomes significant. Multiple remote nodes spread throughout the plant would protect against this.

The problems of using broadcast relate to the increase in complexity and the increase in loading. Complexity is increased because it becomes more difficult to

maintain consistency between remote nodes. Firstly a problem associated with using broadcast is determining which node should respond when a local node requires to recover its LSVS. This can be solved by increasing the complexity (and therefore the time) of the recovery protocol. Also in a broadcast protocol it is not easy to achieve positive acknowledgements for messages in the Save protocol. Thus when recovery occurs it may be from a node that has an RSVS that is not entirely correct. Again, by increasing the complexity (and thus the time) of the Save protocol this could be overcome.

Increased resource loading is also a problem in broadcast RER. As all nodes must be able to handle all state variables from all other nodes this means that they must all have far more spare CPU processing capability and memory than would be required in a more selective pairing of nodes such as in the point-to-point topology. This spare capacity is rarely available in most controllers. As the controllers are operating in a real-time situation against hard time constraints an overload of a resource such as CPU processing power can cause the controller to fail (to meet its timing requirements).

From the above it can be seen that the apparent increase in reliability offered by broadcast RER may be lost by the increase in complexity and resource loading. In practice these factors must be examined to determine if broadcast should be used.

5.3 DYNAMIC RSVS ALLOCATION

Dynamic RSVS allocation is applicable to point-to-point topologies only. It is a strategy that is introduced to cater for failures in the remote node. In the point-to-point topology described above we assumed that each node has one remote node associated with it containing its RSVS. This is the static RSVS topology and protects the local node against a certain class of failures that it may suffer as long as the remote node is operational. It is also necessary to analyze what happens when the remote node fails and to find a solution to the ensuing loss of recoverability.

In a static RER system each local node is assigned a node to hold its RSVS. If the remote node fails permanently then the RER capability of the local node is lost. Transient failures of the remote node are only a problem if the local node also fails at the same time its assigned remote has failed. If the local node does not require

to call upon the remote node for recovery exactly at the time the remote node has failed, then the RER capability will be preserved. Simultaneous failure of the local and remote nodes is discussed further in section 5.4.

If, after recovery from a transient failure, a remote node has lost its RSVS then the local node must again initialise the remote with its entire LSVS.

If the remote node suffers a permanent failure, then under the static scheme described above, the local node that had entrusted its RSVS to that node will have lost all RER capability. It is therefore desirable to consider a scheme of dynamic remote node allocation whereby the RER ability of the system is not degraded if a node fails.

The process would operate as follows: When a node detects that its remote node has failed, it proceeds to search among the remaining nodes for an eligible remote node. Having found one it will update this node with its entire LSVS and thus create a new RSVS. This allows remote nodes to be removed from operation for say maintenance or failure, while still preserving the RER capability for the local node.

A number of strategies are possible when selecting the new remote node. This selection depends on a variety of resource related factors. The selection of the new remote can either be based on a list of eligible remote nodes or selection can be done at runtime when the remote failure is detected.

a) Preferred Remote List (PRL)

A list of preferred remote nodes in a prioritised order of selection is defined in each node when the node is configured. At startup the first node in the list is established as the remote. If that node fails, the next available node on the list is chosen. When the local node fails, it must, on recovery, poll the nodes in its PRL to establish which node contains the RSVS.

The advantage of having a fixed list of possible remote nodes is that the system designer can take into account the impact on performance that can occur if a node suddenly acquires an additional RSVS to manage. This is also the only way to prevent the snowball effect (described below). The disadvantage of the PRL is that a node may end up without a remote node for its RSVS even though other nodes in the system are still operational. This can happen if all the eligible nodes on the list

are disabled, while another node not on the list could operate as a remote node.

The PRL can also be inflexible in some situations due to the fact that many control systems grow and evolve over the life-cycle of the plant. This means that when new nodes are added or existing nodes are modified or upgraded, the structure of all the PRLs may have to be revised.

b) Runtime Selection of Remotes

The alternative to the fixed PRL approach is to perform the selection at runtime. When a node detects its remote has failed it posts a request for a stable storage service to all other nodes in the system. The first node to respond positively becomes the remote node. In this approach the decision of which node becomes the remote node, rests with the remote node and not with the local node. The advantage of this lies in the fact that best use is made of the resources available at any time. It allows new nodes to be entered into the system as stable storage sites without modification to existing nodes. The disadvantage is that all nodes need to be capable of determining on a dynamic basis whether they are suitable candidates for the stable storage of the requesting node. In this case each node must contain the algorithm to determine if it is a suitable node. Furthermore this algorithm must be able to prevent the snowball effect.

c) Snowball Effect

Every RSVS allocated to a node consumes memory and CPU processing resources in that node. The resources consumed depend on such factors as the number and rate of creation of state variables. In a system with dynamic remote allocation that is undergoing a number of failures, there is a danger that all or a large number of RSVSs may migrate to one particularly reliable node. This problem is serious because nodes in an RER environment are real-time nodes operating in class II systems with event inputs. This means that strict timing constraints are imposed. In most such systems there is very little spare CPU time available and furthermore it is not easy to determine in advance with precision the impact of adding another RSVS to this node. The result of all of this is that the node becomes overloaded and cannot fulfil its original controller function.

The snowball effect must be prevented by imposing a limit on each node that is

capable of operating as a stable storage node. The first limit covers the memory allocation required for each additional RSVS. This computation is done at the time the node is requested to setup as a remote node and is defined in chapter 4, note 3. The second limitation involves the incremental load on the CPU incurred in servicing the RSVS. This calculation is based on the number of state variables and the rate of their update. The effect on CPU loading is calculated in chapter 6. The setup protocol would have to contain this information. This requires an extension to the SETUP Request Message (defined in chapter 4 note 2) as follows:

Param #	Parameter Description
1	Local Node ID
2	Setup Request ID
3	Synchronisation Time
4	RSVS Size
5	MNBS Mean No. of Bits per Save (see section 6.2)
6	MTBS Mean Time Between Saves (see section 6.2)
7	Checksum

However, even if outright overload is prevented there is still the problem of nodes having a variable load due to failures of other nodes. This can have many unpredictable effects in a control system. Because of these reasons runtime selection of remote nodes should be used with caution.

5.4 NESTED RECOVERY PATHS

In the previous section we discussed the impact of a remote node failure and how a new remote node may be selected. The assumption there was that the local node does not fail simultaneously with the remote node. If this happens in a broadcast topology, the local node will simply restore its LSVS from the first available remote node. In the point-to-point topology a special strategy must be used to cater for simultaneous failures of local and remote node. This is the technique of nested recovery discussed in this section.

In RER each local node saves its state variable data in a remote node for use on recovery. The question arises whether data stored for another node becomes part of the remotes node's LSVS. The argument whether the RSVS in a node becomes part of the LSVS of that node centres around the impact on reliability.

A remote node typically performs two basic functions. The first is that of a controller of some subsection of the plant. This function generates state variables which are kept in the LSVS. The second function is that of stable storage of an RSVS for some other node. If we want the RER process to restore both these functions of the node, both the LSVS and the RSVS being managed for some other node, must be restored after a failure. This means that they must both be saved in the remote node for this node.

If the RSVS is not part of the LSVS, it is lost when the remote node fails. If the local node had a simultaneous failure, the ability to recovery the local node using RER would be lost. The way to recover the local node in the event of simultaneous failure of local and remote nodes is to give the remote node the ability to restore its RSVS when it recovers. The justification for making the RSVS part of a node's LSVS is therefore the increased reliability it offers. In this section we look at the issues that arise when the RSVS is made part of the LSVS.

Consider the following nodes A, B and C:

A has state variables $\{a_1 .. a_i\}$
B has state variables $\{b_1 .. b_j\}$
C has state variables $\{c_1 .. c_k\}$

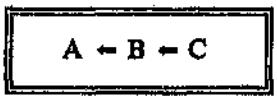
Define the symbol ← : "is a stable storage remote node for"
Then



reads:

B is a stable storage remote node for C.

C saves its LSVS (c_1 .. c_k) in B which has its own LSVS (b_1 .. b_j) and an RSVS for C. Similarly if we extend the chain as follows:



then A contains its LSVS (a_1 .. a_i) and B's RSVS (b_1 .. b_j).

If the RSVS in a node is considered part of that node's LSVS then clearly A will end with its LSVS (a_1 .. a_i) and B's RSVS (b_1 .. b_j) and C's RSVS (c_1 .. c_k). The advantage of this scheme is that if B and C fail simultaneously then on recovery B can restore its LSVS (b_1 .. b_j) and C's RSVS (c_1 .. c_k). After this C can restore its LSVS from B.

The problems that arise when considering the RSVS as part of the node's LSVS are as follows:

a) RSVS Runaway
b) Deadlock
c) Resource utilisation

a) RSVS Runaway

In a typical design every node would be allocated a remote node for stable storage of the LSVS. A scheme as follows may be used:

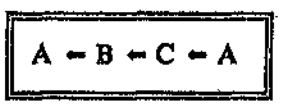


It is clear from this that unless additional intelligence is put in the management of RSVSs the size of the RSVS in each node will grow continuously each time a remote is updated with a state variable. This circulating growth will continue until failure or until some system parameter is exceeded.

When a node saves its LSVS and RSVSs to a remote node it must first check that one of the RSVSs that it contains is not the RSVS for the remote node it wishes to use for its LSVS. If this is true then the RSVS for the remote node must be removed from the items to be saved in the remote node.

b) Deadlock

In the case of nested RSVSs each node must wait for its immediate predecessor to recover its LSVS before the node can recover its RSVS from the predecessor. In the case of a loop arrangement as shown below:



and when simultaneous failure of all three nodes occur, deadlock can ensue as each node waits for its predecessor to recover its LSVS. Thus any recovery algorithm must cater for this.

This is done by means of the WAIT status and a timeout as defined in chapter 4 note 7.

c) Resource Utilisation

Even if the problems described above in a) and b) are solved by adequate design of

RER algorithm, the nesting scheme has some of the problems of the broadcast approach and the dynamic remote allocation approach described above with a variable and increased use of memory, processor and network resources. System level consideration of these constraints will determine whether this is a viable approach for an application.

5.5 CONCLUSION

In this chapter we have examined the various ways in which controllers can be put together to cooperate in an RER system. The two basic types are the single-point topology where a node has its RSVS located in just one node somewhere on the network, and broadcast topology where a node has its RSVS located in all suitable nodes on the network. We found that the broadcast topology can be potentially more reliable but at the expense of increased complexity and resource utilisation. Within the single-point topology the issues of remote node failure and particularly remote node failure simultaneously with a local node failure were examined. We have found that complex models of RER can be devised which can cope with this simultaneous failure of lo and remote node but also at the expense of increase loading and resource utilisation.

CHAPTER SIX

6. EVALUATION of REAL-TIME ERROR RECOVERY

6.1 FAULT DETECTION LATENCY

6.1.1 General

Fault latency is an important concept to consider in any fault tolerant strategy as it can have a significant bearing on the success or failure of the fault tolerance mechanism. All failure events have a latency associated with them. If the latency is too large, the problem of fault containment arises and there is a danger of erroneous information propagating to neighbouring processors or to other nodes in a distributed system [ROS82]. The duration of the latency can vary considerably from nearly instantaneous to infinite as in the case of indistinguishable faults. However, as pointed out by [LAM78], the entire concept of failure is only meaningful in the context of physical time. Without physical time there is no way to distinguish a failed process from one which is just pausing between events. Thus fault latency is always present.

A latent fault occurs when a failure event takes the computer system into a faulty state that is not measured or detected. The system continues operating with the fault present. This is shown in figure 3-3 as the secondary failure events arc. In the best case if the fault produces no incorrect outputs or behaviour of the system, it is called an indistinguishable fault by [SWE87]. An example of an indistinguishable fault is a failure in a memory cell that is not used in the current mission of the computer. In such a case the latent fault could be permanent and not cause any misbehaviour of the system. Of course any change to the computer such as expanding its functionality, or a new mission in which the cell in question is used, would cause the fault to manifest itself.

Faults whose latency is less than infinite are called distinguishable faults [SWE87]. An example of fault latency common in computer systems is found in software

faults, usually occurring as a result of an inadequate specification. A particular module may have a subtle fault in it that will not manifest until the module is executed and even then, perhaps only if the particular set of input parameters are present (see Plastic Faults in section 3.2.5). The fault latency in this case could be fairly long (software faults have been known to occur years after the system has been in operation) or it could be very short if the fault is in software used frequently such as the operating system executive.

6.1.2 Factors Determining the Latency Period

The latency period terminates upon detection of the fault. Detection can be defined as the successful isolation of the failure to its originating module (hardware or software) by a voting comparator or self-test program and the subsequent removal from usage of that module. In this study where we are looking at state restoration techniques of fault tolerance, the issue of self-test programs is very important. As can be seen from figure 3-3, if error detection does not occur, the system will continue executing secondary failure events until it ends up in the final fault state, in other words it "crashes". Speedy detection of a fault is therefore paramount for successful recovery from that fault. Self-test programs, also known as Built-in Test (BIT) software, are programs that exercise the hardware and software in a controlled fashion so as to cause latent faults to manifest themselves. Once the fault has been detected and isolated, error recovery processes can be instituted. A wide range of fault detection schemes are used such as sneak circuit analysis [MIL84], specification and symptom-based real-time analysis [FUC86], [EIF84], [MAX86] and [LIN86], as well as pattern recognition techniques [PAU74]. Analyses have also been done on some of these to determine the detection latency [BLO87] and [LIN90]. Studies such as [CHI86] have shown that where there is no BIT to detect the fault, fault latency can become very long. In this study stuck-at-0 and stuck-at-1 faults in a VAX computer running UNIX had average fault latencies of between 15 and 90 minutes.

Most self-test programs run on a cyclic basis testing for latent faults. Let this self-test cycle time be T_{BIT}. The worst case latency L for faults which are detectable by the self-test program is:

$$\max L = T_{BIT} \qquad (6.1)$$

Therefore, in order to reduce the latency period, the BIT cycle time should be as short as possible.

6.1.3 Impact of Fault Latency on RER

Fault latency effects the RER process in two ways. Firstly it increases the validity time requirements of state variables and secondly it influences the structure of the RSVS. These two factors are discussed below.

As discussed above the latency time for a fault is related to the cycle time of the self-test process. This we called T_{BIT}. In this analysis we assume that the self-test is capable of finding 100% of latent faults. This is never completely true in practice but in computer systems BIT coverage figures of over 99% are achievable.

In the earlier analysis of the validity time in section 4.5.2 we assumed that the fault latency period was zero. In this situation the worst case time for a fault to occur was a very short period, ϵ, before the end of the transfer time to the RSVS. This is shown again in figure 6-1 below. In the case where the fault latency is not zero the fault event itself will not be discovered until a time, at most T_{BIT}, later. This is a worst case because we assume that the entire self-test cycle (T_{BIT}) must be completed before the fault is found. Once the fault has been discovered, the recovery process begins. This lasts for the duration T_{RST} which is the Restore Transaction Time as defined in appendix A. From the above and based on equation (4.9) it can be seen that the new requirement on the validity time of the state variable s_i is:

$$Tv_i = Tc_i + T_{SAVE} + T_{RST} - \epsilon + T_{BIT} \qquad (6.2)$$

Thus the effect of fault latency on the validity time requirements is to extend the time by the amount T_{BIT}.

If we refer to figure 6-1 we can see that when the fault is discovered (at the end of T_{BIT}) the transfer of state variables to the RSVS is complete. Having discovered the fault it is now not possible to abort the transfer and revert to the previous RSVS as was the case for zero latency. The result is that when recovery is performed it will restore state variables from the RSVS that may have been corrupted by the fault that triggered the recovery in the first place.

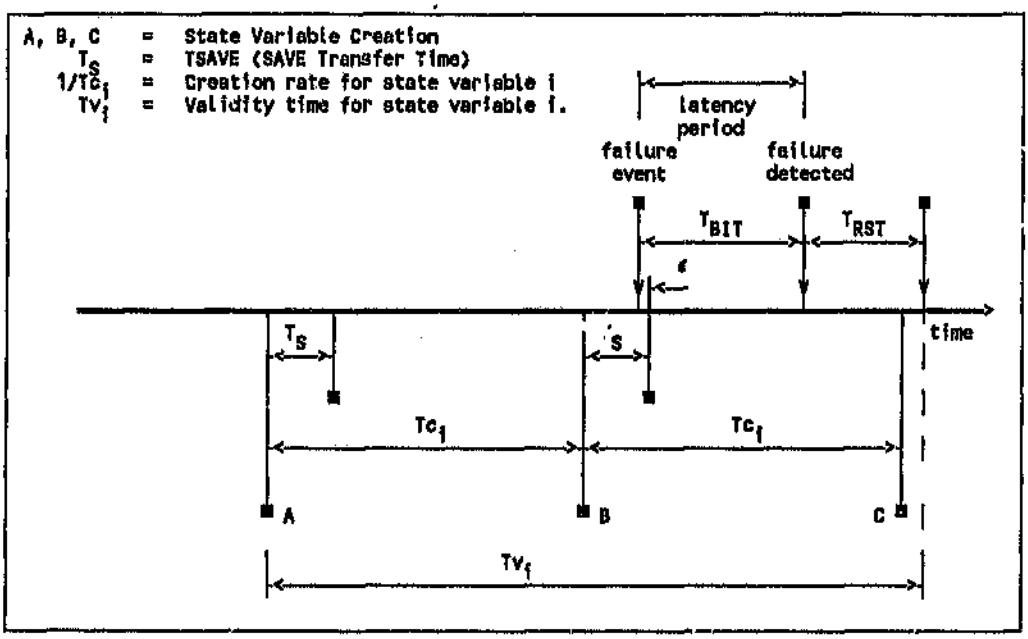


Figure 6-1: Components of Validity Time with Latency

From this it can be seen that it is necessary to implement a double buffered mechanism in the RSVS as shown in figure 6-2. The RSVS manager must perform a two-stage shift register type of function every time a new state variable arrives. The shift is on that new state variable only. When recovery takes place state variables are read only from $RSVS_b$ by the REMOTE GET mechanism.

It can be shown that the number of stages of the RSVS that is required depends on T_{BIT} and Tc_i, the creation interval of the state variables. For a two-stage RSVS the following expression holds:

$$Tc_i + T_{SAVE} + T_{BIT} - \epsilon < 2\,Tc_i \qquad (6.3)$$

hence $$T_{SAVE} + T_{BIT} - \epsilon < Tc_i$$

hence $$T_{BIT} < Tc_i - T_{SAVE} + \epsilon \qquad (6.4)$$

In other word the BIT cycle time determines how many generations of state variables must be buffered in the RSVS in order to obtain an earlier valid state variable set once the fault manifests itself at the end of the BIT cycle time.

Another way to look at this problem is that the number of RSVS stages in the

remote node that is required due to fault latency is:

$$N_{RSVS} = 2 + INT (T_{BIT} / Tc_i + \epsilon) \qquad (6.5)$$

where: INT = the integer portion of the expression

ϵ = the very short time shown in figure 6-1

This requirement to have a number of RSVSs in the remote is analogous to the requirement in backward error recovery to have a number of checkpoints depending on the fault latency as described by [THA86].

From the above analysis we see that fault latency has a serious impact on the performance of RER. It means that the validity time of the state variables must be extended to cover the latency period. It also means that the complexity of the RSVS must be increase making it an N stage shift register.

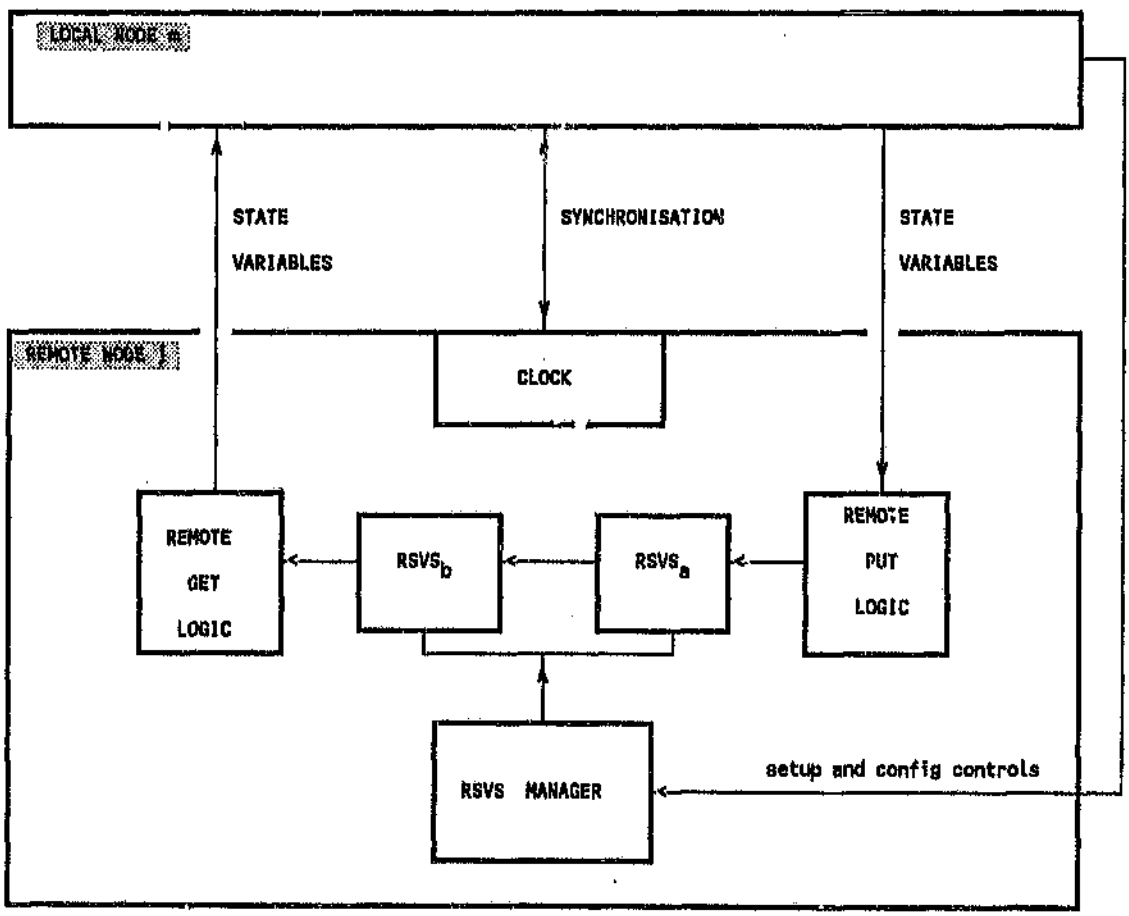


Figure 6-2: Double Buffered RSVS to Cater for Latency

6.2 CPU AND COMMUNICATION MEDIUM LOADING

6.2.1 General

This section analyses the impact of RER on the CPU load in the local and remote processors and the loading on the communication medium between these two processors.

The loading is analyzed for the SAVE phase only as this is the "steady state" phase of operation. The INITIALISATION and RESTORE phases are once-off events of short duration and the additional loading during these times are of less significance.

The timing model used for the CPU and communication medium loading is defined in appendix A. Also, throughout the analysis a relationship between the input variables and the state variables is assumed based on the following: In the processor state variables are created by combining input variables with existing state variables in a cyclic computation thus creating a new state variable. In general not all the input variables are sampled at the same rate, but the state variable computation will be performed at the rate of the variable(s) with the highest sampling rate. Call this input variable v_i. In other words v_i is the input variable, whose sampling rate determines the creation rate of the state variable s_i.

6.2.2 CPU Loading

We wish to calculate the percentage contribution to CPU loading in the local and the remote node incurred during the SAVE phase of RER. In order to do this a number of parameters must first be defined.

a) Calculation of Mean Time Between Saves (MTBS)

Consider the ith input variable v_i:

From equations (4.7) we have:

$$Tc_i = tc_i - (T_{SAVE} + T_{RST})$$

where Tc_i is the creation rate for state variable . In the RER process the state variable is saved to the RSVS every time it is created. This implies that Tc_i is also the rate for the SAVE transactions for state variable i.

For n state variables, i.e., $S = (s_1 \ldots . s_n)$ we can calculate the mean as follows:

The Mean Time Between Saves is:

$$MTBS = \frac{1}{n} \sum_{i=1}^{n} Tc_i \qquad (6.6)$$

$$= \frac{1}{n} \sum_{i=1}^{n} tc_i - (T_{SAVE} + T_{RST})$$

From this equation we can see that the key parameter affecting the MTBS is tc_i which from equation (4.4) can be seen to be based on the first derivative of the input variable. The second factor influencing MTBS is the parameters T_{SAVE} and T_{RST} which are determined by the timing factors of the RER architecture and design.

b) Definition of Mean Time To Save (MTTS)

For the ith state variable s_i we define the time for the SAVE transaction (see appendix A) as T_{SAVE}. The portion of T_{SAVE} that involves the local and remote node CPUs is :

Local node : T_{LCPU_i}
Remote node: T_{RCPU_i}

These parameters are defined in appendix A.

for n state variables, ie $S = (s_1 \ldots . s_n)$

For local node: $$MTTS_i = \frac{1}{n} \sum_{i=1}^{n} T_{LCPU} \qquad (6.7)$$

For remote node: $$MTTS_r = \frac{1}{n}\sum_{i=1}^{n} T_{RCPU} \quad (6.8)$$

c) CPU Load Calculation

The relationship between MTBS and MTTS in the RER process is shown in figure 6-3 below:

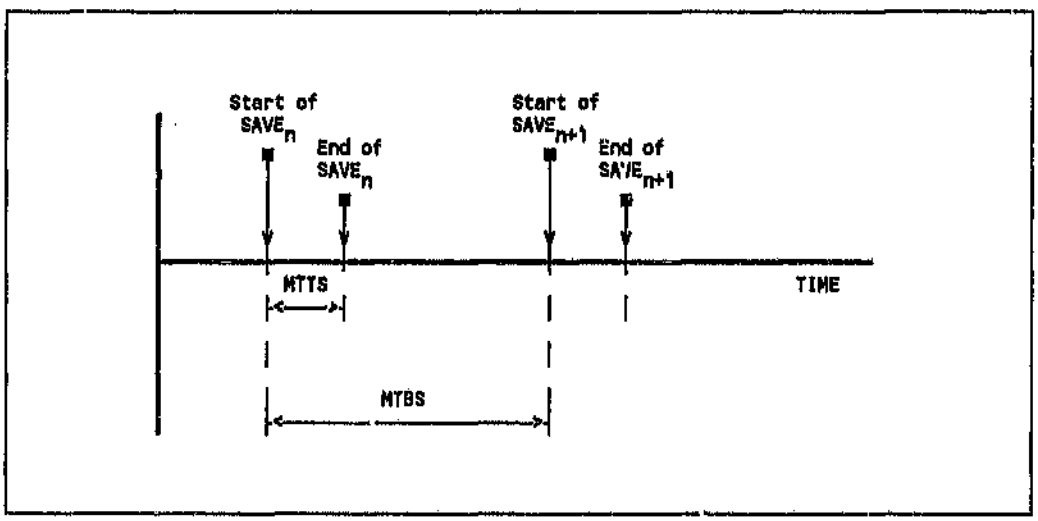


Figure 6-3: Relationship Between MTBS and MTTS

From this it can be seen that for both the local and remote nodes :

$$\text{Mean CPU load} = n \times \frac{MTTS}{MTBS} \quad (6.9)$$

$$= n \times \frac{\frac{1}{n}\sum_{i=1}^{n} T_{CPU_i}}{\frac{1}{n}\sum_{i=1}^{n} tc_i - (T_{SAVE} + T_{RST})} \quad (6.10)$$

d) Example System

Thus far in the thesis the RER strategy has been defined in generalised terms. Based on the performance and validity time equations developed in this thesis it is clear that the technique will not be suitable for all real-time applications and computer systems. This unsuitability can stem from the fact that the state variables may be computed at too high a rate for the local node to implement RER (such as an inertial navigation system), or that the number of state variables may be so large that the local node cannot perform RER (such as a large supervisory control system). Of course combinations of high rate and a large number of state variables would exacerbate the problem. However there is a large class of applications of embedded and dedicated real-time controllers performing specific control tasks in enviroments that are relatively slow changing, such as the control of electrode slip in an electric arc furnace, leak detection on oil and gas pipelines and cab display controllers in the British Columbia Railways LIC train control system. Such systems usually are ruggardised microcomputer or PLC based systems that form front ends to a larger minicomputer based supervisory system as discussed in [DEH88]. For the example in this thesis we select a microcomputer based architecture as shown in figure 6-4.

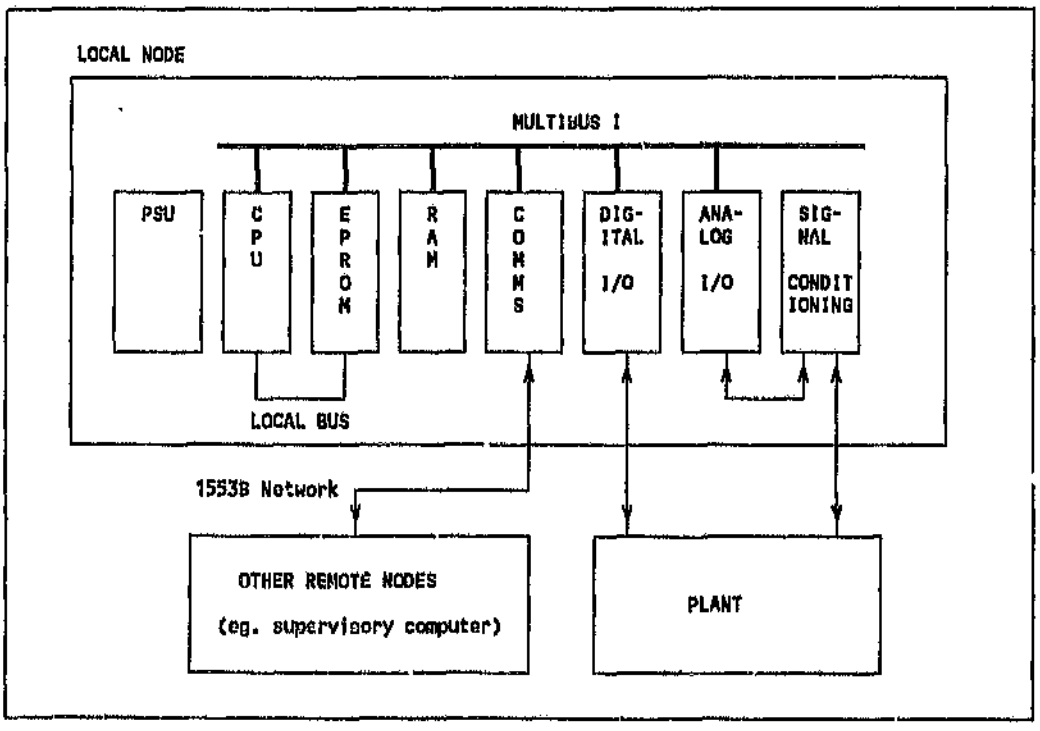


Figure 6-4: Example Local Node

The example is based on a 10 MHz Intel 80286 processor running in protected mode from EPROM using a real-time executive, VRTX286 [HUN86], and programmed in PLM and assembler. The communication medium is MIL-STD 1553B [SCI82] which is a synchronous bus running as 1 Mbit/second.

The application produces 20 state variables, each of which is updated every second. Assume the SAVE transaction time is 40 milliseconds per state variable and the RESTORE transaction time is 150 milliseconds in total for all state variables. The CPU load for the SAVE phase in the local node is 5 milliseconds per state variable and in the remote node it is 2.5 milliseconds per state variable. Calculate the mean CPU load in the local and remote nodes due to RER.

Given:

n	=	20	Number of State variables
tc_i	=	1	State variable update rate (seconds)
T_{LCPU}	=	5	Local Node CPU time to perform SAVE. (Milliseconds)
T_{RCPU}	=	2.5	Remote Node CPU time to perform SAVE. (Milliseconds)
T_{SAVE}	=	40	SAVE transaction time (Milliseconds)
T_{RST}	=	150	RESTORE transaction time (Milliseconds), made up of 100 millisecond T_{lin} and 50 milliseconds for the RESTORE transaction.

As the CPU time for the SAVE and RESTORE transactions is given as a constant for all state variables in both the local and remote nodes we have:

$MTTS_l = T_{LCPU}$
$= 5$ milliseconds

$MTTS_r = T_{RCPU}$
$= 2.5$ milliseconds

$MTBS = 1000 - (150 + 40)$
$= 810$ milliseconds

Therefore, from (6.9) Mean CPU loading

$$\text{local node:} = 20 \times \frac{5}{810}$$
$$= 0.1235$$
$$= 12.35\ \%$$

$$\text{remote node:} = 20 \times \frac{2.5}{810}$$
$$= 0.0617$$
$$= 6.17\ \%$$

These figures show that in principle, a system of the type described could implement RER without an excessive additional CPU burden. In a system designed to implement RER effort could be expended to optimise those aspects of the design that influence the RER induced CPU load. This should include optimising the restart time which in this example is given at 100 milliseconds. For a dedicated EPROM based system this restart time could readily be reduced by carefull design. Also the CPU time associated with communications can be reduced to a minimum by avoiding the use of fully layered ISO standard communication packages such as DECNET and implementing efficient low level drivers, at the expense of generality, of course. For this sort of application, synchronous communication systems such as the MIL-STD 1553B Multiplex system is well suited to high priority messages of short length.

6.2.3 Communication Medium Loading

We wish to calculate the contribution to the loading on the communication medium made by RER. For this we need to define a parameter :

the Mean Number of Bits per Save (MNBS)

Consider ith state variable s_i consisting of d_i data bits. This is saved in a Save Request Message which has a number of overhead bits = F.

Total length of Save Request Message for $s_i = F + d_i$

For n state variables, ie $S = (s_1 \ldots\ldots s_n)$, the mean number of bits per save is:

$$MNBS = F + \frac{1}{n} \sum_{i=1}^{n} d_i \qquad (6.11)$$

The communication bus can transfer data at a certain number of bits per second. Call this bus bandwidth B. We are interested in the number of bits transferable in the time between saves. This is MTBS in seconds from equation (6.6) above.

Then:

$$\text{Mean Bus Loading} \simeq n \times \frac{MNBS}{B \times MTBS} \qquad (6.12)$$

This relationship between these parameter is shown in figure 6-5 below:

Example:

We will use the same example data defined in 6.2.2 (d). The MIL-STD 1553B Multiplex system has an effective bandwidth of 800 Kbits per second. The state variables are each 4 bytes in length with an overhead of 2 bytes for the expiry time tag. There are 20 state variables created every second.

Given:

n	=	20	Number of state variables
B	=	8×10^5	Effective bus bandwidth
Tc	=	1	State variable update rate (in seconds)

From equation (6.11)

$$MNBS = 8 \times (2 + 4)$$
$$= 48 \text{ bits}$$

From equation (6.6)

$$MTBS = 1 \text{ second}$$

Therefore from equation (6.12) Mean Bus loading

$$= 20 \times \frac{48}{8 \times 10^5 \times 1}$$

$$= 12 \times 10^{-4}$$

$$= 0.12\ \%$$

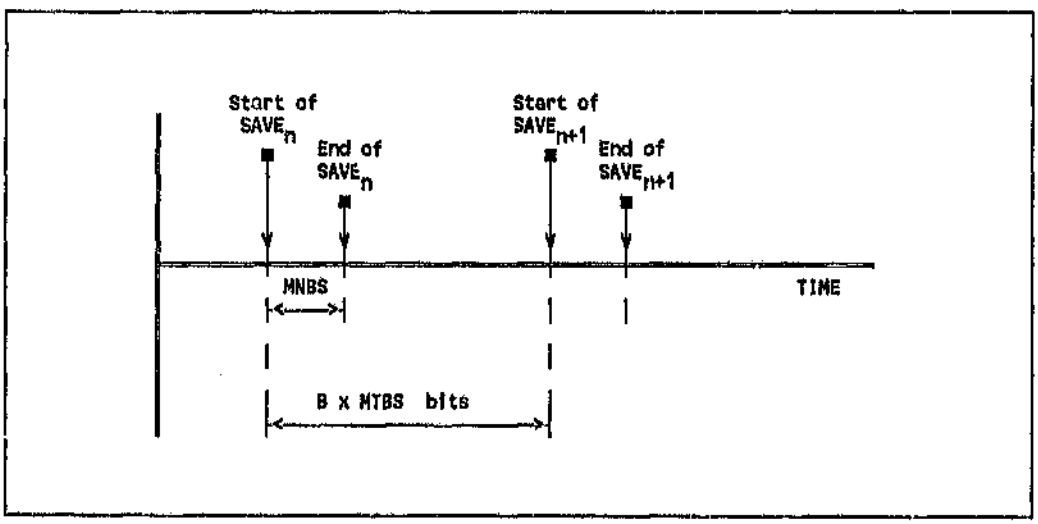


Figure 6-5: Relationship Between MNBS and B (total bandwidth)

6.3 RELIABILITY ANALYSIS

6.3.1 General

As we discussed in section 3.2.5, RER is a technique that provides recovery of the local node from transient or intermittent faults. This restriction to transient faults occurs because the state variables saved in the remote node have an associated validity time. Therefore in saying that RER protects the local node against transient faults we mean that it protects against faults for which the recovery time is less than the minimum validity time of the state variables in the remote node. If a permanent fault occurs in the local node then, by virtue of the fact that the expiry times are not infinite, the validity time of the state variables will be exceeded and no recovery will be possible. From this we see that in analyzing the effectiveness of RER to contribute to the reliability of the local node we must confine ourselves to failures of a transient nature. This is not such a significant limitation. Most computer systems required in high reliability environments go though a period of burn-in before commissioning in order to eliminate infant mortality under controlled conditions. Thus by the time a typical computer controller has been installed and is operational it is on the flat part of the familiar "bathtub" reliability curve. As semiconductor devices and software do not suffer wear-out in the normal sense, it has been realised by [SAV86] and [HESS88] that a significant number of failures during this period are soft faults or digital circuit upsets. These soft faults are amenable to repair by the RER process.

The RER architecture can be seen as providing reparability for the local node. The definition of reparability given in [MIL84] is:

> **The probability that a failed system will be restored to operable condition within a specified active repair time.**

If we now consider what happens in the RER recovery process we see that if the local node can restore its LSVS before the state variables expire, it will have effected a self-repair of a transient fault. In section 4.5 we defined an entity called T_{RST}, the Restore transaction time, as the time from when a fault is detected in the local node to the time when recovery of the LSVS has been completed. When comparing this to the reliability literature such as [MIL84] and [OCO85] we see that this is similar to the definition of the Mean Time to Repair (MTTR) of a unit. The MTTR is a statistical quantity while T_{RST} defined for the local node is not as we

assume that it is a constant for the purposes of our analysis. In this section we refer to the Restore transaction time as the Time To Repair (TTR) of the local node, where the TTR is a constant equal to the MTTR.

From the above we see that in developing the model for reliability in an RER architecture it is necessary to take into account the transient nature of the failures and the reparable nature of the local node.

6.3.2 Reliability Models

We wish to develop a general expression for the reliability of the local node in an RER architecture. The approach taken in this thesis is that we wish to give an indication of the order of magnitude of the reliability improvement that can be achieved using RER. We do this by establishing similarity to an existing model. For a more exact calculation a more sophisticated approach should be adopted such as the development of a Markov model. In this thesis we examine reliability models given in [MIL84] until we find one that is sufficiently similar to the RER architecture. Clearly the remote node provides a form of redundancy that enhances the reliability of the local node. This implies that one of the redundancy models is required. Furthermore the remote node is powered on during the operation of the local node. This fact eliminates the standby redundancy models. This leaves operative (also called active) redundancy models. We see that the normal operative redundancy model is not sufficient because in operative redundancy the secondary unit takes over operation from the failed primary unit. In RER, however, the remote node never takes over operation from the local node. It acts only as a secure repository for the state variables of the local node. Furthermore, once the primary unit has failed, it is not repaired and therefore the secondary unit must operate without failure until the end of the mission. This is shown in figure 6-6. This eliminates the operative redundancy model and we must look at more complex models. We consider now the "redundancy-with-repair" model and have the following definition from [MIL84]:

> **Redundancy-with-repair can be made to approach the upper limit of reliability (unity), contingent on the rate with which element failures can be detected and repaired or replaced. The system thus continues on operational status while its redundant elements are being repaired or replaced, as long as these repairs are completed**

before their respective redundant counterparts also fail.

This too has similarities and differences to RER. The important similarity lies in the fact that the repair must be completed before the redundant component also suffers a failure. In RER architecture this corresponds to the fact that the restore phase must be completed without the remote node failing (in the simple RER architecture without nesting or broadcast). The period for which the remote node must not fail is the restore time plus the time to send a full set of state variables to the remote node. This is one complete set of SAVE transactions for all state variables and one RESTORE transaction for all state variables. The difference between RER and redundancy-with-repair again lies in the fact that the remote node does not take over the operation of the local node. However, this fact does not influence the calculation of the reliability because the remote node must be operational while the local node has failed. We therefore conclude that the redundancy-with-repair model fits the RER architecture from the point of view of calculating the local node reliability. The timing diagram for these events is shown in figure 6-7 below.

In looking at the reliability analysis of the RER architecture there are two distinct cases that must be considered:

1) $TTR > \text{Min } Tv$ (6.13)

2) $TTR \leq \text{Min } Tv$ (6.14)

Where Min Tv is the minimum validity time of a state variable in the RSVS and TTR is the Time To Repair.

a) $TTR > \text{Min } Tv$

By definition if TTR > Min Tv, the state variables will have expired by the time recovery is complete and for S-Correct operation (see section 3.2.3) the local node will not be recoverable. In other words it will have suffered a permanent failure. As there is no recoverability there is also no reparability and hence the reliability model for the local node becomes normal standalone reliability.

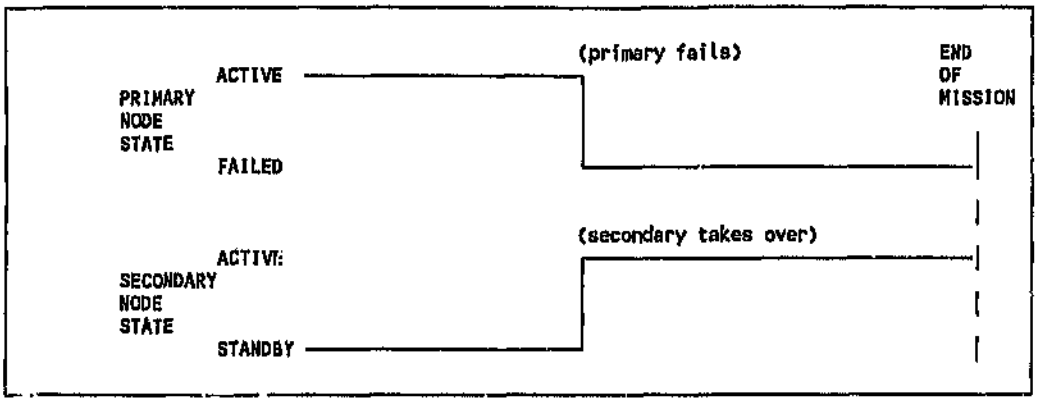


Figure 6-6: Timing Diagram for Simple Redundancy Architecture

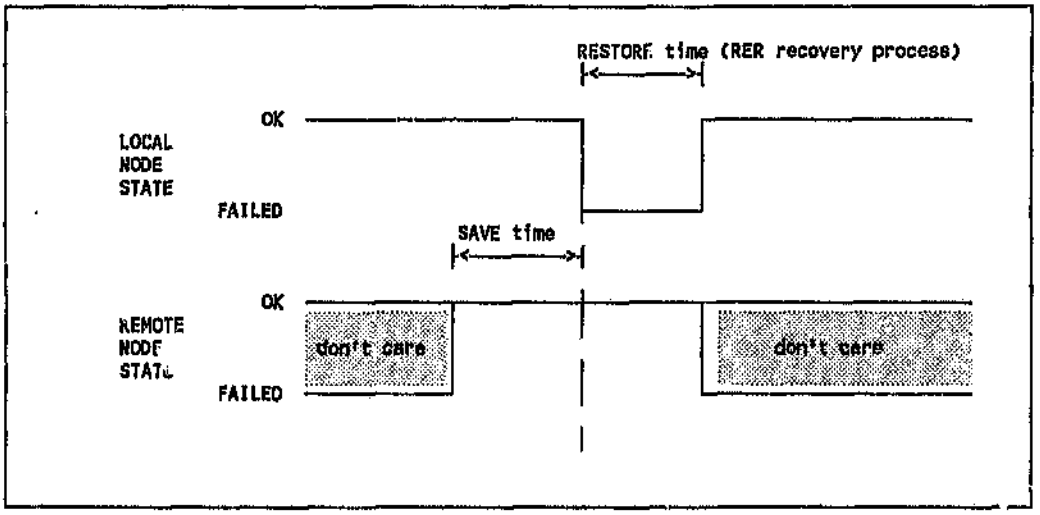


Figure 6-7: Timing Diagram for Redundancy-with-Repair Architecture

If we make the assumptions of :

i) Exponential probability density function for time to failure in the local and remote nodes.

ii) Constant failure rate, Lambda.

iii) Independent failure modes for the local and remote nodes.

then, based on [MIL84], we get the reliability equation as:

$$R(t) = e^{-\lambda t} \tag{6.15}$$

$$\text{MTBF} = \frac{1}{\lambda} \tag{6.16}$$

There are many reasons why TTR could be greater than Min Tv, for instance if the fault is permanent or if the restart of the local node is too time-consuming. In a later section we look at the factors governing the restart time.

b) TTR ≤ Min Tv

In this case the system can be seen as a redundancy-with-repair system with the local node as the primary unit and the remote node as the secondary unit. The justification for using this model has been defined above. The important criterion is that the remote node must be operational when the local node fails. Once the local node has repaired itself by restoring its LSVS operation can switch back from the remote node to the local node. Therefore the failure probability density function of the remote node is not important except in the interval of failure and repair of the local node (actually from the time of last transfer of state variables to completion of recovery). This assumes that there is no reliability penalty for the restore process.

The model for system reliability of the simple RLP architecture is derived from [MIL84] based on redundancy-with-repair with continuous monitoring. This gives the reliability function as:

$$R(t) = \frac{S_1 e^{S_2 t} - S_2 e^{S_1 t}}{S_1 - S_2} \tag{6.17}$$

where in the case of operative redundancy:

$$S_1 = -\frac{1}{2}\left[(3\lambda + \mu) + \sqrt{\mu^2 + 6\mu\lambda + \lambda^2} \right] \tag{6.18}$$

$$S_1 = -\frac{1}{2}\left[(3\lambda + \mu) - \sqrt{\mu^2 + 6\mu\lambda + \lambda^2} \right] \tag{6.19}$$

Example:

Consider a local and remote node each with MTBF of 500 hours. The TTR (i.e., the SAVE time plus the RESTORE time) is 10 seconds. Calculate the reliability after

14 days of operation for the cases where:

a) TTR > minTv

b) TTR ≤ minTv.

Case a)

$$R(t) = e^{-\lambda t}$$

$$R_a = e^{-888/500}$$

$$= 0.5106861834$$

This gives a probability of failure P_a:

$$P_a = 1 - R_a$$

$$= 0.4893138$$

Case b)

$$S_1 = -360.005999978$$

$$S_2 = -0.000000022$$

Substituting into equation (6.17) we get:

$$R_b = 0.9999926081$$

This gives a probability of failure P_b:

$$P_b = 1 - R_b$$

$$= 2.2 \times 10^{-6}$$

Comparing P_a and P_b we get :

$$P_a / P_b = 222\,415.36 \qquad (6.20)$$

Thus we see that the reduction in the probability of failure for transient failures achieved through the successful use of RER is a factor of approximately 2.2×10^5.

This is for the simple architecture of RER. If reliability enhancing features of nesting or broadcast were used, the factor would be even higher.

6.3.3 Methods for Reducing TTR

From the above discussion it is clear that for RER to have a significant impact on the reliability of a system we must have TTR < minTv. This means that TTR must be minimised. The timing model of RER gives the factors that constitute TTR as:

1) Fault detection time (fault latency)
2) Local node initialisation
3) Restore transaction time
4) LSVS restoration

1) The fault latency time has been discussed in section 6.1. Clearly the shorter the fault latency the more effective the RER process will be.

2 As can be seen from Appendix 3, local node initialisation (T_{lini}) is an important parameter in minimising TTR. Local node initialisation occurs after the fault is detected and resets the local node into an initial state by physically resetting the hardware, reloading the program and initialising the variables. This time is governed by such factors as:

   a) Hardware reset circuitry.
   b) Whether the program is stored in ROM or on a disc. ROM-based systems will be capable of much faster restarts. As described in the example in 6.2.2 (d) a typical EPROM based system can restart in the order of 100 milliseconds. The main activity in this time is initiating the tasking structure, which for real-time executive kernels such as VRTX is a few milliseconds. In disc based systems such as traditional supervisory control systems, program loading can easily take up to five minutes. This would eliminate these systems as

candidates for RER in all but the most exceptional cases where the state variables all have validity times greater than this. However, the advent of relatively new technologies such as flash EEPROMS [SAN89] makes it feasible to run large applications from non volatile memories and thus achieve fast restarts after fault detection. This has become especially true for PC based supervisory systems with the availability of low cost bulk flash EEPROM boards [WSZ90].

c) The need for startup diagnostics. These would delay the initialisation process and thus interfere with the recovery and hence should be avoided in RER designs until the LSVS has been recovered.

3) The Restore Transaction time is governed by the transfer rate of the communication medium as well as possible buffering delays in the local and remote nodes. These should be minimised and RER transactions should receive high priority, preempting other transactions waiting for transmission.

4) The process of restoring the LSVS is discussed in Note 9 of chapter 4.

When designing a system for RER the above factors must be considered, especially if they contribute to delays that are of the same order of magnitude as the validity times of the state variables.

6.3.4 Conclusion

From the above analysis it can be seen that RER improves the reliability of the local node by a factor of at least 10^5 when considering failures of a transient nature if TTR < minTv. By minimising the repair time of the local node we broaden the set of transient failures that the local node can be protected against. Protection of the local node against permanent failures would require online replacement of permanently broken components of the local node. This is possible in principle and in section 6.5 we consider the usefulness of RER in online maintenance.

6.4 A-CORRECT OPERATION

We have shown in earlier discussion that RER can provide an extremely good enhancement of reliability of the local node if the restore process completes successfully before the expiry of the state variables in the RSVS. We have also seen that there is a very sharp cutoff in reliability if the expiry time is exceeded. This ratio can be seen quantitatively from equation 6.20. In this section we wish to analyze whether this all-or-nothing reliability improvement is in fact necessary. We wish to analyze whether there is not perhaps a gradient of degradation that can be achieved between strictly correct restoration (s-correct) and failed restoration. This is the state of acceptable correctness (a-correctness) defined in chapter 3.2.3.

Most of the input variables we deal with in an RER controller are continuous functions. Based on these continuous function variables we can calculate a sampling frequency that is a function of the first derivative of the input variable as shown in equation 4.5 and also a function of the RER process parameters as shown in figure 4.8. As defined in section 4.5.1 the state variables are some combination of these input variables. As the input variables are continuous functions we conclude that the state variables themselves are also continuous functions.

If a state variable is a slowly varying function in time and if it is restored just outside of its expiry time, the restored value will be incorrect by some small amount. This amount is a function of the rate of change of the state variable and the delay from the expiry time to the current local clock time.

For example:

State variable s_i is restored at local time T_l^k such that $T_l^k > TE_l$. Also the first derivative for s_i at T_l^k is ds_i/dt. From this we can deduce that the correct value of si at time T_l^k is :

$$s_i = s_i + (T_l^k - TE_l) \frac{ds_i}{dt} \tag{6.21}$$

From this we see that if we have the first derivative of the state variable available at the time of restoration then, by simple extrapolation, the state variable can be restored to an acceptably correct value. If the rate of change of the first derivative

(i.e., the second derivative) is more slowly varying than the first derivative, the calculated value for the first derivative will be valid for longer that the value of the state variable itself. Thus, to extend the validity time, first derivatives are calculated for state variables with these characteristics and made a part of the LSVS. The restored first derivatives are then utilised to extrapolate state variables if the state variables themselves have expired. It is possible, although probably not necessary, to have a second derivative of the state variable to restore the first derivative. In practice it is possible to use the expired first derivative to obtain an a-correct state variable if the expiry time has not been exceeded by too much.

Thus we can see that by adding further redundancy into the system by means of additional state variables (first derivatives of state variables), we can extend the expiry time of the state variables and introduce a graceful degradation in the acceptability of the restored state variables.

6.5 DEALING WITH PERMANENT FAULTS USING RER

In our discussion to date we have stressed that RER is designed to cater for transient faults only. The problem with dealing with permanent faults relates to the fact that the state variables stored in the RSVS have an expiry time. As a permanent fault persists indefinitely, the state variables in the RSVS will expire while the local node is still faulty and thus recovery will fail.

Dealing with permanent faults in a system usually involves removal of the faulty component from the controller and the subsequent replacement with a new working component or perhaps with the repaired original component. In general, this replacement process is manual and takes a considerable amount of time, especially if logistical time to get spare components from a store is included.

However if we postulate a situation in which the faulty component can be replaced by a spare component within the expiry times of the state variables in the RSVS then there is no reason in principle why RER cannot be used in the correction of permanent faults. This requires the concept of online replacement of spares and such systems have been designed and implemented such as FTMP [HOP78] and Bell's ESS processors [TOY78].

This illustrates the fact that RER could be used effectively in conjunction with a reconfiguration/replacement strategy in order to bring the new replacement component into a state necessary to resume control. This in effect means restoring its LSVS. An important difference is that a component replaced in this manner must not implement a cold startup where, in accordance with figure 4-4, it initiates the RER INITIALISATION phase, but must implement the RER RESTORE phase and thus retrieve the LSVS of its predecessor.

Consider figure 6-8 as an example. This figure shows the combination of a passive standby redundancy architecture with the RER architecture. If the active local node fails permanently then the switch is thrown and the standby node performs the RESTORE phase and assumes valid control.

We have shown that RER is a usable process when used together with replacement/reconfiguration strategies for catering for permanent failures during operation. In the same way, in principle, preventative maintenance can also be implemented using RER. The local node may be shut down for preventative

maintenance and restarted again with the correct state. Again this is possible if the maintenance period is less than the expiry time of the state variables in the RSVS.

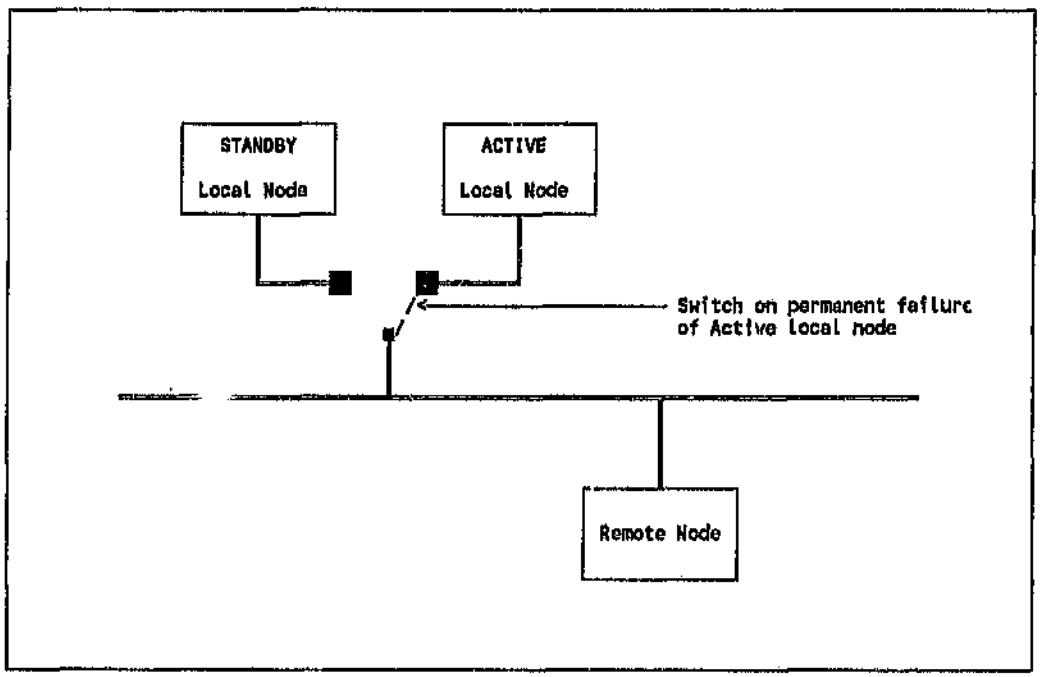


Figure 6-8: RER used with Standby Redundancy Architecture

CHAPTER SEVEN

7. CONCLUSION

In this thesis we set out to explore the concept of protecting real-time process control computers against soft or transient errors without the use of specialised hardware. The objectives were to reduce the problem of startup in systems containing a history state to the process of restart, where all the information necessary to resume control in a valid state was obtained from the environment.

Based on the analysis performed in chapter 2 we concluded that the type of error recovery process we were looking for needed to cope with systems that contained state variables, and in which timing was critical because the environment could evolve to new states while the controller was not functioning.

Real-time Error Recovery has been proposed as a recovery technique to deal with transient or soft faults occurring in systems requiring complex error recovery. We defined in chapter 2 and again in chapter 4 that one of the central obstacles to simple error recovery is the presence of state variables in the controller and argued that if the controller were stateless, the problem of error recovery reduces to the matter of performing a restart from a predetermined state, i.e., a reset of the controller. We extended this line of reasoning in chapter 4, proposing that it is possible to export the history state of a "statefull" controller in real-time to a stable storage node such that when recovery is required this data can be restored to the original failing node within the validity time constraints of the state variables. Thus, when we examine the behaviour of a RER controller recovering from a transient failure, we see that the process involves reading state variables from a remote node, reading the current inputs from the environment and continuing its normal control operation. What we have achieved is a situation where all the information necessary for the controller to begin operation in a valid state (a state relevant to the state of the environment) is now available to the controller from its environment. This is precisely the situation of simple error recovery we have been striving towards. The only limitation is that the information saved in the

controller's environment (i.e., in the RSVS) will not remain valid indefinitely.

What this process in fact produces is a "soft error filter", which in effect filters out transient failures of duration less than the minimum validity time of the state variables. The validity time itself is determined by the nature of the process under control. The remote nodes acts as a state variable "capacitor". This concept of a low pass transient filter as a model for Real-time Error Recovery was developed in chapter 4.

An important aspect of the proposed Real-time Error Recovery is the architecture of the stable storage node. We have shown that a common architecture in the sort of environments we are interested in is a local area network interconnecting a number of loosely coupled autonomous nodes. We have utilised this architecture by making it possible for each one of the autonomous nodes to act as a stable storage medium for the other nodes. In order to meet our original goal of not significantly loading the resources of the controllers specifically for failure recovery, the remote nodes are required to provide only an area of memory for the storage of state variables in the form of the RSVS. This memory utilisation will not normally be large in terms of today's memory capacities. For example, a controller with 50 state variables each of which is 4 bytes in length would require an RSVS of only 200 bytes in the remote node. The steady state CPU loading in the remote node can be reduced to one message transaction per state variable at the sampling rate of the state variable. The loading on the remote node is therefore related to the storage and retrieval of data and does not require the remote execution of procedures as in other proposals such as [AGR88] and [LOQ86].

The impact of the reliability of any local node was shown to be very high, of the order of 10^5 improvement over a standalone node not implementing Real-time Error Recovery. This is achieved by the fact that RER provides a form of real-time repair of the local node. The component of the local node being repaired is the Local State Variable Store. In other words it is a repair of data or information. Thus in a node not implementing RER which suffers a transient failure it may be possible to consider the hardware and fixed programs as self-repaired when the transient failure has disappeared, but the node itself will have suffered a permanent failure as the history data will be lost. RER transforms the permanent damage into a repairable condition by providing for restoration of the history data in the LSVS. The local node therefore implements self-repair in real-time before the validity of the state variables expire thus making a transient failure transparent to the

controlled process.

At a number of places in the thesis, while describing the RER algorithm, choices arose as to which way the implementation could take place. Examples include the architecture of the nodes in chapter 5, whether the save transaction needs to be bidirectional or unidirectional and dependent versus independent restoration of the LSVS, etc. Future research into these areas could fruitfully be done by development of a prototype network implementing the RER process. The objective would be to evaluate and quantify some of the design options discussed in this thesis for the purpose of developing design rules for the implementation of RER.

Finally, in assessing the significance of the RER process we see that it is an effective mechanism for dealing with transient faults which are a class of faults of growing importance as circuit sizes diminish. The process provides efficient recovery of transient faults by identifying the minimum set of components that require redundant storage. These are the state variables containing the history state of the processor. The limitation of the technique which is discussed extensively in the study centres around the fact that the validity of the state variables will expire in a finite time. This time determines the maximum duration of the transient fault that can be coped with using this technique.

APPENDIX A

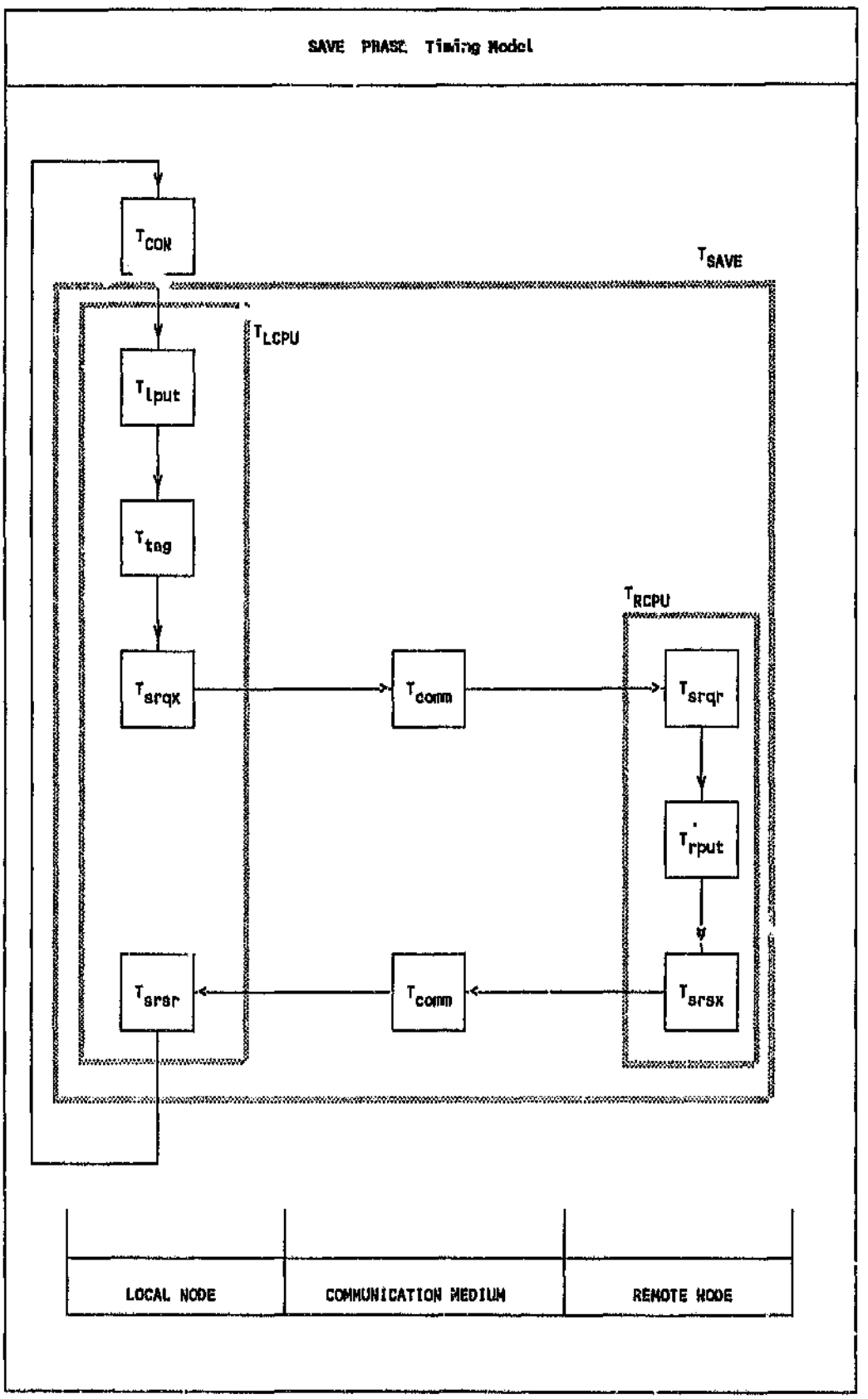
A. RER Timing Model

A.1 INTR)DUCTION

This appendix contains the timing model for the RER process. Only the SAVE and RESTORE phases are covered as they are the only timing critical phases. The SAVE phase is cyclic while the RESTORE phase is a once-off event that occurs when recovery from a failure is required.

A.2 SAVE Phase Timing Model

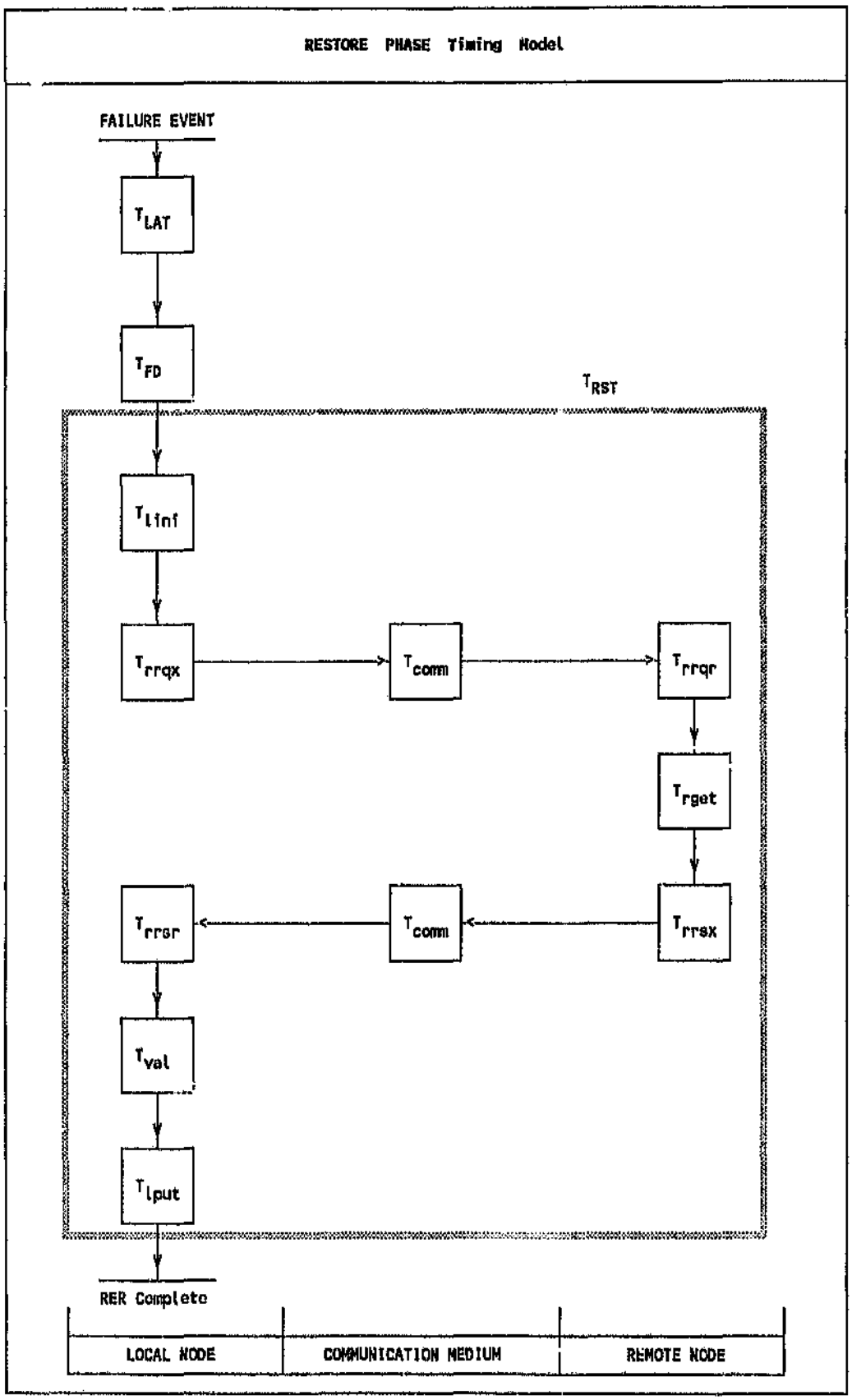
This section contains the timing diagram which shows the sequencing of the events during the SAVE Phase and in which element of the generic RER architecture the events occur. The table that follows gives a definition of each of the mnemonics used in the timing diagram and defines the steps that make up the SAVE transaction.


SAVE PHASE Timing Model
T_{CON}
T_{SAVE}
T_{LCPU}
T_{lput}
T_{tag}
T_{RCPU}
T_{srqx}
T_{comm}
T_{srqr}
T_{rput}
T_{srsr}
T_{comm}
T_{srsx}
LOCAL NODE
COMMUNICATION MEDIUM
REMOTE NODE

PHASE	ACTION	TIMING MNEMONIC
S , E	SAVE Transaction	T_{SAVE}
	Put state variable into LSVS. Local PUT function.	T_{lput}
	Tag s_i with offset into RSVS and length of data string.	T_{tag}
	Transmit SAVE Request message to remote node.	T_{srqx}
	Physical transmission on LAN.	T_{comm}
	Receive SAVE Request message from local node.	T_{srqr}
	Store into RSVS. Remote PUT function.	T_{rput}
	Transmit SAVE response message to local node .	T_{srsx}
	Physical transmission on LAN.	T_{comm}
	Receive SAVE response message from remote node. Check status and terminate SAVE transaction.	T_{srsr}
CPU UTILISATION SUBPHASE	Local node CPU utilisation for SAVE transaction.	T_{LCPU}
	Remote node CPU utilisation for SAVE transaction.	T_{RCPU}

A.3 RESTORE Phase Timing Model

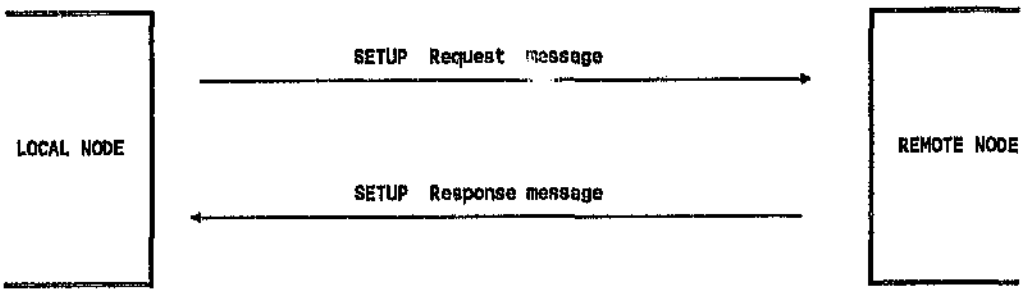
This section contains the timing diagram which shows the sequencing of the events during the RESTORE Phase and in which element of the generic RER architecture the events occur. The table that follows gives a definition of each of the mnemonics used in the timing diagram and defines the steps that make up the RESTORE transaction.


RESTORE PHASE Timing Model
FAILURE EVENT
T_{LAT}
T_{FD}
T_{RST}
T_{Linf}
T_{rrqx}
T_{comm}
T_{rrqr}
T_{rget}
T_{rrsr}
T_{comm}
T_{rrsx}
T_{val}
T_{lput}
RER Complete
LOCAL NODE
COMMUNICATION MEDIUM
REMOTE NODE

PHASE	ACTION	TIMING MNEMONIC
RESTORE	Restore Transaction	T_{RST}
	Fault latency period	T_{LAT}
	Fault Duration	T_{FD}
	Initialise the local node.	T_{lini}
	Transmit RESTORE Request message to remote node.	T_{rrqx}
	Physical transmission on LAN.	T_{comm}
	Receive RESTORE Request message from local node.	T_{rrqr}
	Get state variables from RSVS. Remote GET function.	T_{rget}
	Transmit RESTORE Response message to local node.	T_{rrsx}
	Physical transmission on LAN.	T_{comm}
	Receive RESPONSE and restore S_i to LSVS.	T_{rrsr}
	Validate the state variables against their expiry times.	T_{val}
	Store the state variables into the LSVS.	T_{lput}

A.4 RER Transaction Summary

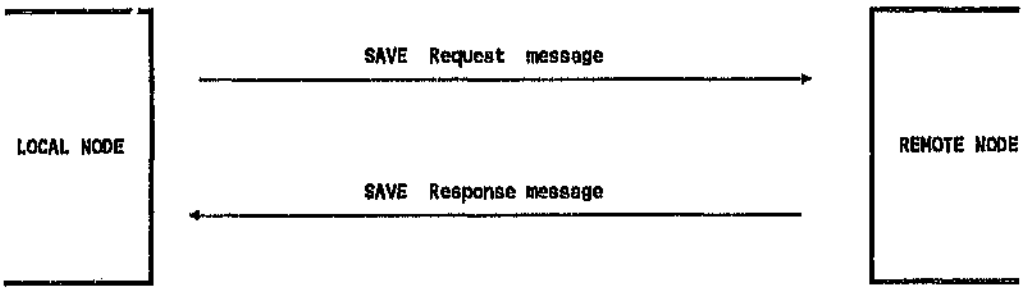
A.4.1 SETUP Transaction



SETUP Request Message	
Param #	Parameter Description
1	Local Node ID
2	Setup Request ID
3	Synchronisation Time
4	RSVS Size
5	Checksum

SETUP Response Message	
Param #	Parameter Description
1	Remote Node ID
2	Setup Response ID
3	Status (OK / NOTOK)
4	Checksum

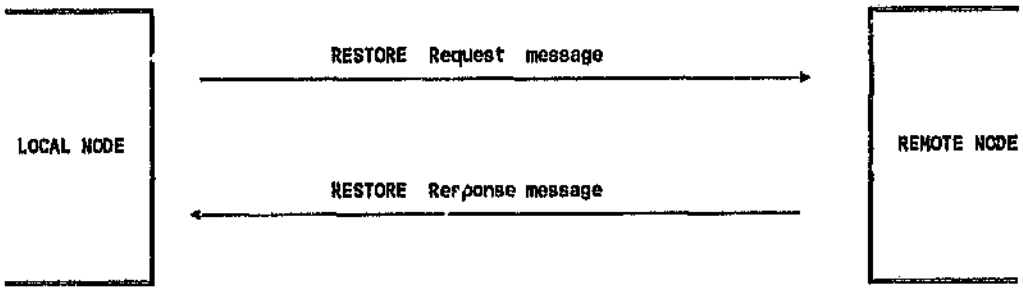
A.4 ? SAVE Transaction



SAVE Request Message	
Param #	Parameter Description
1	Local Node ID
2	Save Request ID
3	O_i , b_i
4	State variable i
5	TE_i
6	Checksum

SAVE Response Message	
Param #	Parameter Description
1	Remote Node ID
2	Local Node Id
3	Save Request ID
4	Status (OK / NOTOK)
5	Checksum

A.4.3 RESTORE Transaction



RESTORE Request Message	
P[illegible]	[illegible]arameter [illegible]escription
[illegible]	[illegible]l Node ID
[illegible]	RESTORE Request ID
3	RSVS size
4	Sequence Number
5	Checksum

RESTORE Response Message	
Param #	Parameter Description
1	Remote Node ID
2	Local Node Id
3	RESTORE Response ID
4	Status (OK / NOTOK)
5	Current Time in Remote node
6	S_i and TE_i i = 1 to n
7	Checksum

REFERENCES

[AGR88] Agrawal, P. Fault Tolerance in Multiprocessor Systems without Dedicated Redundancy. IEEE Transactions on Computers. Vol 37, No. 3, March 1988.

[AND81] Anderson, T., P.A. Lee. Fault Tolerance: Principles and Practices. Prentice Hall 1981.

[AND82] Anderson, T,. P.A. Lee. Fault Tolerance Terminology Proposals. The Twelfth International Symposium on Fault Tolerant Computing (FTCS 12) 1982. Pages 29 to 33.

[AVI78] Avizienis, A. Fault Tolerance: The Survival Attribute of Digital systems. Proceedings of the IEEE. Vol. 66, No. 10. October 1978.

[AVI82] Avizienis, A. The Four-Universe Information System Model for the Study of Fault Tolerance. The Twelfth International Symposium on Fault Tolerant Computing (FTCS 12) 1982. Pages 6 to 13.

[AVI86] Avizienis, A. Dependable Computing: From Concept to Design Diversity. IEEE Proceedings. Vol 74, No. 5, May 1986.

[AVI88] Avizienis, A., M.R. Luy. On The Effectiveness of Multiversion Software in Digital Avionics. AIAA/IEEE 8th Digital Avionics Systems Conference. Oct 1888.

[BLO87] Blough, D.M., G.M. Masson. Performance Analysis of a Generalised Upset Detection Procedure. The Seventeenth International Symposium on Fault Tolerant Computing (FTCS-17). 1987.

[BLO88] Bloch, G., I.M. MacLeod. An Analysis of Error Recovery Problems in Distributed Computer Control Systems. Proc. 8th. IFAC Workshop on Distributed Computer Control Systems. Vitznau, 1988.

[CHI86] Chillarege, R., R.K. Iyer. Fault-Latency in the Memory - An Experimental Study in VAX 11/780. The Sixteenth International Symposium on Fault Tolerant Computing (FTCS-16). 1986.

[DEH88] Dehning, R.W., D.P. Weeks. Some Aspects of Networking Needed for an Integrated Distributed Supervisory Control System. 5th IFAC/IFIP Symposium on Software for Computer Control. SOCOCO 88, Johannesburg, South Africa, 26-28 April 1988.

[DZW88] Dzwonczyk, M., H. Stone. A Fault Tolerant Avionic Suite of an Entry Research Vehicle. AIAA/IEEE 8 th Digital Avionics Systems Conference. (DASC88). October 1988.

[EIF86] Eifert, T.B., J.P. Shen. Processor Monitoring Using Asynchronous Signatured Instruction Streams. The Fourteenth International Symposium on Fault Tolerant Computing (FTCS-14). 1984.

[FUC86] Fuchs, W.K. A Specification Based Approach to Concurrent Structure Verification in Multiprocessor Systems. Proceedings, IEEE. International Conference on Computer Design: VLSI in Computers. ICCD'86. October 1986.

[GAI88] Gaitanis, N. The Design of Totally Self-Checking TMR Fault-Tolerant Systems. IEEE Transactions on Computers. Vol. 37, No. 11. November 1988.

[GAR81] Garman, J.R. The "Bug" Heard Around the World. ACM SIGSOFT. Software Engineering Notes. Vol. 6, No. 5, October 1981.

[GEL79] Gelenbe, E. On the Optimal Checkpoint Interval. Journal of the ACM. Vol. 26, No. 2, April 1979.

[HEC76] Hecht, H. Fault Tolerant Software for Real-Time Applications. ACM Computing Surveys. Vol 8. No. 4. December 1976

[HES88] Hess, R.F, L.J. Yount, et al. Sharing The Protection of Aircraft Electronic Systems Against the Effects of High-Level Electromagnetic Environments Between Traditional Protection and System Architecture. AIAA/IEEE 8 th Digital Avionics Systems Conference. (DASC88). October 1988.

[HIL88] Hills, A.D., N.A.Mirza. Fault Tolerant Avionics. AIAA/IEEE 8 th Digital Avionics Systems Conference. (DASC88). October 1988.

[HOP78] Hopkins, A.L., T.B. Smith, J.H. Lala. FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. Proceedings of the IEEE. Vol 66, No. 10. October 1978.

[HOP80] Hopkins, A.L. Fault-Tolerant Systems Design: Broad Brush and Fine Print. IEEE Computer. March 1980.

[HUN86] Hunter & Ready, Inc. VRTX: Timing Summary for VRTX/286. Document No. HRI MC02220. 1986

[HUN87] Hunt, D.B., P.N. Marinos. A General Purpose Cache Aided Rollback Error Recovery (CARER) Technique. The Seventeenth International Symposium on Fault Tolerant Computing (FTCS-17). 1987.

[IHA78] Ihara, H., et al. Fault-Tolerant Computer System with Three Symmetric Computers. IEEE Proceedings. Vol. 66, No. 10, October 1978.

[IHA84] Ihara, H., K. Moria. Autonomous Decentralised Computer Control Systems. IEEE Computer. Vol. 17, No. 8, August 1984.

[JOH89] Johnson, B.W. Design and Analysis of Fault-Tolerant Digital Systems. Addison-Wesley. 1989.

[KNI86] Knight, J.C., N.G. Levenson. An Empirical Study of Failure Probabilities in Multi Version Software. The Sixteenth International Symposium on Fault Tolerant Computing (FTCS-16). 1986.

[KOH81] Kohler, W.H. A Survey of Techniques for Synchronization and

Recovery in Decentralized Computer Systems. ACM Computing Surveys. Vol. 13, No. 2, June 1981.

[KOO87] Koo, R., S. Toueg. Checkpointing and Rollback Recovery for Distributed System. IEEE Transactions on Software Engineering. Vol. SE-13, No. 1. January 1987.

[KOP80] Kopetz, H. Future Requirements and Trends - Reliability and Error Recovery. Real-time Data Handling and Process Control. North Holland Publishing Co. 1980.

[KOP82] Kopetz, H. The Failure Fault (FF) Model. The Twelfth International Symposium on Fault Tolerant Computing (FTCS-12). 1982. Pages 14 to 17.

[KOP87] Kopetz, H., Ochsenreiter, W. Clock Synchronization in Distributed Real-Time Systems. IEEE Transactions on Computers. Vol C-36 Number 8. August 1987.

[KOP88] Kopetz, H., Kim, K. Consistency Constraints in Distributed Real Time Systems. Proc. 8th. IFAC Workshop on Distributed Computer Control Systems. Vitznau, 1988.

[KUH86] Kuhl, J.G., S.M. Reddy. Fault Tolerance Considerations in Large Multiple Processor Systems. IEEE Computer. March 1986.

[LAM78] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM. Vol. 21, No. 7, July 1979.

[LAM82] Lamport, L., R. Shostak, M. Pease. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems. Vol. 4, No. 3. July 1982.

[LAP82] Laprie, J.C., A. Costas. Dependability: A Unifying Concept for Reliable Computing. The Twelfth International Symposium on Fault Tolerant Computing. (FTCS-12). 1982.

[LEE82] Lee, P.A., D.E. Morgan. Fundamental Concepts of Fault Tolerant Computing. The Twelfth International Symposium on Fault Tolerant Computing. (FTCS-12). Pages 34 to 38. 1982.

[LEV83] Levenson, N.G. Software Fault Tolerance: The Case for Forward Error Recovery. 4th AIAA Conference of Computers in Aerospace. 1983. Hartford.

[LEV86] Levenson, N.G. Software Safety: Why, What, and How. ACM Computing Surveys, Vol. 18, No. 2, June 1986.

[LIN86] Lin, T.Y., D.P. Siewiorek. Towards On-line Diagnosis of Trend Analysis. Proceedings, IEEE. International Conference on Computer Design: VLSI in Computers. ICCD'86. October 1986.

[LIN90] Lin, T, K.G. Shin. Location of a Faulty Module in a Computing System. IEEE Transactions on Computers. Vol. 39, No. 2, February 1990.

[LIT87] Littlewood, B., D.R. Miller. A Conceptual Model for Multi-Version Software. The Seventeenth International Symposium on Fault Tolerant Computing (FTCS-17). 1987.

[LIT89] Littlewood, B., D.R. Miller. Conceptual Modelling of Coincident Failures in Multiversion Software. IEEE Transactions on Software Engineering. Vol SE-15, No. 12. December 1989.

[LOM81] Lombardo, T.G., N. Mokhoff. Shuttle Firsts Put to the Test. IEEE Spectrum. Vol 18 No. 8. August 1981.

[LOQ86] Loques, O.G., J. Kramer. Flexible Fault Tolerance for Distributed Computer Systems. IEE Proceedings. Vol 133, Pt 5, No. 6. November 1986.

[MAC83] MacLeod, I.M. A Study of Issues Relating to Real-Time in Distributed Computer Systems. Phd Thesis, Department of Electrical Engineering, University of the Witwatersrand, 1983.

[MAC88] MacLeod, I.M. Reliability and Availability Enhancement Through Active Redundancy in Distributed Computer Control Systems. 7 December 1988.

[MAX86] Maxion, R. A. Distributed Diagnostic Performance. Proceedings, IEEE. International Conference on Computer Design: VLSI in Computers. ICCD'86. October 1986.

[MCG88] McGough, J., D. Mulcare, W.E. Larsen. A Method of Measuring Fault Latency in a Digital Flight Control System. AIAA/IEEE 8 th Digital Avionics Systems Conference. (DASC8). October 1988.

[MIL84] MIL-HDBK-883 Military Handbook. Electronic Reliability Design Handbook. US DoD. 15 October 1984.

[MIL85] Mili, A. Towards a Theory of Forward Error Recovery. IEEE Transactions on Software Engineering. Vol. SE 11 no 8. August 1985

[MOR87] Morris, M.A. The Development of a Software Fault Tolerant System for a Real Time Control Environment. Department of Computer Science, UNISA. 1987.

[OCO85] O'Conner, P.D.T. Practical Reliability Engineering. Second Edition. John Wiley & Sons.

[PAR85] Parnas, D.L. Software Aspects of Strategic Defence Systems. Communications of the ACM. Vol. 28, No. 12, December 1985.

[PAU74] Pau, L.F. Applications of Pattern Recognition to the Diagnosis of Equipment Failure. Pattern Recognition. Vol 6. Pergamon Press. 1974.

[RAN78] Randell, B. Reliable Computing Systems. Operating Systems, An Advanced Course. Springer Verlag. 1978.

[RAN78a] Randell, B., P.A. Lee, P.C. Treleaven. Reliability Issues in Computing System Design. ACM Computing Surveys, Vol 10. No. 2. June 1978.

[RAN83] Rancy, L.H. The Use of Fault-Tolerant Software for Flight Control Systems. Digital Avionic Systems, IEEE/AIAA, Conference '83. October 31 - November 3 1983.

[ROB82] Robinson, A.S. A User Oriented Perspective of Fault Tolerant System Models and Terminologies. The Twelfth International Symposium on Fault Tolerant Computing (FTCS 12) 1982.

[ROD88] Rodd, M.G., K.D. Muller. Real Time Distributed Computer Control Systems - A Realistic Perspective. Proc. 8th. IFAC Workshop on Distributed Computer Control Systems. Vitznau, 1988.

[ROS82] Rose, C.W. On the Contribution of Operating Systems to Reliability and Safety in Real-Time Systems. IFAC International Workshop on Safety and Reliability of Industrial Computer Systems. SAFECOMP'82. 1982.

[RUB83] Rubertus, D.P. Self Repairing Flight Control Systems Overview. Digital Avionic Systems, IEEE/AIAA, Conference '83. October 31 - November 3 1983.

[SAN89] Santo, B. Technology '89 - Solid State. IEEE Spectrum. Vol. 26, No. 1. January 1989.

[SAV86] Savaria, Y., N.C. Rumin, et al. Soft-Error Filtering: A Solution to the Reliability Problem of Future VLSI Digital Circuits. Proceedings of the IEEE. Vol. 74, No. 5, May 1986.

[SCI82] SCI Systems, Inc. MIL-STD 1553B. Mulitplex Applications Handbook. SCI Systems Inc. 1982.

[SHE82] Shepherd, J. A Method of Designing Fault Tolerant Software. Certification of Avionic Systems. Royal Aeronautical Society. 27 April 1982.

[SHI87] Shin, K.G. Special Issues on Real-Time Systems. IEEE Transactions on Computers. Vol C-36. Number 8. August 1987.

[STA84] Stallings, W. Local Networks. ACM Computing Surveys. Vol 16, No. 1. March 1984.

[SWE87] Swern, L., Bavuso, S.J., Martensen, A.L., Miner, P.S. The Effects of Latent Faults on Highly Reliable Computer Systems. IEEE Transactions on Computers, Vol C-36, No. 8. August 1987.

[TAY80] Taylor, D.J., D.E. Morgan, J.P. Black. Redundancy in Data Structures: Improving Software Fault Tolerance. IEEE Transactions on Software Engineering. Vol SE-6 No. 6. November 1980.

[THA86] Thanawastien, S., R.S. Pamula, Y.L. Varol. Evaluation of Global Checkpoint Rollback Strategies for Error Recovery in Concurrent Processing Systems. The Sixteenth International Symposium on Fault Tolerant Computing (FTCS 16) 1986.

[TOY78] Toy, W.N. Fault-Tolerant Design of Local ESS Processors. Proceedings of the IEEE. Vol. 66, No. 10. October 1978.

[TSO88] Tso, K.S., A. Avizienis. Community Error Recovery in N-Version Software: A Design Study with Experimentation. The Seventeenth International Symposium on Fault Tolerant Compu. g (FTCS-17). 1987.

[VER78] Verhofstad, J.S.M. Recovery Techniques for Database Systems. ACM Computing Surveys, Vol 10, No. 2. 2 June 1978.

[WEN78] Wensley, J.H., et al. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. Proceedings of the IEEE. Vol. 66, No. 10. October 1978.

[WSZ90] Wszola, S.J. Flashdisk: Not Your Father's RAM Disk. BYTE, Vol15, No. 9, September 1990.

[YOU74] Young, J.W. A First Order Approximation to the Optimal Checkpoint Interval. Communications of the ACM. Vol. 17, No. 9, September 1974.

[YOU83] Yousay, W.J., et al. AFTI/16 DFCS Development Summary - A Report to Industry in Redundancy Management System Design. Digital Avionic Systems, IEEE/AIAA, Conference '83. October 31 - November 3 1983.

Author: Bloch Gerald.
Name of thesis: An Error Recovery Technique For Real-time Distributed Computer Systems.

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2015

LEGALNOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page)for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.