

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Reliable Distributed Computing in Parallel Processing Systems

By

Yong Qian Tian

B.S., ShanXi University, ShanXi Province, R.P.China, 1978
M.S., University of Newcastle Upon Tyne, UK., 1988

THESIS

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in the Department of Computer Science
at the University of Waikato, December 1994

Hamilton, New Zealand

Abstract

This thesis presents a software approach which uses efficient diagnosis along with a recovery strategy to achieve fault tolerance in a multiprocessor system. It proposes two algorithms: a hybrid diagnosis algorithm, and an adjacent checkpointing and recovery synchronisation algorithm for multiprocessor systems.

The hybrid diagnosis algorithm is t diagnosable, and is able to identify hardware faults or errors correctly, given t processor failures in a k-regular connected network (where $t=\frac{k}{2}$).

It requires only one broadcast message to determine the global state in the absence of failures, and 2k+2 messages to detect a failure in the worst case. The number of messages required is constant as the size of the network varies. Compared to other algorithms that require at least O(n) messages for deciding the global state, this algorithm offers a significant improvement.

The adjacent checkpointing and recovery synchronisation algorithm is designed for dynamic partial failure recovery in an array parallel processing system. This algorithm can be incorporated into a parallel operating system, and be independent of the application. It provides continuing operation in the presence of multiple processor failures, or failed inter-processor links, without the requirement for spare hardware modules. The algorithm is constructed by combining a process replication strategy and a process migration technique. The discussion is centered on two issues: efficient checkpointing, and network synchronisation for failure recovery. The algorithm uses neighbouring nodes to store recovery points, and checkpointing is executed in parallel.

In the thesis, we estimate the performance penalty imposed by the fault-tolerant algorithm in the fault-free situation and judge the performance degradation when the system suffers one or more processor failures. The thesis also presents a fault tolerant numerical weather prediction system designed for the New Zealand Meterological Service.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my thesis supervisor, Professor I. Graham, for his encouragement, support and guidance, and the many things I have learnt from him throughout this thesis research.

I also would like to thank Professor I. Witten, Professor J. Cleary and Dr. G. Holmes for their valuable suggestions and for commenting upon this research work from earlier proposals to the drafts of the thesis. I have much to learn from their examples of how to advise students and to show the steps neccessary to reach academic success.

I am also indebted to Drs. M. Kidwell, L.Smith, B. McQueen, Z. Halim, Mr. K. Hopper and Mr. D.Whitaker for helping me with the thesis proposal, and being collaborators in the research over the years. I wish to thank Wayne Schou and Michael Day for sharing the experience of their research. Several people have donated their time and support in the research. I thank Hazel Leslie, Phil Treweek, Brent Summers, Craig Lynch-Blosse and all people studying in the graduate room for providing an enjoyable place to work and live.

Of course, finally, I would like to thank my family, my wife Wei Huang, and son Jack Tian, for the love, patience, support and encouragement, which they gave me during this period.

Financial support for much of the work described in this thesis was provided by grants from the New Zealand Meteorological Service and by scholarships from MERT and Waikato University.

Contents:

Chapter 1 1
Introduction 1
1.1 Motivation 1
1.2 Parallel Processing Architectures 2
1.3 Failure Characteristics 4
1.4 Research contributions 4
1.5 Assessment 6
1.5.1 Performance of the NZ fault-tolerant NWP system 7
1.5.2 Evaluation of the checkpointing and recovery algorithm 7
1.6 Thesis Organisation 8
Chapter 2 10
Reliability, Fault Tolerance and System Structures 10
2.1 Introduction 10
2.2 Reliability 11
2.3 System structures of fault tolerance for multi-processors 11
2.3.1 Active replication 13
2.3.2 Passive replication 13
2.3.3 Fault tolerance through replication and masking 15
2.3.4 Fault tolerance through diagnosis, repair, and recovery 16
2.3.4.1 Fault detection 17
2.3.4.2 Failure Diagnosis 19
2.3.4.3 Reconfiguration 19
2.3.4.4 Failure Recovery 20
2.3.5 Checkpointing methods 23
2.4 Summary 25
Chapter 3 27
Fault-Tolerant Systems 27
3.1 Introduction 27
3.2 Tandem Non-Stop System 29
3.3 Sequoia Computer System 33
3.4 Stratus Computer System 36
3.5 SIFT 38
3.6 Mars 41
3.7 FTMP 42
3.8 MAFT 44

3.9 Summary....47
Chapter 4....52
A Fault-tolerant NWP system....52
4.1 Introduction....52
4.2 System model....53
4.2.1 Transputer network architecture....54
4.2.2 Characteristics of parallel processing for NWP application....54
4.2.3 Reliability issues....58
4.2.3.1 Failure modes
4.2.3.2 A Petri Net model for NWP system....59
4.3 System modifications to improve reliability....60
4.4 Network controller....62
4.4.1 Scheduler....62
4.4.2 Starter....67
4.4.3 Server....68
4.5 Diagnoser....70
4.6 Summary....76
Chapter 5....79
A hybrid diagnosis algorithm....79
5.1 Introduction....79
5.2 Diagnosis algorithms for multiprocessors....81
5.2.1 Centralised diagnosis....81
5.2.2 Distributed diagnosis....83
5.2.3 Processor membership....86
5.3 Preliminary....88
5.4 A hybrid diagnosis algorithm for multiprocessors....89
5.4.1 Fault Models....90
5.4.2 Algorithm....91
5.4.3 Communication overhead analysis....96
5.5 Summary....97
Chapter 6....100
Consistent recovery from partial dynamic failure in parallel processing....100
6.1 Introduction....100
6.2 Adjacent checkpointing and recovery synchronization....102
6.2.1 System overview....102
6.2.2 Adjacent Checkpointing....105
6.2.3 Failure recovery synchronisation control....111
6.3 Related work....114

6.3.1 Chandy & Lamport global state detection algorithm 114
6.3.2 Network Sweeping Algorithm 116
6.3.3 Sync-and-Stop Algorithm 118
6.4 Summary 120
Chapter 7 123
Reliability evaluation of ACRS Algorithm 123
7.1 Introduction 123
7.2 Processor pipeline 124
7.3 The ACRS with toroid mesh topology 124
7.4 Derivation of the reliability of a system tolerant of two module faults 127
7.5 Summary 130
Chapter 8 131
Performance Evaluation 131
8.1 Introduction 131
8.2 Validating the simulation model 133
8.2.1 Simulation of parallel image processing 134
8.2.1.1 Hardware resource generation 134
8.2.1.2 Software process activation 136
8.2.2 A program on real system for validation 141
8.3 Simulation of ACRS fault-tolerant system 145
8.4 Performance analysis 149
8.5 Summary 153
Chapter 9 155
Conclusions and Future Work 155
9.1 Hybrid diagnosis algorithm 156
9.2 ACRS algorithm 158
9.3 Limitations and Future Work 159
Appendix
A1
data for ACRS performance overhead 161
A2
data for ACRS performance degradation 162
References

Chapter 1

Introduction

The reliability of parallel processing systems is becoming an increasingly important attribute. More and more applications require high computer processing performance. This requirement can be satisfied by using the power of parallelism from a large group of processing elements (PEs). Interconnecting these PEs to form a distributed computing system is one of the solutions which can provide a structure of parallelism. An overall computational load can be spread across the available processors in a way that keeps them all busy. Like ordinary computer systems however, a parallel system cannot avoid failures occurring either in the hardware or the software. Moreover, the probability of a failure increases as the number of processors in the system grows. In a non-fault-tolerant parallel processing system, one processor or communications failure results in the failure of the entire system. Without fault tolerance therefore, the high performance of parallelism can not be applied without loss of reliability

1.1 Motivation

Reliability improvement has been heavily dependent on fault-tolerant technology. Most existing fault-tolerant methods for improving the reliability in parallel and distributed systems require modification of the hardware architecture. Many of them were especially designed for highly critical applications. These fault-tolerant systems for special applications are usually expensive. There are many applications that demand a fault-tolerant system to secure their performance but cannot afford the modification of their hardware structure.

Current ordinary commercial parallel processing environments have their focus on performance, and provide little support for system reliability or availability. For example, the Helios parallel processing operating system detects processor or link failures by using timeouts rather than a more elaborate message protocol involving acknowledgments. Its recovery mechanism from failures is just simply to retry sending the message or to reboot the failed processor [HOS89]. Helios expects the occurrence of hardware failure to be very rare, and it doesn't consider any fault-tolerant strategy to recover the system to a consistent state and then continue the processing in the event of failure. As with Helios, many parallel processing systems have left fault tolerance of processor or link failures to the application programmer, and application programmers have been largely unable to respond to the challenge. However as there is an increased use of parallel processing in everyday life, systems are greatly in need of software approaches for fault-tolerance at a reasonable price.

The motivation for this research was a request from the New Zealand Meteorological Service (NZMS) for improvement in the reliability and speed of its large-scale numerical weather forecasting computation. Neither purchasing a commercial fault-tolerant parallel machine, nor developing a fault-tolerant application program was practical for the NZMS. This reflects the situation for many applications. This research investigates a software fault-tolerant approach for improving system reliability on parallel processing systems. It is supposed to be implemented at the operating system level, and be independent of both hardware architecture and application software. It requires no special hardware modification and is applicable to a wide range of applications.

1.2. Parallel Processing Architectures

The hardware architectures for parallel processing are very similar to those of distributed computing. Distributed computing is mainly used for resource sharing and

distributed data transactions in database systems. Distributed computing systems are more autonomous so that the failure of an individual processing component will not collapse the whole system. Parallel processing is used for increasing the processing power in high speed computations and real-time applications. Its typical form is data parallelism in which every processor runs the same set of tasks as other processors in the network but on different data. Furthermore, parallel processing involves more inter-processor actions, and the progress on each node may not be autonomous, i.e., one processor's failure would crash the whole system.

Parallel architectures can be broadly divided into two major categories: shared memory multiprocessors (SMM), and no remote memory access multiprocessors (NORMA). In a shared memory multiprocessor, all main memory is accessible to and shared by all processors. In NORMA architectures, each processor has its own local memory that is not shared by other processors in the system.

NORMA multiprocessors are simpler to design and to build than SMM, and have become the architecture of choice for current parallel processing systems like transputer parallel machines, Intel iPSC machines, the Thinking Machines CM-5, the recent Cray Research Cray3, and others. A typical NORMA multiprocessor consists of a number of processors interconnected with high speed links or a bus. Processors directly access local memory, and communication between processors is carried out through point-to-point message passing. This kind of multiprocessor is also defined as "loosely coupled" as there is no central physical clock, and no shared memory.

One of the most important parts of the loosely coupled multiprocessors is the interconnection communication network. It should be is scaleable, high-bandwidth, high speed and with a low latency. Popular interconnection networks include hypercube, two-and three-dimensional meshes, and fat trees. These types of systems are used to provide high performance, but unfortunately the probability of failure will

increase as the system size grows.

The research presented in this thesis focuses on loosely coupled multiprocessors interconnected by links. Such systems provide a good basis for constructing a fault-tolerant system due to their inherent redundancy. They are called multiprocessors throughout this thesis.

1.3 Failure Characteristics

Three types of failures are considered in this investigation. They are processor failures (site failures), communication failures (link failures) and storage medium failures. The semantics of failure can be either fail-silent or fail-arbitrary. Our fault-tolerant approach assumes that processors have fail-silent semantics. That is, a processor either produces the correct response or no response at all. Fail-silent semantics usually requires an efficient diagnostic program. All errors or failures should be detected before affecting other processors. The diagnostic algorithm presented in this thesis deals with fail-arbitrary semantics and causes the failed processor to stop once errors or failures are detected. The diagnostic algorithm transfers fail-arbitrary semantics to fail-silent semantics as required by the recovery algorithm.

1.4 The research contributions

There have been many previous efforts to improve the reliability of multiprocessors e.g. design diversity for fault detection in both hardware and software [Avi84], creating fault-tolerant programming environments [Shr91], through masking or hiding of the effects of faults, and finally, by identification of the sources of failure, followed by undertaking action to appropriately compensate for the effects of identified failures [KR86]. This thesis, studying a software approach to achieve fault tolerance in a multiprocessor system, fits into the last category, and uses efficient diagnosis along with a recovery strategy. A hybrid diagnosis algorithm is proposed, together with an

adjacent checkpointing and recovery synchronisation algorithm for multiprocessors.

This research makes two main contributions:

(i) A hybrid diagnosis algorithm:

This algorithm is t diagnosable, that is able to identify hardware faults or errors correctly, given t processor failures in a k-regular connected network (where $t=\frac{k}{2}$). It requires only one broadcast message to determine the global state in the absence of failures, and 2k+2 messages to detect a failure in the worst case. The number of messages required is constant when the size (n) of the network varies. Compared to other existing algorithms that require at least O(n) messages for deciding global state, this algorithm offers a significant improvement.

The algorithm is called the hybrid diagnosis algorithm because it is implemented with mixed features of centralised, distributed and processor membership diagnosis algorithms, and with its own characteristics of parallelism and logical analysis policies for distributed test syndromes. It can cooperate with a dynamic failure recovery algorithm to improve a system's reliability.

(ii) The adjacent checkpointing and recovery synchronisation algorithm:

This algorithm takes advantage of the natural redundancy in a homogeneous multiprocessor interconnected with an 3D mesh topology, and it causes nodes to checkpoint regularly to some of their adjacent processors during the absence of failures. In this manner, error-free states for each processing component are distributed over the network. If a failure occurs, system recovery is carried out by redistributing the tasks left on the failed location to a functioning processor.

There are two options for redistributing the tasks left by failed processors: firstly, the recovery process double loads the tasks left on the failed processor to one of its neighbours, which already contains the replica image of the failed processes. The

system can then resume with some performance degradation. Alternatively, the replica images of all the processes on the failed processor are transmitted to one of a few shared spare redundant processors, and system execution continues with the same performance as before the failure.

The adjacent checkpointing algorithm can be classified as a consistent checkpointing technique, which requires one processor to synchronise all of the processors in the network to coordinate in checkpointing the global recovery state. Previous consistent checkpointing algorithms used a shared disk for the storage of recovery states [Rus80]. Their maximum reliability improvement was limited by the reliability of this shared disk device. Failure of the shared device compromised the integrity of the whole network under normal operating conditions.

Moreover, checkpointing messages from n sources converge at one destination simultaneously. This generated communication congestion so that what was initially a parallel action results in a sequential performance. Due to the distribution of the recovery states among the neighbours of each processor, the adjacent checkpointing algorithm saves and transmits recovery state entirely in parallel between processor pairs over the whole network, and enables the system to recover from any individual processor failures.

1.5 Assessment

The reliability improvement of the adjacent checkpointing and recovery synchronisation algorithm was estimated using the Markov chain model, and its performance was evaluated using simulation techniques under the Simscript simulation programming environment. Full implementation of the hybrid diagnosis algorithm was left as future work. However, we present the performance and implementation details of the New Zealand Meteorological Service (NZMS) fault-tolerant numerical weather prediction (NWP) system. The hybrid diagnosis algorithm was developed based on the diagnosis

algorithm in the fault-tolerant NWP system and these two diagnosis algorithms should have similar performance results, so effectiveness of the hybrid diagnosis algorithm should reflect that of the NWP system. The design of the NZMS fault-tolerant NWP system was the starting point for the thesis.

1.5.1 Performance of the NZ fault-tolerant NWP system

Fault tolerance in the NZMS weather prediction system was achieved by diagnosing the system and locating errors or faults in a parallel processing system. The application program is executed in a loop. During each loop, it requests a segment of data from the workfile server on the host machine, and distributes this data across the processors in the network. It then sends the processed data back to the workfile server after completing the computation. Regularly it invokes a diagnoser to check the system. If a fault is located, the system terminates processing, and reloads the terminated data segment into the remaining multiprocessors, whose configuration of functioning resources is explored by the diagnosis program. The computation is then continued from the last data segment rather than from the beginning. This system has been working reliably at the NZ Meteorological Service for more than three years.

1.5.2 Evaluation of the checkpointing and recovery algorithm

The software fault tolerance approach is able to cope with up to $n/2$ processor failures (n is the total number of processors in the system). In the best situation of $n/2$ failed processors, all other functioning processors are evenly double loaded and the system performance is degraded about 50% in processing power. When one and two processor failures were modelled with the Markov chain mathematical model, the analysis result demonstrates that the improvement of the system's reliability is significant.

Tolerating multiple processor failures by double loading to neighbouring processors, or transmitting redundant replicas to shared spare processors, results in a different system

performance after recovery. The second of the two options may cause light communication overheads during recovery, but it imposes no performance degradation once the system has recovered. The first option recovers the system from up to $n/2$ processor failures with about 50% performance degradation. It might result in less degradation if the processor which takes over the failed processes was lightly loaded prior to the failure. The performance variation was simulated with a parallel simulation model which also identified the performance overhead imposed by this checkpointing algorithm during the absence of failures.

The performance overhead imposed by the checkpointing algorithm is a very important issue, since the system performing spends the majority of its time in the absence of failures. The simulation results tell us that the checkpoint overhead is less than 5% of the target application processing. This overhead rate is acceptable to most of applications.

1.6 Thesis Organisation

The rest of the thesis is organised as follows. In Chapter 2 some basic background concepts, and the software fault tolerance framework are provided. In Chapter 3 some existing commercial fault-tolerant architectures are reviewed, and their features analysed. They employ a range of passive and/or active replication techniques to achieve fault-tolerance. This chapter reviews how each checkpointing and control recovery synchronisation has been achieved in previous systems.

In Chapter 4, a fault-tolerant numerical weather prediction system is described, in which a diagnosis algorithm plays a very important role. In Chapter 5 the Hybrid diagnosis algorithm is described, and its efficiency is analysed. The originality and advantages of the algorithm are described through comparisons with other similar efforts.

Chapter 6 is about the adjacent checkpointing and recovery synchronisation algorithm for tolerating dynamic failures. Similarities and differences are pointed out by comparison with some related works. Its reliability improvement for a multiprocessor system with toroidal mesh topology is analysed theoretically in Chapter 7.

In Chapter 8 the performance penalty imposed by the fault-tolerant algorithm in the fault-free situation is estimated, the delay for recovery is assessed, and the performance degradation when the system suffers one or more processor failures is judged. The performance was tested by simulation methods, based on parameters obtained from real systems. The simulation results demonstrate the efficiency and practicality of the fault-tolerant system. The structure of the simulation model for the checkpointing and recovery algorithms is also introduced. Conclusions and ideas for future research are presented in Chapter 9.

Chapter 2

Reliability, Fault Tolerance and System Structures

2.1 Introduction

It is important to understand the difference between fault tolerance and reliability. Fault tolerance is a technique that can improve reliability, but a fault-tolerant system does not necessarily have a high reliability. Likewise, a highly-reliable system is not necessarily fault-tolerant. In other words, the system can achieve a high reliability without possessing the attribute of fault tolerance. If however a system is made fault-tolerant, its reliability should be improved. The concepts and fault-tolerant methods to be discussed in this chapter provide a rigorous setting for explaining the principles of improving reliability in computing and the framework of implementing a fault-tolerant system. We will briefly discuss the techniques from a historical perspective. More details can be found in [And89, Joh89, SS82, AKL87; Fis90].

As will be shown, to support reliable distributed computing, diagnosing followed by backward recovery is an efficient mechanism. The backward recovery technique requires checkpointing to error-free states in order that system can roll back and resume the execution from a checkpointed state. The features of several checkpointing methods and some recovery strategies for the distributed systems are also reviewed in the chapter. Further analysis of some existing commercial fault-tolerant distributed systems and our solution to these problems will be developed in subsequent chapters of this thesis.

2.2 Reliability

Reliability of a system, R(t), is a function of time, defined as the conditional probability that a system will perform correctly throughout the interval [t_0,t], given that the system was performing correctly at time t_0. In other words, reliability is the probability that the system will operate correctly throughout a complete interval of time. Many computing systems designed for applications such as aircraft or nuclear power plant control [Wen78; Lal86], require a high R(t) because a single error could be fatal. For long-life unattended systems such as space craft, the probability of multiple PEs becoming faulty increases dramatically with mission time. Automatic repairs must be made to maintain reliability over the life of the mission, although some performance degradation may be acceptable during these repairs. The reliability of a system may be based on reliable components, but once one of the components fails the whole system fails.

In order to increase the reliability of computing systems, there are two complementary approaches which can be adopted. The most obvious and direct course of action is to expose and eliminate remaining faults before reliance is placed on the system. This approach is termed fault prevention, it aims to prevent faults from being present in the operational system. The second approach considers the situation where an implemented system will not be perfect and it was recognised that a further procedure was often required to enable the operational system to cope with the faults that remain or develop. The aim of fault tolerance is to carry on system operation correctly, possibly with degraded performance, in the presence of the failures. This latter approach, termed fault tolerance, is the major issue to be discussed in the thesis.

2.3 System structures of fault tolerance for multi-processors

Fault tolerance using software methods is one of the active research issues for improving the reliability in parallel and distributed computing systems [Nel90]. However many problems remain. Firstly, the complexity of architecture and

communication facilities results in correspondingly complex mechanisms for dealing with the effects of failures. Secondly, the methodologies for achieving fault tolerance must be distributed among the processors, probably as an integral part of the architecture and software of the system nodes. Making these facilities robust without imposing high overheads in use of system resources is a difficult problem. Last but not least, fault tolerance is likely to be a basic requirement for any large multi-processor system; the costs and overheads associated with such facilities must be factored into any proposed system design and can be significant. State-of-the-art research in fault-tolerance design to improve reliability of parallel and distributed computing is mostly about balancing these three issues, complex mechanisms, high overhead and high cost.

There have been many attempts to attack the problems from various angle, and levels, such as design diversity for fault avoidance in both hardware and software [BKL87; BKL90], creating fault-tolerant programming environments [Shr91] through masking or hiding of the effects of faults and finally by identification of sources of failure, followed by undertaking action to appropriately compensate for the effects of identified failures [KT87]. Those strategies have been considered in four different layers; hardware architecture, operating system, programming environment and application program.

Although a fault-tolerant system can be achieved in different ways, redundancy is an essential technique used in most fault-tolerant mechanisms in computing systems. In case of hardware failure, it can be hardware redundancy or time redundancy. Hardware redundancy indicates that spare components are available to logically replace a faulty one, whereas time redundancy means that operational resources are assigned the tasks (computations) that faulty resources would perform if they were operational.

When redundancy for a fault-tolerant system is at the processor level, the software processes have to be replicated on the separated nodes. Replication techniques can be active or passive. Active replication techniques replicate software components and

concurrently execute each replica on an independent processor. Passive replication techniques provide for the re-execution of software components on a non-faulty processor [SB89].

2.3.1 Active replication

N-Modular Redundancy (NMR) is a form of active replication in which each processor is replicated to form a node and each processor replica within the node executes the same set of software components [EST89]. When the degree of replication is three, the technique is known as Triple Modular Redundancy (TMR) which can tolerate the failure of a single node processor. Communication between the nodes in the form of messages, passes through a voting mechanism by which processor failures are masked. So masking or hiding of the effects of faults [Shr87; Man86] often is the way to conduct fault tolerance for active replication.

In this case, the failure of a bounded number of processors can be masked without a further time penalty. It often takes the form of majority voting, where a process must vote both on inputs produced by other replicated processes and outputs decided by itself and its replicas. Majority voting may impose a performance overhead during fault free execution. Also the cost of hardware replication for masking failures is clearly quite high. To tolerate m failures requires at least $(2m+1)$ replications of hardware in the case of node-level masking. Even though processors might be plentiful in a large multi-processor system, maximising performance and justifying cost dictates that they should be used as efficiently as possible within the reliability constraints of the system.

2.3.2 Passive replication

Passive replication usually replicates and checkpoints the computation to the devices which fail independently from the primary processor. In this method, only primary processes are acting in the fault-free situation. The replicas of processes do not execute until the primary processes fail. One way to achieve fault tolerance with passive

replication is through diagnosis and recovery [KR86].

Passive replication is particularly attractive when temporary disruptions of services can be tolerated. In this type of system re-execution of processes on a failed node moves to a non faulty processor from a state prior to the failure. It is clearly desirable to minimise the resources dedicated to fault-tolerance-related activities and to provide an acceptable degree of performance in the event of failure.

In contrast to masking failures, a second potential approach, diagnosis followed by recovery, to fault tolerance in multi-processors involves attempts to force failures to exhibit themselves. Once the failures are identified, explicit mechanisms for dealing with their effects are invoked. Formally, the processes can be categorised into

(1) fault detection,
(2) fault diagnosis (location and types)
(3) system repair or reconfiguration, and
(4) system recovery.

The first two steps concern the detection and location of a source of failure to the appropriate level of granularity (typically the node level in a multi-processors). The third step involves logical or physical removal of the failed component from the system, along with rearrangement of the remaining system elements to compensate for the loss of the failed component. Reconfiguration of the inter-processor communication facilities may also be required. The final step in the process, recovery of the system state, is perhaps the most challenging problem for large multi-processors. This step involves restoring data and computations in the system to a consistent state (a state equivalent to that prior to the occurrence of failure). This may involve rolling back computations to a pre-failure state and then restarting them.

2.3.3 Fault tolerance through replication and masking

Fault tolerance through replication and masking is an ideal mechanism for real-time applications. The appropriate level for N-modular redundancy techniques in multi-processors would be the processor or node level. Figure 2.1 shows a triple modular-redundant implementation of a node. If a multiprocessor system was constructed as an interconnection of such nodes, then all single failures of processors in the system would be masked at the node level and all internode interaction would be correct.

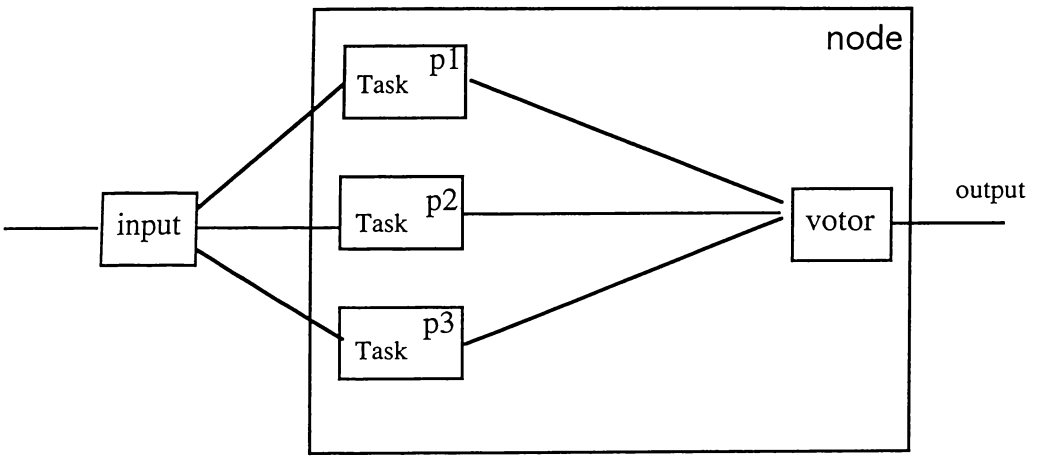


Figure 2.1 Triple Modular-redundancy.

There are, however, several problems with the application of modular redundancy in this fashion to large multi-processors. Firstly the problems of software voting in failure-prone environments could destroy interaction consistency, or more probably, Byzantine agreement [LSP82]. Byzantine agreement basically states the following:

> In a multiprocessor system, allow that up to *m* processors may fail, with faulty processors allowed to act in any arbitrarily malicious manner. Suppose that one processor wishes to send a message to each of the other (n-1) processors. Byzantine agreement is achieved if and only if fault-free processors can agree on the same version of the message, and if the version of the message agreed upon is the correct one (assume original generator is fault-free).

Secondly the potential sources of non-determinism could lead to a divergence of state among software component replicas which could then produce inconsistent responses to identical input messages in an identical order, thereby defeating the NMR voting mechanism. Those sources of non-determinism could be timeouts and prioritised messages. To prevent state divergence, we have to use well-defined generic mechanisms to keep process execution in order. Also, when a system is suffering from a failure, process re-ordering mechanism can be used to keep system consistent.

The Byzantine agreement problem and identical ordering problem have sparked considerable interest. A number of variations of the problem have been investigated, together with a variety of applications of the agreement and re-ordering algorithms [LSP82; Tul91] to the design of fault-tolerant distributed systems.

2.3.4 Fault tolerance through diagnosis, repair, and recovery

The high overhead and complexities of the active replication schemes, for instance NMR, may be justified by the reliability requirements of some applications. It expends a large amount of resources during a fault-free situation to assure minimal disruption of operations in the event of failure. If real-time response and total confinement of errors is required, as in a critical control application such as an aircraft flight system, this approach may be appropriate or even necessary. However, if some temporary disruptions of service can be tolerated, as in the execution of large computationally-intensive jobs without real-time constraints or with the cooperation of time tightness and lateness, then the use of more passive form of fault tolerance may be possible. The passive replication concept in redundant techniques attempts to incorporate fault detection and fault recovery into the system at the expense of eliminating the fault-masking capability.

A general-purpose computational environment may allow a graceful performance degradation following a short delay from failure recovery. Graceful degradation means

that a system is able to decrease its level of performance moderately to compensate for component failures and errors. If this is the case, resources devoted to fault tolerance can be avoided by replacing active replication schemes with passive replication schemes and hence more resources are available for normal computing. The key issue is whether or not the application can afford this short delay for failure recovery; performance degradation can always be compensated by adding more computation components.

If the application can tolerate a short delay without disruption, one way to achieve fault tolerance is to detect and diagnose a fault, followed by an efficient recovery mechanism rather than fault masking. Once a fault is diagnosed, a procedure can be invoked to isolate the faulty component from the system. Then, the state of computations that may have been damaged by the failure must be restored and normal system operation can be resumed. The proper term for isolating the faulty component from the system is reconfiguration. The reconfiguration can be logical and physical. Although the cost of diagnosis, reconfiguration, and recovery may be relatively large, it is only incurred following failures, so the impact on the system may not be unduly large. If we keep the cost of fault detection low enough, then this approach will be more attractive from the standpoint of system throughput.

The process described above may sound straightforward, but its application to large distributed systems poses non-trivial problems at each step. We will now briefly discuss each of the four aspects: fault detection, diagnosis, reconfiguration, and recovery in the context of large multi-computer systems.

2.3.4.1 Fault detection

Fault detection is the process of recognising that faults have occurred. It is often required before any recovery procedure can be implemented. Many techniques are available to detect faults, such as parity check, consistency checking, and protocol violation. Unfortunately, these techniques are not perfect, and an arbitrary period of

time may pass before detection occurs. This time is called "fault latency."

Failure at processor level can be detected in two ways: external detection and internal detection. External detection implies that responsibilities for detection of failure of a node are given to facilities external to the node (foreign nodes in the system). These methods can be:

(1) interacting with a node to detect any deviation of the node from expected behaviour.

(2) performing some testing activities on a node under test, at appropriate intervals to force failures to exhibit themselves. There are two reasons for this type of test: first, processing elements are complex entities, with a large number of internal states and many points of potential internal failure. Second, the testing through the internode communication media does not provide the tester with a large degree of direct control or observation of the node under test.

Internal testing places the failure detection mechanisms for a node within that node, such as (1) hardware design to check itself, (2) using concurrent error detection schemes, (3) employing simple replication and matching (two copies required to detect any mismatch between outputs from the copies). Once a failure is detected, notification to external nodes must be made.

For the internal testing, the mechanisms for performing external notification may have to be very robust, because the results of the testing must be known outside the detecting nodes. One example is to make the processing stop as soon as a fault is detected, assuming that other nodes can detect the complete loss of the node.

A combination of internal and external detection mechanisms may be used; internal mechanisms for detection of failure within the node, and external mechanisms for validation. For example, designating pairs of nodes to execute identical software

modules and exchange information in order to compare results. In this case, the system can detect faults but it cannot distinguish which is the faulty node, so that a failure has to be confirmed by a subsequent diagnostic process.

2.3.4.2 Failure Diagnosis

If the fault-detection technique does not provide information about the failure location and/or properties, a diagnostic step may be required. Diagnosis refers to the process of determining the location of a fault or faults in the system. Correct diagnosis of faults is necessary to allow proper reconfiguration and recovery actions to be carried out. Usually, diagnosis is controlled by two factors: (1) the nature of the fault model assumed, and (2) the extent of the diagnosis required.

In conventional system level fault diagnosis, the results of the testing are utilised to compute a single, centralised diagnosis for the system. But in a distributed system, each fault-free node in the system must independently produce its own tests and the testing information must be received from other nodes through system communication facilities.

Diagnosis in a distributed system has to consider the following situations:

- Failure affecting the internode communication facilities.
- Systems containing nodes with limited or no ability to perform testing.
- Allow the dynamic repair and return to service of formerly faulty facilities.

2.3.4.3 Reconfiguration

If a fault is detected and a permanent failure is located, the system may be able to reconfigure its components to replace the failed component or to isolate it from the rest of the system. In other words, the configuration can be physical or logical.

Physical reconfiguration is to control the electrical circuits and isolate failed components physically. It is unlikely that much effort would be expended on on-line hardware reconfiguration unless it involved a critical resource, without which the system could not function adequately in its previous configuration. Because any facilities to switch-in spare nodes, redirect communication paths, etc. would have to be distributed throughout the system, the cost of such facilities would rise at least in proportion to the size of the system. In addition, there is the obvious concern about the robustness of the reconfiguration facilities, since failure of any of them to operate correctly could have system-wide consequences.

Logical reconfiguration is an alternative to physical configuration. In logical reconfiguration, each fault-free processing node can reorient its view of the system to avoid interaction with faulty facilities. In this way failed nodes and communication links can be effectively isolated from the system without physical removal. This solution is an attractive means for the recovery strategy with dynamic reconfiguration

2.3.4.4 Failure Recovery

No matter what kind of reconfiguration you choose, application programs which are affected, or potentially affected, by a diagnosed fault must be recovered to a guaranteed error-free state. Normally, the system operation is backed up to some point in its processing that preceded the fault detection, and operation recommences from this point. This form of recovery, often called "rollback recovery" [MR78; Ran75], usually entails strategies using backup files, checkpointing, and journaling. In the recovery, error latency becomes an important issue because the rollback must go far enough to avoid the effects of undetected errors that occurred before the detected one.

If we assume that the fault is at the processor level, tasks executed on the faulty node must be logically removed from the processor, and replaced with alternative versions

outside the faulty node. The place where the alternative versions are saved depends on the system under consideration and the strategy employed. There are three possible storage alternatives: (1) shared devices (sharing nodes or disks), (2) redundant components (3) neighbouring nodes.

The alternative version of each task must be in a state consistent with that of the original task prior to its failure. It is widely recognised that error recovery problems become difficult in a distributed environment, and it is clear that much additional work is needed to fully understand and solve these problems. One of the difficulties is the establishment of a common point set from where distributed operation can resume consistently. The common point set is called the recovery line which is formed by linking a set of recovery points between the tasks such that the system can be restarted with global consistency between the tasks.

A recovery point (or checkpoint) is a point at which the target program is restored upon recovery. These recovery points are supposed to be stored with the alternative version of the task. Checkpointing involves saving a copy of all necessary information about the current state of the task. The necessary state information is that which will allow the task to be restarted at the point of the setting of the recovery point, and in the same state. The state includes all data objects, registers, message state etc. Checkpointing in multi-processor systems can also be used for process migration and playback debugging.

While setting a recovery point for an individual task may be relatively simple, coordinating the establishment of a set of recovery points for a group of tasks (recovery line) may be very difficult. The source of the difficulty is a phenomenon know as the domino effect [And85; Ran77]. This effect is illustrated in its simplest form in Figure 2.2. The thin solid lines between tasks indicate points of communication (or access to common data objects) between tasks. The thicker line represent the recovery line

established for the tasks. Each small circle marks an active recovery point to which the state of the corresponding process can be restored.

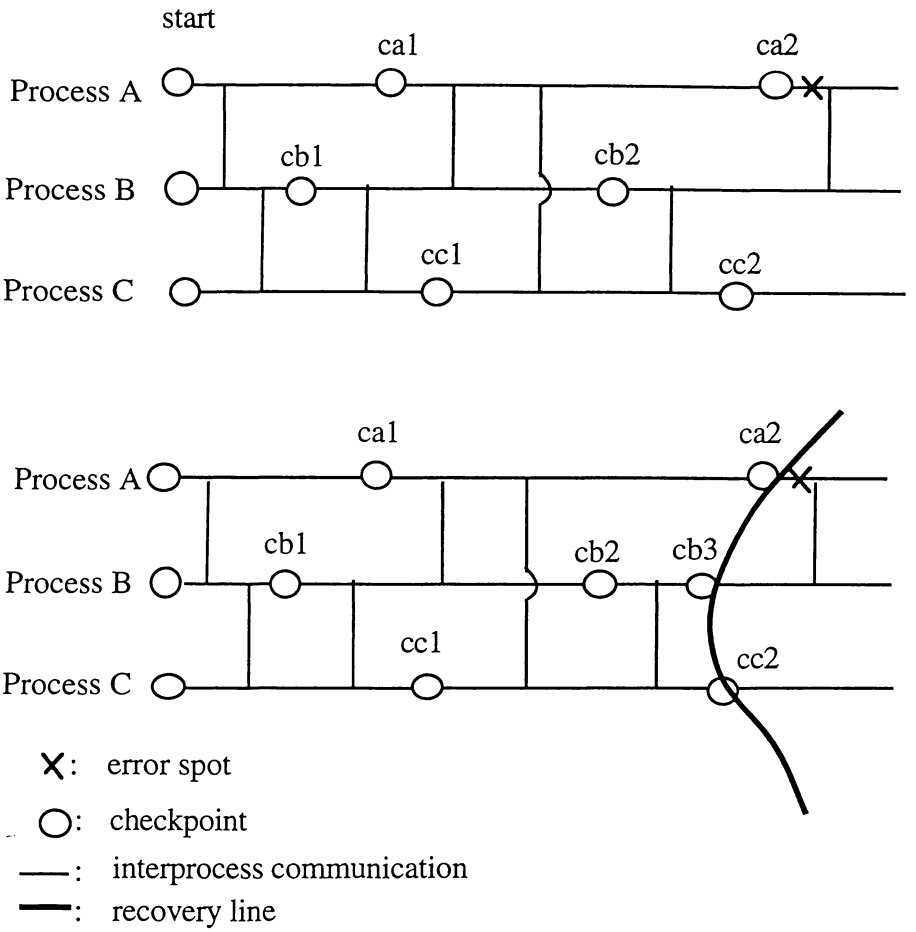


Figure 2.2 Illustration of domino effect and recovery line for three communicating processes

Now, suppose a fault occurred after recovery point ca2 in process A. Then process A should be restarted on a different processor from point ca2. But since process A may have communicated with process B subsequent to establishing ca2, process B must be rolled back to maintain proper synchronisation with process A. Furthermore, rolling back process B to point cb2 results in the need to roll process C back to point cc1, which requires process A to roll back to ca1, etc., until eventually all three processes are rolled back to their starting points. Thus, an uncontrolled 'roll-back' is precipitated, with recovery points being restored in sequence like a line of toppling dominoes.

If we modify the first diagram into the second in Figure 2.2 by setting one more checkpoint cb3 following its communication with process C, then it could be assured

that no failure would cause a backup of the task force beyond points ca2, cb3, and cc2. In this case the set of recovery points {ca2, cb3, cc2} are said to constitute a recovery line for the task force.

In the distributed environment, the dynamic (automatic) maintenance of recovery lines for large task groups that exhibit a high degree of inter-task communication requires complex protocols and a potentially large amount of overhead. Reducing the overhead of checkpointing to establish recovery lines regularly in a distributed system is a very important issue. Because a fault-tolerant system mostly performs in the absence of failures, non-efficient checkpointing would seriously affect normal system performance, so consistent checkpointing has been well researched for distributed systems. In addition to those for database computations, checkpointing algorithms for distributed computing fall into three general categories: Pessimistic, Optimistic and Consistent/coordinated algorithms.

2.3.5 Checkpointing methods

Pessimistic algorithms assume frequent failures of the system, and thus take strict precautions to ensure the reliability of the computation. Pessimistic algorithms tend to impose heavy penalties on failure-free performance for the sake of reliability. In these schemes, each process is backed up by a mirror process and all messages have to be sent to all mirrors of destination and initiator. The system must guarantee that this operation be atomic. When failure happens, it takes time to make a new mirror and catch up with the current state.

Optimistic checkpointing algorithms [JZ90; SY85] are the opposite. They assume that failures occur relatively infrequently, and therefore attempt to push most of the overhead of checkpointing into the recovery process. In this scheme, processors independently checkpoint themselves and maintain a log of the messages that they receive and the state of the processors that send them messages. Each processor can

checkpoint whenever it wants. They are not affected by the checkpointing on other processors. When a processor fails, all processors cooperate to decide to which state they should restore themselves. This is decided by a complex distributed algorithm, it may involve rolling back to previous checkpoints again. This algorithm is optimistic during failure-free operation, and allows a large degree of autonomy. But rolling back and replay can be very time consuming activities. On parallel multi-processor systems, autonomy is not as important as in a distributed system, since one processor failure would bring down the whole system. It makes sense to give up autonomy and cooperate more tightly than in optimistic checkpointing.

Consistent/coordinated checkpointing algorithms try to strike a balance between optimistic and pessimistic checkpointers. In other words, they attempt to provide good failure-free performance, and simple recovery. The consistent checkpointing algorithms stem from a paper by Chandy and Lamport [CL85]. Most previous consistent checkpointing algorithms use a shared device as the storage for all the secondary replicas [Pla93; CL85]. In these algorithms, all processors cooperate at checkpoint time to define a consistent state, and write it to disk.

Recovery is straightforward, all processors read their states from disk, and computation may proceed. Consistent checkpointing seems to be best suited to parallel multi-processor checkpointing, as it does not overly penalise the fault-free performance, and recovery is simple and fast enough to be used for general computation interaction, and message swapping.

However, in the existing consistent checkpointing methods, the maximum system reliability improvement is limited by that of the shared device. Failure of the shared device may crash the whole network in normal operation. Moreover, coordinated checkpointing requires all processors to save and transmit their error-free state to replicas. So checkpointing messages from n resources are sent to one destination at the

same time. This generates communication congestion so that the parallel action results a sequential performance. The fault-tolerant algorithm presented in this thesis uses neighbouring processors as locations for the replicas, so the replicas are distributed over the network with multiple copies. Our algorithm will solve the checkpointing bottleneck and sequential performance problems.

2.4 Summary

Reliability can be improved by using various fault-tolerant and fault prevention approaches. Ideally, failures can be prevented if all faults are detected and eliminated before execution of the application. However fault prevention can only reduce the possibility of a failure, it cannot guarantee that no failure occurs during processing. Fault tolerance is a very important strategy for improving reliability. In the fault-tolerant design, combinations of redundancy and process replication techniques can provide the information needed to negate the effects of the failures and continue the service on functioning processors.

Fault masking using N-modular redundancy is a form of static redundancy with active replications. Although N-Modular redundancy can provide continuos service in the presence of N/2-1 processor failures without performance degradation, its interactive consistent voting processing on input and output could introduce high overheads in the fault-free execution. It also has problem of expensive construction and difficulties of process ordering and Byzantine agreement.

In contrast to fault masking, fault tolerance through diagnosis, reconfiguration and recovery is an attractive mechanism if an application can be disrupted with a short delay. It can minimise the amount of resources dedicated to fault-tolerant related activities. It is a mechanism which is able to tolerate multiple processor failures. However, fault tolerance through diagnosis, reconfiguration and recovery requires high capability of fault detection and diagnosis and needs efficient checkpointing and a

backwards recovery strategy to support it. A non-efficient checkpointing technique leads a high overhead performance during fault-free situation. The complexity of the recovery protocol could be resource of failure too.

There have been many works on fault-tolerant distributed computing in both fault masking and identifying failure followed by recovery mechanisms. In the next chapter, we introduce several interesting fault-tolerant distributed systems and outline their distributed diagnosis and checkpointing algorithms.

Chapter 3

Fault-Tolerant Systems

3.1 Introduction

The research work presented in the thesis offers an alternative software approach to avoid hardware component failures in parallel architectures. Existing fault-tolerant systems are mostly designed for specific applications, such as data transaction processing, and real-time control functions, and also their fault-tolerant methods were considered for common communication media networks and their redundancy employed in much finer granularity units [AL81; HLD88]. In this chapter we review some commercial fault-tolerant distributed computing systems with emphasis on fault diagnosis, checkpointing and restarting, and interactive consistent synchronisation. The content in this chapter is part of investigation of fault-tolerant techniques achieved previously. Further related work regarding diagnosis and checkpointing/restarting mechanisms are presented in Chapter 5 and Chapter 6 respectively through a comparison with thesis contributions. The following examination of several systems shows how each maintains operation in the event of failure.

Existing fault-tolerant distributed systems can be distinguished with their process replication techniques with assumptions of failure semantics. As mentioned in Chapter 2, failure semantics can be fail-silent (crash failure) and fail-arbitrary (arbitrary failure), and replication techniques can be active and passive. With this criteria, the systems may be divided into three groups as shown in Figure 3.1.

In the first group (Tandem passive model), processors are assumed to have fail-silent semantics, allowing passive replication techniques to be used for process backup. The fail-silent semantic assumes the outputs from a functioning processor are either correct or failure can be detected and the process can be forced to stop before transmitting any data out of the processor [SS83].

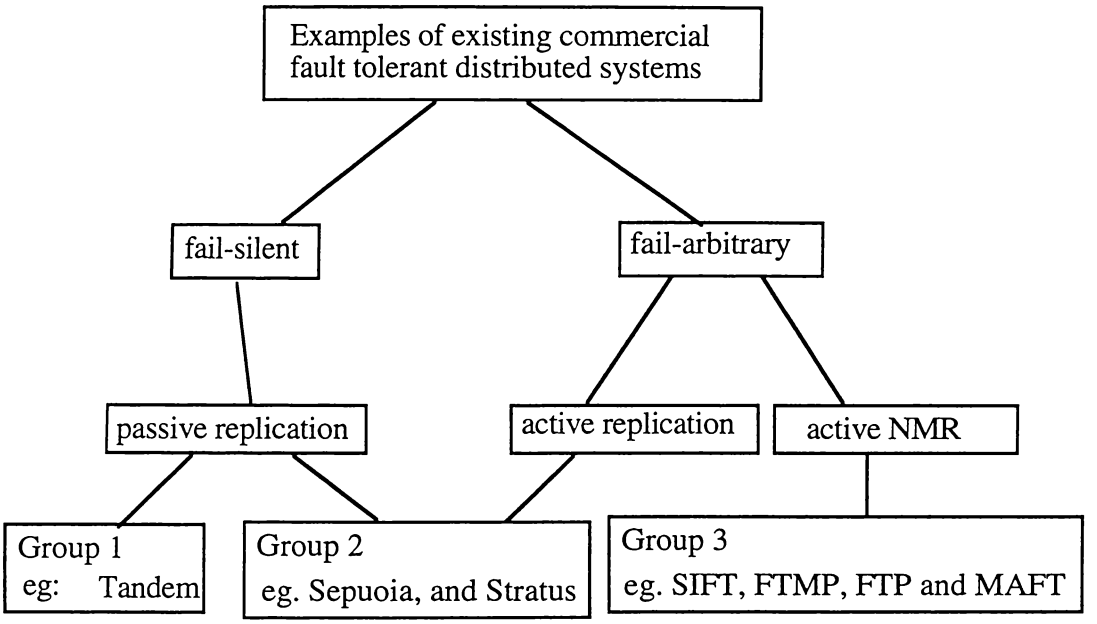


Figure 3.1 Examples of existing FTDC systems classified by failure semantics and replication techniques.

NMR: N-modular redundancy, FTDC: Fault-Tolerant Distributed Computing

In the second group (Sequoia, Stratus active model), system architecture is organised in a hierarchy with a mixture of passive and active replication formats as well as failure semantics. System reliability depends on each processing module (PM) and a PM is composed of two passive replicated processing elements (PE), and each PE consists of two processors. The active replication in such organisation is implemented at the processor level. Fail-arbitrary semantics are assumed for processors although fail-silent semantics are guaranteed for PMs.

In the third group (SIFT, FTMP, FTP, MAFT active model), processors are assumed

to have fail-arbitrary semantics. The active replication takes the form of N-modular redundancy for each critical unit and the same tasks are allocated to each replication processor and executed in parallel. It employs majority voting on inputs and outputs to mask any performance deviation.

In addition to fail-semantics and replication techniques, the employment of independent redundancy requires some form of synchronisation among the independent data resources. In general, synchronisation can be considered in either "soft or loose", or "hard or tight" approaches [HSL78; SCL89]. Soft or loose synchronisation involves such operations as buffering, comparing or voting, signalling consensus, and marking completed intervals. These usually are implemented in software, given suitable intermodule data links. Hard or tight synchronisation involves hardware comparison or voting, and a common time reference, whereas loose synchronisation can employ separate time references.

In the rest of this chapter, we analyse fault-tolerant systems from various aspects which are related to this research.

3.2 The Tandem Non-Stop System

The Tandem Non-Stop system [Bar78, HC85] is a commercial distributed computing system for on-line transaction processing. When the system was designed in 1976, the view was taken that active replication of processes through hardware redundancy would not be cost effective for that application. So Tandem assumes that processors have fail-silent semantics, then uses passive replication to tolerate failures. A simplified representation of the Tandem architecture is shown in Figure 3.2. Its key architectural features are processor replication, dual-access I/O controllers, a redundant power system, and a message based operating system.

A Tandem system is composed of 2 to 16 computers interconnected by two message

oriented buses (Dynabus). It is a loosely-coupled architecture instead of a tightly coupled, shared memory architecture. Each processor has its own power supply, memory, and I/O channel. All devices have redundant power supplies and are connected by means of dual paths to all other system elements (disk controller, I/O controller), processors are replicated and disks are mirrored (two identical copies of files on two independent disk drives). The processors communicate over a high-speed, duplexed, 16-bit parallel bus system called Dynabus. Any single point of failure can be avoided by using the redundant device through an alternative access route.

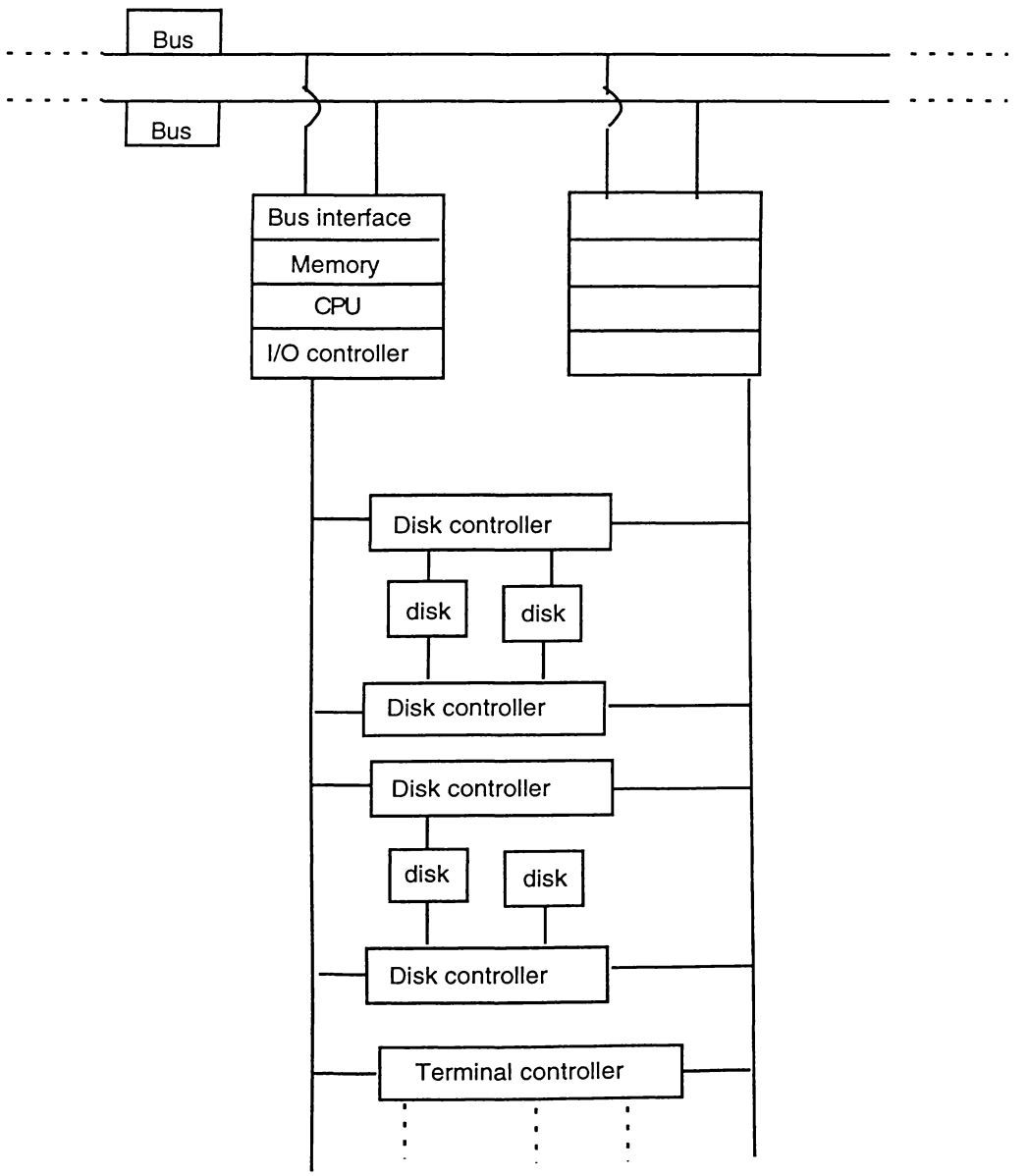


Figure 3.2 Tandem's system organisation

Upon this hardware structure the software builds a process-oriented system with all communication handled as messages. Due to its design nature for transaction processing, all hardware and software resources can be shared in a distributed manner through the network. Each processor in the system contains its own copy of the Guardian operating system, which maintains local tables reflecting the status of all available system resources. The local and global status of resources contained on each processor allows the blurring of the physical boundaries between processors and peripherals. It can support large number of terminals.

Since the Tandem system uses fail-silent semantics, the hardware and software modules are designed to be "fast-fail", that is, to terminate processing immediately after detection of errors. The failure detection techniques employed in hardware modules include checksums on bus messages, parity on data paths, error-correcting-code memory, and watchdog timers. Software modules employ consistency checks and defensive programming techniques. Stalls (infinite loops) and processor crashes are detected by the absence of watchdog timer messages, which each processor is required to periodically broadcast over the Dynabus to all other processors.

Checkpointing is the key recovery mechanism in the Tandem system. All software items such as processes, messages, files and data are maintained in the form of pairs. For each running process designed or stored as primary, there is an identical, but semi-inactive, backup process in another processor. This semi-inactive process implies that the process is created but it will not be executed until its primary fails. Passive replication techniques are reflected in static backup mechanisms.

All messages are delivered to the primary process, then the primary process sends a duplicate of message with checkpoint information to the backup so that it can take over if the primary's processor or access path fails. Those primary and backup processes are kept on physically distinct processors. Thus, in the event of physical failure or

isolation of the primary, the backup file is up-to-date and available. The operating system in each processor "wakes up" the relevant backup process upon discovering that its corresponding primary has failed. The backup can then resume the task from the state defined in the last checkpoint.

The checkpoint information is inserted in the corresponding memory locations of backup process, as opposed to the more usual approach of updating a disk file. This approach permits the backup process to take over immediately in the event of failure, without having to perform the usual recovery journaling and disk accesses before processing resumes.

The philosophy of primary and backup techniques and watchdog timer failure detection used in Tandem systems is somewhat close to that proposed in the thesis. However its algorithms cannot be adopted in modern parallel and distributed computing systems. Firstly, Tandem systems take the form of redundant pairs for all devices (including processors). During absence of failure, those spare components are not active and wait for failure to occur. This form is suitable only for a small multicomputer system and is not a suitable practice for massive multiprocessors with array architecture.

Secondly, the Tandem system was designed for transaction processing. The progress of processes on each computer is independent of those on other processors. Communications between processors are for resources and data access, there are no interprocess actions. In such a constrained situation, checkpointing activity has no requirement of interprocess consistency, and is between the primary processor and its replica. Recovery from a processor failure involves only that appropriate process replicas roll back to the last checkpoint and continue service without any performance degradation. Other processors even don't have to notice the failures. It is just the opposite in parallel and distributed computing systems, where the major effort is in interprocess consistency and network synchronisation for recovery.

Thirdly, communication between the processors in Tandem systems is through the common media network. Any processor is able to send messages to others directly. An individual node failure does not crash the whole network. But in multiprocessors, communication between processors is through message passing, and one node failure affects not only its own function but also could break the routing path in which it plays an intermediate role. To recover from such processor failure, process migration and network reconfiguration are appropriate techniques which the Tandem system did not need to consider.

A similarity between Tandem systems and our proposed algorithm is that both use the checkpointing and rollback recovery technique. The concepts of primary and backup passive replication and fail-silent semantics are in common.

3.3 The Sequoia Computer System

Sequoia [Ber88; Ber90; Sie90; Cri91; Tul91] is a tightly coupled distributed computing system designed for on-line transaction processing. The system uses active replication at processor level and passive replication at the level of processing element (PE). Unlike Tandem which assumes fail-silent semantics, Sequoia assumes that processors are fail-arbitrary and employs active replication (a processor pair) to implement a PE's function with fail-silent semantics. A simplified representation of the Sequoia architecture is shown in Figure 3.3.

The Sequoia system is composed of three major element types: PEs, memory elements (MEs), and I/O elements (IOEs). Elements are interconnected by a dual, segmented system bus (Processor local bus, Global bus, Memory local bus). Each PE is constructed using two processors that execute the same instruction stream in lockstep and continuously compare their results to detect any difference in behaviours and ensures that PEs have fail-silent semantics.

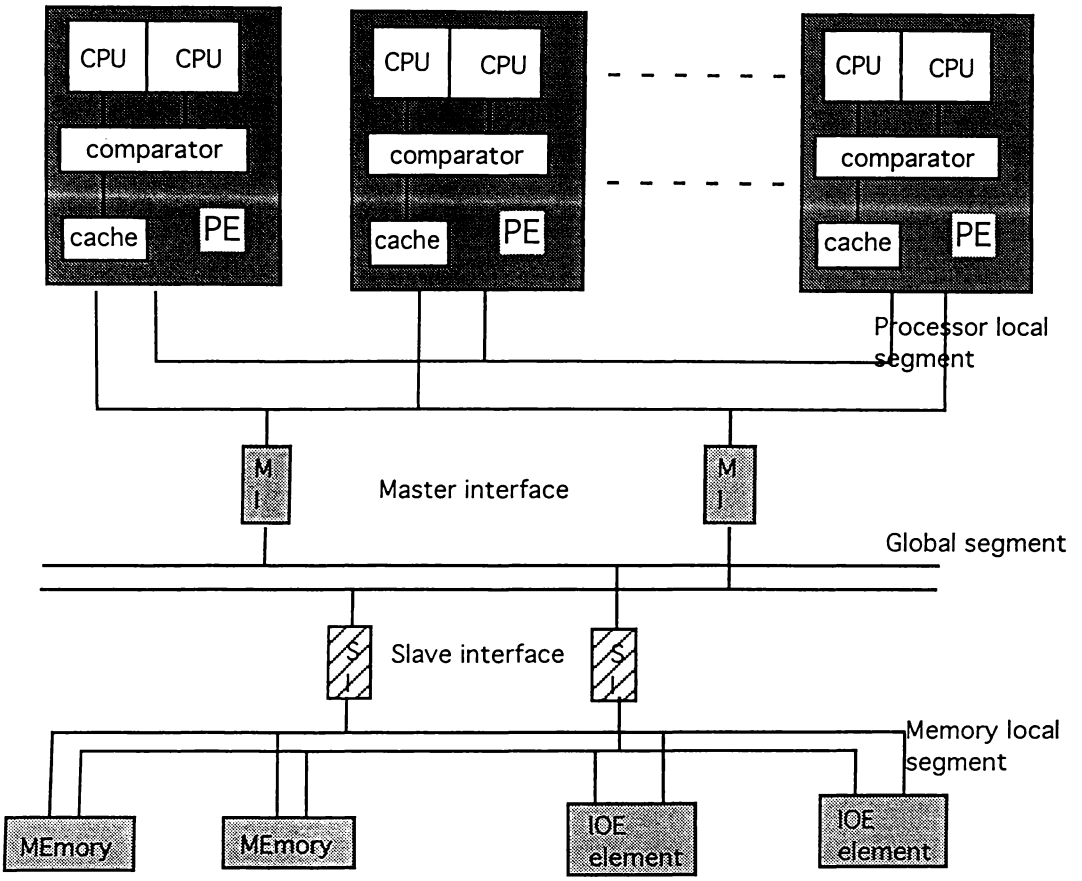


Figure 3.4 A simplified Sequoia architecture

Each PE contains a cache memory. A process P executing on a PE uses this cache memory to store its data; all read and write requests operate on the cache. Periodically, the contents of the cache are flushed to two distinct MEs, thereby establishing a replicated checkpointing. If a PE fails, each process residing on the failed PE is restarted on another PE by executing the code stored in an ME from its last checkpoint. Fail-silent semantics are reflected to PE failures in this manner. Failures of MEs or busses, on the other hand, are dealt with on a much finer granularity. Three types of mechanism are used: error-detecting coding, comparison of duplicated operations, and protocol monitoring [CJ91].

Coding: Data is protected by error-detecting codes while stored in a PE cache, in an ME or while in transit over a bus. The hardware implementing all data storage transfer paths is partitioned so no two bits of the same byte have any common components.

Consequently, single-component failures can only produce a single error in any byte and all such errors are detectable.

Comparison of duplicated operations: Sequoia uses hardware duplication and comparison to detect processor failures. A comparator tests operation in the background on a bit-by-bit basis.

Protocol monitors: prevent message omission failures by detecting violations in the sequence and timing of inter-element communication.

Together, the fault detection techniques discussed above provide immediate detection of all single error resulting from hardware faults. Once a failure is detected, the system disconnects the failed component physically then resumes the execution on its duplicated component. This is fail-silent semantics. For example, if a ME fails, a PE can always obtain data from the other ME to which it checkpoints. Read-only data is treated as a special case; it is not checkpointed. Actually, it is stored on only one ME. If that ME fails, data is restored from disc.

All Sequoia components thus have fail-silent semantics, either through active replication (PEs) or error detection (MEs and busses). Replication of all code and data, using checkpoints and disc backups, ensures that the execution of a failed process may be resumed elsewhere.

Because Sequoia is a tightly-coupled system, where each processor has direct access to all memory and I/O resources, its checkpointing involves no interprocessor data transmitting. Due to its special design for transaction processing applications, there is no interprocessor dependent processing. In addition to autonomous execution on each individual processor, the common media dual busses prevent any single point failure from affecting system continue operations. So communication synchronisation and recovery strategy are very different from loosely coupled multiprocessors.

3.4 The Stratus Computer System

The Stratus computer system [Cri91; And85; Sie90] is another commercial distributed computing system for on-line transaction processing. Stratus introduced the systematic use of duplication and matching for implementing CPU, I/O and communication controller hardware servers with fail-silent semantics. It assumes that processors have fail-arbitrary semantics and employs active replication (a processor pair) to implement a processor with real fail-silent semantics (like Sequoia). Active replication is then used again to mask the fail-silent failure of a processor pair.

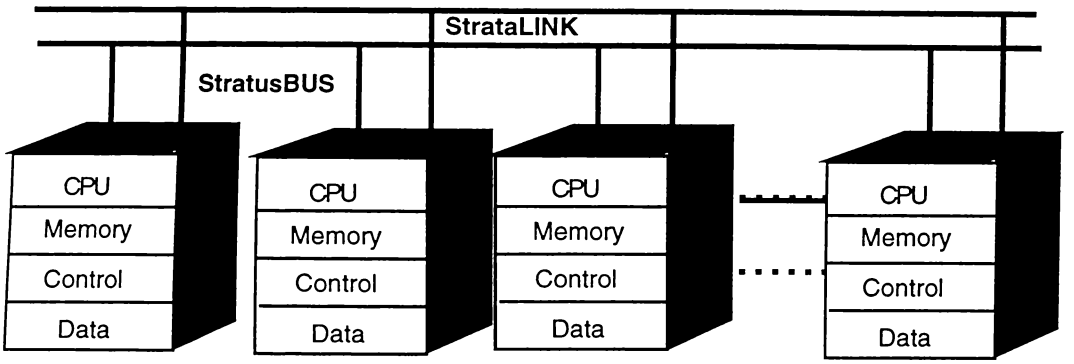


Figure 3.4 The Stratus Computer System

A Stratus system consists of a number of processing modules (PM) connected by a dual bus called StrataLINK, as shown in Figure 3.4. Each processing module consists of two identical processing elements (PE), connected by a dual bus call StrataBUS as shown in figure 3.5. Each processing element consists of a number of components (CPU, memory etc) which constitute the smallest replaceable unit (SRU) of the system.

A CPU component is constructed from two identical processors operating in lock-step execution (like Sequoia). A failure can be detected by two processors comparing their outputs in hardware. As long as they match, outputs are gated onto StrataBUS. If a disagreement is detected, the outputs are not propagated, the component is isolated from StrataBUS and thus has fail-silent semantics.

A memory component contains error detection/correction circuits which prevents the propagation of erroneous data. A memory component therefore also has fail-silent semantics.

Stratus may also contain a number of other component types (not shown) such as disc controllers. In all cases, components have fail-silent semantics.

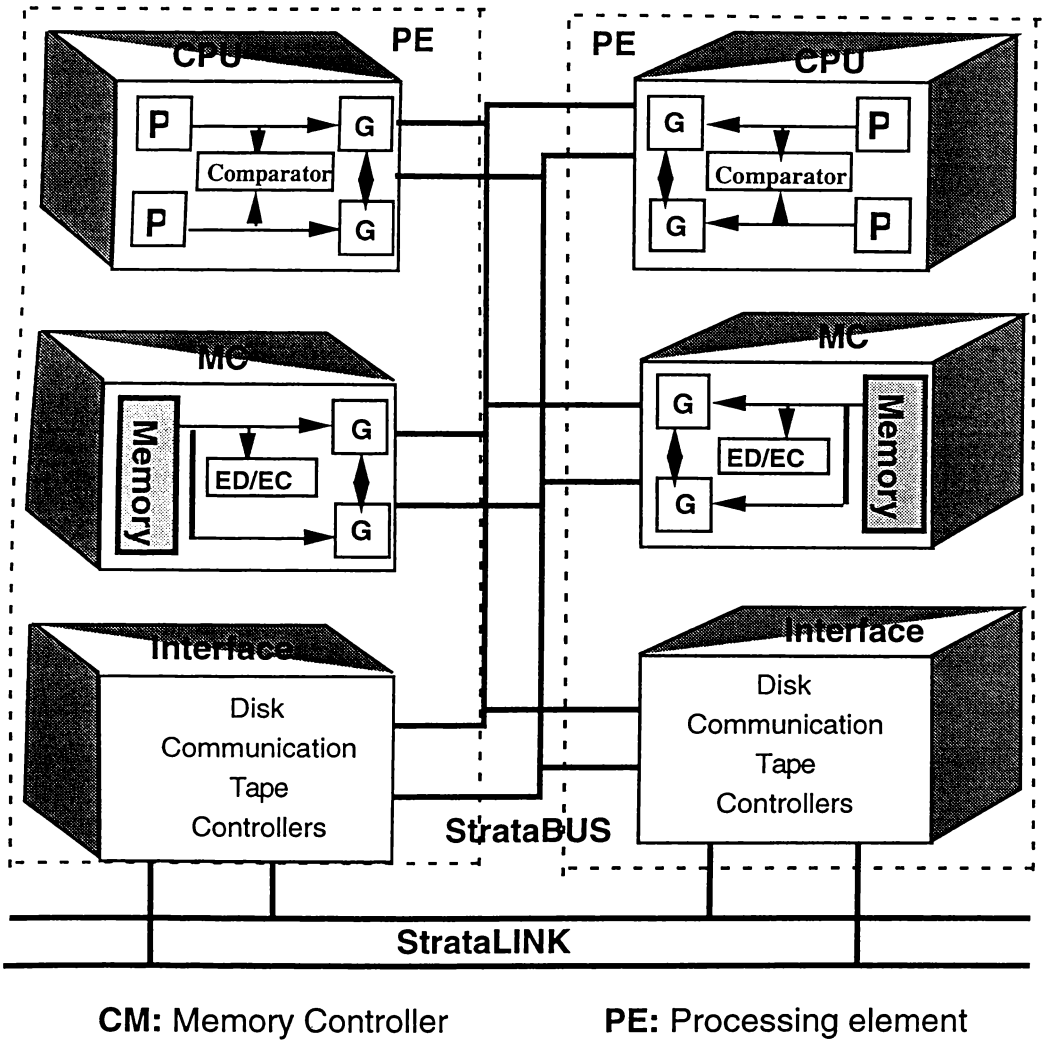


Figure 3.5 A Stratus Processing Module

When a component in a PE issues/receives a message on StrataBUS, the corresponding component (in the other PE of the same PM) also issues/receives the same message (in the absence of failures). So when a single failure occurs, there is always at least one

message issued/received and the failure is masked.

The Stratus solution is a hardware approach which detects a failure or transient error at the point of failure, so that data is always protected from corruption. The processors which make up the two CPU components in a processing module are fed exactly the same inputs in the same order in lock-step. Thus all four processors in a processing module operate in lock-step execution. If one component fails, operation can be continued by another processor element, so there are no requirements of checkpointing and restart programming for recovery.

In addition to fail-silent semantics, Stratus also has the two features of fine granularity and group masking. Most of its elementary hardware servers are customer replaceable units with their own power supply. Group masking is achieved at the hardware level by pairing elementary hardware servers with fail-silent semantics.

3.5 SIFT

SIFT (Software Implemented Fault Tolerance) was developed at Stanford Research Institute for the National Aeronautics and Space Administration [Sei82; LAB+90; Wen78; Wei80]. It is an ultra-reliable computer achieved by actively replicating tasks among processing units. The SIFT approach makes no assumptions about failure modes, it only distinguishes between failed and non failed units.

The physical architecture of the SIFT system is shown in Figure 3.6. It is composed of a number of processors, each processor, P_i, with its own memory M_i. Processors and memory are directly connected to multiple busses. A processor may read or write its own memory but may only read the memory of other processors.

Every P_i executes one or more processes. Every process is replicated and each replica executed on a different processor. The calculations are performed on main processing

modules and input/output is carried out on I/O processing modules. Data produced by a process replica is deposited in its own memory. Input data is read from the memory which contains replicas. Each read operation is performed over a different bus, and the results are combined through majority voting to produce a single input to the process. In this way, the failure of a single process replica (through the failure of its host processor) or the failure of a bus will be masked.

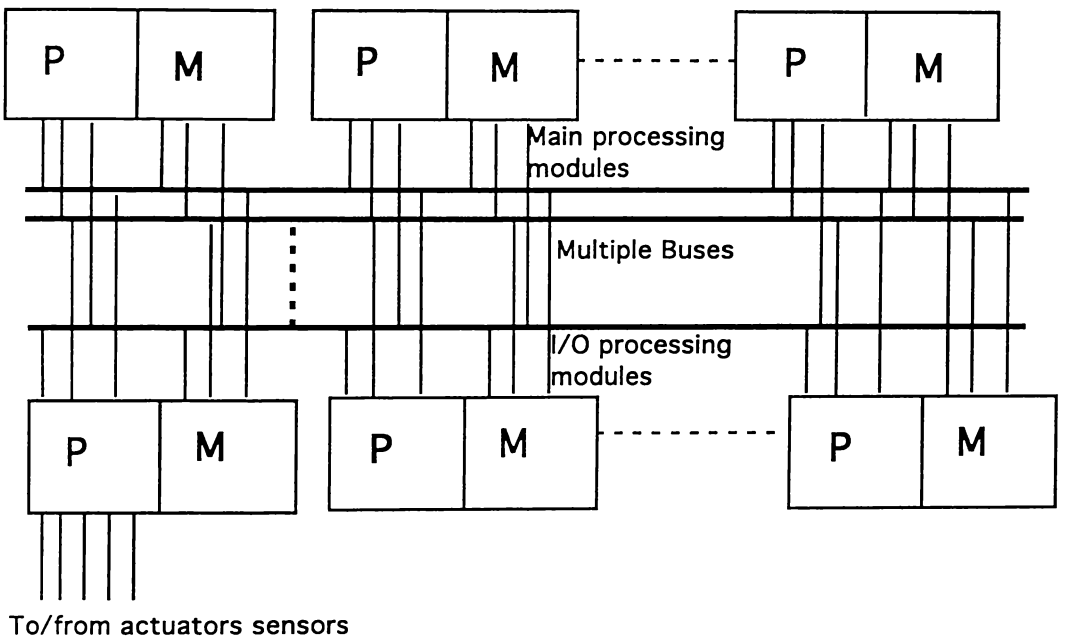


Figure 3.6 The physical structure of the SIFT system

The input data for each processor is voted on at the beginning of each iteration. This has important implications for the problem of synchronising the different processors. Executing the same iteration of the task on different processors allocated to a replica must be ensured, so synchronisation between voting processors is necessary. The synchronisation method developed for the SIFT system is a loose synchronisation, in which the individual physical clock on each processor is synchronised through intertask and interprocessor communication mechanisms.

The clocks of all processors are loosely synchronised to within 50μs (clock synchronisation is another form of the interactive consistency problem, discovered and

solved by the SIFT designers). So the time frames of different processors will also be synchronised to within 50μs. Now suppose the clock of P_b is running 50μs faster than that of P_a, and P_a outputs its data to the buffer right at the end of a time frame (as determined by the clock of P_a). If P_b inputs its data from the buffer right at the start of a time frame (as determined by the clock P_b), it could precede the data output by 50μs and the data would thus be two iterations old. It is necessary therefore to separate the input and output of data by, for example, preventing the inputting of data until at least 50μs into the time frame. The proportion of the time frame available for input is then reduced by 50μs.

In SIFT, fault detection is based upon the processor/bus error table, a *m* by *n* matrix, where *m* is the number of processors and *n* the number of buses in the system. Each processor has its own processor/bus error table that is maintained by its local executive's error handler.

Each processor has a error reporting task which analyses the processor/bus error table. It will inform the global executive that the processor or the bus is faulty. The global executive task makes the final decision about which unit is faulty. If two or more processors report that the same processor is faulty, then the global executive decides that the processor has indeed failed. If two or more processors report the same bus is faulty, then the global executive task decides that the bus has failed.

In order to avoid being confused by an error report, the global executive in SIFT also can allocate its appropriate diagnostic task to confirm information about the error. Once an error is detected, the system can take appropriate recovery actions. A faulty unit cannot crash the system, as the majority voting technique will mask the errors and prevent the system failure.

SIFT is a loosely coupled distributed computing system. Its fault-tolerant strategy starts with error detection and analysis. After identifying the faulty unit, the system will be

reconfigured to prevent that unit from having any further effect. The faulty unit will be isolated and will be replaced by using specially designed redundant buses and units. If the faulty unit is a processing module, then the tasks that were assigned to it will be reassigned to other modules. If it is a bus, the processors will request their data over other buses. After reconfiguration, the system will be able to withstand a new failure, assuming that there are enough non faulty units remaining. In SIFT, all these error detection and analysis and system reconfiguration functions are performed by software.

3.6 Mars

The Maintainable Real-Time System (Mars [Kop89; KM85]) was designed for real-time control applications where reliability, availability and safety are of paramount importance. The architecture of a Mars cluster is shown in Figure 3.7.

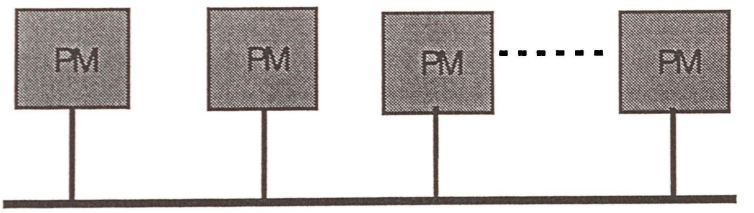


Figure 3.7 A Mars Cluster

It consists of a number of processors which are assumed to have fail-silent semantics, connected via a TDMA bus, called Mars BUS. A number of Mars clusters may be connected in a hierarchical structure (not shown).

The clocks of all processors are synchronised to create a fault-tolerant global timebase called system time. The scheduling of tasks is pre-planned and conforms to one of a finite set of schedules, which may be changed to give the system a new phase or mode.

Processes communicate using state messages and event messages. The semantics of a state message are similar to that of a global variable. That is, a new version of a

particular state message updates (overwrites) an old one and state messages are not consumed when read. The semantics of an event message are similar to that of a conventional message. These event messages are queued and an event message is consumed when read.

All messages carry a validity timestamp which, when it expires, causes a state message to be become invalid or an event message to be discarded.

Mars uses active replication. If processes were allowed immediate access to messages, state divergence could occur, because message arrival is asynchronous with respect to process execution. Instead, all messages are buffered. Periodically, a clock-driven interrupt routine is executed which acts on all buffered messages to create a new image set, representing a new global state.

Process scheduling, bus scheduling and clock-driven interrupts are all synchronised. So when process replicas perform a read operation, they will all read from the same image set and state divergence will be prevented.

3.7 FTMP

The Fault-tolerant Multiprocessor (FTMP) [Sei82; LAB+90] was designed as part of a commercial aircraft fly-by-wire program. The physical architecture of an FTMP system is shown in Figure 3.8.

Like SIFT, it could be composed of an arbitrary number of processors, each processor, P_i, with its own local or cache memory M_i, collectively termed a processing module. Unlike SIFT, however, only processors are connected to the multiple data busses, the memory remains private to its attached processor. Instead, there are an arbitrary number of global memory modules, also connected to the redundant serial buses and accessible to all processors. Any module, whether a processing or memory module, may receive

data on all busses but may send data on only one (the choice of this bus is quasi-static but reconfigurable). Modules are logically grouped into triads such that each member sends its data on a different bus. Every module contains a voting element to mask bus disagreements.

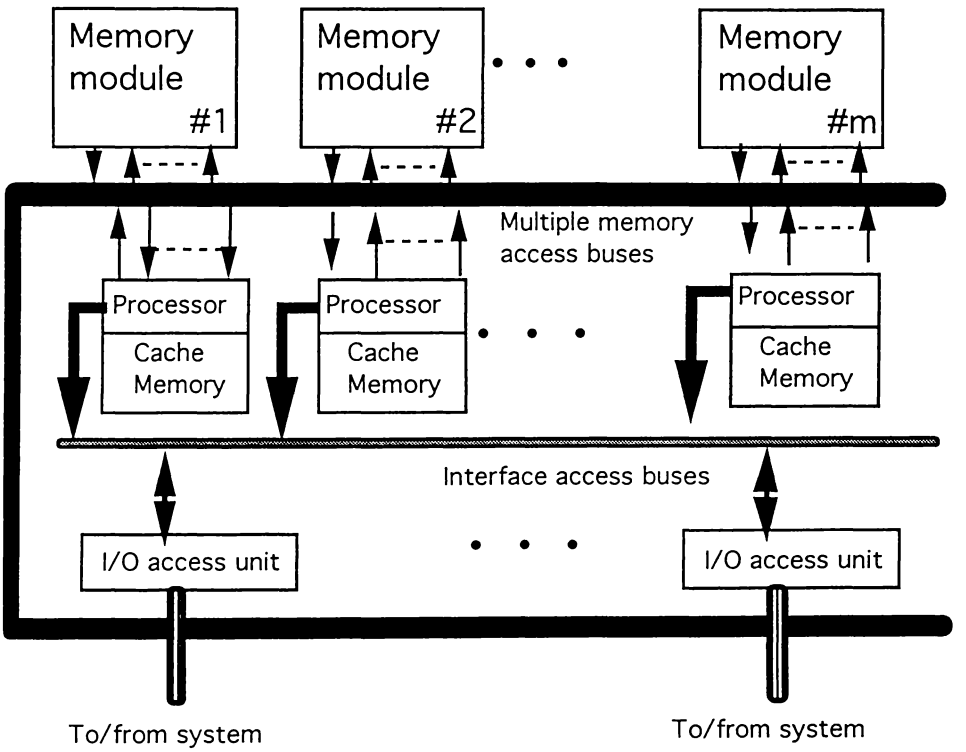


Figure 3.8 The simplified physical architecture of FTMP

Every processor executes exactly one process (at any given point in time). Every process is replicated (to degree of three) and each replica is executed on a different member of a processor triad. When a process triad sends a message to another triad, three message are sent, one from each replica. These three messages will necessarily be sent on different busses (a bus triad). Each replica in the destination triad receives all three messages, one from each bus, combining them in hardware to form a majority. As with SIFT, the failure of a single process replica (through the failure of its host processor) or the failure of a bus will be masked. Although the majority is formed by hardware, each member of the destination triad has its own voter, so a failure of the voter will be treated as a failure of the attached module; there is no trusted hardware.

All module clocks are tightly synchronised using a fault-tolerant hardware clock. So the processes of a triad remain in lock-step execution and will perform data input and output requests synchronously. Although process triads operate independently, their access to the multiple busses is strictly controlled such that the processors of a triad are granted access simultaneously and exclusively, even if the two bus triads concerned have no common members (as shown in Figure 3.8 for process triads). That is, at any point in time, there may only be one active bus triad.

Fault detection is carried out by testing the data on arrival for disagreement at the voters of a processor triad. It discovers the module, bus or other isolated element in which the failure resides. The recovery mechanism in FTMP is to reassign the task on a failed unit to a substituter, and then initialise it. The unit could be modules, or voter and transmitter selection for buses. The recovery actions are accomplished by the bus guardian units upon receipt of commands from active triads executing system software.

Although there are multiple busses as in SIFT, the enforced sequentiality of bus traffic limits the bandwidth to that of a single bus no matter how many busses are available. This bandwidth would be insufficient for large systems. FTMP also achieves fault tolerance by using active replication in a degree of three; it employs three times the resources normally required by the application, plus an arbitrary level of spares, plus the hardware and software overhead necessary to manage the redundancy, i.e., fault detection and isolation, reconfiguration, and recovery. The cost for this triple redundancy limits its extensibility to general applications.

3.8 MAFT

The Multicomputer Architecture for Fault-Tolerance (MAFT) system [KWF88; WKF85] is a distributed computer system designed to combine extreme reliability with high performance in a real-time environment. It is similar to SIFT in that it consists of a number of semi-autonomous processors connected via multiple busses. MAFT

employs active replication and design diversity to achieve majority voting for its input and output. It is assumed that any error is possible, no matter how malicious (arbitrary fault semantics), such as multiple coincident faults, "Byzantine" faults, and built-in or "generic" faults.

Each node in MAFT consists of an Application Processor (AP) and a hardware intensive Operation Controller (OC). The APs which may be heterogeneous, execute application processes. The OCs execute all the voting and consistency algorithms and are responsible for process scheduling on the APs. Processes are replicated to a degree specified by the application and each replica is executed on a distinct AP (as an alternative, the replicas may be diverse software implementations).

OCs communicate using messages. Each OC transmits on a different bus but may receive on all buses. Every transmission by an OC thus constitutes a broadcast (non-atomic) to all other OCs. Each OC also communicate with its own AP.

An application process executes in serial atomic actions. The clocks of the OCs are loosely synchronised through the exchange of system state (SS) synchronised messages (and pre-synchronised messages) whose transmission marks an atomic boundary. Thus all atomic boundaries of all OCs are also loosely synchronised.

The interactive consistency algorithm in MAFT acts by interprocessor message exchanging among all nodes to reach agreement on the existence and content of each branch condition (BC) message transmitted, even in the presence of a single malicious (Byzantine) failure. Each OC may then execute any deterministic scheduling algorithm which uses the interactive consistency information, and the consistent scheduling algorithm will be maintained.

The OCs maintain a form of distributed shared data memory on behalf of their AP

processes, so a request for data from an AP may be satisfied immediately. When an AP process wishes to modify part of this data, it sends the new value to its OC. The OC forms a data message by appending to this value a data identification descriptor (DID) which uniquely labels the data in the message then broadcasts the message to all other OCs.

When an OC receives a data message, it performs an acceptance test. Messages deemed to be correct are passed to a voter. The voter performs an instant vote using the new data message and any other data messages previously received from other replicas. The voting algorithm is based on median voting to cope with the inconsistent values produced by diverse software implementations of the process replicas. It is conceivable that a faulty OC could broadcast inconsistent values to other OCs. This could result in a difference in the outputs of their voters. But since all data messages must undergo an acceptance test, voter outputs would be bound to an acceptable range of values. Thus approximate agreement is achieved in the presence of arbitrary (Byzantine) failures.

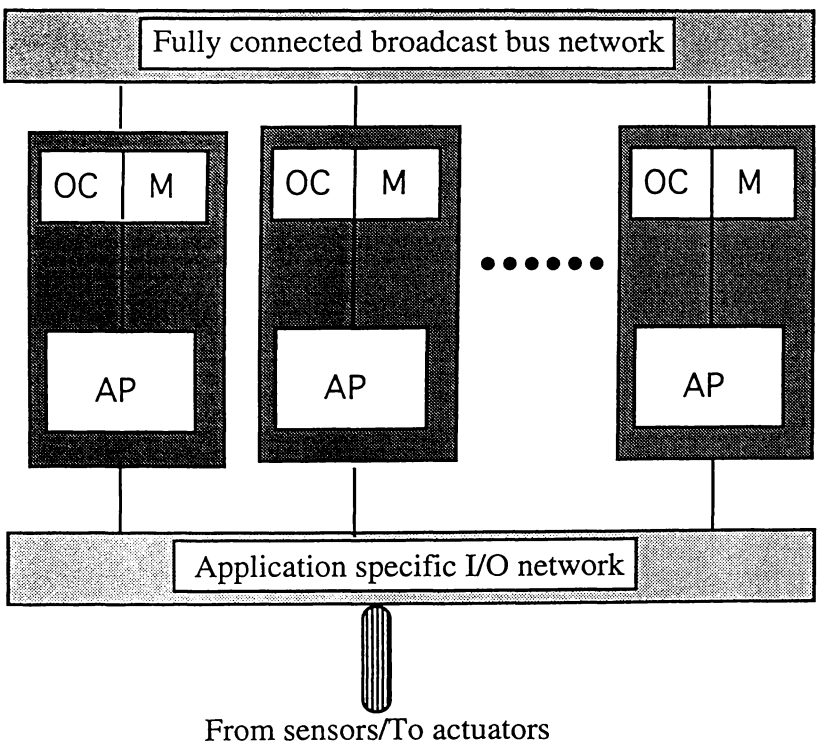


Figure 3.9 A simplified MAFT system architecture

The MAFT OC uses an interactive consistency algorithm to determine when a given process replica has terminated, and to agree on the BC it returned. The value of BC returned by a given process replica need not be consistent among non-faulty replicas.

The contents of all internode messages are protected by a strict link protocol and an error control code (ECC). However, the ECC and link protocols cannot guarantee the detection of individual corrupted messages, but they do probabilistically preclude the persistent undetectable corruption of multiple messages.

The MAFT system shown in Figure 3.9 has a highly reliable distributed architecture designed to provide computation in real-time control systems. Its majority voting and design diversity strategies for fault masking require multi-version hardware and software modules. This method is suitable in small systems for very critical applications. However the high cost, due to the extensive redundancy, makes the architecture difficult to extend to large array architecture systems.

3.9 Summary

We has reviewed several commercial fault-tolerant systems that use a variety of techniques to improve the reliability in distributed systems. The common points of these systems are that they all require spare hardware redundancy for replication whether passive or active, and all fault-tolerant schemes are developed on a common media bus communication network. The degree of reliability that each system aims to achieve varies, and it is orientated to a specific application. For data transaction processing, reliability is considered to be more important than performance and cost, and for real-time control applications, both reliability and performance are more important than the cost of building the systems.

Fault tolerance in those systems, like SIFT, FTMP, MAFT, is achieved through modular redundancy and voting of data from replicated tasks. The issues of concern in

these active replication and majority voting on I/O data are intertask or internode clock synchronisation, efficiently organising redundancy, error detection and system reconfiguration. A single error or single unit failure is masked out and will not affect performance. Once an error or failure in a unit is detected, a non faulty unit will be switched in and replace the faulty unit, so there is no requirement to roll the program back and restart its execution from a previous state. These active replication systems sacrifice significant performance overhead to gain reliability.

Hardware duplication with a mixture of active and passive process replication is used to improve fault tolerance in Sequoia and Stratus. These systems provide immediate detection of all single errors resulting from hardware faults by matching operations on replicated processors. Recovery processing is also arranged in hardware to disconnect failed components physically, then resume the execution on duplicated components. This hardware operation for failure detection and failure recovery may achieve high reliability in systems which are relatively small in size and suited to applications of data transaction. In a large distributed system with array architecture, any interprocessor action has to be distributed over the network and dynamically reconfiguring the network connection is very complex. Such complicated operation could introduce more unreliable factors into the system. Furthermore, an algorithm implemented in hardware is difficult extend to a wider arrange of applications.

In the Tandem fault-tolerant system, active replication of processing would not be cost effective. So, Tandem took the form of passive replication with assumption of fail-silent semantics. It constructs the system by using redundant pairs for all devices (including processor, communication buses, I/O controller, power supply and memory. Any single point of failure can be eliminated by invoking redundant devices through an alternative access routine. The Tandem system has reduced the number of components for redundancy required for active replication of majority voting. But in the absence of failure, half its system devices contribute nothing to the process performance. This

shortcoming can be optimised in a parallel and distributed system with grid connection by taking mutual passive replications. The details of new passive replicating scheme and its recovery algorithm will be discussed in chapter 6.

Every system has its features in hardware architecture, way of redundancy, and interprocessor communication protocols. However, communication between processors is through common media networks and communication speed between any two nodes is topology independent. There is no requirement of establishing a channel before transmitting messages to an internal or external processes. Every site is an immediate neighbour to all other sites. Total communication bandwidth is fixed by bus technology. Individual site failure does not always crash whole systems.

In an interconnected grid topology network, communication between processors is through point-to-point message passing. Each processor can only send a message directly to its immediate neighbours. Otherwise, the message has to be relayed by intermediate processors. Message travel time from initiating node to destination node is very much topology dependent. Channel establishment is required before any communication. Communication media failure can only affect message passing between directly connected nodes. Because of arbitrary topology, communication is very complicated and can become a significant overhead to system performance.

Because of the differences between common media network distributed systems and interconnected grid topology distributed networks, fault-tolerant algorithms used in common media network systems cannot be efficient, and even not feasible in point-to-point message passing system. The efficiency of the distributed diagnosing, checkpointing and synchronising for recovery cannot be discussed without considering the specific system architecture.

From fault-tolerant systems discussed in this chapter, failure or error detection methods

used for diagnosing, may be efficient on bus networks, but they are difficult in array architecture systems. Because the number of the communication routines in array multiprocessors varies between nodes, and an intermediate node failure or fault will affect correct diagnosis. The wide communication bandwidth provides a better structure for diagnosing processes to worm through the entire system in parallel. In Chapter 5, we will introduce a distributed diagnosis algorithm which can detect and locate the failure given multiple node failures. The desired result of the algorithm has been demonstrated on a parallel system for numerical weather predication.

The checkpointing mechanisms used in the Tandem Non-Stop system were well suited to the situation of no cross processor dependency, but their efficiency is limited in parallel processing multiprocessor systems. Checkpointing used in Sequoia is for shared memory architectures, and it is achieved simply by duplicating process states to multiple memory sites. In array architectures for parallel processing, checkpointing algorithms introduced in this chapter cannot guarantee that error-free states are globally consistent. We will present a new consistent checkpointing algorithm in chapter 6, which can be used to establish global consistent states in multiprocessor systems and be executed in a parallel manner so that it would not affect system performance significantly during error-free execution. When failure occurs a failure recovery algorithm, introduced in Chapter 6, can remove the failed component and resume normal processing with graceful degradation.

Once again, the analysis of fault-tolerant techniques used in some existing distributed systems in this chapter is part of investigation of previous similar efforts, more related work about distributed diagnosis and consistent checkpointing in multiprocessor systems will be introduced when discussing our algorithms in chapter 5 and chapter 6.

A successful fault-tolerant system using passive replication techniques must be supported by an effective error detection algorithm, because a passive replicated

process cannot be invoked to take over a failed task without failure or error being correctly diagnosed. As mentioned earlier, in a bus-oriented network, an individual site fault would not affect diagnostic processing on other sites. This is not true in interconnected multiprocessors with point-to-point message passing. An error on an intermediate node could lead to a wrong diagnostic result and mislead the recovery action. Therefore, we will discuss a distributed diagnostic algorithm for multiprocessors in Chapter 5. In the next chapter, we present a fault-tolerance strategy in the numerical weather prediction parallel system designed for the New Zealand Meteorological Service.

Chapter 4

A Fault-tolerant NWP system

4.1 Introduction

In this chapter, we present a fault-tolerant multi-transputer system to improve the reliability of a Numerical Weather Prediction (NWP) application [GT90, GTP90; GTP91]. This system was in daily production for weather prediction at the New Zealand Meteorological Service, operating correctly for more than three years. It detects and locates errors and faults that emerge, automatically removes the failed components, then resumes operation.

The Meteorological Service is moving towards ever more reliance on automated systems, so that reliability becomes critical. Customers expect products to be delivered without fail, and no requirement for staff to cope with frequent problems. The NZMS requested an effective diagnosis program which is able to detect any hardware failures as soon as they occur and get the system repaired with a short delay, in order that data processing for timely prediction can be continued from the last processing state. Our approach to this problem is divided into two stages.

Firstly, the diagnosis process is invoked while initiating the system. This diagnosis is intended to detect static failures and remove them before loading the application program. Static failure is a failure which previously exists. If a static failure is detected at this stage, the removal of the failure is accomplished by isolating the failed

processors and links logically and spreading application programs over available resources.

Secondly, the diagnosis process is invoked periodically during the execution of the application program. Diagnosis at this stage is intended to detect dynamic failures, then terminate the processing and restart the system. The procedure of removing failures and restarting the system is the same as that used at the first stage. Processing can be resumed on the repaired system with the processing state that was terminated to deal with the system failures.

Terminating the system and restarting is not always an efficient method for recovering dynamic failures. However, it is OK if the application program is executed in an iterative manner on altered data and allows a short time delay for recovery. The weather prediction program used in NZ Meteorological Service is one of this kind of applications. We introduce more details about this application model and system architecture in the following section before going to the development strategy.

4.2 System model

The New Zealand Meteorological Service uses forecasting products from several global weather models run on computers overseas. It also runs its own Numerical Weather Prediction system, which is mainly used at present to produce timely analyses of current conditions for use by weather forecasters. The parallel processing system installed at the NZMS for this NWP model was implemented on a transputer system.

A transputer is a programmable VLSI component with communication links for point-to-point connection to other transputers [Inm90]. A transputer module contains memory, a processor and four standard external hardware links. A transputer-based system has asynchronised computation and synchronised inter-processor communication.

4.2.1 The transputer network architecture

The hardware system for the parallel NWP was composed of 9 transputer modules as shown in Figure 4.1, each with a T800 processor and 2 Mbytes of memory. These transputer modules reside on a series of Transtech motherboards connected through a Caplin Cybernetics QT0 Q-bus/transputer interface to a MicroVax computer which acts as the host processor. The transputer system that supports the numerical weather prediction is arranged in a toroidal mesh, with C004 link switches which are used to make other connections between the processors.

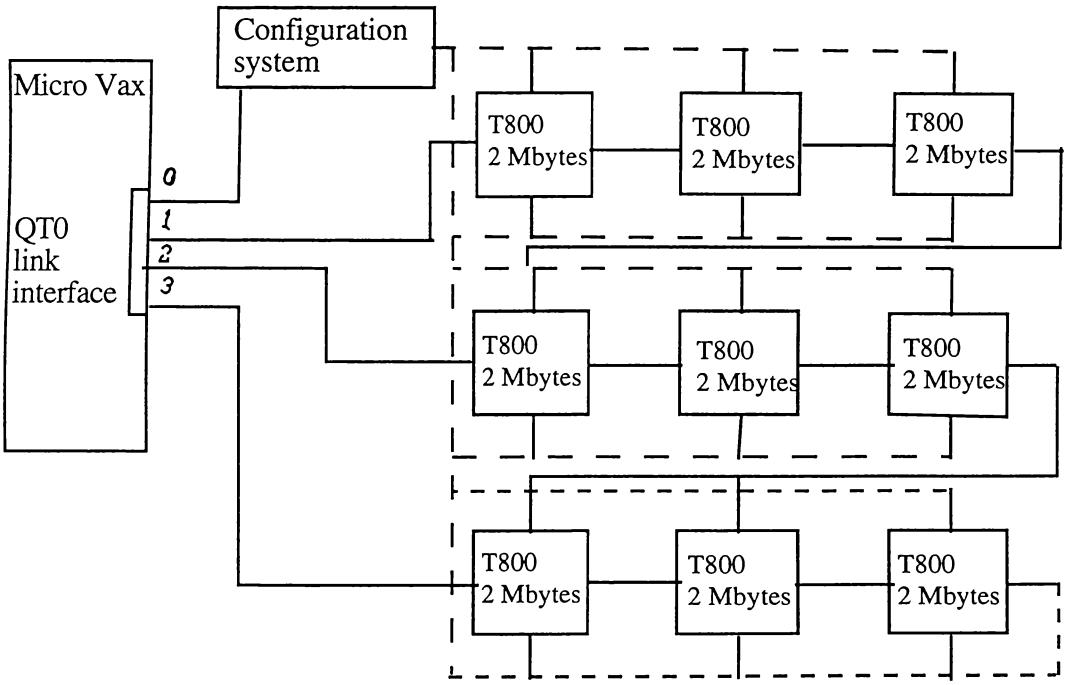


Figure 4.1 Nine transputer network with toroidal mesh topology.

Dashed lines in Figure 4.1 show connections into C004 link switch configuration system, solid lines are hard-wired links. Each of the horizontal pipelines of processors can be extended indefinitely.

4.2.2 The characteristics of parallel processing for NWP application

The parallel NWP program consists of five working units: analysis code, database, workfile utility, multiplexor, and transputer server. Its processing model is

geographically decomposable - parallelism could be set up to deal with separate geographical areas without the need for any communication between the processes. This application is arranged in two phases; analysis and prediction. The code for the analysis phase updates the atmospheric model with observations taken in the preceding few hours. This updated model is then used as input to the prediction program, to give a picture of the atmosphere as it is expected to appear at say 6, 12 and 24 hours in the future. Each transputer runs a copy of the analysis code.

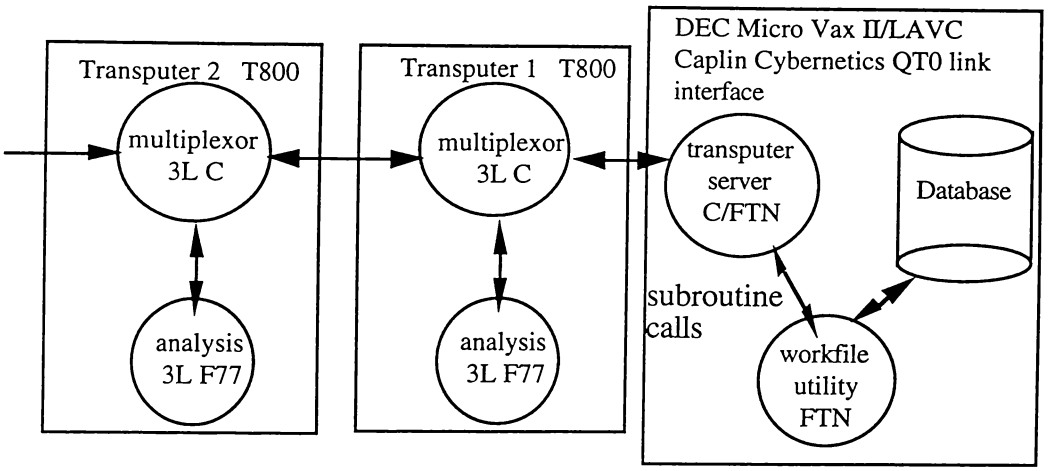


Figure 4.2 Operational system software configuration

The database is used to contain the atmospheric model, prediction and observations. The workfile utility is a database management system that controls the access to this database. The transputer server implements a subset of the afserver (INMOS alien file server) protocol, to which has been added a workfile message protocol. Messages from the analysis processes are passed by subroutine calls to the workfile server frontend. A multiplexor process was introduced between multiple copies of the analysis code and the workfile server. Subroutine calls to the workfile server in the analysis program were converted into messages that were passed to the multiplexor and relayed by the transputer server.

The transputer server merges requests from the various analysis processes and passes them on, again by a mailbox, to the frontend of the workfile server. This frontend converts the messages back to the correct subroutine calls to the workfile, and then

returns the data back through the transputer server and multiplexer to the originating analysis process. Each copy of the analysis program is passed parameters which tell it how many copies are running on an individual processor and which one it is. It can then calculate the geographical area for which it should update the model.

In an operational system model as shown in Figure 4.2, each transputer runs a copy of the analysis code, written in 3L's Parallel FORTRAN 77. Input and output from these programs goes through a multiplexor process, again on each transputer, and is passed by the QT0 driver to a transputer server (written in VAX C and FORTRAN), running on the MicroVAX. This server acts as central controller which allocates tasks evenly to each individual processor, and forwards the processed results or requests from each processor to workfile server through corresponding routine. The MicroVax is part of a Digital Local Area VAX Cluster (LAVC), and has access to disk resources anywhere on the cluster.

The analysis code is in 3L FORTRAN 77, an extra procedure in the message path is used to convert the VAX floating point format to and from the IEEE 754 format of the transputer. Mailbox functions were replaced with channel input and output, using an extension of the afserver protocol to support long messages to the workfile. This protocol allows the transmission of data records up to 16 Kbytes in length in our application to reduce data transfer overheads.

When a process makes a request to the workfile server it waits for a reply, and no other process may use the server while this handshake is in process. Normal FORTRAN input and output are also used by the analysis processes, for such functions as creating and writing to log files, and for error messages. Care has been taken that no two processes attempt to create files of the same name. The analysis processes identify themselves by using the binding to message ports which is possible in 3L's FORTRAN.

The multiplexor process is written in Parallel C, and runs with high priority. It has to handle message input and output as well as FORTRAN input and output from the analysis programs. The process creates two threads, one of which waits for input from the neighbouring transputer, the other for input from the analysis program on the same transputer. If a higher degree of interconnection of the processors were used a thread could be created for each additional input. When an input is received a common protocol function is called, which forwards the message to the server program, and waits for the reply. Thus, one communicating transputer will lock out all other transputers from the workfile server until the whole transaction is complete.

This is quite reasonable, as the present version of the workfile server can only deal with one request at a time. Sometimes several messages must be sent to the workfile server before a reply is expected. This case is managed by adding a lock command to the multiplexor protocol. The lock command causes the multiplexor to listen on one channel only, and to pass the command on to the next multiplexor in the chain. The transputer server program need take no special action on receiving the lock command. When the transaction is complete the analysis program issues an unlock command, and the multiplexers resume their normal mode of operation. Of course, a process that issues a lock command and fails before issuing the unlock command will cause the whole system to deadlock.

The code for the entire transputer system was developed for a simple configurer/loader which will load an arbitrary network with a set of identical processes [Sch90]. This does not require a separate configuration step, and the loading process is faster than loading a configuration file for networks of more than two transputers.

The transputer server is an implementation of most of the afserver protocol in VAX C and FORTRAN 77, with some modifications to enable it to deal with messages from multiple transputer processes. Commands have been added to the protocol to open

communication with the workfile server, and for input and output to and from the workfile. The workfile is opened by the first open request; subsequent requests from other processes are not ignored but are counted, so that the transputer server knows how many copies of the analysis program are running. In the same manner, termination messages from the analysis programs are counted, and the transputer server terminates when it detects that all the analysis programs have finished. More details of the multi-transputer numerical weather prediction system can be found in [GTP91].

4.2.3 Reliability issues

The possibility of a failure in such parallel processing system as described above increases as the number of processors grows so that parallel high performance power cannot be obtained without loss of reliability. Failure of any component could cause the whole system to crash, so that reliability is as important as system performance. To improve the reliability for this NWP system, we first studied most probable failure modes then modified the software so that it is tolerant of these failures.

4.2.3.1 Failure modes:

The failure modes we are concerned with in this fault-tolerant NWP system are two types: processor failures and communication failures. The effects of the failures can be analysed as follow.

Processor failures

(i) If the *i*th transputer fails during the booting and loading period, the whole system will fail, with the booting process timing out after several minutes.

(ii) If the *i*th transputer fails during calculation time, then transputers located after the *i*th position can neither receive data from the workfile, nor send results back. The transputer server will wait indefinitely for a terminating message that will never come.

(i) Link failure:

There are three kinds of links in the system, links from the QT0 interface to the transputer system, the link between the two motherboards, and links (only two out of four being used) on the motherboards between processors. If any one of these links fails the system will fail; the effect is similar to processor failure.

(ii) Soft communication error.

If a message has an error from noise or a memory problem, it could render the analysis model results incorrect. The present system includes a checksum in each message designed to detect communication errors, but any such error results in termination of the computation. Thus a single soft communication error results in the failure of the system.

4.2.3.2 A Petri Net model for NWP system

We have emphasised above failure due to hardware problems. In the system before applying the fault-tolerant scheme, a failure in any hardware component led to complete failure of the system. If any of its components fails, no further processing can take place. A static failure causes a system crash during loading with analysis codes and multiplexor into the transputers. A dynamic failure makes the system hang infinitely without any further progress. This situation is depicted in a Petri net model as shown in Figure 4.3.

In this figure, circles indicate the processing states and rectangles indicate transitions between processing states. The processing states are labelled p_i (i=1, 2, ..., 6) and transitions are labelled t_j (j = 1, 2, ..., 6). The token in the circle represents the current processing state. The token moves while the program is executing. The strength of

each directed arc (p_i, t_j) or (t_j, p_i) is denoted respectively by $s(p_i, t_j)$ or $s(t_j, p_i)$. In this case, $s = 1$ for all situations. $\forall j$, $j \in (1, 2, ..., 6)$, if a transition t_j has a token in its source, t_j is ready to fire a transition from source state to sink state. If the token is in a circle which has two output branches, there is a relation function F which directs arcs from states to transitions. In the graph depicted in figure 4.3, F is associated with the states p_3 or p_4. If an error occurs in either of these states then transitions are made to t_5 or t_6 respectively, otherwise transitions are made to t_3 or t_4 respectively.

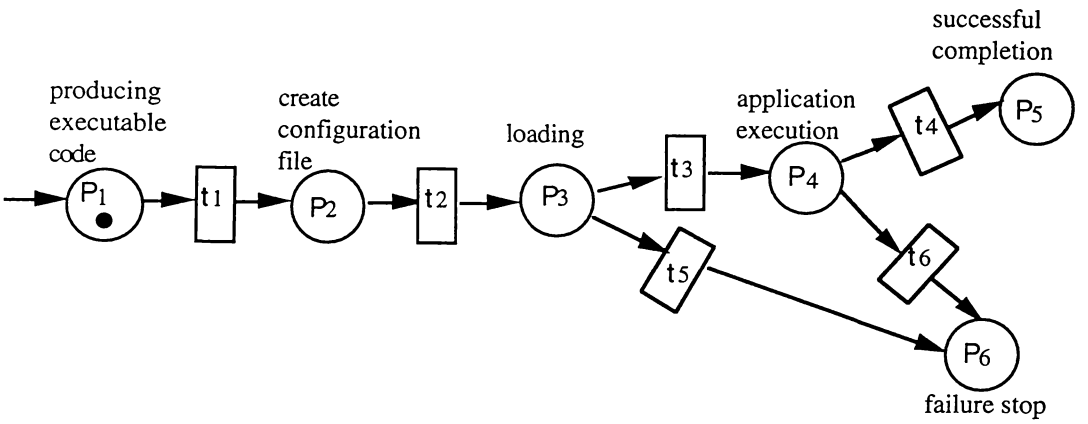


Figure 4.3 A Petri net model for the parallel NWP system without fault tolerance

From Figure 4.3 we can see whenever the network configuration or software program changes, the user has to modify the configuration description file (CDF). Because it is difficult to change CDF automatically, if failure occurs the system is unable to restart the service after the failed processor is isolated. If the system falls into the failure state, it will hang indefinitely until the failed state is manually removed by the user.

4.3 System modifications to improve reliability

We have identified some of the possible failure modes of the present system. In this section we present our approach to enable these failures to be diagnosed and to allow the system to continue in operation in the presence of partial failure.

The failure modes we have identified are:

(1) Processor failure.

(2) Hardware link failure.

(2) Communication soft error.

It is impossible to have approaches which are completely successful in every condition. A proper fault-tolerant technique directly affects system reliability and performance. Our approach is to diagnose the system to determine system resources before an application is run, then periodically diagnose it while the application programs are executing. It terminates its performance if a failure is diagnosed, and restarts the system again after isolating the failed component. The design of this fault-tolerant NWP system consists of four distinct executable images:

- A *scheduler* which is always executing on the host. It controls the submission of the tester or workfile server-job to the transputer network.
- A *diagnoser* which is run periodically by the scheduler, testing transputer memory and links and returning the configuration of available resources.
- A *starter* which is invoked by the user whenever the workfile server (or possibly some other application) is to be run.
- A *server* which is started by the scheduler in response to a request from the starter program.

The relationship between these modules is illustrated in Figure 4.4. They communicate via three mailboxes: a permanent (system-defined) mailbox, and two temporary mailboxes created by the scheduler prior to spawning the diagnoser and server processes. The scheduler, diagnoser and server are in the same process group, but the starter can be invoked from any source (and hence the need for a permanent system-defined mailbox). The scheduler is always executing, but only one of the diagnoser or server processes can be executing at any time.

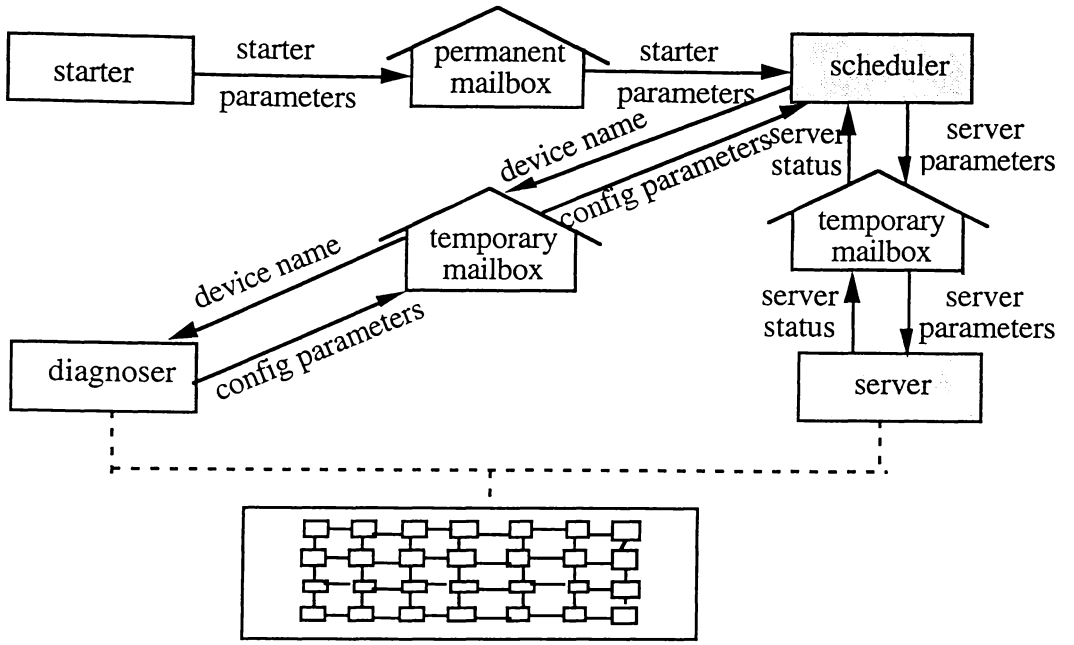


Figure 4.4 Relationship between the four image modules

These four image modules shown in figure 4.4 also can be categorised into two groups: Network controller (scheduler, starter and server) and distributed network exploring program (diagnoser). We explain the functions of these program modules in the two groups respectively in the next two sections.

4.4 The network controller

Three distinct executable images: scheduler, server and starter in the group of network controllers run on the VAX computer. Only one module provides a service at any one time. This synchronisation is controlled by the module scheduler. The details of their services are as follows.

4.4.1 The Scheduler

As mentioned above, the scheduler is a program which controls other NWP program modules. It is always executing unless the diagnoser or server images are active. The scheduler is responsible for detecting any server requests from the starter program, scheduling the execution of diagnoser and server programs, and maintaining a log file. The scheduler has been designed to continue running indefinitely, irrespective of

whether the diagnoser or server programs execute correctly.

The scheduler is invoked on the VMS command line with three optional parameters: wait_time, device_time and diagnoser_timeout. The wait_time is the time interval between diagnoser invocations; the device_name is the QT0 device name; the diagnoser_timeout is the time before the diagnoser is deemed to have failed. As illustrated in Figure 4.5, the scheduler begins by initialising the user-interrupt handler and obtaining the command line parameters. The scheduler then opens the transputer log file and assigns a channel to the permanent mailbox. It also empties the mailbox of any old server requests. It then loops indefinitely, periodically diagnosing the transputer network by invoking the diagnoser program. It also listens on the permanent mailbox for server requests. If a server request is received, then the server program is run using parameters supplied by the starter, in conjunction with configuration information supplied by the diagnoser.

Figure 4.5 is in fact a slight simplification of how the scheduler operates. In particular, the scheduler will not respond to a server request if the diagnosis program detects *zero* transputers in the network. Also, if the server program fails to execute correctly then the scheduler will re-diagnose the network then re-load it again, repeating this process until the server completes successfully. After each invocation of diagnoser or server program, execution is delayed for a few seconds — this allows time for the spawned image to run down, so that no more than one subprocess is present at any one time.

The wait between diagnoser invocations is implemented in the event-driven style using an asynchronous read on the permanent mailbox. First a scheduled wakeup is initialised, then after the read is queued the process is sent into hibernation using the VMS system services call `sys$hiber()`. The process is then woken by either the scheduled wakeup, or by the AST (asynchronous system trap) routine which executes when the I/O request actually completes.

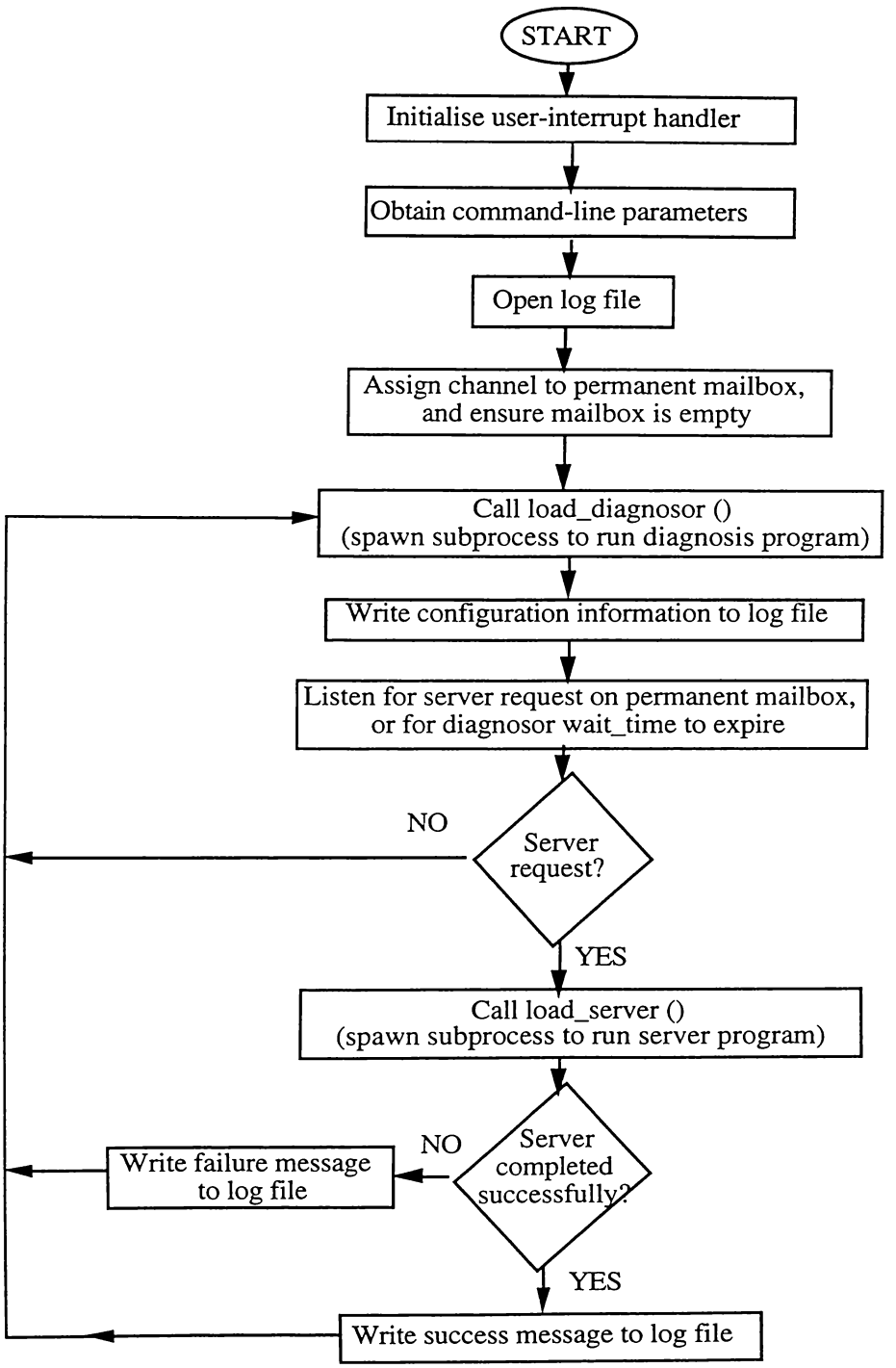


Figure 4.5 operation chart of the scheduler program

The above approach (an asynchronous I/O request in conjunction with a scheduled wakeup) is used throughout the scheduler program when communicating with the

diagnoser and server temporary mailboxes. It allows failure of any I/O request to be detected by simple examination of the I/O status block immediately after the process has been woken. However, care must be taken to ensure that both scheduled wakeups and pending I/O requests are cancelled when no longer needed, otherwise the AST routine's wakeup can re-activate a subsequent sys$hiber() statement before its own I/O is complete. Of particular note is that the I/O dequeuing request sys$cancel() also causes the AST to fire (and thus causes a sys$wake() to be issued). This is implemented in the scheduler using sleep (CANCELL_TO_TIME), where CANCELL_TO_TIME) is normally a few seconds.

Loading the diagnoser

Loading diagnoser is dominated by the main loop of the scheduler program (see Figure 4.5). It runs the transputer diagnoser program by creating a diagnoser subprocess, communicating with it via a temporary mailbox. This isolation of the diagnoser program allows any failure of the tester subprocess to be detected by the scheduler (using a timeout), and hence allows the scheduler to continue execution in the event of failure.

The diagnoser loading procedure is illustrated schematically in the left of Figure 4.6. It begins by creating a temporary mailbox, then obtains the current terminal name and creates a spawned diagnoser subprocess. The terminal name is passed to the sys$creprc() (create process) system call, so that any messages which the diagnoser subprocess generates will be echoed on the scheduler's terminal. If the sys$creprc() system call returns an error code then this may well be due to the presence of an idle diagnoser or server process---in this case diagnoser loading has to try again with after a short wait. It then sends to the mailbox a string (of type device_name_type) specifying the QT0 device, and waits for the diagnoser routine to execute and return the network configuration parameters. This diagnosing process takes 10-20 seconds,

depending upon network size, memory configuration, and link configuration. Both the sending and receiving I/O statements operate asynchronously, each having an associated timeout. If the diagnoser completes successfully (without timeout or other error) then the value OK is returned to the scheduler's main routine---otherwise an error code is returned.

Loading the server

The program for loading the server is analogous to that of loading the diagnoser. It is responsible for starting the NWP Workfile Server (or any other application program specified by the starter program). Once again, it does this by creating a subprocess and communicating via a temporary mailbox. If the server fails then the loading procedure will detect this and return to the main scheduler program, which re-tests the network then re-submits the server request.

The operation of loading the server is illustrated schematically in the right half of Figure 4.6. After creating the temporary mailbox and spawning a server subprocess, this loading event passes the starter parameters (read from a starter program connected via the permanent mailbox) and the configuration parameters (supplied by the diagnoser) to the server subprocess. It then attempts to read from the temporary mailbox, timing out and exiting if it does not periodically receive a "report-back" message from the server subprocess. The length of this reporting interval is specified by the report_interval field of the starter parameter structure.

The loading server process is also responsible for checking each "report-back" message to make sure it is a different one from the last report back message, and can thus detect if the server has stalled. When the server status indicates that the server has completed successfully (with status code SERVER_COMPLETED) then the procedure for loading the server returns with status OK.

The diagnoser and server loading operations have been slightly simplified in figure 4.6. In fact, all operations which have a possibility of failure (including creating mailboxes) are either guarded with a timeout or have their return status examined to see if they have been successful. This approach has hopefully made the scheduler completely "crash-proof", and thus capable of performing the diagnoser/server cycle indefinitely.

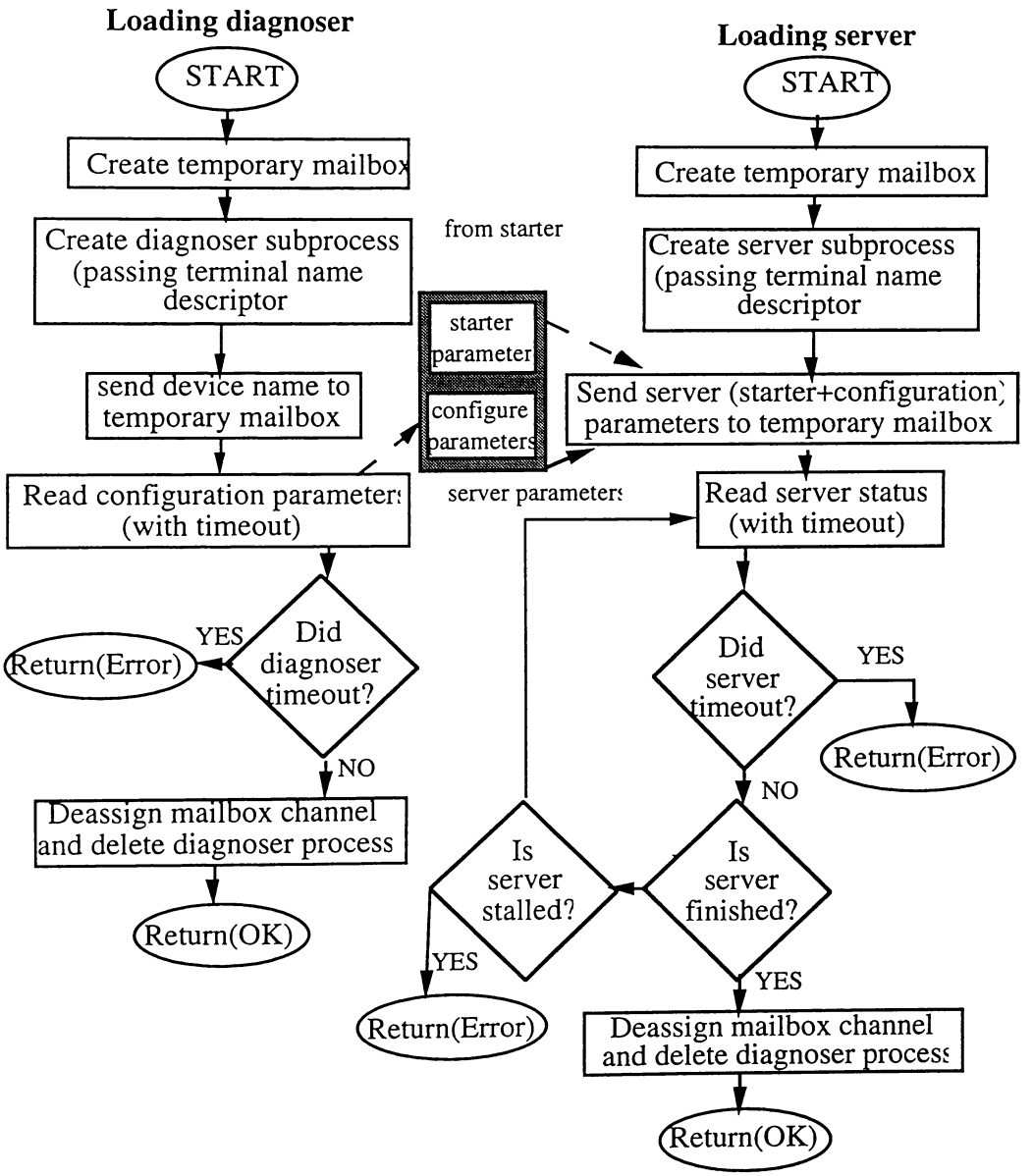


Figure 4.6 Schematic of load_diagnoser() and load_server() routine

4.4.2 The Starter

The starter is a simple program responsible for initiating a server request. The starter

does this by building a structure of parameters specifying details of the request, then sending this structure to the permanent mailbox. When this transmission is complete the starter exits, and further processing is left to the scheduler and server programs.

The starter program is activated by the user to submit an application task. The user provides information like: name of executable server program (e.g. "server.exe"), name of transputer diagnoser image file, time between NWP server watchdog messages, soft channel connecting multiplexor task to transputer network, time between server "report-back" messages, QT0 device name, and name of the NWP data file.

Although the starter program has been designed to invoke the NWP Workfile server, it could easily be used to start any other transputer-based application which can be loaded using the distributed loader.

4.4.3 The Server

The server program is responsible for initialisation and booting (loading) of the transputer network, and servicing of NWP Workfile requests. It executes as a spawned sub-process of the scheduler. Operation of the server is illustrated schematically in figure 4.7.

The server program begins by assigning a channel to the temporary mailbox already created by the scheduler, then reads the mailbox to obtain the starter and configuration parameters supplied by the starter and diagnoser programs, respectively. It then initialises a timer using the VMS system call sys$setimer(), which expires every report_interval seconds, generating a call to the AST (asynchronous system trap) routine timer_AST(). This routine simply returns the current value of the global variable server status, then resubmits the timer. Hence every report_interval seconds the workfile server code is interrupted and a status message is sent back to the scheduler

via the temporary mailbox.

Having initialised the timer, the server then creates a logical name equivalence between the logical name NWP_WORKFILE and data file name specified in the starter structure --- this logical name is used by the workfile server code. The server then initialises the network, and loads the network using a distributed loader. The distributed loader uses some of the parameters specified in the starter and configuration substructures, loading the transputer code into the longest pipeline.

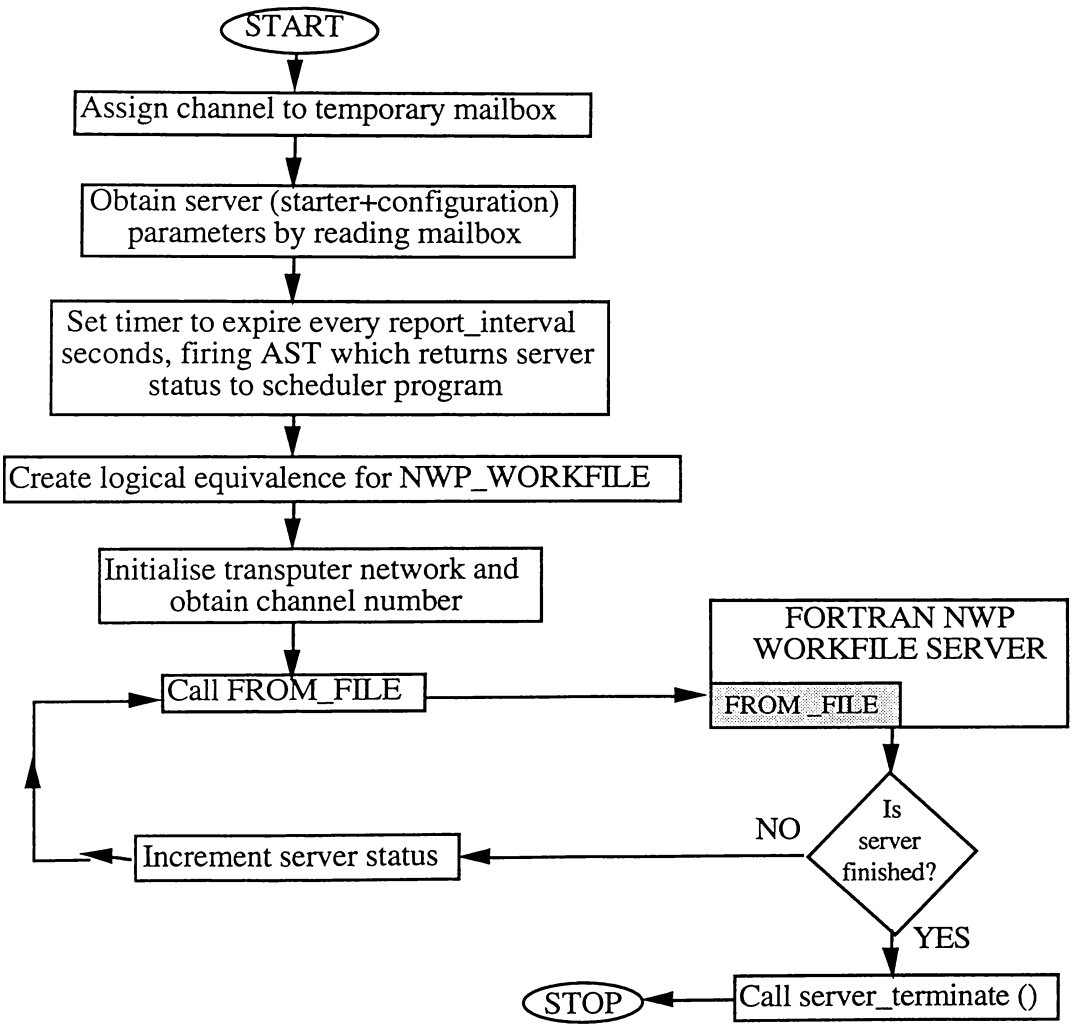


Figure 4.7 Schematic of server program

Finally, the server calls a routine in a continuous loop, incrementing the variable server_status between each call so that the timer AST routine reports to the scheduler that progress is being made. The transputer server is forced to exit only when the

transputer server routine responsible for reading messages from the transputer link detects that there are no more requests to be serviced. In this case, the server writes a success code (SERVER_COMPLETED) back to the scheduler via the temporary mailbox, then deassigns the transputer and mailbox channels and exits, causing the server image to run down. At this point the scheduler program reawakens and continues the diagnoser/server cycle.

4.5 Diagnoser

The diagnosing program is a tool which is used to detect faults on processors and on external links. In this NWP application, this diagnosis program is invoked before starting application program execution to worm through the processors and the links between them. After the network mapping, it produces a network configuration map as well as resource information, and then compares its findings with the topology produced by a C004 link switch program(CLSP). In such a way, all faults and errors are detected and located accurately in the system.

If a fault or error is found at this stage, the diagnosis program feeds its explored configuration map into the CLSP to reconfigure the network, thus isolating faulty components. It ensures that no faults can be present when the system is in use.

After a failure is detected and removed, the diagnoser program then runs again to check the new network topology. Alternatively, it waits for the user to get the problem fixed in order to start the application with the original configuration. A final version of the processor link map obtained from the diagnosis program is fed to an intelligent loader [Sch90], which loads the application code in an efficient manner without requirement of predetermined CDF.

The diagnoser begins by assigning a channel to the temporary mailbox created by the scheduler process. It then performs a (synchronous) read on the mailbox, obtaining the

name of the transputer device (QTAx: or something similar). After exploring and diagnosing the network, it returns the name of the transputers in the current functioning configuration and storing the associated link connections in a linear array. It then sends the results of the diagnosis back to the scheduler via the temporary mailbox. The entire diagnosis process is illustrated schematically in Figure 4.8.

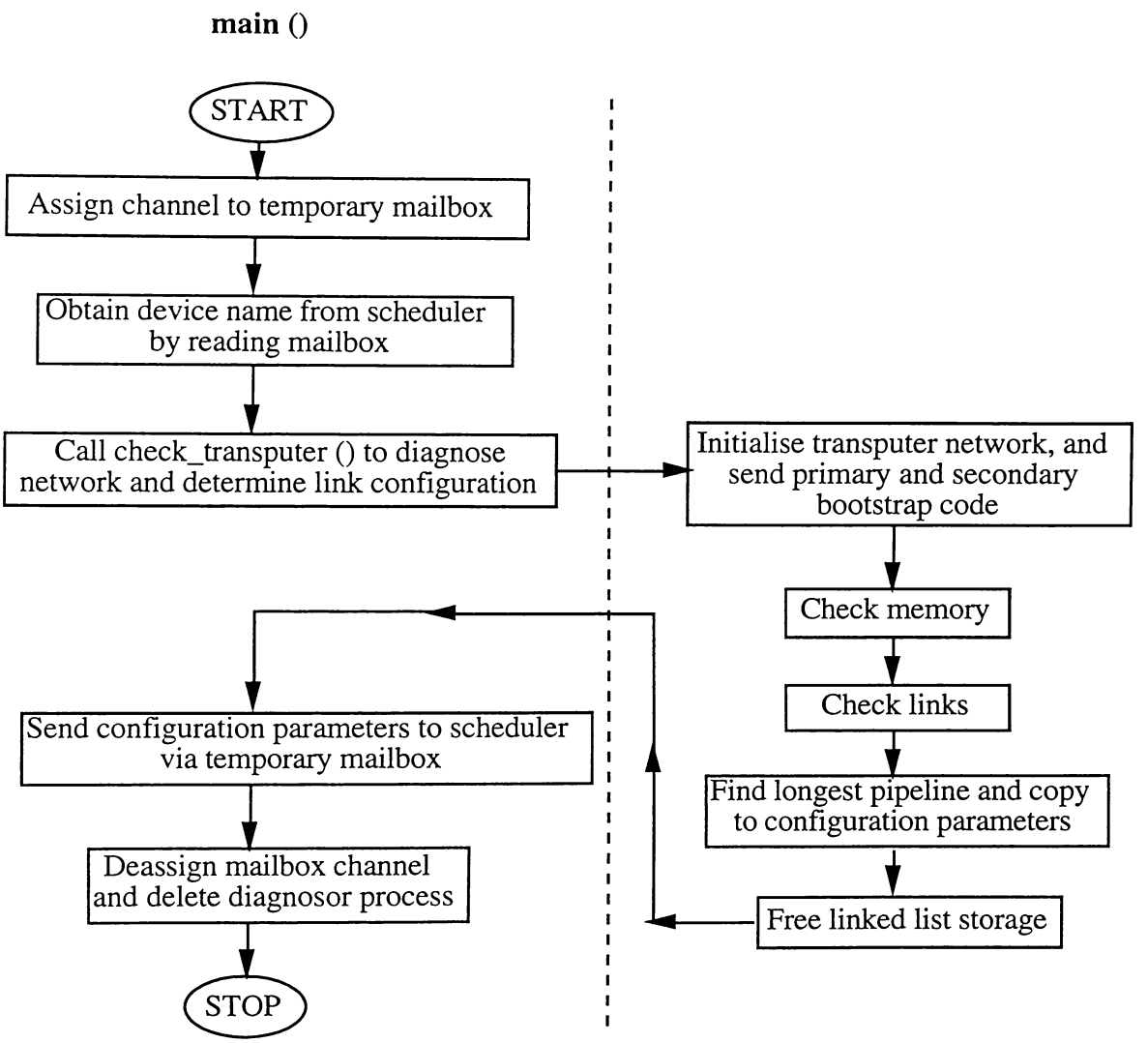


Figure 4.8 Schematic of tester program.

The diagnosing program consists of four sections, configuration explorer, filter, and memory and communication link tester.

The configuration explorer

The configuration explorer contains a primary bootstrap(PBS), and an exploring code. The primary bootstrap is used to match the transputer array, and it boots, loads and

numbers transputers in a distributed manner. Once the network is booted, the exploring code is called frequently to explore the connections to direct neighbours, memory, processor type, and communication information. It also does the testing on the processor it resides on and its neighbours. The primary bootstrap is always sent to an unbooted processor by exploring code, and after the bootstrap has booted a processor, it loads exploring code into this processor then finishes its job.

The configuration explorer on each node tests three links' (except the father link, through which the current processor was booted) connection status by using pokes and peeks with time out. If it cannot receive the data from a peek within a certain time (usually 5 ticks is the minimum) after poking, a time up process kills the poke and peek process to end infinite waiting. We assume that the link timed out connects to nothing. Information about configuration and communication routes are forwarded back to the server on the host machine for failure detection.

Filter

A filter on each transputer acts as a packet passing controller which is responsible for forwarding or accepting the message. Message forwarding is achieved by the filter's maintaining a list of transputer ids for each of its links. Only those packets destined for the filter's processor are accepted by the filter. The filter also interprets the accepted packet based on a protocol. The accepted message roughly can be classified as follows:

(1) Packet for poking and peeking. As network mapping is in depth first order, explorer code might poke and peek a transputer which is already booted. When a booted transputer receives a poke and peek packet, it ignores the message and replies to advise its booted situation and its id.

(2) Packet for terminating the diagnosing over the sub-branch of the network. The sub-net to each link of the transputer will send a termination packet back when all transputers have been booted and initialised over it. After receiving termination signals

from all three son links, the transputer sends the same kind of termination signal to its father transputer. Eventually the server on the host will receive termination packets over the entire transputer network; then it is ready for diagnosing memory and link communication errors. The route for relaying termination packets back to the server is exactly the same as reversing the booting network route.

(3) Packet for procedure switch. The filter accepts the message and calls the procedure implicated. Communication packets are simple byte arrays. The first word contains the size, and the following three bytes are destination, source and classification. The real message is packed from the eighth byte to the end.

(a) Open decoders: The total memory cannot be truly addressed.
(b) No access.
(c) Multiple access: More cells than the addressed ones are accessed.
(d) Crosstalk between cells.
(e) Stuck cell.
(f) Pattern sensitive interaction between cells.
(g) Chip select line stuck.
(h) Read/Write line stuck.
(i) Data register stuck.
(j) Crosstalk between data lines or cells of the data register.
(k) Drivers stuck.
(l) Cell can be set to 0 but not to 1 (or vice versa)

Figure 4.9. List of functional faults

External Memory checking:

There are two algorithms used for transputer external DRAM chip checking. One is *filling address* to each word memory location and the other is *checker board.* Just as with ordinary processors, the transputer memory model can be divided into four functions, address decoder, memory cell array, read/write logic and refresh logic. We are interested only in detecting the memory fault and not which function the fault exists in. The address decoder handles addressing the right cell or word. The read/write logic handles the transport of data from and to the memory cell array. The refresh logic keeps data status and renews the data in the cell array. A fault in any of these functions can cause a memory fault. Figure 4.9 lists some functional faults that can occur in a

DRAM (list is not complete) [TGS91].

Filling address

The transputer memories are often organised with wrap-around addresses. Filling locations with addresses is one way to detect the size of the external memory (ET). It also checks faults in the address decoder and some other faults. Its algorithm is simply to write each word address number into its corresponding location from the top to the beginning of the ET, then read it back to check if the data matches its address word by word from the first word toward the top of the ET. This process continues until a mismatch is identified. The last address which succeeds is the top address within available error-free memory. If the difference between the top and the beginning address is equal to desired ET size then no error exists. Otherwise, an error has been detected.

Checker board

We used the Checker board algorithm [SR81; VV90] to check the cell array as well as read/write and refresh logic for every bit situation. The strategies of Checker board can be summarised as four steps. (1) Generate a 255 byte pattern block in the internal memory with a pattern 55 and AA alternating one after another and generate a 255 byte buffer. (2) Move the first 255 bytes to the buffer then copy the pattern block into this emptied block in the ET. (3) Match these two blocks between internal and external memory. If they are the same, go on to the next step; otherwise error detected. (4) Move the previous block in the ET to the area starting right after it and go to step (3). Steps 3 and 4 continue until the number of bytes left to reach the top address is less than a block. When the pattern 55-AA has been tested, the pattern block in the internal memory is replaced with pattern AA-55 and the algorithm goes from step 2 to 4 again. After AA-55 pattern, it changes back to pattern 55-AA for one more time. In this way, each cell has been transited from 0 to 1(T 0->1) and (T 1->0). The algorithm covers all

of the faults in the list of Figure 4.9 except some cases of faults d and f. The time spent on moving blocks should be enough to detect a refresh logic fault.

The two memory checking algorithms have different emphases, but they complement each other. The combination of Filling address and Checker board algorithms is sufficient to detect most faults.

Link communication testing

The interprocessor communication over a link is a synchronous protocol. Communication faults could interrupt the synchronisation and cause process suspension at both ends. An efficient preliminary diagnosis on link communication is absolutely necessary.

There are various possible link communication errors or faults but our diagnosis program tests only hardware faults. The possible faults, then, would be two: firstly, link broken, where some or entire messages cannot get through; this can be confused with processor failure. The server on the host will work out the difference. Secondly, data passing errors from noise or from signal transitions, where the effect is that the receiver gets messages different from the ones the sender sent.

There are two processes on each transputer to detect link faults, a test data sender and a receiver. To each hard link, the sender on one transputer sends a 1k byte packet, with pattern AA, through the link to a transputer. At the other end, the receiver acquires the packet and checks to see if the message has an error in it. If an error is found, it records the id of the link and reports the error to a sending transputer; otherwise, it acknowledges the normal situation. After having done this for the AA pattern, another 55 pattern is tested in exactly the same way. Link testing is bi-directional; once the receiver is finished checking for both patterns the sender will check the link in the other direction. Over the network, the receiver on each transputer always waits to test the

father link first, then the son links in order.

4.6 Summary

The general system local diagnosis is based on internal and external test results. The internal test is that each processor tests itself and sends test result to the processors from which it has received a request. The external test is that each processor sends a request to its neighbours and uses peek and poke techniques to examine the memory the testing program resides on. These internally and externally tested results provide sufficient information for local diagnosis decisions. The entire network's health can be diagnosed by analysing the combined test results from the individual processors.

The effectiveness of the fault-tolerant NWP multi-transputer system was tested by running it over a network which contained faulty processors and a network which generated failures deliberately. It identified failed processors and explored functioning network topology. It always provides a healthy system to the application with maximum number of processors. Although we have not monitored its performance after it was used for weather prediction at the NZ Meteorological Service, we did not receive any complaints about its performance. Therefore, we can say that the fault-tolerant NWP system has been working correctly and reliably for more than three years in daily performance as far as we know.

The modified NWP system with fault tolerance features can be summarised as depicted in a Petri net model in Figure 4.10.

(a) The system is initiated with system starting and job submitting states. The job submitting state at this stage means a new job arrived. With initiating state, the only t_1 is enabled so that system comes to diagnosing and job submitting state.

(b) If a fault is detected during diagnosing, the diagnoser will ignore the component which the fault resides on and only records the available functioning resource

configuration. After this exploration, the t_4 is enabled then submitted new job will be loaded over the functioning processors those diagnoser explored.

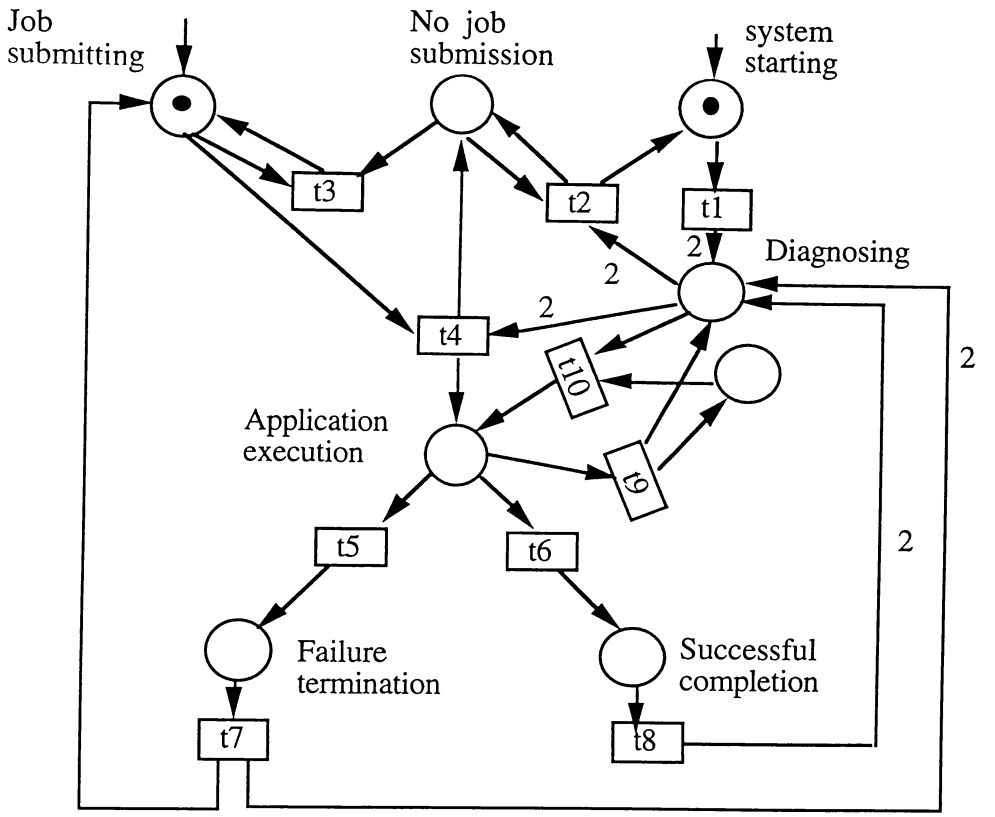


Figure 4.10 A abstract Petri net model for fault-tolerant NWP system

(c) While application programs are executing over the processor in the network, the diagnosis program is invoked periodically to detect the dynamic failures.

(d) If a failure is detected before the application program is successfully completed, the processing will be terminated immediately. Then the system would re-submit the job which terminated and re-diagnose and explore the current functioning resource configuration. This process might be iterated more than once until the application is completed successfully.

(e) When the job submitted is completed successfully, the system comes to the diagnosing state. This diagnosis is iterating until a new job is submitted or the system is forced to shut down.

The diagnosis algorithm described in this chapter has two assumptions: (1) any processor in a network is always diagnosed by a healthy processor, (2) once a processor is diagnosed as a healthy processor, then all testing results, regarding other nodes, produced by this processor are correct. However, in a truly parallel and distributed diagnosis algorithm, every processor in a network should start to diagnose its neighbour processors asynchronously and roughly at the same time. In such case, assumption (1) and (2) can not be guaranteed. In another word, a faulty processor might report a healthy processor as a faulty processor and no diagnosis is produced by a predetermined healthy processor. Also if there are more than one faulty processors in a network, then a faulty processor might wrongly report another faulty processor as healthy processor. In fact a correct diagnosis to a processor should be produced by analysing the results that are tested by a group processors rather than that is tested by an individual processor. This problem will be discussed in the next chapter by introducing a hybrid diagnosis algorithm.

The hybrid diagnosis algorithm is a further development over the diagnosis algorithm designed for the fault-tolerant NWP system. It offers significant improvement compared to other existing diagnosis algorithms.

Chapter 5

A hybrid diagnosis algorithm

5.1 Introduction

The problem of fault diagnosis in multiprocessor systems has become important with both the growing complexity of the systems and the critical application areas in which they are used. Classical techniques for identifying system failures, such as error-detecting codes and massive redundancy with voting, are unable to yield the high level of reliability required without a tremendous amount of hardware overhead and/or performance degradation. As the growing level of parallelism significantly increases the difficulty of recovering from faults, efficient diagnosis is a very important step for fault-tolerant multiprocessor systems to achieve their fault tolerance through error detection, diagnosis, reconfiguration and recovery.

Correct diagnosis of faults is necessary to allow proper reconfiguration and recovery actions to be carried out. There has been much research work on efficient diagnosis for distributed systems. The most popular existing diagnosis algorithms comprise three groups: centralised, truly distributed and processor membership. The centralised algorithm is restricted by the reliability of the central supervisor processor. The truly distributed diagnosis algorithms impose heavy communication overheads for assessing the status of the system. Processor membership is mostly used to monitor processors departing and joining in the system and it can be used to diagnose the network only in the situation in which a processor fails with fail-silent semantics.

Interconnecting multiprocessors to form a grid architecture can provide natural redundancy for failure recovery. However, when the number of processors contained in the network grows, the diagnosis of the state of the network has to increase its accuracy in locating failures and in analysing the syndrome more efficiently. Any mechanisms for the detection of failures would likely need to be distributed throughout the network, and any diagnosis of the source of the network failures would also need to be accomplished in a distributed fashion.

In this chapter, a hybrid diagnosis algorithm is presented. This algorithm is especially suitable to parallel and distributed systems. It can be used to determine system resources and identifies the faulty processing units by incorporating a fault-tolerant procedure [GTP90; TGS91]. It is named the "hybrid diagnosis algorithm" because it has the features of many different types of diagnosis algorithms.

The hybrid diagnosis algorithm executes like a distributed diagnosis algorithm while faults are absent, and it works like a centralised diagnosis algorithm only when a fault emerges during the execution. This reduces the number of messages required in all cases compared to ordinary algorithms. This algorithm also employs the processor membership mechanism to monitor and recover from central supervision processor failures. It is guaranteed to work correctly for a given $\lfloor \frac{k}{2} \rfloor$ number of faults present in a k--regular connected multiprocessor system.

The complexity of the diagnosis problem for large multiprocessor systems is governed by two factors:

(1) the nature of the fault model assumed, and

(2) the extent of the diagnosis required (who needs to know what?)

In the most general and difficult formulation of the problem, the answers to these questions are that a faulty node is assumed capable of any arbitrary or even malicious behaviour, and that every fault-free node is required to diagnose the condition of all other facilities in the system. The work in this thesis on diagnosis covers the fault conditions assumed above. The objects of diagnosis are processors and communication links.

This chapter reviews some important attributes of existing diagnosis algorithms, presents details about the hybrid diagnosis algorithm, and analyses the reduction in the number of messages compared to other algorithms.

5.2 Diagnosis algorithms for multiprocessors

Three types of diagnosis, centralised diagnosis, distributed diagnosis and processor membership diagnosis, are directly related to the hybrid diagnosis algorithm. However, there are many other kinds of solutions using different philosophies in diagnosis, such as set diagnosis, adaptive testing, probabilistic diagnosis and so forth. The details of these types of diagnosis in fault-tolerant system design may be found in the literature [KR86; Dah88; FS80; KH87; KY86, and ML80].

5.2.1 Centralised diagnosis

The problems of centralised diagnosis in parallel and distributed systems have been subjected to intensive research over the past three decades. The most extensively studied is the so-called PMC model [PMC67]. This model describes a class of systems which can be decomposed into units that are capable of testing each other.

The PMC model was studied initially for centralised system level fault diagnosis and it formed the framework for much of the subsequent research in the system diagnosis area. It considered that a PE could test other PEs and that results could be used to find

the state of the system. The test results were known as a "syndrome". A centralised arbiter interpreted the syndrome and deemed each PE to be either faulty or not. PMC originally dealt with the special case where all of the failures were hard or permanent faults. It assumed that a fault-free processor was always able to accurately determine the condition of the PE it was testing and that a faulty processor produced unreliable test results.

Preparata [PMC67] were primarily interested in a *t*-diagnosable system in which given the test syndrome, all faulty PEs may be uniquely identified, without repair, provided that the number of faulty PEs does not exceed *t*. The diagnosis without repair is also called one-step diagnosis. If the system is repairable, then it is only necessary to locate at least one faulty PE if it exists. In this case, after a PE is diagnosed as faulty, it can be repaired and the testing continued to eventually diagnose all of the faulty PEs. Such diagnosis is called *k*-step diagnosis, or diagnosis with repair. The characterisation problem is to find the necessary and sufficient conditions for a test assignment to achieve a given level of diagnosibility for a given fault model. [HA74] gave a general solution for one-step *t*-diagnosable networks.

The assumptions made by PMC model are not necessarily valid nor desired in a fault-tolerant and distributed network. Later work has attempted to remove these restrictions. One of PMC's drawbacks is supervisor controlled diagnosis. This implies that all test data must be gathered and analysed with the result distributed back to the system by a centralised supervisor which is a single PE. This is costly in terms of time, message passing, and system reliability, which is directly related to the reliability of this supervisor.

Methods for distributed diagnosis have been developed in which every processor decides independently what is the fault-free population. Thus, as long as the bounds for diagnosibility are met, the hardware to perform the diagnosis is available.

5.2.2 Distributed diagnosis

One problem with removing the centralised supervisor is the forwarding of the diagnostic information to the system user. If the user is unable to test a processor, then it is a problem to decide from which PE to take the system diagnosis information as any of them might be faulty. In [KH88] Kerutzer and Hakimi discussed this issue. Basically, they sought the minimal connections required between a centralised observer, i.e., the user, and a *t*-diagnosable system in which every fault-free processor has the correct diagnosis of the system. They found that if authenticated messages, or message passing, were available, then only *t+1* PEs needed to be queried to learn the correct diagnosis. Without authenticated messages, $2t + 1$ PEs needed to be probed.

[Nai78] provided the first formulation of the system diagnosis problem in a distributed environment. His concept was a roving-diagnosis approach in which only a subset of the system's PEs performs diagnosis as the subset "roves" over the entire set of PEs. [Smi79] gave simple system diagnosis algorithms that do not use a centralised observer and that can be applied regardless of the system structure. However, he did not describe the manner in which the test data would be distributed, nor did he exhibit his discussion in a distributed-system framework.

[CGS81] saw the problem as one of distributing the centralised analysis of the syndrome. In their approach, testing was carried out as required for *t*-diagnosibility. Once this was completed, results were not sent to a centralised observer, but rather were disseminated using a consensus protocol similar to Byzantine agreement. Then, each PE would calculate the system diagnosis from these results. The problem with this approach is that while it assures that all processors have a consistent view of the diagnosis of the system, the syndrome dissemination is expensive and halts useful processing. As a result, techniques resulting in dynamic diagnosis have been explored.

The term *distributed diagnosis* was first introduced by [KR86]. In their model, a

processor in a distributed environment has reliable information about only those PEs in its neighbourhood, for example, those that it can communicate with via direct communication paths. Data about the rest of the system is indirectly available from PEs outside of the neighbourhood. Each fault-free PE should be able to independently and correctly diagnose the entire system, then use this knowledge to refrain from dealing with any elements deemed to be faulty, and initiate fault recovery techniques. The authors assume that fault-free PEs could accurately test other PEs and that faulty PEs conducted tests with unreliable results.

In [HKR84] Hosseini *et al.* presented a series of algorithms called SELF. SELF2 was developed based on the original distributed diagnosis work and assumed that faults are permanent. Each processor P_i calculates a fault vector F_i whose jth element is a 0(1) if P_i concludes that P_j is fault-free (faulty). A processor P tests each of its neighbours and completes part of the fault vector. If a neighbour is faulty then this condition is broadcast to all the fault-free PEs, that themselves test P (obviously P has a direct link to each of these PEs). Whenever P receives a diagnostic message about some faulty processor Q that was previously considered fault-free, it first checks that it believes the last relay PE of the message is fault-free. P tests this sender again, and if it passes, (1) the information about Q is saved in the fault vector and (2) the message is forwarded to all those fault-free PEs that test P. Otherwise the diagnostic message is ignored, the sender is marked as faulty, and this information is sent to the testers of P.

In SELF3, [KR80; KR81] extended the SELF2 algorithm to cover message corruption caused by faults in the communication paths or by fowarding a message through a faulty processor. Further to the algorithm SELF3, [HKR85] modified it into NEW_SELF such that fault-free PEs are never temporarily misdiagnosed as faulty.

At Carnegie Mellon University, system diagnosis was applied to an Ethernet, connecting over 100 workstations, by adopting the distributed-diagnosis algorithm

NEW_SELF [BGN90]. The CMU diagnosis solution called EVENT_SELF enhances the NEW_SELF algorithm by (1) using time to synchronise tests and detect faulty PEs, (2) allowing for PEs to join and leave the set of fault-free PEs, (3) calling for reconfiguration testing after PE failures, and (4) assuming system stability to minimise overheads. It does not test for faulty communication links.

EVENT_SELF uses self-tests initiated by outside PEs. When a processor A tests another processor B, it actually sends a request to B that it test itself. B spawns a subprocess which reads from the disk and executes floating-point operations. Thus, the operation of the disk, the operating system software, and some portion of the CPU are tested without starving any productive tasks of resources. The diagnosis algorithm treats the result of this test as complete, and the authors reported that this simple test caught every processor fault that occurred over a period of two years.

One of the major concerns of this work was the communication overhead required by the NEW_SELF algorithm. A significant reduction may be had by combining test requests and results. If processor P is being tested by all of its neighbours, then it need run the self-test only once and report the same result to these testers. If processor P is testing some processor q and receives a request for another test of q, then p may combine these requests and disseminate the test result accordingly. These simplifications are convenient for the implementation of the test as a self-test.

Another technique used for reducing message transmissions was to take advantage of the stability of the system. That is, a stable system has no faults occurring and no new nodes joining it, and thus requires no new diagnosis. The authors [BGN90] devised an event-driven algorithm in which diagnostic information is not passed unless a fault or join occurs. They also continued the experiment at CMU with an adaptive distributed-system diagnosis algorithm appropriately called Adaptive DSD. The Adaptive diagnosis locates a fault-free processor and uses it to diagnose further. Once a PE is diagnosed as

fault free, the tests it performs are considered reliable, and therefore, any other PE should be only tested once by this fault-free PE to correctly determine its status.

Adaptive DSD assumes that the system is logically fully connected and that communication links are reliable. [SBB92] relaxed these assumptions in their algorithm which works on-line in arbitrarily connected topology even in the case of unreliable communication links. The algorithm maintains a minimally strong testing graph in each component of the system; for instance, if the system becomes disconnected, each component will continue to be diagnosed correctly. Testing is performed according to the testing graph which is in turn determined by the topology of the system. Therefore, the performance of the algorithm varies with the topology of the system on which it is executing.

5.2.3 Processor membership

Processor membership [Bir85] is a technique which is used to maintain a list of fault-free processors without aid from a centralised observer. A processor is in the membership, or fault free, if it can maintain a timely schedule of "present" messages. The problem is to keep all of the fault-free PEs informed of the membership regardless of whether PEs are joining or leaving the system. [Cri91] presented three protocols for processor membership: the periodic-broadcast protocol, the attendance list protocol, and the neighbour surveillance protocol. Each of them detects processor faults in a timely manner and at the message-passing level. The periodic-broadcast protocol requires that every processor broadcasts a "present" message at some predetermined and globally known frequency. A new membership is created when a processor broadcasts a "new membership" message. An obvious problem with the periodic-broadcast protocol in the point-to-point communication network is that the overhead generated with "present" signals at every testing round could be prohibitive.

In the attendance list protocol, one PE generates a token periodically, and circulates it

around a logical cycle of the members. If any PE does not receive a token in time, it is assumed that an error has occurred, and a "new-group" request is made. Such a technique cannot be used in a large array architecture because the sequential token pass consumes on intolerable amount of time.

The neighbour surveillance protocol works in a manner similar to the attendance list protocol. A logical cycle of the processor membership is specified and given a direction. Periodically, each processor requests a neighbour confirmation of its predecessor. If the confirmation is not received during the correct time frame then a failure has occurred, and a "new-group" request is initiated to establish the new membership.

The processor membership problem is similar to the distributed-system diagnosis problem, as both strive to determine which processors are faulty and which are not. Whereas distributed diagnosis explicitly utilises a test to do this, an implicit test is assumed in the processor membership model, in that a PE is determined to be fault free if it can transmit and forward messages in a timely manner. One of the major defects in processor membership solutions is that they are only able to some processor faults. Communication link failures are not included in this mechanism.

Three existing techniques in fault diagnosis for multiprocessor systems have been reviewed. The major problem with centralised diagnosis is that system reliability is limited by the reliability of its centralised supervisor. Also a non-supervisor facility could pose a reliability bottleneck for the entire network after failure, or incorrect operation.

The limitation with some truly distributed techniques is that hardware reconfiguration in response to a diagnosed failure may not be possible (or desirable). This stems from the lack of any central authority in the networks capable of performing such activities. The

processor membership technique has a fault diagnosis focus in the time domain; it is also only useful to determine processor departing and joining information, and cannot cover communication link faults.

The problems pointed out above are improved in the fault diagnosis algorithm presented in this chapter. The approach is based on the concept of mutual testing at the processor level where each processing element (PE) is capable of testing its immediate neighbours. It is formed by combining some of the features from existing methods and special situations, in a parallel and distributed structure, with array topology, and point-to-point message passing communication protocol.

5.3 Preliminary

The diagnosis algorithm developed in this thesis has the feature of testing and communicating to near neighbours only. Otherwise, ensuring correct delivery of test results is very expensive. This approach has also been adopted in other research [HKR84]; their algorithm is optimised for parallel processing applications. A PE A tests another PE B, by sending a request to B so that B can invoke a test program to test its own situation, and send the tested results to A. A also uses a peek and poke procedure to test the memory of the PE upon which the test program resides. The tester itself could be faulty and produce arbitrary test results. In a comparison test approach [MM81; DSK87], the results produced by two PEs for identical computations were compared by one of the PEs or by a third PE independently to produce test results. The hybrid algorithm uses an alternative method to solve this problem.

The complexity of a diagnosis algorithm in a parallel and distributed system depends upon the test, and upon the communication interconnection network of the system. Most multiprocessor systems have regular topology such as toroidal rectangular, hexagonal, and octagonal grids in two-dimensional meshes, rings, and hypercube. A regular interconnected structure is formally defined as follows.

Definition 1: An interconnection structure is said to be k-regular if the considering network is homogeneous and each node has k neighbouring nodes. That is, in mapping from one node to another, any node in the network has at least k different nodes that it can map to. For example, in a toroidal mesh, each of its nodes has four neighbouring nodes. So in this case a toroidal mesh has 4-regular connection.

5.4 A hybrid diagnosis algorithm for multiprocessors

The hybrid diagnosis algorithm is designed to identify faulty PEs in a loosely coupled multiprocessor system for parallel processing applications. The algorithm is hybrid because it was formed by absorbing the features of centralised, truly distributed and processor membership diagnosis algorithms. It is also unique in that it performs in parallel and uses minimum communication messages for making accurate diagnoses. The algorithm is t-diagnosable where $t = \frac{k}{2}$ if the network is k-regular connected. The diagnosis algorithm can identify the situation of any PE given $t = \frac{k}{2}$ processor failures. The worst case is where t faulty processors are the neighbours of a same processor. In more usual situations, the algorithm might be able to diagnose the system in the presence of more than t failures.

The algorithm is also able to diagnose faulty communication paths between pairs of fault-free nodes in the network. The whole diagnosis procedure is accomplished in two steps:

(1) distributed local semi-diagnosis for fault-free situations and

(2) centralised final diagnosis when faults are reported.

The requirement of a centralised processor in the algorithm is derived from the nature of the parallel processing application. Unlike truly distributed systems, parallel processing systems require a centralised facility to allocate the processes onto each node, and to

distribute raw data to, and collect processed data from, all processors. When failures occur, it is necessary for a centralised facility to reconfigure the network topology either logically or physically. However, centralised supervision in this algorithm is different from that used in the PMC model, since this diagnosis algorithm allows the system to survive in the presence of the centralised processor failing by using the processor membership technique. If it fails, a "new-group" request will be broadcast to the network [Bir85]. One of the functioning processors will take over the centralised supervisory role after the establishment of a new group processor membership.

Parallel processing uses the coordination predicate i.e. the progression of one process on a PE depends upon the progress of different processes on other PEs. If one PE fails, the system must be recovered once the fault has been detected. Since the tasks on each individual PE are as important as the task of the whole system, if any single processor fails to produce a correct output on time, the entire system's results will be corrupted. In contrast to distributed systems, the diagnosis algorithm for parallel processing systems does not allow processors to freely join and depart.

5.4.1 Fault Models

Knowing how a processing element fails is key to making realistic assumptions and creating workable algorithms to detect the faulty PE. Examination of the characteristics of faulty processors has resulted in the proposition of many fault models, and a wide range of system diagnosis algorithms. The Hybrid diagnosis algorithm considers the Byzantine fault which covers every fault possible in the system model. This model is the universal fault set which includes an arbitrary or malicious fault, such as when one processor sends differing messages during a broadcast to its neighbours. This algorithm will give a correct diagnosis under all these conditions.

A major problem in trying to achieve an efficient diagnosis of failures in an environment in which faulty facilities may act arbitrarily, is that diagnostic messages

may be altered, destroyed, or erroneously issued by the faulty facilities. Thus, a node cannot trust the diagnostic messages reaching it to be complete or accurate. The algorithm given in this section deals with the problem by using a majority vote on testing information from neighbouring processors .

5.4.2 Algorithm

If a homogeneous parallel processing system consists of n PEs interconnected to form a k-regular array architecture, this type of architecture implies that there are k direct links on each PE, which connect to its k neighbouring processors. It is assumed that each processor in the network has unique identification and its k links are numbered clockwise. A processor P_m takes the role of central supervisor. Given a network as described, the Hybrid diagnosis algorithm can be viewed as follows:

P_m initiates a system diagnosis by broadcasting a request over the network. This broadcast message is firstly sent to its neighbours in the link number order. When a request has been sent to all its neighbours, it invokes the testing process to detect if there are any faults. In the same manner as P_m, when a processor receives a request for diagnosis, it forwards this request further down to all its neighbours before starting its testing process. The routes that requests are mapped through forms a spanning graph. For example, a spanning graph for a 4-regular connected toroidal mesh multiprocessor system with 16 nodes is shown in Figure 5.1. This spanning graph is equivalent to its configuration graph and it displays all possible paths for broadcasting between any two nodes.

For any processor p_i in the network, as soon as processor p_i has completed its test processing, it sends the test result about itself to its four neighbours. Following the mutual transmitting of test results, a processor uses peek and poke techniques to test the memory where the test program resided on every one of its neighbouring nodes. It combines the test results, and peek and poke results, to make a local sub-diagnosis

about its neighbours. A processor reports the diagnosis information back to P_m if, and only if, it detected an abnormal situation on its own node or on a neighbouring node. The reason it is named a local sub-diagnosis is because diagnosis effectiveness is restricted to its own processor, and it might be changed if it conflicts to the global view.

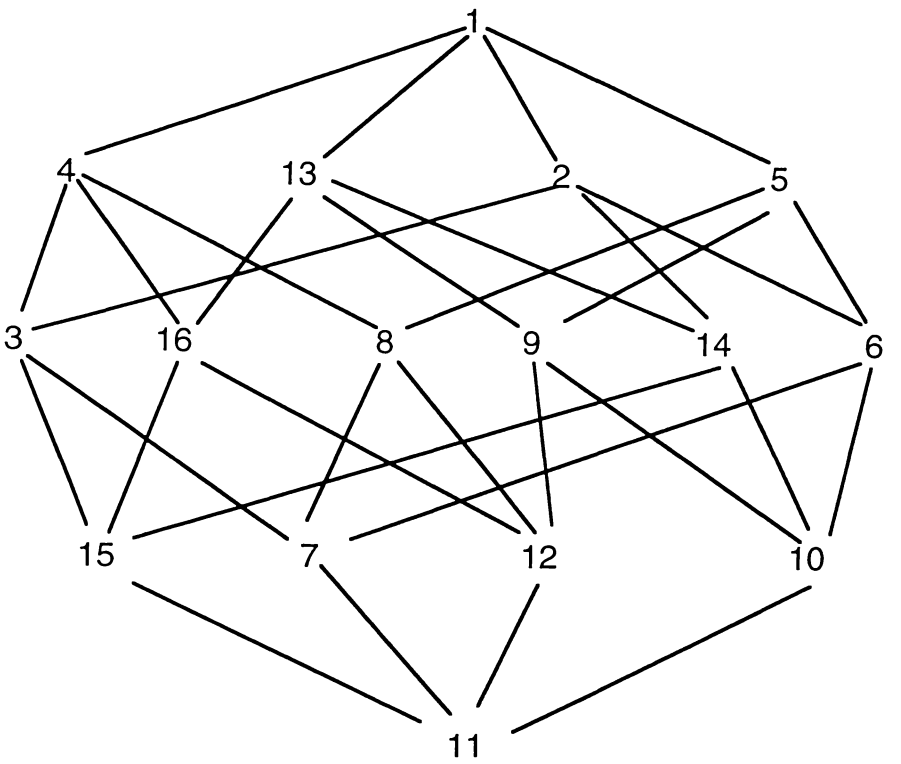


Figure 5.1 A spanning graph for a toroidal mesh with 16 processors

For example, processor B is one of the neighbours of A. If A diagnosed a fault on B, processor A reports the fault to P_m by sending a message, then waits for a confirmation about the fault report from P_m. Otherwise, if A completes its diagnosis with no fault detected, then it just transfers to a waiting state unless a broadcast message concerning some other faulty processor arrives from P_m.

Whether A has diagnosed B as faulty or not, there are always two circumstances in which this result is correct or incorrect. Firstly assume A correctly diagnosed B as a faulty processor and sent a report about this diagnosis to P_m. In this case A's local diagnosis result on B is consistent with the global view, because it will be identical to

the result confirmed from P_m. Secondly, assume that this local diagnosis is incorrect, then there would be a confirmation about A's failure or a failed link between A and B instead of $B's$ failure.

In the same way, if A diagnoses B as a non-faulty processor incorrectly, then P_m will recognise this faulty report from it being inconsistent with reports from B's other neighbours and broadcast a message about A's failure or a failed link between A and B. There is no requirement for A to send a report about B to P_m if A diagnosed B as a non-faulty processor correctly, and A's local diagnosis is consistent with the global view of B's state.

The global diagnosis is carried out by the centralised processor P_m and is based on individual distributed local sub-diagnoses. After P_m broadcast a diagnosis request over the network, it diagnoses itself and its neighbouring processors. The only time that P_m receives local diagnosis messages from other processors is when they report faults. When P_m receives the first fault report, it starts a timer and waits for possible fault reports from other nodes. Once the time is up, it analyses the received reports to produce a global diagnosis by majority voting. In a k-regular connected multiprocessor network, P_m identifies a processor failure if more than $\frac{k}{2}$ nodes reported a fault on the same processor. If less than $\frac{k}{2}$ nodes reported a fault on the same processor, P_m can determine that the failure is either in the reporting processor, or one of its links. The precise diagnosis can be explained in the following example.

In Figure 5.2 A is a node in a 4-regular connected network; it has four neighbours B, C, D and E. All nodes have an initial error-free global state. Node A can be locally diagnosed by its neighbouring nodes B, C, D and E. If there is no fault existing in the network, then diagnosis about A by B, C, D and E will be fault-free. Therefore, P_m will not receive any diagnosis message about A from any of A's neighbours, so that the global state about A on B, C, D, and E will continue to be the same as their

previous local sub-diagnosis state. If A is a faulty node, then its fault will be diagnosed and reported to P_m immediately. P_m should receive messages about A's fault from B, C, D and E. It might also receive a message from A about its own fault. Based on this majority reporting, P_m determines A to be faulty processor, and then broadcasts this information to change the global state of A in the network.

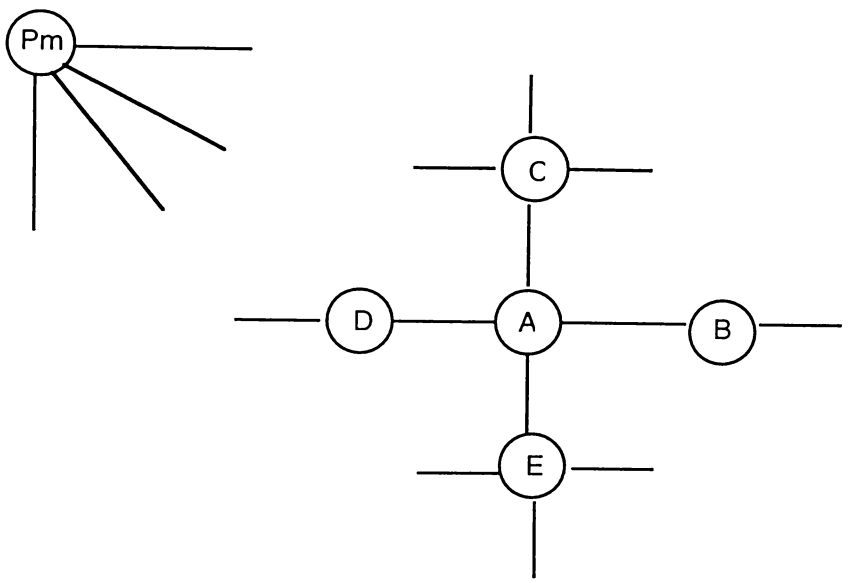


Figure 5.2 A situation in a 4-regular connected multiprocessors

In the situation that two faulty processors are located at either end of a link, a faulty node could lie about its diagnosis of a neighbouring node. Without loss of generality, if two faulty nodes are A and B in the figure 5.2, B could incorrectly diagnose A as a non-fault node so that P_m will not receive a report about A's fault from B. However the information about A's fault will be reported from A's other neighbours C, D, and E. When P_m receives the first report about a fault, it waits for timeout. After the time is up, P_m analyses the syndrome by taking the majority vote. Because three out of four A's neighbours reported A's fault, P_m will diagnose A as a faulty processor, and broadcast an announcement to change the global state of A in the network. Because B failed to send correct information about A, there is one more failure on either B, or the link between A and B. P_m will work out the precise location by referring to the diagnosis information from B's neighbours. If none of B's neighbours reported B's failure then P_m announces a link failure. Otherwise B has failed.

The worst case is when two faulty nodes are the neighbours of a same node, these two faulty nodes could make a false diagnosis of a non-faulty node as a faulty node. P_m is still capable of identifying these two faulty nodes. Again in figure 5.2, if B and C reported that A is a faulty node, then P_m can recognise that B and C are two faulty nodes, and inform the global states of B and C to the network immediately. This is because if B and C are not faulty nodes, then A, D, and E must be faulty nodes. Otherwise P_m should have received a report about A's fault from D and E. However a 4-regular connected network cannot have more than $t = \frac{k}{2} = \frac{4}{2}$ node failures for correct diagnosis. So P_m 's determination of the failures of B and C are correct. Although two out of four neighbours of a node claim it as a faulty node, P_m has no difficulty in making the correct decision. If its decision is wrong, the system has three faulty nodes i.e. A, B, and C. This situation is in conflict with the assumption of that the algorithm is t-diagnosable where t=2. The same idea can be extended to any k-regular connected network.

A link failure can be detected by its two end processors. If P_m receives two reports about the failure of each other, and no other reports about their failures, then P_m diagnoses a link failure occurred between these two nodes. For instance, if two nodes A and B are neighbours to each other, A reported that B was a faulty node, and B reported that A was a faulty node, then P_m could broadcast the failure of the link between A and B to the network.

The situations in which t faulty processors can be identified has been analysed. No matter how these t faulty processors are distributed in the network, it will not block the broadcasting messages between any other node to P_m. Even faulty processors have fail-stop semantics. From figure 5.1, we can see that there are at least k routes between P_m and any other nodes. If t nodes failed, there would still be k-t routes, which is more than t, available ($t = \frac{k}{2}$).

Central processor failure

Fault tolerance is the ultimate goal of system diagnosis; therefore, centralising the diagnosis function, and thereby creating a "weak link," is usually unacceptable. However, in the Hybrid diagnosis algorithm, failure of P_m is handled by the processor membership mechanism. There are k neighbouring processors surrounding P_m so that the status of P_m can be diagnosed by its k neighbours if its failure occurred after a request had been issued. P_m also diagnoses its k neighbours. If P_m fails to broadcast a diagnosis request, this is detected by using time-out among its k neighbours. This neighbouring diagnosis and time-out for detecting the failure is only an individual local view. Because of suspicions concerning P_m's failure, the global state of P_m has to be figured out among its k adjacent processors using Byzantine agreement.

Byzantine agreement in this special situation can be translated as follows. Once a neighbour of P_m has detected the failure of P_m, it broadcasts a message about P_m's failure to the other neighbours. When other neighbours of P_m receive information about P_m, they make their own diagnosis on P_m and broadcast it to others. Based on these messages exchanged among the k neighbours of P_m, each of them takes the majority and changes its global state of P_m. If the result of global diagnosis on P_m is negative, then the processor which is one of the neighbours of P_m, and which has the lowest identification, takes the role of P_m and announces it to the network. In other words, a new membership is established. Therefore the system is still able to make correct diagnoses even in the case of P_m failure.

5.4.3 Communication overhead analysis

As always, algorithm efficiency is of interest. It is necessary to reduce the number and size of the messages until their effect on other communications is virtually transparent. Otherwise, diagnosis might become a system bottleneck.

The hybrid diagnosis algorithm proposed in this section avoids the massive communication overhead imposed by conventional diagnosis algorithms. In particular, in the absence of a fault, this diagnosis algorithm requires only one diagnosis request for identifying the global state of every processor in the network. This improvement is significant, since most of the time the diagnostic process is coping with system performance without the appearance of failures. Of course, there is the possibility of the worst situation occurring. The worst case for this algorithm is that t processors fail in between two diagnosis requests, and that they are not physically adjacent to each other. In order to detect and locate these failed processors and confirm the global state of the network, it involves $tk + 2$ diagnosis communications(broadcasts) over the network. (Where tk messages for fault reports are made: k messages for each faulty processor from their surrounding neighbours to P_m, and one diagnosis request and one message for global state announcement about those faulty processors from P_m).

When a network topology is determined, k is a constant number, while the network size n may alter. If $k = 4$ as in a 4-regular configuration network, then t is 2, so that $tk + 2 = 10$ messages are required in its worst case. This number stays constant even in a large multiprocessor system.

5.5 Summary

We have presented the hybrid diagnosis algorithm and analysed all possible situations of t faulty node distribution. The algorithm is capable of identifying t faulty processors and all non-faulty processors completely, no mater how they are distributed.

The hybrid diagnosis algorithm contains features of the centralised diagnosis methods. The centralised supervisor is used to identify the global state of each processor in the network. However, it improves on traditional centralised diagnosis methods in two ways: (1) when the centralised supervisor is faulty, the system can survive by employing the processor membership mechanism. (2) It reduces diagnosis

communication overhead significantly to O(tk) rather than O(n) messages as in traditional centralised diagnosis. Obviously when the size of the network increases, an overhead of O(n) would be intolerable. In the hybrid diagnosis algorithm, the messages required for identifying t faulty processors are a constant $tk+2$ for the worst case. Its efficiency is especially desirable in a large network.

The hybrid diagnosis algorithm uses processor membership methods only to monitor the failure of the centralised supervisor P_m. This monitoring is only among the k - neighbouring processors of P_m and it is not restricted to P_m having fail-silent semantics. It involves only k monitoring messages for detection of the P_m's failure and k messages to make Byzantine agreement among k neighbouring processors of P_m. Because k is a constant determined by the network topology, messages for diagnosing P_m's failure are independent of network size n. Again it is efficient in a large network.

With a truly distributed diagnosis algorithm, the minimum number of messages required for identifying the states of all processors in the network is O(kn) [BH91]. The diagnosis of t faulty PEs requires a minimum $2t+1$ queries of fault-free PEs to learn the correct diagnosis in the worst case [KH88]. The hybrid diagnosis algorithm however requires only one diagnosis request message for global state diagnosis in a fault-free network and a maximum of $tk+2$ messages in the worst case. It needs no queries to fault-free PEs and is able to learn the correct diagnosis based on fault report information from t nodes. This logical induction reduces the operating overhead enormously.

Further to the improvements over centralised, processor membership and distributed diagnosis algorithms, this hybrid diagnosis algorithm parallelises the testing processes on each individual processor. Moreover, wide communication bandwidth in multiprocessors with an array architecture and a point-to-point message passing protocol would enhance the system performance. This diagnosis algorithm plays an

important role in an adjacent checkpointing and recovery synchronisation algorithm to be presented in the next chapter.

Chapter 6

Consistent recovery from partial dynamic failure in parallel processing

6.1 Introduction

In this chapter we present a software algorithm which can be used to protect continuing operation from partial system failure in a parallel and distributed architecture. The failure modes are the same as we discussed in chapter 5, that is processor and link failures caused by hardware faults, as well as software errors derived from hardware faults. Since the algorithm concentrates on recovery strategies, it assumes that processors have fail-silent semantics, they produce either correct responses or no response at all. Otherwise, it supposes that failure is detected by a separate facility which cooperates with the algorithm. The failure detection facility used in this chapter is based on the hybrid diagnosis algorithm presented in Chapter 5. Once an error is detected, the processing on the failed node will be forced to stop.

The issue of checkpointing/restarting plays a very important role in fault-tolerant parallel processing. An inefficient checkpointing/restarting algorithm could seriously affect the system performance in error-free operation, and many applications require fast recovery to meet time constraints. Although this issue has already received much attention in parallel processing, most existing methods checkpoint error-free states to shared storage. However, checkpointing and recovery is slowed down significantly by the limited communication bandwidth to the storage. The reliability of the system would

be largely dependent on the reliability of the shared storage used for checkpointing.

We have classified checkpointing algorithms into three general groups: Pessimistic, Optimistic and Consistent algorithms. Our algorithm is a consistent checkpointing/restarting algorithm. This algorithm has two major differences from previous consistent checkpointing algorithms: firstly each processor uses a small amount of memory storage on its neighbouring nodes for storing checkpoints. It reduces the risk of system crash due to the failure of shared memory used for checkpointing states. Secondly every processor takes advantage of wide communication bandwidth between any two adjacent processors, and saves and transmits the error-free states across the links in parallel. This method scatters heavy data transmission for checkpointing over each individual link evenly and lessens communication congestion. The arrangement of distributed checkpointing storage makes checkpointing in parallel possible.

System reliability can be increased, in theory, by checkpointing to more than one neighbour. When a failure is detected, the system is recovered by rolling execution back to an error-free state and resuming the process again. The task left by the failed processor can be continued with small degradation on one of the neighbour nodes which has time redundancy, or migrated to a spare redundant processor to continue with the same performance as before the failure occurred.

Since the algorithm uses memory on the neighbouring processor to keep error-free states that must be consistent over the processors in the system and checkpointing has to be synchronized in a distributed manner, we name this algorithm as adjacent checkpointing and recovery sychronization (ACRS) algorithm.

This fault-tolerant algorithm is suitable for a system with a homogenous array architecture, where communication between processors is carried out through point-to-

point message passing. It assumes that parallel and distributed systems are loosely coupled, which implies no central physical clock, and no necessity for shared memory. Such systems provide a good basis for constructing fault-tolerant systems due to their inherent redundancy.

There have been many investigations of efficient consistent checkpointing algorithms. In the following sections, we firstly present our consistent checkpointing algorithm, and then discuss some existing consistent checkpointing algorithms and some other related work. By comparing ours with these existing algorithms, we exhibit the originality and advantage of the ACRS algorithm.

6.2 Adjacent checkpointing and recovery synchronization

The ACRS algorithm employs the backward error recovery concept [Ran77]. When a processor failure is detected, the affected task has to be logically removed by isolating the faulty node from the system, and replaced with an alternative version. In order to bring the alternative version of the task to a state consistent with that of the original task prior to its failure, it is necessary to make passive replicas for all target tasks to the separated storage and to checkpoint regularly to these replicas. In a large multiprocessor system, there are two possible places to store the passive replicas: shared devices or neighbouring nodes. The mechanisms for checkpointing to shared devices [Sie90] have been well studied. Our algorithm checkpoints to neighbouring processors as described below.

6.2.1 System overview

The AC algorithm takes advantage of a point-to-point connection feature in which an arbitrary topology can be formed by reconfiguring the connection between the nodes. A topology can be called well structured, if any node in the configuration has more than two distinct routes to reach any other node, in other words there is no single node in the

network where failure would cause a partition of the network. If a multiprocessor system is well structured, then any node in the system has at least two neighbouring processors. The algorithm in this paper considers well structured networks only.

This study was based on the toroidal mesh topology as in Figure 6.1. In this case, each node has four neighbours. The system is initiated by loading primary processes into corresponding processors before execution, then each node duplicates its primary processes onto its neighbours as secondary replicas. This duplication is conducted simultaneously between processors over the network.

When a processor receives the secondary processes, it installs them into free memory and initialises the newly created processes. The initialisation involves allocation of memory for saving the process state, ie for registers, stack, data, and inward and outward queued message status. This state information will be updated regularly with checkpoint data from primary processes. We call this updating of the state information of replicas "installing checkpoints". The newly created replicas are always suspended until they are invoked by the process manager.

The number of neighbours the secondary replica should be copied to is predetermined. Figure 6.1 is an example of mutual redundancy in a simple mesh connected network, in which each processor contains four external links and processes could be copied to all the neighbours. For example, in Figure 6.1 secondary task T_{05} of primary task T for processor 05 is replicated on its neighbouring processors. If more adjacent processors are used as locations for replicas, a more reliable system should result. However, if a processor fails, only one of its neighbours will take over its primary tasks. The decision is made through a rule of election among all the neighbours. If only one neighbour is used for backing up storage, such system is still quite reliable, as the probability of two adjacent processors failing at the same time is very low. Using only one secondary replica greatly reduces the memory and checkpoint communication

overheads, and also eliminates time during recovery spent deciding which processor will take over execution of the failed processes.

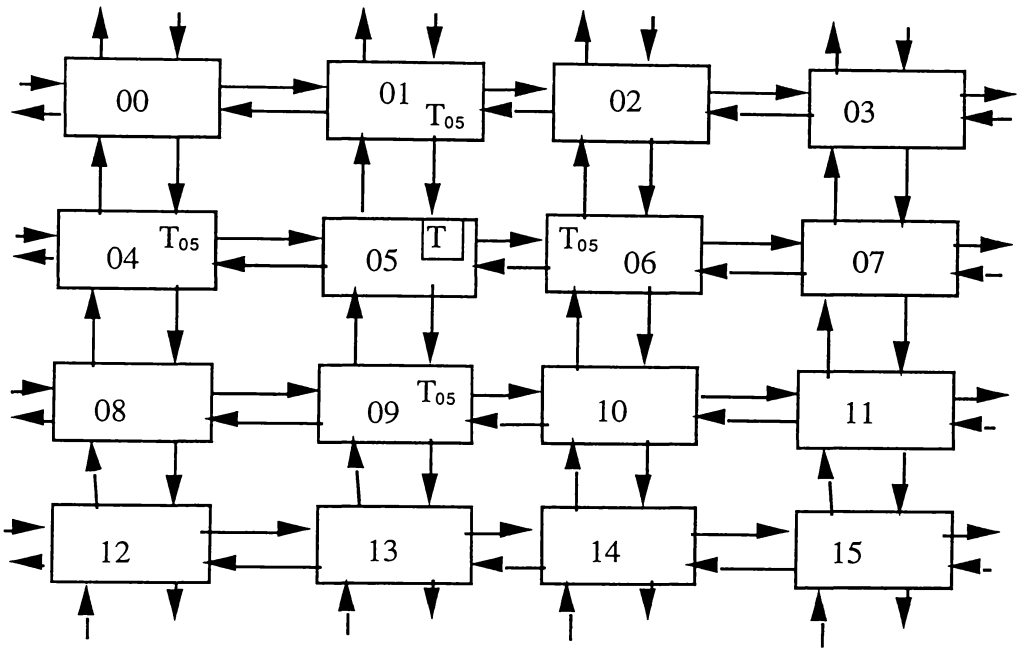


Figure 6.1 A simple mesh topology used for the fault tolerant algorithm study

When a processor failure is detected, the recovery action is to isolate the failed processor logically and migrate the execution of the failed primary processes onto an adjacent site or onto one of a few spare redundant processors. However, the emphases of migration between load balancing and fault tolerance are quite different. The primary use of process migration strategies [AF89; DO87; LS92; PM83] has been to accomplish load balancing in distributed computing systems. In order to make more efficient use of the power of multiprocessor systems, process migration allows processes on a heavily loaded processor to move to an idle or lightly loaded processor. In load balancing, both source processors and destination processors are live. But in fault tolerance, processes have to migrate from a dead processor to a live processor. The process migration in our fault-tolerant algorithm is achieved by removing the failed primary process and invoking its secondary replica. These redundant secondary processes on different processors are passive replicas, they will not be executed until their originals fail.

Process migration to tolerate processor failures requires cooperation with checkpointing techniques. The system must trace the progress of each primary process by regularly checkpointing to the process replicas. In this manner, if a processor terminates execution unexpectedly, its execution can be continued from the latest checkpoint on some other functioning processor which contains its processes' duplicates. The checkpointing algorithm determines the state of the process to be migrated, transferring it to the destination processor, and updating the state of the corresponding redundant process.

In the presence of failure, a neighbour of the failed processor which has time redundancy could execute both its own primary tasks and the migrated task. In this case, the processor taking over the migrated tasks will be heavily loaded and may cause a certain amount of performance degradation after recovery. This degradation is dependent on the time redundancy that the processor has. The solution to this problem is to add more processors to the network or to keep a few processors spare when initiating the system. Adding more processors would increase the time redundancy evenly on all processors and also expand the time tightness of performance. The latter option would retain the same performance as prior to failure, but it will take longer to recover from a failure, because the failure-affected secondary replicas would have to be transmitted to a spare redundant processor.

Other performance concerns are the penalty during normal operation and the delays caused by the recovery procedure. This penalty and delay can be reduced by design of efficient algorithms for checkpointing and network synchronisation. We discuss the details of these two challenging issues for the fault-tolerant algorithm separately below.

6.2.2 Adjacent Checkpointing

Checkpointing is used to periodically pass the current state of a process to its replica on a different site. It should include the necessary data to ensure the redundant process can

be restarted properly from the last checkpoint. The state of a process consists of two parts: the state of the computation, and the state of the processes communication with other processes.

Checkpointing for an individual task can be relatively simple, but coordinating the establishment of a set of consistent checkpoints (recovery line) between processes may be very difficult. The source of the difficulty is a phenomenon known as the domino effect. Many papers have proposed that consistent checkpointing is best suited for loosely coupled multiprocessor systems [Kim78; Koh81]. However their consistent checkpointing algorithms use a shared device for the storage of the secondary replicas [Pla93; CL85]. Details of their approach to consistent checkpointing are introduced in section 6.3.

One of the difficulties in checkpointing for multiprocessors is establishment of a recovery line which is formed by linking a set of recovery points between the tasks so that the system can be restarted with global consistency between the tasks. To avoid domino effects, a recovery line is necessary to keep interprocess actions consistent and their communication synchronised. The mechanism we adopted is centralised coordinating. A processor is chosen to play a role of master in the network. The master processor (Pm) sets the timing interval between each checkpoint installation. When it is time for a checkpoint it broadcasts a signal to the other processors. While each processor is checkpointing, the network will freeze actions which are irrelevant to checkpointing. The master processor will be acknowledged when a processor finishes its checkpointing. When it has received acknowledgments from all processors, it sends another signal to restart ordinary processing. In this manner, all newly set checkpoints form a consistent recovery line.

The processor which assumes the role of master may change between checkpoints to protect a master failing. The process on Pm which is used to synchronise the checkpointing in the multiprocessors is distributed over the processors, but with only

one acting at any time. If we assume that this synchronised controlling is an atomic action then the master processor can be replaced by invoking a replica of the process on one of the other processors if the current acting Pm fails during normal execution. The further investigation of this issue of Pm failure is left to future work.

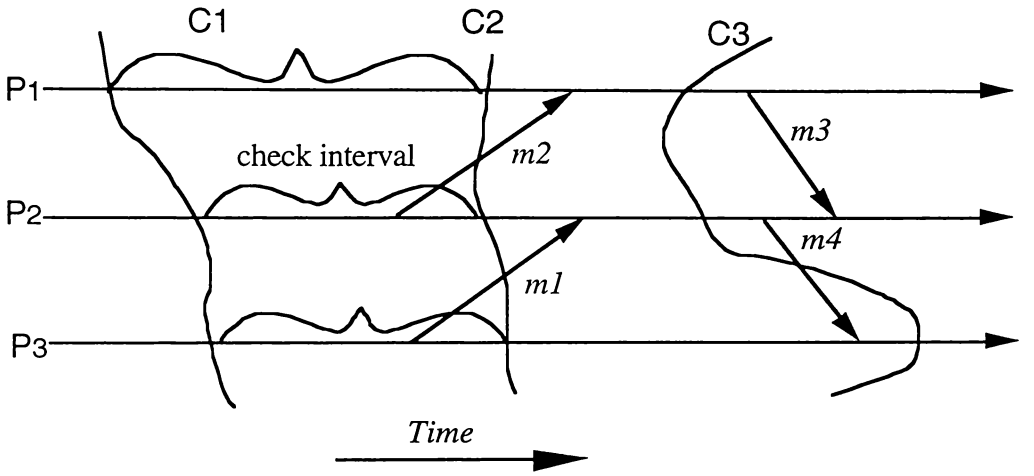


Figure 6.2 A Sample Distributed system

The standard way of visualising a distributed group of processors is to use horizontal lines that denote the relative progress of the processors over time. A message between processors is denoted by an arrow from the point at which the sending processor sends the message to the point at which the receiving processor receives it. A checkpointing action or a cut is represented by a dotted line that crosses each process's time line exactly once. A message is said to cross the cut line from left to right if its sending point is to the left of the cut line, and its receiving point is to the right of the cut line. Crossing from right to left is defined similarly. A consistent cut is a cut in which no message crosses the cut from right to left. Other cuts are called inconsistent. A checkpointing interval is the time between cuts.

Figure 6.2 shows a distributed system with three processors, four messages, three checkpointing actions (cuts) and checkpointing intervals between those cuts on each processor. The first two cuts ($C1$ and $C2$) are consistent as no messages cross them from right to left. The last ($C3$) is inconsistent due to message m_4.

The ACRS algorithm for checkpointing in a three processor system can be explained as shown in figure 6.3. Its protocol is also summarised as follows:

1. The system appoints one processor as a master node Pm, (can be a special coordinating processor or processor with lighter loading) This processor will start checkpointing and notify completion for the whole network.

2. Regularly, Pm stops running its target program and broadcasts a special message M1 to stop normal computing on every processor in the network.

3. Once a processor P receives M1, it stops its target program and sends an acknowledgment message M*a* to P*m*.

4. If Pm received Ma from all processors, it broadcasts a second message M2 to start checkpointing on all processors. Then Pm checkpoints itself. Otherwise, Pm will time out of the waiting state and call an exception handler for failure recovery.

5. When P receives this second message M2 from Pm, it checkpoints its local states across the links to its replicas on adjacent sites. After completing the checkpointing, it sends a message Mc indicating the completion of checkpointing on node P to Pm.

6. Once Pm has received Mc from all the other processors, the checkpointing on the whole network is complete. Pm notifies the other processors by broadcasting M3 allow normal computing, paused for checkpointing, to proceed.

7. When P receives this message M3, it performs tidy up activities: deletes the previous checkpoints; then restarts the target program.

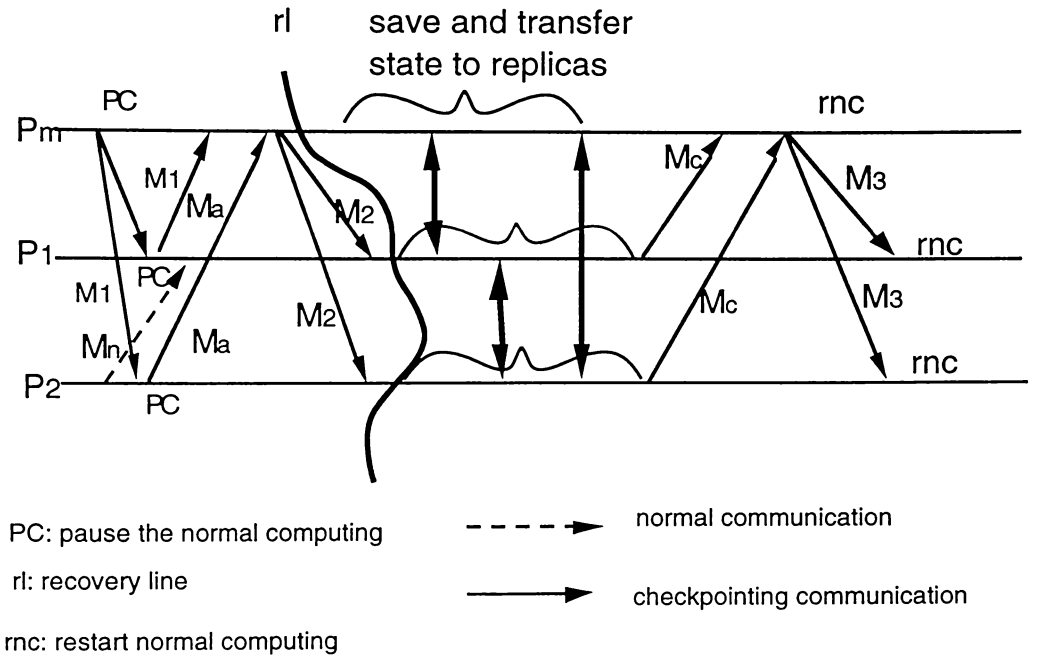


Figure 6.3 The proposed checkpointing algorithm at work

We can prove message forwarding in this situation will not violate the system consistency.

Theorem. A message initiated from a source processor and arriving at a destination processor in different checkpointing interval, providing the message is suspended on a processor if checkpointing started before it reaches its destination, will not violate the consistency of the network performance.

Proof. There are three conditions where this theorem could be false. They are (a) the message is lost, (b) the processor suspending the message fails and (c) one of the processors on the path from message source to its destination fails. We verify the correctness under these three conditions separately as follows:

(a) A processor P_i sends a message M_n to P_j through the route (P_i,... P_k, P_w,... P_j), if the message is lost from P_k to P_w, P_k will not receive the acknowledgment from P_w. P_k will either resend the message or report the failure to P_m. If resending is successful then the message will not violate the consistency. Otherwise, P_m will invoke

a recovery action, the execution will roll back from previous checkpoint and reinitiate the message via an alternative route. Therefore the message cannot violate the consistency.

(b) A processor P_i sends a message M_n to a processor P_j through the route (P_i,... P_k, P_w,... P_j). Assuming the network is scheduled for *m*th checkpointing as the message reached onto processor P_k, the P_k's failure before forwarding the message to the next processor will cause the execution roll back to *(m-1)*th checkpoint. The message lost on processor P_k will be reinitiated and reach its destination through an alternative route.

(c) If P_k suspends a message from P_i to P_j during *m*th checkpointing, the message forwarding will be continued after completion of the checkpointing. Assuming a processor P_w on the path from P_k to P_j is failed, after completing the mth checkpointing and before the message reaches onto it, the execution will roll back to mth checkpoint and will re-forward its suspended message again. This message will not violate the consistency. []

Transferral of the state of the process communications generates problems of how to handle current and pending messages during checkpointing and if a failure occurs, it creates the problem of how to retransmit pending messages and to re-route future messages headed to the failed processor. In process migration for load balancing, pending and future messages can be forwarded by the source site. But for fault tolerance, the failed node is unable to forward any message so all the messages since the last checkpoint have to be retransmitted to the migrated processor directly.

During failure recovery, the system will suspend network activity in order to handle future communication to the old location. It updates the routing list on each processor and enroute messages with new information. The enroute messages are the messages in the output queue when network freezes the activities for checkpointing. The snapshot

of the local process checkpoint on each processor includes the state of the enroute messages. It is used to roll back to the checkpoint by copying enroute message back to the current message queue and re-delivering them. If there are messages addressed to the failed node, their destinations have to be updated. The protocol used for recovery control is explained in next section.

6.2.3 Failure recovery synchronisation control

The checkpointing algorithm is only used to save the error-free state periodically and to provide the condition which enables the system to roll back to a previous state to continue execution. The checkpointing algorithm is not responsible for failure detection and recovery. Failure detection and recovery are dealt with in a separate component facility based on the hybrid diagnosis algorithm we discussed in Chapter 5. During execution of the application tasks, regularly, the system invokes the diagnosis process to diagnose the network health about communication links and processors. The unhealthy components identified will be broadcasting to the network immediately. Once a failure or fault is detected and the information about the faulty component is broadcast to the network, the system takes action to recover from the failure. The algorithm for this fault recovery is discussed as follows.

When the type of failure is that of broken links, for example, a link between P_i and P_j has failed, then the only recovery action is to find substitute paths which are the shortest from P_i to P_j and from P_j to P_i in the routing list to replace the paths broken by the link between P_i and P_j. After the routing list modification on both P_i and P_j, P_k (where k=i if $i < j$ otherwise k=j) informs the network that the failure has been corrected and signals processing to restart on the network.

When the failure is a dead processor, those processors which contain the replica of the failed processor, and which are not loaded with other extra tasks left by a failed processor, broadcast messages over the network to apply for taking over the tasks.

Each processor receives the application message then makes its local decision about a winner based on a rule. The rule for this local decision is that the winner is always the processor with the smallest ID number among the applicants for taking over the tasks. When a processor has made its local decision, it sends its decision to all others in the network. The global decision about the winner is made by majority voting on each individual processor.

The reason for making majority voting for the global decision is to avoid misinterpreting the individual local decision. Although we don't deal with the full degree of Byzantine failures and also the types of the failure are considered in diagnosis process, a majority voting here is to guarantee that this global decision will not be affected even by a processor with the behaviour of Byzantine failure at this stage. The final winner broadcasts a message to indicate it is the one that is taking over failed tasks left by the failed processor, and to restart the ordinary performance from the last checkpoint.

An exceptional case is that the processor which took over a task left by a failed processor could fail as well. If this double loaded processor fails, the system has to be able to migrate its primary and secondary processes to different neighbour processors. A double loaded processor has to be responsible to the checkpointing for both its primary task and the task taken over from the failed processor. For the secondary taken-over task, it can be done either to one of the neighbours of the failed processor through an intermediate processor, or by creating new replicas for the secondary task to neighbours of the currently acting processor. Further details about recovery from double loaded processor failure are left to future work. However the probability of two adjacent processor failures is relatively low.

To give a bit of insight into how this recovery synchronisation control algorithm works, we present an example in which the multiprocessor array in Figure 6.1 takes

recovery action from a node failure. Although this multiprocessor network is very small and simple, it is sufficient to illustrate our points.

Example 1. In this example, we discuss link failure and processor failure separately. We assume that the types of the failures and their location have been advised to the network before this algorithm is applied. We show how they take recovery actions and resume the execution.

Firstly, we assume that the link between P_6 and P_{10} has failed, and the network has been informed of this failure. The algorithm proceeds as follows.

* Both P_6 and P_{10} go through their path routing list to find the shortest substitute routing via which they can reach to each other and replace broken routing.

* The P_6 informs of the completion of recovery and readiness to resume normal processing by broadcasting a message to the network.

* Every processor in the network rolls back to the last error-free state and start the execution.

Secondly, if the fault resides in processor 10 and it has processors 6, 9, 11, and 14 as its neighbours, all containing a copy of replicas of the processes on node 10, then the progress of events is detailed below (refer to Figure 6.1).

* Each of processors, P_6, P_9, P_{11} and P_{14}, broadcasts a message over the network to apply for taking over the tasks left by processor 10.

* When processor P_6, P_9, P_{11} and P_{14} have received application messages, they select P_6 as their local decision about the winner, and broadcast the decision to the network. This is because we assume that the system has a protocol of the candidate

processor which states that the one with smallest ID will win the election.

* When P_6, P_9, P_{11} and P_{14} have all received decisions from the other three, all of them know that P_6 is the global winner, taking over the tasks of the failed processor.

* P_6 broadcasts a message to the network to notify that it is the one which executing the tasks left by P_{10}.

* After having received this notification message, processors P_6, P_9, P_{11} and P_{14} in the network update the network state including path routing lists, to forward future messages destined for P_{10} to P_6, then roll back to the last error-free state and resume performance together with others in the network.

Both the AC algorithm and the algorithm for recovery synchronization control should be implemented inside the kernel as it assumes that it will have access to basic functions: I/O management, packet routing control, memory management, process management and interface between tasks and I/O management. The algorithm must access the primitives served by I/O management to communicate with other processors in the network to maintain their synchronisation and consistency.

In the next section, we introduce some other consistent checkpointing and restarting algorithms as well as some similar efforts in achieving checkpointing. After describing each algorithm, we compare the differences between our algorithm and other efforts.

6.3 Related work

6.3.1 The Chandy & Lamport global state detection algorithm

Consistent checkpointing schemes stem from a paper by Chandy and Lamport [CL85]. Their algorithm attempts to detect the global state in a distributed system. The Chandy-

Lamport algorithm has all processors periodically define a cut and make a distributed checkpoint. The algorithm works as follows: A processor initiates the algorithm by recording its local state, and then broadcasts a special *marker* message m_{CL} to all processors to which it has a channel. All processors p then perform the following:

* If p receives m_{CL} and has not taken its local checkpoint yet, then it takes its local checkpoint, and broadcasts to all processors to which it has a channel.

* Afterwards, if p receives a non-marker message m along a channel c, and it has not yet received m_{CL} on c, then m is a message which should be in the current recording state and must be logged. Otherwise, m is a message which should be in the next recording state.

* When all processors have received m_{CL} on all their incoming channels, then there are no more incoming messages which should be in the current recording state to be logged: the checkpoint is finished.

The Chandy & Lamport algorithm is correct because it defines a consistent global state in a distributed system. Specifically, it logs all cross-cut messages that don't generate domino effects and leaves non-cross-cut messages. Thus they are all identified and logged properly.

In the Chandy & Lamport algorithm, any cross-cut message must arrive at a node after it has checkpointed, but before it receives m_{CL} from the sender. Although the algorithm doesn't require freezing of all processors, instead, it only suspends process executions on its own processor for local state recording. The time spent on local state recording and waiting for receiving m_{CL} from all incoming channels may be longer than the time which is used for our AC algorithm in a medium size network. For example, in a simple network which consists of three nodes with a tree structure as

shown in figure 6.4. Assume that p1, p2 and p3 spend t1, t2 and t3 time length for their local state recording. When p1 has recorded its process state, it broadcasts m_{CL} to its neighbours p2 and p3. After both p2 and p3 have finished their state recording and send a m_{CL} back to p1, p1 can terminate its current state recording. In such case, state recording time for p1 is t1 + max(t2, t3) + message transmitting time. However in our AC algorithm, checkpointing time for this structure would be max(t1, t2, t3) + message transmitting time. In a loosely couple parallel processing system, interprocessor communication links are relatively short and fast. So max(t1, t2, t3) + message transmitting time could be less than t1 + max(t2, t3) + message transmitting time.

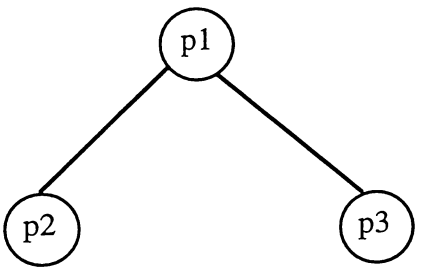


Figure 6.4 A simple distributed system

Furthermore, if the network being checkpointed is fully connected (the worst case: every processor has a channel to every other processor), every processor must send m_{CL} to every other processor so that the algorithm requires $O(n^2)$ messages with n processor. Comparing to O(n) in AC algorithm, this is a less efficient algorithm.

6.3.2 The Network Sweeping Algorithm

Li, Naughton, and Plank developed a network sweeping (NS) algorithm [LNP92] which reduces the number of marker messages that a processor must send. This scheme makes use of the special properties of wormhole routing networks to satisfy this new set of requirements. The number of the marker messages required in the NS algorithm is less than $O(n^2)$.

The NS algorithm uses three types of marker messages to determine a consistent cut, to

identify cross-cut messages, and to guarantee that there are no more cross-cut messages. These three types of markers are the:

1. m_{CL} message which is used to define a consistent cut and help define the cross-cut messages. Unlike the m_{CL} message in the Chandy & Lamport algorithm, it need not be sent to all neighbouring processors. The m_{CL} message in the NS algorithm can be sent between arbitrary pairs of processors in a multiprocessor system.

2. m_{NS} message which used to determine when there are no more cross-cut messages for the processor to log. The m_{NS} messages are sent only to neighbouring processors i.e. those that have a physical channel between them.

3. m_S message which is used by all processors to determine when the checkpoint is over.

The NS algorithm assigns one processor p_C to be the coordinating processor which initialises and terminates the checkpoint. When it is time to initialise checkpoint i, p_C first takes a local checkpoint, then broadcasts m_{CL} to all other processors. All other processors perform as follows:

(1) When p receives first m_{CL} or m_{NS}, then p takes its local checkpoint.

(2) After taking its local checkpoint, p may start sending m_{NS} messages to the processors which are directly connected to p by a physical link. It does so when it is sure that no more cross-cut messages can be sent along that link. The processor p is finished sweeping if p has sent m_{NS} on all its outgoing links and received m_{NS} on all its incoming links.

(3) Between starting and finishing the i th checkpoint, if p sends a non-marker message m to a processor p which has a direct channel to p, and p hasn't sent m_{CL} to p', then it must send m_{CL} with a sequence bit of (i mod 2) to p' before sending m.

(4) Between starting i th checkpoint and finishing i th sweeping, if p receives a non-

marker message m from processor p', and p has not yet received m_{CL} from p', then m is a cross-cut message so p logs m.

(5) When p has finished sweeping, then it knows that it will receive no more cross-cut messages. then, p commits all of its checkpointing information (including its local process states and message log) to disk. Afterwards, it notifies p_C by sending it m_S.
(6) When p_C receives m_S from all processors, checkpoint i is over. It broadcasts m_S back to all processors to let them know. After p_C has sent m_S and all other processors have received m_S, i th checkpoint is over, the system is free to start the next checkpoint.

If L(n) is the number of physical communication links in the network, then the NS algorithm requires O(L(n)) marker messages. On a hypercube, L(n) is O(nlogn); on a mesh connected multiprocessor system, L(n) is O(n). Compared to the Chandy & Lamport algorithm, the NS algorithm reduces the number of marker messages greatly. It allows the application programs on each processor to be resumed asynchronously. Some processors could finish their checkpoint earlier than others that take indefinite time for waiting m_{NS} from their all incoming channels. This might be efficient for a distributed system on which processes are independent of others, or small parallel systems on which checkpointing time difference could be significant. However, in a large parallel system, the system performance usually requires synchronization over the network. Systems that have particularly completed their checkpoints could not continue their performance without their dependent processor's cooperation.

6.3.3 The Sync-and-Stop Algorithm

After publication of our ACRS algorithm [TD92], Plank presented a Sync-and-Stop (*SNS*) algorithm in his Ph.D thesis [Pla93]. The *SNS* algorithm has great similarity to our ACRS algorithm in checkpointing synchronization, as both algorithms require that the processors first synchronize to define a consistent cut, and then stop to take a global checkpoint. The SNS algorithm can be briefly described as follows:

(1) One centralised processor regularly initiates checkpoints by broadcasting a marker message, m_{SNS}, over the network.

(2) When the m_{SNS} arrives on a processor p, p suspends its application program. If p has sent a message m to another processor p' before receiving m_{SNS}, then p has to wait for acknowledgment for m from p' before moving on.

(3) After all acknowledgments have been received, p sends m_{SNS} back to the central control processor.

(4) When the central control processor has received m_{SNS} from all other processors, it rebroadcasts m_{SNS} and saves its local checkpoint to disk.

(5) Whenever the second broadcast m_{SNS} reaches a processor, p, it saves its local checkpoints to disk. After completing its local checkpoints, it sends a m_{SNS} message to inform the central control processor.

(6) When the central control processor has received m_{SNS} to notify the completion of checkpoints from all processors, it broadcasts a final m_{SNS} message to all processors to announce the completion of the checkpointing. After all processors have received this final message, the application programs on each processor resume their normal processing until next checkpoint command.

The *SNS* algorithm requires all processors to save and transmit their error-free state to shared disk storage. The maximum system reliability improvement with *SNS* algorithm is limited by the reliability of this shared disk device. Failure of the shared device may crash the whole network in normal operation. Moreover checkpointing messages from n resources are sent to one destination at the same time. This generates communication congestion so that the parallel action results a sequential performance.

Comparing *SNS* with the ACRS algorithm, which checkpoints to neighbouring processors, the states of checkpoints are distributed over the network with multiple copies. Individual failure of a site would be unlikely to affect system health. During

checkpointing, each processor transfers its data through different links to distinct destinations. This parallelizes checkpoint processing completely and reduces the time which is required for checkpointing significantly.

Furthermore, according to the stage (2) of the *SNS* algorithm, a processor has to wait for the acknowledgment of cross-cut messages before proceeding further. For example, if a processor Pi receives an initiating checkpoint message m_{SNS} from the central control processor immediately after having sent out a normal message m to a processor Pj (this m is a cross-cut message), then Pi could not signal the readiness for checkpointing to the central control processor with message m_{SNS} until it has received the acknowledgment for m from Pj. In a large multiprocessor system, the big distance between Pi and Pj can delay checkpointing severely.

However, in our AC algorithm, message acknowledgment is between two directly connected processors. For example, the route for Pi to send a message m to Pj is (Pi, Pk,.....Pw, Pj). Pi is acknowledged by Pk before Pk forwards the message further. This saves the time for Pi to wait for the reply from Pj before proceeding. If Pk receives m after having paused its normal target processing for checkpointing, it logs the message and forwards it as soon as checkpointing is completed. In this manner, the *ACRS* algorithm reduces the time used in the *SNS* algorithm waiting for acknowledgments of cross-cut messages.

Summary

The ACRS algorithm presented in this chapter is specially designed for parallel and distributed structures. It has the following four features:

1. It is able to tolerate at least k -1 processor concurrent failures in the worst case in multiprocessors which are k regular connected. The worst case is that all k-1 failed

processors are the neighbours of the same processor. It should be noted that the possibility of the worst case emerging is relatively low.

2. The states of checkpoints are distributed over the network so that the system reliability is not limited by any individual devices.

3. Checkpointing for the network is carried out in parallel. Because the algorithm is specially suitable for multiprocessors interconnected with point-to-point package switch communication protocol, it has wider communication bandwidth. Data saving and transferring is evenly distributed over the network, and it has less communication congestion.

4. The checkpointing is synchronised over the entire network by defining a consistent cut first, and then stopping to take a global checkpoint. There is no need to distinguish a cross-cut message, because all possible messages arriving during checkpointing, are cross-cut messages. This method simplifies the checkpointing procedure significantly and is best adapted to synchronised parallel processing applications.

Many other consistent checkpoint algorithms, such as Chandy & Lamport, Network sweeping, and SNS, have similarity to ACRS algorithm. The first two are asynchronous checkpointing algorithms, and they do not require that the whole network suspends its normal execution, instead it only stop its local processing for checkpoints. But they can be delayed while ensuring that there are no more incoming cross-cut messages, and also since starting and finishing checkpointing for a same global state could have a big time difference, the algorithms are not very appropriate for synchronised parallel processing applications. The *SNS* algorithm synchronises to define consistent checkpoints and it contains the features which are required by parallelism. But its reliability improvement can be limited by the shared disk device for checkpoint storage and it can cause bottleneck effects because of data saving and data

transmitting from n resources to one destination.

The idea of consistent checkpointing in which all processors coordinate to stop target programs and resume them after completion of the checkpointing is different from blocking checkpointing introduced in Borg's algorithm [BBG83]. The Borg's algorithm requires that every message be logged synchronously as it is sent or received. This synchronous logging of messages was achieved by blocking the receiver until the message is logged, or by blocking the receiver if it attempts to send a new message before all received messages are logged. In his scheme, each process is backed up by a mirror process and all messages have to be sent to all mirrors of destination that means logging of messages.

The advantage of the blocking checkpointing algorithm is quick recovery, but the overhead of synchronously logging every message makes this scheme inappropriate for multiprocessor application checkpointing. Our algorithm does not use blocking interprocessor communication except during the checkpointing period. Blocking checkpointing is specially used for a shared memory multiprocessor system. The blocking checkpointing treats data saving and transmitting for checkpoints as atomic actions which blocks the memory being used for checkpoint to prevent interference from other processors.

Our adjacent checkpointing and recovery synchronization algorithm can improve the system reliability significantly and it performs efficiently both in the absence of failure and in recovering from a failure. This algorithm is able to provide the system with continued services in the presence of multiple failures with some degree of performance degradation. Analysis of the reliability improvement and performance variations of the algorithm are introduced in Chapter 7 and Chapter 8 respectively.

Chapter 7

Reliability evaluation of ACRS Algorithm

7.1 Introduction

The principles and the design of ACRS algorithm have been described in the previous chapter. Here we supply the quantitative evaluation of reliability improvement when the algorithm is applied to a multi-transputer network with toroid mesh topology. In the mesh configuration network, our ACRS fault-tolerant algorithm can tolerate up to $\lfloor \frac{n}{2} \rfloor$ processor failures in the best case. In the worst situation, the algorithm can provide correct operation in the presence of three processor failures where failed processors are neighbours of the same processor. This evaluation is modelled with Markov methods [GT90]. It shows that the system reliability has been much improved with the support of the ACRS algorithm.

As addressed earlier, reliability of a system, R(t), is a function of time, defined as the conditional probability that a system will perform correctly throughout the interval $[t_0,t]$, given that the system was performing correctly at time t_0. Defining $R_m(n, k, t)$ as the reliability of a toroid mesh system supported by the ACRS algorithm, where n is the number of modules used in the system, k is the maximum number of failed modules the system can tolerate, and t is the time when reliability is considered. When a system can tolerate no module failures, its system failure rate is the same as that in a system with a pipeline structure.

7.2 Processor pipeline

In the pipeline architecture, each of the elements is required to function for the system as a whole to function correctly. If we consider only reliability based on the processor modules, then the reliability of the entire system can be calculated as the probability that none of the modules will fail. We define $R_i(t)$ as the reliability of module i at time t, and $R_p(t)$ as the reliability of the whole pipeline system. By definition, we have $R_p(t) = \prod_{i=1}^{n} R_i(t)$. If each module i has a constant failure rate of λ_i and satisfies the exponential failure law such that the reliability of each module is $R_i(t) = e^{-\lambda_i t}$ [4,5,6] then $R_p(t) = e^{(-\sum_{i=1}^{n} \lambda_i t)}$, the system failure rate is $\lambda_s = \sum_{i=1}^{n} \lambda_i$. If all the modules have the same failure rate λ then R_p can be simplified further as $R_p(t) = e^{-n\lambda t}$. That is $R_m(n,0,t) = R_p(t) = P_n(t) = e^{-nlt}$. The calculation procedures used in this section can be found in [GTP91].

7.3 The ACRS with toroid mesh topology

The ACRS parallel processing system with toroid mesh topology can tolerate multiple processor failures. The more failures it can tolerate, more reliable the system will be. The reliability improvement of the ACRS algorithm with toroid mesh connection can be modelled by a Markov process.

The two main concepts in the Markov model are system state and system transition. The system state is a combination of faulty and fault-free modules [Joh89]. We can define the state of this system as $S = (S_1, S_2, \ldots S_i, \ldots Sn)$ where $S_i = 1$ if module i is fault-free, otherwise $S_i = 0$. There will be 2^n possible states (1,1, ..., 1) ... (0,0, ..., 0) in a system which consists of n modules.

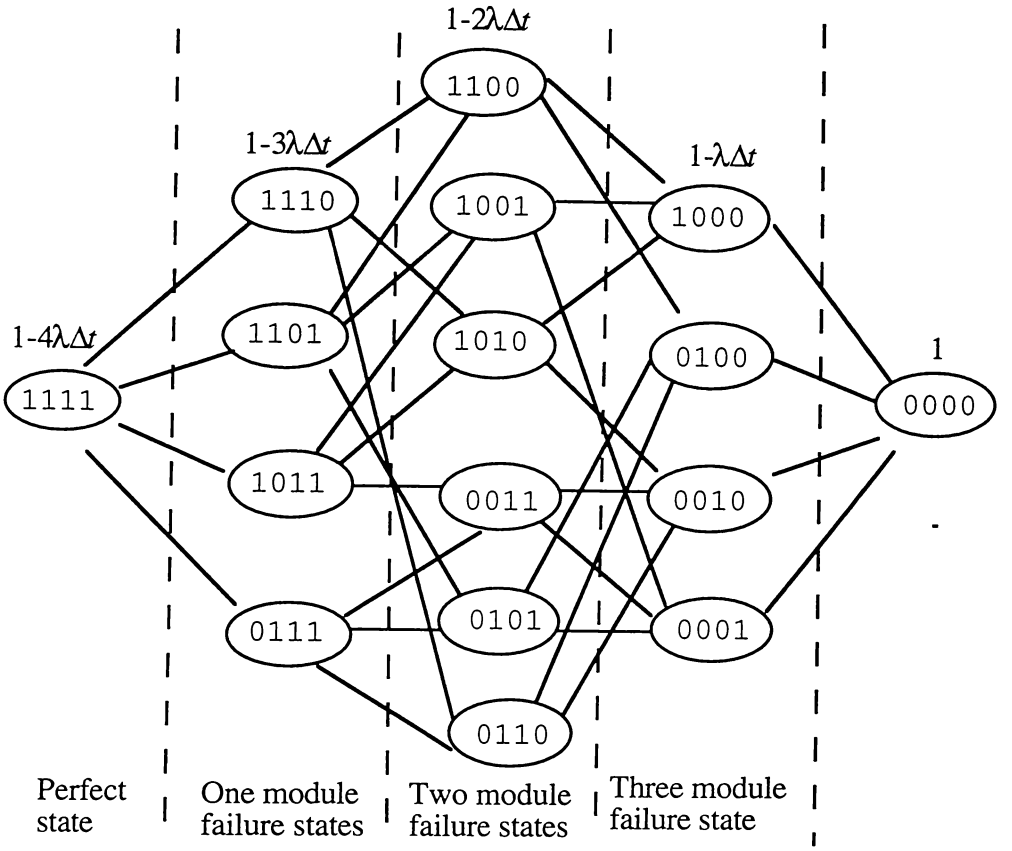


Figure 7.1 All possible states and transitions for the Markov model of a four processor module system. If the system can tolerate the failure of two modules then only the three and four module failure states represent complete failure of the system. The formula at the head of each column of states shows the probability of remaining in that state. The probability of each change of state is λΔt.

If all the elements in a state are 1, for example (1,1,1) for $n = 3$ the state is a perfect state in which all the modules are fault-free. Positions with 0's in the state show faulty processors, thus (0,0,1) means that modules 1 and 2 are faulty. A state transition is a change from one state of the system to another state. Figure 7.1 shows all the possible states and transitions for a Markov model of a four transputer mesh-like system, which can tolerate two processor failures.

We now assume that each processor module in the toroid mesh system obeys the exponential failure law, and all of the modules have some constant failure rate of λ. The probability of a module being faulty at some time $t_0 + \Delta t$, is given by $1 - e^{-\lambda \Delta t}$

Applying this to our toroid mesh system with $n = 4$, the probability of the perfect state is 1 minus the sum of the probabilities of transition to one of the module failure states. The probability that the system stays in the state (1,1,1,1) is $1 - 4\lambda\,\Delta t$. In the same manner we can derive the probability for each of the other states keeping its state. These results are also shown in Figure 7.1.

The system will be in the failure state after three modules have failed. Figure 7.1 presents *all* of the possible system states, but this is equivalent to an alternative form of Markov model with minimum states, shown in Figure 7.2. The corresponding model for n states is shown in Figure 7.3.

(assuming the module was not faulty at time t_0), Δt is a time period in which the module can fail. If Δt is very small, then $1 - e^{-\lambda\Delta t} \approx \lambda\Delta t$.

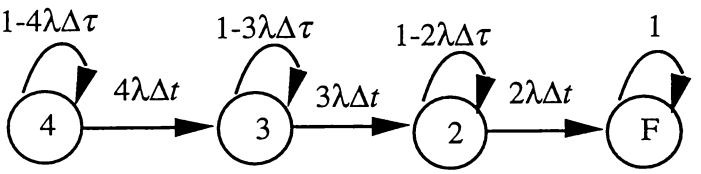


Figure 7.2 Reduced Markov model of a four processor system that can tolerate two failed processors. This is derived from Figure 7.1 by combining states with the same number of processor failures, and representing as the failed state those with either three or four failures. The figure in the state shows the number of operational processors, and the state transition probabilities.

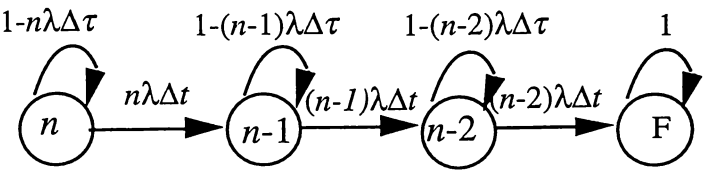


Figure 7.3 Reduced Markov model of an n processor system that can tolerate two processor failures

7.4 Derivation of the reliability of a system tolerant of two module faults

Assuming $P_i(t)$ is the probability of the system being in state i at time t, and $P_F(t)$ is the probability of being in state Failure at time t,

Then

$$P_n(t+\Delta t) = (1 - n\lambda\Delta t)P_n(t) = P_n(t) - n\lambda\Delta t P_n(t). \quad \text{(no failures)}$$

$$P_{n-1}(t+\Delta t) = n\lambda\Delta t P_n(t) + (1-(n-1)\lambda\Delta t)P_{n-1}(t)$$

$$= n\lambda\Delta t P_n(t) - (n-1)P_{n-1}(t) + P_{n-1}(t) \quad \text{(one failure)}$$

$$P_{n-2}(t+\Delta t) = (n-1)\lambda\Delta t P_{n-1}(t) + (1-(n-2)\lambda\Delta t P_{n-2}(t)$$

$$= (n-1)\lambda\Delta t P_{n-1}(t) + (n-2)\lambda\Delta t P_{n-2}(t) + P_{n-2}(t) \quad \text{(two failures)}$$

$$P_F(t+\Delta t) = (n-2)\lambda\Delta t P_{n-2}(t) + P_F(t) \quad \text{(three, and therefore system, failure)}$$

It readily follows that;

$$\frac{P_n(t+\Delta t) - P_n(t)}{\Delta t} = n\lambda P_n(t)$$

$$\frac{P_{n-1}(t+\Delta t) - P_{n-1}(t)}{\Delta t} = n\lambda P_n(t) - (n-1)\lambda P_{n-1}(t)$$

$$\frac{P_{n-2}(t+\Delta t) - P_{n-2}(t)}{\Delta t} = (n-1)\lambda P_{n-1}(t) - (n-2)\lambda P_{n-2}(t)$$

$$\frac{P_F(t+\Delta t) - P_F(t)}{\Delta t} = (n-2)\lambda P_{n-2}(t)$$

when $\Delta t \rightarrow 0$ then

$$\frac{dP_n(t)}{dt} = n\lambda P_n(t) \qquad \frac{dP_{n-1}(t)}{dt} = n\lambda P_n(t) - (n-1)\lambda P_{n-1}(t)$$

$$\frac{dP_{n-2}(t)}{dt} = (n-1)\lambda P_{n-1}(t) - (n-2)\lambda P_{n-2}(t) \qquad \frac{dP_F(t)}{dt} = (n-2)\lambda P_{n-2}(t)$$

Using the Laplace transform we obtain:

$$SP_n(S) - P_n(0) = n\lambda P_n(S)$$

$$P_n(S) = \frac{1}{S+\lambda n}$$

$$SP_{n-1}(S) - P_{n-1}(0) = n\lambda P_n(S) - (n-1)\lambda P_{n-1}(S)$$

$$P_{n-1}(S) = \frac{\lambda}{(\frac{S}{n}+\lambda)(S+\lambda+\frac{\lambda}{n})}$$

$$SP_{n-2}(S) - P_{n-2}(0) = (n-1)\lambda P_{n-1}(S) - (n-2)\lambda P_{n-2}(S),$$

$$P_{n-2}(S) = \frac{(n-1)\lambda^2}{(\frac{S}{n} + \lambda)(S + \frac{\lambda+1}{n})\lambda(n-2)\lambda+S}$$

Using the inverse Laplace transform we can derive:

$$P_n(t) = e^{-n\lambda t}, \qquad P_{n-1}(t) = ne^{-(n-1)\lambda t} - ne^{-n\lambda t}$$

$$P_{n-2}(t) = \frac{(n-1)\lambda P_{n-1}(S)}{S + (n-2)\lambda} = \frac{(n-1)\lambda}{S + (n-2)\lambda}\left(\frac{-n}{S+\lambda n} + \frac{n}{S+ (n-2)\lambda}\right)$$

$$= \frac{n(n-1)}{2}(-e^{-(n-2)\lambda t} + e^{-n\lambda t}) + n(n-1)(e^{-(n-2)\lambda t} - e^{-(n-1)\lambda t}).$$

$$P_F(t) = 1 - P_n(t) - P_{n-1}(t) - P_{n-2}(t).$$

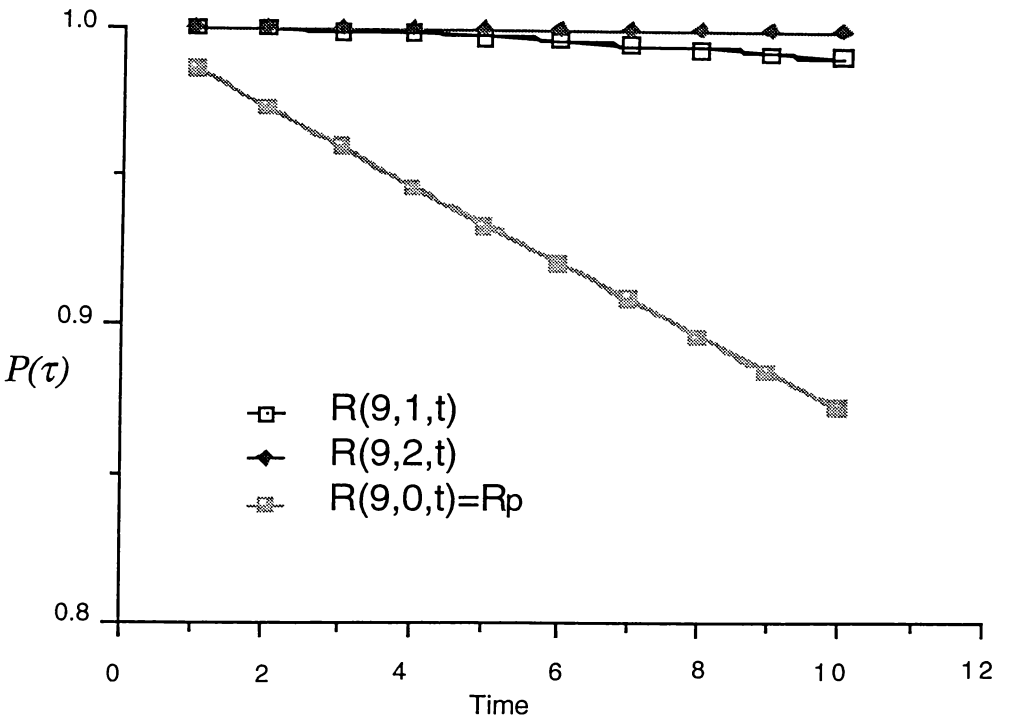


Figure 7.4 Reliability as a function of time for systems that can tolerate 0, 1 and 2 processor failures, for $\lambda = 0.01$

$R_m(n, k, t)$ represents the reliability of a multiprocessor with the toroid mesh connection supported by the ACRS fault-tolerant algorithm, where n is the number of modules used in the system, k is the maximum number of failed modules the system can tolerate, and t is the time when reliability be considered. The failure rate of the pipeline system is the same as the rate for a toroid mesh system which can tolerate no module failures i.e.

$$R_m(n,0,t) = R_p = P_n(t) = e^{-n\lambda t}.$$

The reliability of a system that can tolerate one module failure is then the sum of the probabilities that the system will experience one or zero failures:

$$R_m(n,1,t) = P_n(t) + P_{n-1}(t) = ne^{-(n-1)\lambda t} - (n-1)e^{-n\lambda t}$$

and the reliability of a system that can tolerate two processor failures is obtained by summing the probabilities of zero, one and two failures:

$$R_m(n,2,t) = P_n(t) + P_{n-1}(t) + P_{n-2}(t)$$
$$= \frac{n(n-1)}{2} e^{-(n-2)\lambda t} - n(n-2) e^{-(n-1)\lambda t} + \frac{(n-1)(n-2)}{2} e^{-n\lambda t}$$

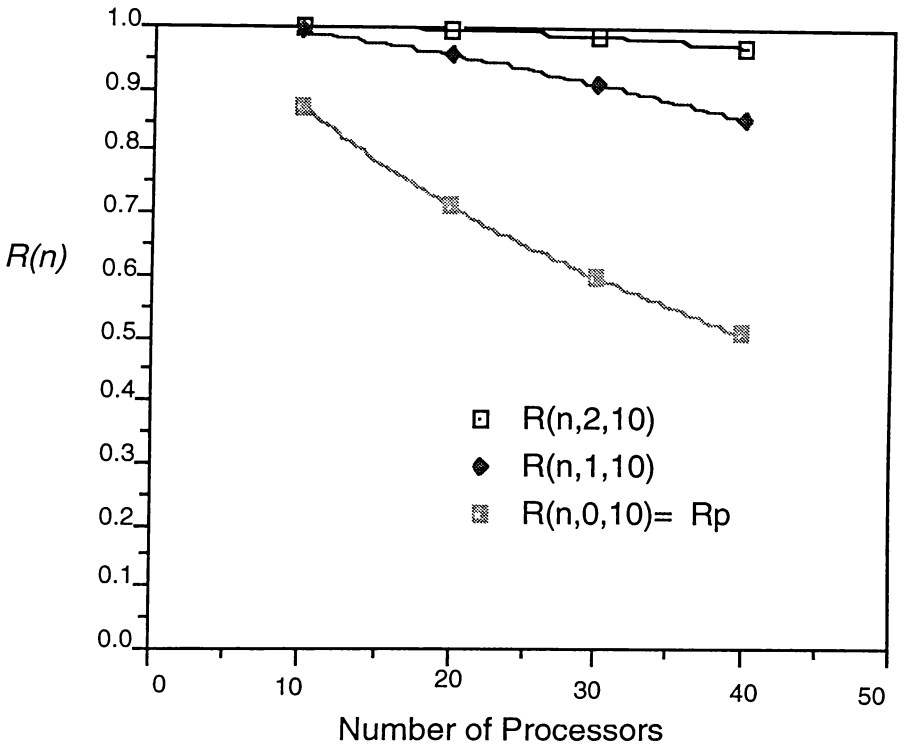


Figure 7.5 Reliability of systems that can tolerate 0, 1 and 2 processor failures as a function of the number of processors in the system

As an illustration the reliability has been computed against time for a system of nine processors (the number installed at the NZMS), for the cases where the system can tolerate zero, one, and two processor failures (Figure 7.4). Here we have assumed that $\lambda = 0.01$, that is there is a 1% chance of failure in unit time interval. The lowest line in the diagram shows the reliability of a system that can tolerate no module failures, this

corresponds to R_p (the reliability of the pipeline system), or $R_m(9, 0, t)$. It is clear that the reliability of the systems supported by the ACRS algorithm in a toroid mesh architecture is very much improved compared to the original system. Figure 7.5 shows the reliability of systems as a function of n, the number of modules, for a time period of ten units, to illustrate how the reliability varies with the size of the system.

7.5 Summary

The reliability improvement with the ACRS algorithm has been analysed for a toroidal mesh system and it shows the modelling results of zero, one and two failures. Analysis shown that reliability is improved. This modelling for ACRS can be extent to a system with broad topology in a similar way. The results also show that a given reliability requirement, when the size of a network is large, the system is required to tolerate more processor failures to meet the reliability standard.

Chapter 8

Performance Evaluation

8.1 Introduction

From Chapter 6, we know that the ACRS fault-tolerant algorithm improves system reliability greatly. However the trade-off between reliability improvement and performance has been a difficult issue to balance. An inefficient checkpointing algorithm could seriously affect system performance in error-free operation, and many applications require fast recovery to meet time constraints. In this chapter we estimate the performance penalty imposed by the ACRS algorithm in the fault-free situation and judge the performance degradation when the system suffers one or more processor failures.

The performance and operation studies of the ACRS algorithm were tested with discrete event simulation technique under the Simscript programming environment [MHK63]. The computation model used for this performance test is a parallel image processing application. Simulation is a research technique which has frequently been employed to predict the performance of a proposed system. It may give understanding and insight in situations where statistics cannot be applied. It is possible to trace out in detail the consequences and implications of a proposed course of action. Simulation is often used as an alternative to the more structured deterministic methods of problem solving.

A discrete event simulation model assumes the system being simulated only changes

state at discrete points in simulated time [Bir79; Fuj90; Gor78]. The simulation model jumps from one state to another upon the occurrence of an event. The simulator of the ACRS fault-tolerant algorithm includes state variables to indicate the length of message queues, the status of communication links (busy or idle), etc. Typical events might include the arrival of a message (including checkpointed states) at some node in the network, forwarding a message to another node, component failures, checkpointing etc. in the network. The sequential discrete-event simulation used for the ACRS fault-tolerant parallel processing system is event driven which are simulated chronologically and its simulation clock is advanced after simulation of an event to the time of the next event. Using a sequential simulation technique will not be time consuming, so it is most appropriate method of pretesting proposed systems before developing expensive prototypes.

The simulation model in this study aims to capture features of the performance overhead and of continuous performance with degradation in the face of failures. It also identifies relevant aspects that affect the efficiency of real systems. The performance overhead is caused by regular transmitting of checkpointing states across processors. The performance degradation originates from recovery processing after failures and the unbalanced execution of migrated processes.

There is always more information available about a system than can be incorporated in a simulation model. However, only the essence of the ACRS fault-tolerant system will be considered in this simulation study. The study simulates the performance for each fault tolerance related event rather than details of task operations. The events basically consist of process replicating from one processor to another, a network synchronization commitment, recording checkpointing on a source processor, installing checkpoint on a destination processor, and rolling back to the last checkpoint.

The performance overhead of the ACRS fault-tolerant system is essentially determined

by (a) the size of the network; (b) amount of process state information on each processor for every checkpoint and interprocessor communication message; (c) the interval between the checkpoints and the number of processors each process must to be duplicated to. Since the test was based on an application of parallel image processing, the above parameters are treated as deterministic variables.

The simulation model may run "correctly" by producing reams of results and passing all kinds of bench-marks, and yet be worthless, because the model may not correctly represent the real system. This subtle problem in this simulation was avoided by comparing some results of parallel image processing programs with those obtained from running on a real multi-transputer system. From this comparison, some parameter values were adjusted and programming errors were identified and removed to accurately model of the real system. This procedure is called validation of model. The validation procedure for this simulation of the ACRS algorithm is introduced after the simulation model is described.

8.2 Validating the simulation model

This simulation procedure was developed in two steps. Firstly, a simulation model of non fault-tolerant parallel image processing was constructed for model validation. Then the ACRS fault tolerance algorithm was added into the validated simulation model and formed a simulation model of fault-tolerant parallel processing system supported by the ACRS algorithm.

Validating is a very important stage in ensuring correct simulation results for performance prediction of a proposed system. The validation in this section is to verify some crucial simulation techniques of an existing system and some time related parameters so that these techniques and parameters can be applied to the proposed system and produce practical outcomes. The program used for the validation is a simplified parallel image processing application without considering some more trivial

details.

8.2.1 Simulation of parallel image processing

The Meat Research Institute of New Zealand (MERINZ) employed parallelism technology to an image processing application to detect the location of bones. In this application, the image data is divided evenly into m small blocks where m is the number of processors in the network. Each processor works on one of the divided segments. To calculate the image accurately on the boundary of each block, data has to be swapped between neighbouring processors. Computation followed by data swapping is iterated many times until the precision requirements are met. This processing involves interprocessor communication because of regular boundary data swapping and this kind of interaction can be iterated in a long run to meet the requirement of accuracy in the image calculations. This simulation model for this ACRS algorithm was based on this image parallel processing application, because it is of general characteristics of parallel processing. The simulation program does the checkpointing frequently during this processing and takes the recovery action for failures generated deliberately.

The Simscript simulation environment uses the notions of process and resource to represent primary dynamic object and passive elements of a model respectively. In other words, a process represents an object and the sequence of actions it experiences throughout its life in the model, and a resource is used to model an object which is required by the process object. With this simulation model, all hardware components are considered as resources and all software processes are considered as processes.

8.2.1.1 Hardware resource generation

The form of resources for this parallel image processing are processors and links. Resources are created before activating processes and other operations. The information about the processors, links, their connections and paths between any

processors is predetermined and provided in a configuration description file. The generation of the hardware resources is performed by three procedures: *start.main, initiator* and *neighbours*.

The start.main procedure is used to create a processor element for each node in the network. The initiator procedure is invoked to generate link components between processors and it also sets a path for each pair of processors. The neighbours procedure is executed to install the neighbour information to each relevant link variable which direct to that node.

The syntax of input file is described as follows:

(1) <input file> :: <Node>

<links>

<Paths>

(2) <Node> :: <number of node>$

<number of node>::{<digit>}*

(3)<Links> :: <total number of links>:<link>

<total number of links>::{<digit>}*$

<link>::{<digit>}*&{<digit>}$

comment: $ stands for line return, and this symbol does not appear in the file.

& stands for space character, a space character is shown in the file instead of symbol &.

(4)<paths>::{<length of the path>:{<path>}*}*<end of paths>

<length of the path>::{<digit>}*$

<path>::{{<digit>}*&{<digit>}*<&|$>}*

<end of paths>:: end of paths$

This simulation has used a multiprocessor network model with toroidal mesh topology which consists of 16 nodes as shown in Figure 8.1. The network connection is described in a text file and is read into the program at the beginning of the simulation.

In reality, each processor requires many small components such as link ports, reserved

memory locations as register store variables and pointers, and each link has the different processors to attach at its ends. These units of resources are defined as attribute variables whose identifiers are derived from their usages. The attributes of transputer and link resources in the simulation model are shown in Figure 8.2.

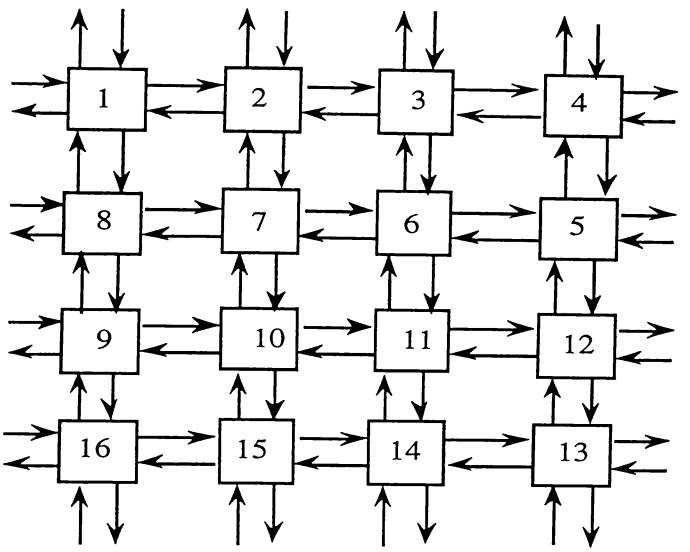


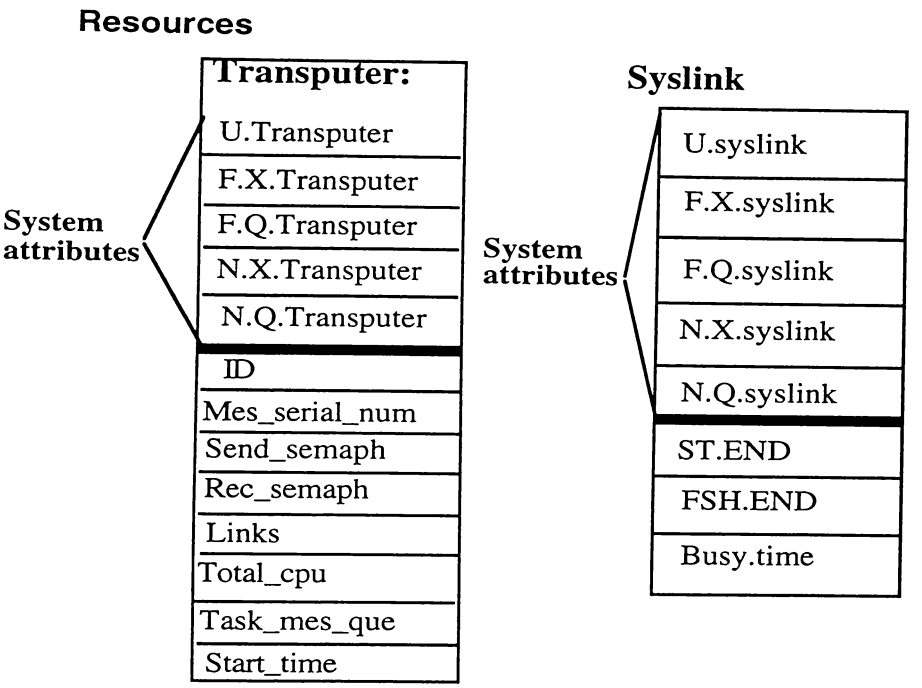
Figure 8.1 Processor arrangement and their connections for the simulation model

Resources are objects which are required by the process objects. If the resource is not available when required, the process object is placed in a queue or waiting line until the resource becomes available. There are two kinds of attribute for each resource object: system automatically defined attributes, and modeller defined attributes. The number of the units of a resource, the number of outstanding requests, and the number of currently satisfied requests are all contained in system defined attributes of a resource. The modeller defined attributes in this simulation are displayed in Figure 8.2 and are chosen to characterize the resources adequately.

8.2.1.2 Software process activation

There are three different processes: kernel (OS: operating system kernel), tasks (Task: application program) and messages (Message: interprocess communication message) in a model. There also are many instances (or copies) of each process in the simulation. Referring to the parallel image processing application, Task corresponding to each

processor resource is the application program which is responsible for a divided block of the image data, messages is the data of block boundary. An OS for each processor resource is used to activate application programs.



System attributes:

U.Transputer (link):	Number of units of the resource
F.X.Transputer (link):	Pointer to the first process which the resource allocation made to
F.Q.Transputer (link):	Pointer to the first process which is in the resource waiting queue
N.X.Transputer (link):	Number of the processes which the resource unit allocation made to
N.Q.Transputer (link):	Number of the processes which waiting for the resource

Modeller defined attributes:

ID:	resource identifier
Mes_serial_num:	to assign serial number to each message
send_semaph:	semeph for synchronize the message sending and receiving
rec_semeph:	semeph for synchronize the message sending and receiving
links:	consists of four array:link0, link1, link2 and link3 that used to store eighbour's information such as ID of neighbour's
total_cpu:	to record total consumed cpu time
task_mes_que:	a queue of task messages waiting for the resource
start_time:	to record initiating timer for substracting by ending timer

Figure 8.2 Processor (transputer) and link resources used in the simulation

Three processes on each processor are activated in a sequential manner, the OS process is actived first, followed by the Task process. The message processes are activated when Task processes start swapping boundary data. The corresponding algorithm for these processes is described as follows in Figure 8.3.

The Task process simulates data processing on one of the block images. It swaps boundary data with all its four neighbours before dealing with numerical computation on current local data. As mentioned earlier, this data swapping and data computation needs to iterate until the precision requirement has been satisfied. The simulation model assumes that 100 iterations can reach the precision requirement.

The Message process represents boundary data and it travels only between two adjacent processors. After a message process is activated, it calculates time required on both sender and receiver processor resources. This calculation is a function of communication rate and message size. The message processes are automatically destroyed once it has reached its destination processor.

```
PROCESS OS
DEFINE T AS INTEGER VARIABLES
   LET T = OSID(OS)
   ACTIVATE A TASK  " active a task process on processor T
END " OS: OPERATING SYSTEM


PROCESS TASK
DEFINE DI AND K AS INTEGER VARIABLES
LET DI = TID(TASK)
FOR K = 1 TO 100
DO
CALL DELAY                  " delay is the operation time for preparing boundary data
CALL HORIZ_SWAP             "swap boundary data with horizontal neighbours
CALL VERTIC_SWAP            "swap boundary data with vertical neighbours
CALL COMPUTATION   " computation on default block image data
LOOP
END " PROCESS TASK


PROCESS MESSAGE
DEFINE DI, DEST, MES_LEN AS INTEGER VARIABLES
LET DI = CUR_SITE(MESSAGE)                          " default processor id
LET MES_LEN = MES_SIZE(MESSAGE)                     " message size
LET DEST = MES_DEST(MESSAGE)                        " message destination
WORK ( MES_LEN / COM_RATE) .MICROSECONDS ON PROCESSOR DI
WORK ( MES_LEN / COM_RATE) .MICROSECONDS ON PROCESSOR DEST
END " PROCESS MESSAGE
```

Figure 8.3 Algorithm description of three processes in parallel image processing simulation model

Processes OS and Task enter the simulation model immediately at its "creation time."

However its lapses of time are not considered until it executes a sequence of time related events. The events which referring to lapses of time are conducted in five situations:

S1: operation overhead for preparing boundary data, this measurement is the function of the message size and is calculated with formula AVERAGE_DELAY = (M_SIZE * 0.000761 + 0.344) * 10000 in the routine "delay." Where (M_SIZE * 0.000761 + 0.344) was regression formula generated with tested data from ACRS validation program on a multitransputer system. It was transferred to ten millisecond unit by multiplying 10000.

S2: time consumed for waiting receiver on destination node ready. This parallel processing program was synchronized globally in a network, the sender cannot transmit until the receiver have started receiving. It is an indefinite parameter depending on the nature of the simulations. This measurement is implemented by accumulating X number of microseconds until the receiver is ready in routine "sleep."

S3: lapses of time on sender for transmitting the message and on receiver for receiving the message. It is a function of message size and communication rate (message length (MES_LEN) / communication rate (COM_RATE). This function is performed in process "MESSAGE."

S4: Time of waiting for receiving a message from a sender. This is another indefinite parameter as mentioned in S2. This measurement is also implemented in routine "READ" by accumulating Y number of microseconds until the message is arrived.

S5: Computation time which is consumed for numerical calculation on image data. This time segment (COMP_TIME) is required in routine computation.

The detailed description of timing activity in the corresponding procedure can be found in figure 8.4 which provides the algorithms for time oriented procedures in the simulation. The executable program for this image processing model has been included in the appendix.

```
ROUTINE HORIZ_SWAP GIVING NODE_ID, STATUS AND M_SIZE
DEFINE NODE_ID AND M_DEST AS INTEGER VARIABLES
DEFINE M_SIZE AS REAL VARIABLES
SELECT CASE NODE_ID
CASE 1, 8, 9, 16
        CALL SLEEP              " waiting for dest ready
        ACTIVATE A MESSAGE GIVING NODE_ID, M_SIZE, NODE_ID AND M_DEST NOW
        CALL READ               " reading message
CASE 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15
        CALL READ               " reading message
        CALL SLEEP              " waiting for dest ready
        ACTIVATE A MESSAGE GIVING NODE_ID, M_SIZE, AND M_DEST NOW
ENDSELECT
END " END OF HORIZ_SWAP

ROUTINE VODIC_SWAP GIVING NODE_ID, STATUS AND M_SIZE
DEFINE NODE_ID, AND M_DEST AS INTEGER VARIABLES
DEFINE M_SIZE AS REAL VARIABLES
SELECT CASE NODE_ID
CASE 1, 2, 3, 4
        CALL SLEEP              " waiting for dest ready
        ACTIVATE A MESSAGE GIVING NODE_ID, M_SIZE AND M_DEST NOW
        CALL READ               " reading message
CASE 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
        CALL READ               " reading message
        CALL SLEEP              " waiting for dest ready
        ACTIVATE A MESSAGE GIVING NODE_ID, M_SIZE AND M_DEST NOW
ENDSELECT
END " END OF HORIZ_SWAP

ROUTINE COMPUTATION GIVING NODE_ID
DEFINE NODE_ID AS INTEGER VARIABLES
  REQUEST 1 UNIT OF TRANSPUTER(NODE_ID)
     WORK COMP_TIME .MILLISECONDS              " time consumed for computations
  RELINQUISH 1 UNIT OF TRANSPUTER(NODE_ID)
END OF PROCESS COMPUTATION


ROUTINE READ GIVING NODE_ID
DEFINE NODE_ID  AS INTEGER VARIABLES
UNTIL MESSAGE_ARRIVED  EQ 1
DO
"REQUEST 1 UNIT OF TRANSPUTER(NODE_ID)
WORK WAITING_UNIT .MICROSECONDS
"RELINQUISH 1 UNIT OF TRANSPUTER(NODE_ID)
LOOP
RECEIVE MESSAGE
END " READ
```

Figure 8.4 Subroutines that contain time related functions

The simulation model operates correctly, and its performance entirely depends on those

time related parameters. Since this model is to be used for further fault-tolerant algorithm analysis, it has to be assured that the algorithm faithfully represents the model as abstracted from the real system. In order to simulate accurately, these time related parameters have to be either predetermined or indefinite.

With the simulation model, parameters: AVERAGE_DELAY in the situation S1, COM_RATE in S3, and COMP_TIME in S5 are predetermined and parameters: X (sleep routine) in S2 and Y (read routine in S4) are indefinite parameters. The determination of these time related parameters are part of the procedure of model validation. Those predetermined parameters are produced from a test by running a real system and the indefinite parameters are adjusted while comparing to the performance result to the real system. The details about the model validation and program used for validation on the real system are introduced in the next section.

8.2.2 A program on real system for validation

The actual system of validating the simulation model is implemented on a multi-transputer network which contains 20 processors, but only 16 processors out of 20 were used for this test. The program was compiled using version 2.05 of the Helios C compiler, and run under Helios version 1.2.2. Its physical configuration, as displayed in Figure 8.1, is a directly linked pipeline circle of 25MHz IMS T805 transputers, each having 40 Mbytes/sec sustained data rate to external memory. The link speeds of the processors were set at 20 Mbits/second. The communication level in the program was the routines from Posix library.

The logical topology for this program was toroidal mesh which was the same as that in Figure 8.1. It makes full use of four links of each processor. This logical connection was constructed by invoking a CDL script.

Because there is not a centralised clock over the transputer network, the task on each

processor would initialise the processing at a different time. In order to make the interprocessor action more cooperatively, the first message exchanging was to synchronize the multiprocessing. The time spent on this first communication was not included in the time measurement. The timing was adjusted after this synchronization statement as shown in Figure 8.6. Then data swapping between neighbouring processors is followed as it required.

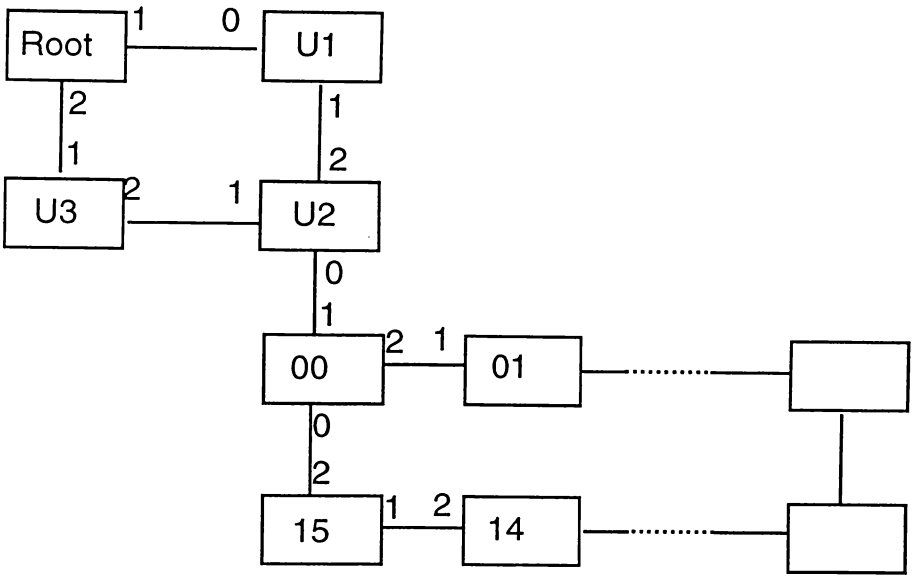


Figure 8.5 Physical connection of a transputer network

This simple image parallel processing model shown in Figure 8.6 was implemented on a multi-transputer system and has the same algorithm as that in the simulation model. its message swapping has been divided into two stages: horizontal swapping and vertical swapping. Since this program was specially designed for this validation, only right side and bottom side border condition was considered to be sent to adjacents. Firstly the processors on the left edge initialize a message about the border condition to their right neighbours. Once having completed sending, they wait to receive the message from their left neighbours. The processors located on the left side of the network array, cannot send their message out until they have received it from their left neighbours. Secondly, when horizontal swapping is completed, those processors

located on the top most side of the network array initialize a message sending to their down neighbours, then wait for an incoming message. Once a processor has received boundary data from its up neighbour, it prepares its boundary data for its downward neighbour processor.

```
int main(int argc, char** argv)
{
      begin = clock();
      for (J = 1; J<=loop; ++J){
         swap_edges(rowidx, colidx, &pre_synch_time, &one_swap_time);
         compbegin = clock();
         for (i = 1; i<900000; ++i) j*=i;
         compfinish = clock();  COMP_TIME = compfinish - compbegin;
       }
         finish = clock();  COMP_TIME = compfinish - compbegin;

};
void swap_edges(int rowidx,int colidx, int *pre_synch_time, int *one_swap_time)
{
  swap_begin = clock();
     switch(processor_id) {              /* partial horizontal swapping procedure
       case 0: case 4: case 8: case 12:
           start = clock();
           send_to_right();              /* this message for synchronising the communications
           receive_from_left();          /* this message for synchronising the communications
           fsh1=clock(); pre_synch_time=((float)fsh2-(float)start)
                                         /*time required for synchronization
         send_to_right();                /* send data in buffer to the processor on the right
         receive_from_left();            /* receive data from the processor on the left
       break;
   default:
           start = clock();
          receive_from_left();           /* this message for synchronising the communications
          send_to_right();               /* this message for synchronising the communications
          fsh1=clock(); pre_synch_time=((float)fsh2-(float)start)
                                         /*time required for synchronization
         receive_from_left();            /* receive data from the processor on the left
         send_to_right();                /* send data in buffer to the processor on the right
       break;
     } end of swaich

    switch(s_id) {
       case 0: case 1: case 2: case 3:
          send_to_downward();            /* send data in buffer to the processor downward
          receive_from_top();            /* receive data from the processor from up
        break;
     default:
        receive_from_top();              /* receive data from the processor from up
          send_to_downward();            /* send data in buffer to the processor downward
       break;
       }
 swap_finish = clock();
 one_swap_time = ((int) swap_finish - (int) swap_start);
                                                    }
```

Figure 8.6 Pseudo-code of a simple parallel processing model implemented on multi-transputer system

The image calculations on each processor will commence when one round of boundary data swapping has been completed. This procedure is supposed to be repeated until it meets the precision requirement. The system produces timing results for both one round of the entire procedure, and some other partial operations. The time consumed for each round of the entire procedure was used for the performance comparison between this application and simulation. These performance results produced on a multi-transputer system varied for each testing, so comparing performance was based on the average result from a seven round test.

The time consumed for the first data swapping (pre_synch_time) and for image computations (COMP_TIME) are used to adjust the parameters: AVERAGE_DELAY and COMP_TIME in the case of S1 and S2 of the simulation program. In fact, using AVERAGE_DELAY = (M_SIZE * 0.000761 + 0.344) * 1000 can produce approximately the same result as pre_synch_time tested on the transputer system. The COMP_TIME used for the simulation program is feeded with the test result of computation time from transputer system.

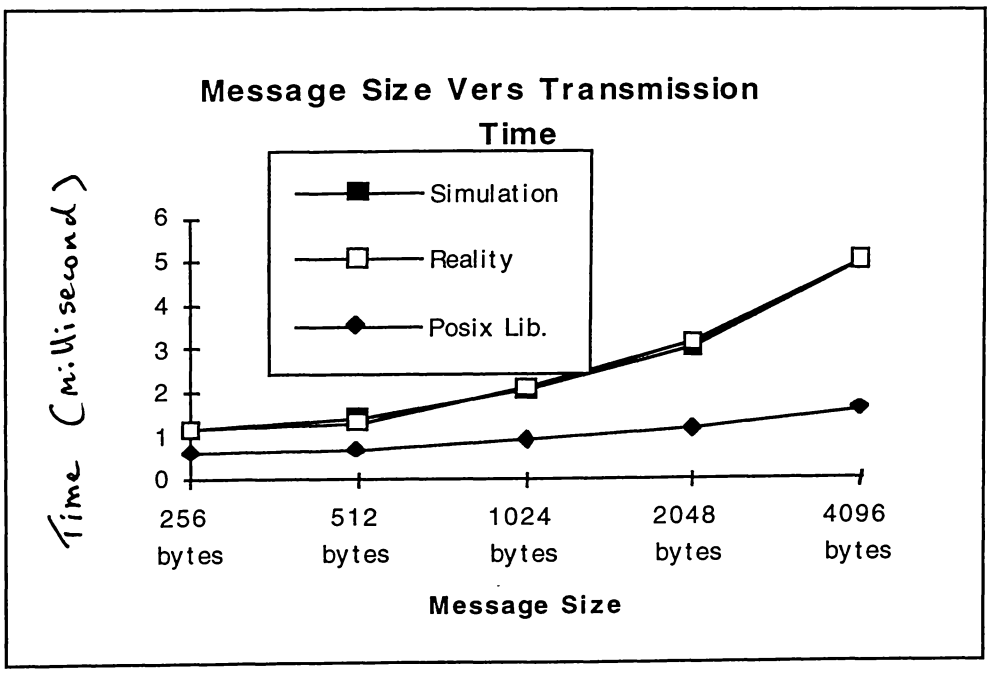


Figure 8.7 performance comparison between simulation results and result tested on transputer system

The infinite time related parameters: X in the case of S2, lapses of time in S3 and Y time unit in S4 in the simulation program was determined so that X in S2 was fixed as one microsecond for each pause, Y in S4 was chosen as 50 microseconds and COM_RATE for the data transmitting time, MES_LEN/COM_RATE, was the transmitting rate tested for Posix library. The performance of the simulated parallel image processing model by using the parameters described above perfectly match the performance tested on a transputer system based on the same algorithm. Their performance comparison is displayed in Figure 8.7 and tested data and communication rate used was shown in table 8.1.

Number of bytes (NB) transmitted	Simulation milliseconds	Reality milliseconds	Posix Lib. milliseconds	Posix bytes/microsec.	posix Kbyte/second
256 bytes	1.14	1.13	0.602	0.17	168
512 bytes	1.41	1.27	0.680	0.30	294
1024 bytes	2.01	2.11	0.890	0.46	454
2048 bytes	3.01	3.09	1.111	0.73	720
4096 bytes	5.03	4.98	1.570	1.04	1019

Table 8.1 Data for figure 8.7

8.3 Simulation of ACRS fault-tolerant system

The simulation model of the ACRS fault-tolerant system was based on the simulation model used for validation. There are three more processes: server, client, and protocol, designed to tolerate processor failures. The process server is supposed to be invoked on root processor resource and is used to synchronise checkpointing regularly. A process client was set on each processor resource to co-operate with the server for checkpointing and failure recovery. Process protocols are acted as messages between server and clients for synchronising corresponding actions. Both processes server and client ran concurrently with other system and task processes. A protocol process was activated whenever either a client or the server sent a protocol message about checkpointing or failure recovery.

At regular intervals, the process server initiate a protocol message to each client to indicate checkpointing cooperation. Once a protocol process was activated, it moved from source processor to destination processor through a path included in a path list. Having landed on a processor, it calculated the CPU time required for the processor to receive and to forward the protocol message further to the next processor along the path. The data transmission rate used here was the same as the communication rate used in the validation model. The formula for this time function is the message length (MES_LEN/PROTO_RATE) divided by transmitting rate. The time consumed for each protocol message on each processor was accumulated and stored into an attribute FT_PROTO_TIME.

After the server signalled to all clients to start checkpointing, it waited for the acknowledgment of this checkpointing initialisation signal from all clients. If *i*th client did not reply within a certain amount of time, then the server broadcast the failure information about *i*th processor to the network and invoked the failure handler to synchronise the network to recover from the failure. If the server received the acknowledgments from all clients in the network, then it sent a second message to them for recording the checkpointing states. Having received the checkpointing completion message from all clients, it signalled to resume Task, application program, over the network.

There was a client process on each processor. The function of the client process is to cooperate in checkpointing and fault recovery. All client processes were created when the system started and dismantled when the system terminated. During normal task computation, the client process was suspended until a message about checkpointing arrived. When a client process received a new checkpointing request from a server, it suspended executions of all application programs on the default processor and indicated its readiness to the server.

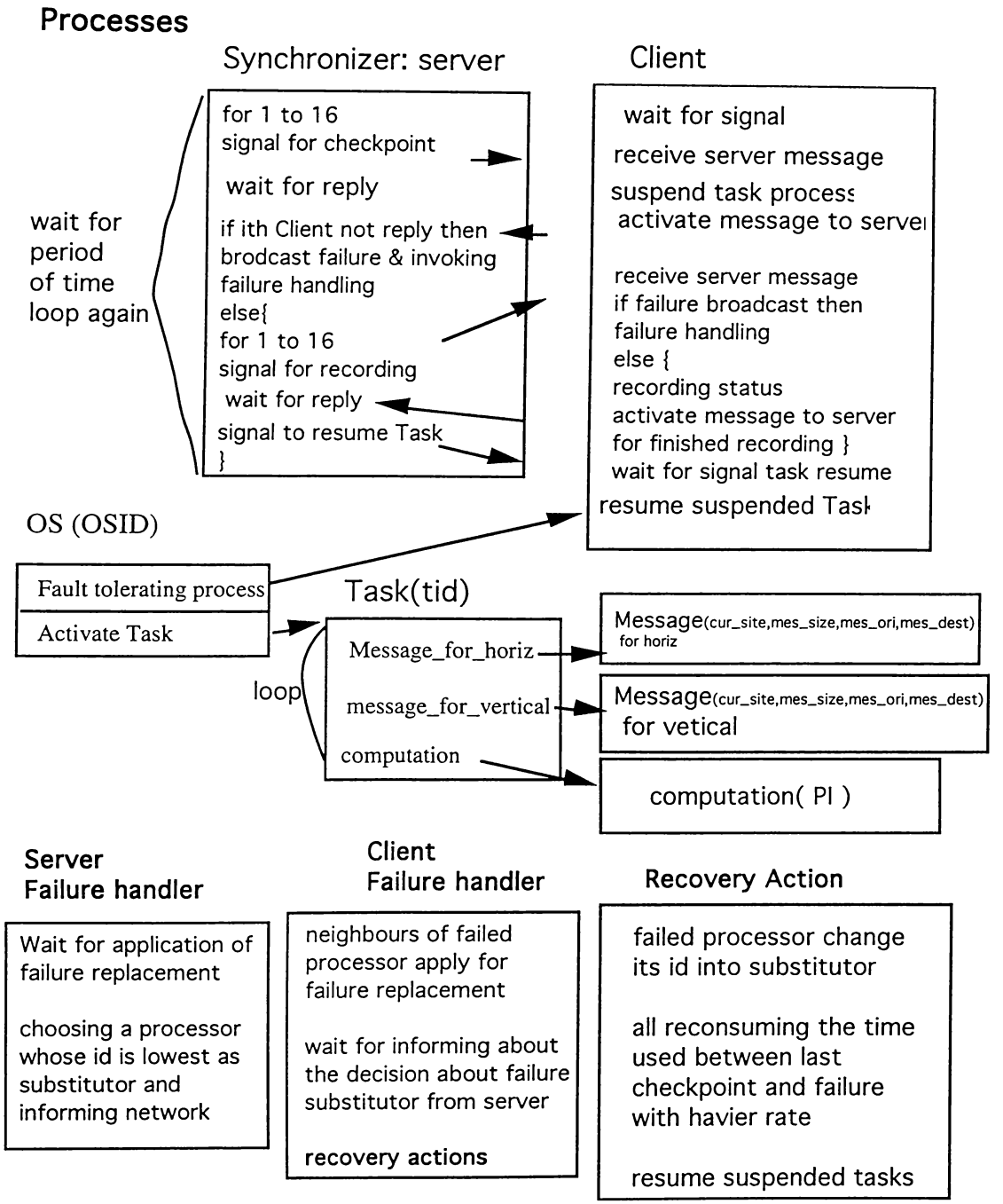
After the second message, which was about installing the error-free states, arrived from server, the client process calculated the time consumed for the checkpointing data installation based on the size of the error-free state. This data installation was the transmission of the program execution state to a neighbour processor in order to be able to restart from this state in the case of failure occurrence. Once data installation was completed, client process informs this checkpointing completion to the server, then waited for a signal to resume the normal task computation from where it was suspended.

If a failure occurred in the middle of processing of the application programs, then both server and client processes were involved in the recovering from the failure. Firstly, the server broadcast the failure to the network, then it assigned a processor to take over the tasks on the failed processor by choosing an adjacent processor. When the client process on the failed processor receive the information about its failure and substitutor, it recovered by changing its processor id into the one same as the substitutor's. In such a way, the processor which was assigned as substitutor would be double loaded with the tasks for both the failed processor and its own original tasks.

In order to make this performance test simpler, the checkpointing requests were issued for every k loops of data swapping and image computation. k was varied from 2 to 10 during the test. Assuming a failure occurred in the ith loop (where i>k), the system has to roll the processing back to (*i-k)*th loop and resumes the execution by redoing the computation for the last i-k loops and proceed for further. A brief outline of the simulation model of the ACRS algorithm is given in Figure 8.8.

The fault-tolerant system supported by the simplified ACRS algorithm described above imposed some degree of performance overhead during error-free situation and performance degradation if processor failures emerged. This simulation produced a

estimation analysis about both performance overhead and degradation in next section.



checkpointing protocol:

0: no checkpointing
1: suspend for checkpointing
2: task answer ready
3: check start
4: check finish
5: resume computation performing
6: recover start

Figure 8.8 Interactions among processes in the simulation for ACRS algorithm

8.4 Performance analysis

The fault-tolerant algorithm is a part of the operating system. The communication messages between the processor entities go through intermediate processor entities and link entities on the path. The time related parameter was adjusted to a multi-transputer system. The estimation of the performance penalty in the simulation of the fault-tolerant algorithm was for a network of 16 processors. Experiments were performed to measure the effects of performance penalty imposed by the fault-tolerant algorithm in an error-free situation and performance degradation after recovery from a processor failure.

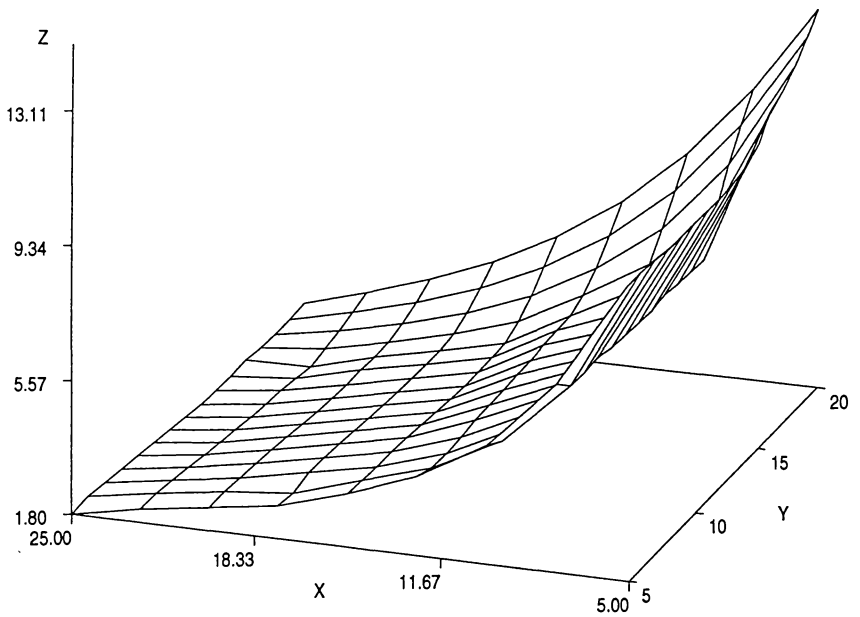


Figure 8.9. The performance overhead imposed by the fault-tolerant algorithm during error-free operations as the function of checkpointing interval (X) and log value of the checkpointing data size (Y).

The fault-tolerant algorithm performance penalty is the difference between the simulation results of parallel image processing with employment of fault-tolerant algorithm and the result without fault tolerance. Figure 8.9 shows the performance penalty while checkpointing frequency and size of the data change. The time unit for the X-axis was in seconds and that for the Z-axis was 10 milliseconds. The value used for the Y-axis is the logged value of the checkpointing sizes in bytes. This figure shows us that the penalty is more severe when the checkpointing interval is small and the

checkpointing data is large. The performance can be improved by optimising the frequency of the checkpointing activities.

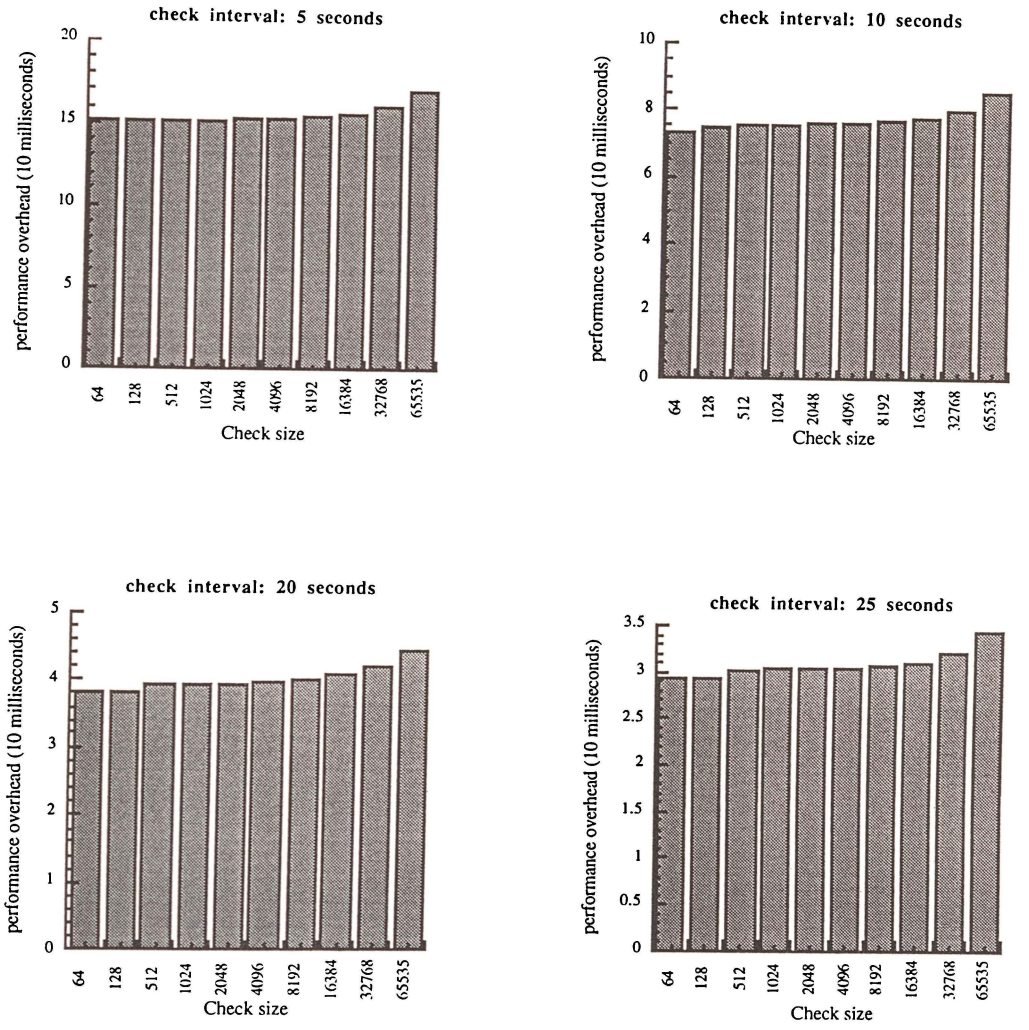


Figure 8.10 Performance overhead while check size changes

Figure 8.10 and Figure 8.11 are further illustrations of performance penalty situation in the absence of failure by considering the checkpointing data size and checkpointing frequency individually. Figure 8.10 shows that the amount of data transmitted to adjacent processors for replicating error-free states does not affect system performance significantly. But the checkpointing intervals are the important factor, high frequency of checkpointing could be costly to the system performance. Figure 8.11 shows that when the checkpointing interval is bigger, the performance overhead is smaller. Minimising

for checkpointing frequency which can meet the reliability standard is necessary.

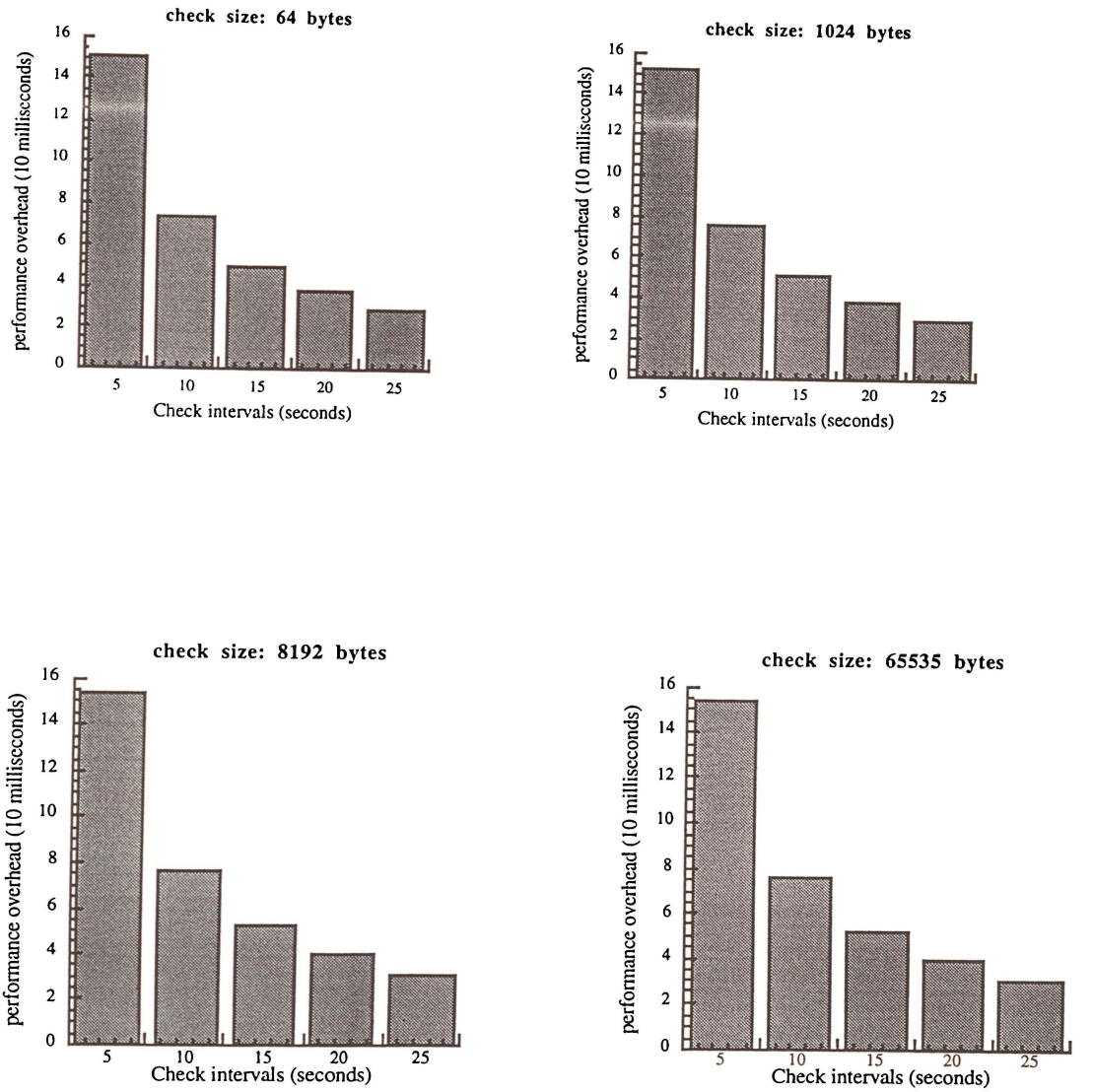


Figure 8.11 Performance overhead imposed by ACRS algorithm while check interval changes

Performance degradation after recovery from a processor failure, is estimated for the worst case in which each processor provides no time redundancy. The figure 8.12 is the percentage of time increased when the a processor failed after 13 seconds. Its performance is decreased less than double that of the "before failure" situation. This is because the processor which is doubly loaded with both the failed task and its primary task slows down interactions between the processes. It is important to note that the degradation would be the same if recovery extends to several processor failures until

some of the processors are more than doubly loaded.

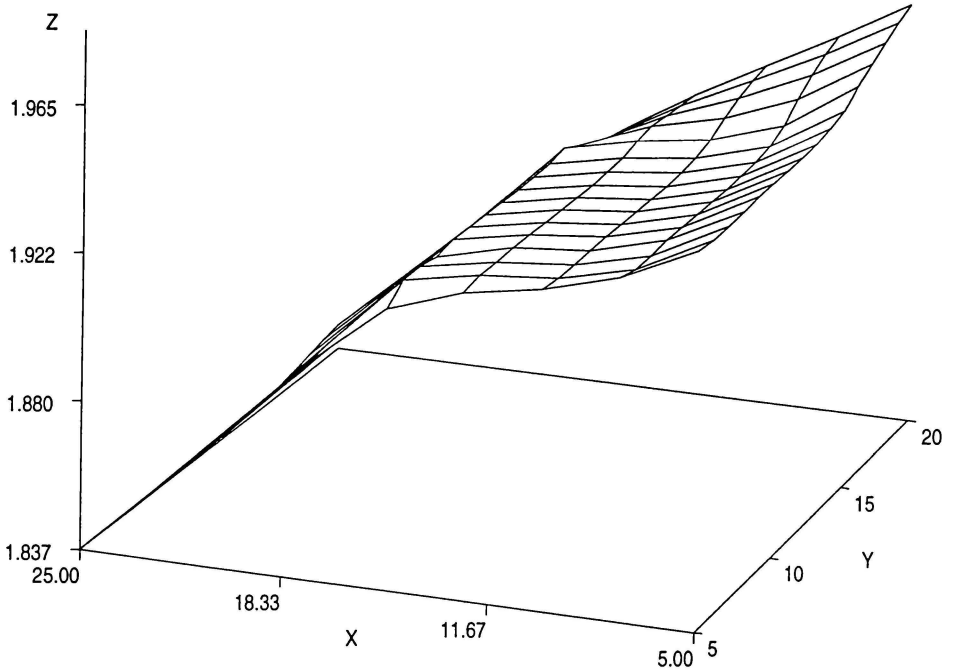


Figure 8.12 Performance degradation in the worst case as the function of checkpointing interval (X) and log value of the checkpointing data size (Y). The value for Z axis presents the percentage of normal performance.

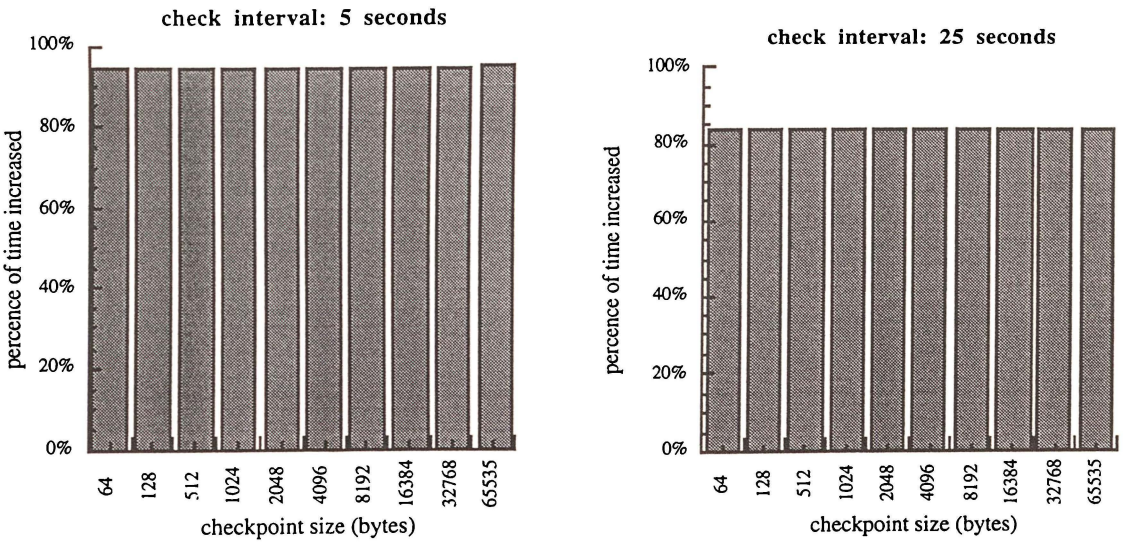


Figure 8.13 Performance degradation while check intervals changes at the presence of one failed processor after 13 seconds

Figure 8.13 and Figure 8.14 are a different view of figure 8.12 showing them as checkpointing size and check intervals change separately. The Y axis in both figures represents the percentage of performance time increased from the recovered system. Again as indicated before, frequent checkpointing would impose heavier performance degradation after recovery from a failure and checkpointing large of data for error-free state doesn't change performance very much.

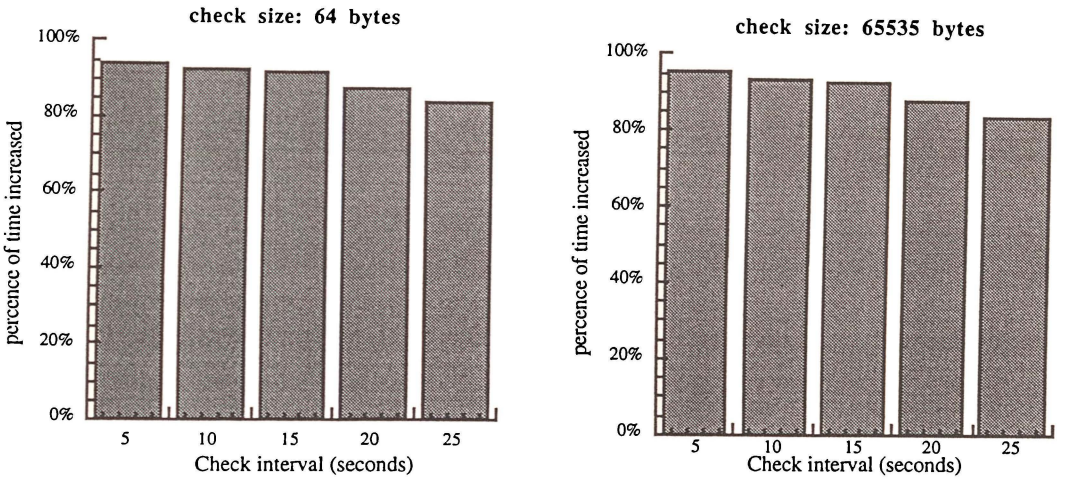


Figure 8.14 Performance degradation while check size changes at the presence of one processor failed after 13 seconds

Performance degradation can be reduced by introducing a few redundant spare processors. The recovery action transmits a replica from a neighbouring site to the spare redundant processor. In this method, it may take longer to recover because of process migration. However it will not impose any penalty on performance after recovery from a failure.

8.5 Summary

The fault-tolerant parallel processing algorithm has been simulated with the discrete event simulation technique. The simulation proved that the fault-tolerant algorithm is operationally correct, and it imposes a small performance overhead during error-free

situation.

Its performance degradation in the presence of processor failures is dependent on whether or not the processors are heavily loaded. In the worst case in which every processor is fully loaded, performance after recovery will degraded to that produced with n/2 processor power. However, based on ACRS algorithm we can infer that this performance degradation will remain the same in the presence of up to n/2 failures.

Chapter 9

Conclusions and Future Work

This thesis has presented a hybrid diagnosis algorithm and an adjacent checkpointing and dynamic recovery algorithm for distributed and parallel computing. These two algorithms play complementary roles in achieving fault tolerance in multiprocessor systems. The diagnosis algorithm is used to identify the sources of failure and the recovery algorithm undertakes actions to appropriately compensate for the effects of identified failures.

The thesis began by introducing some important concepts and existing software approaches for fault tolerance in parallel and distributed computing. It has a focus on fault tolerance mechanisms through diagnosis, reconfiguration and recovery. Because fault tolerance through diagnosis, reconfiguration and recovery has the advantages of cost effectiveness and no requirement of hardware operations over that through replication and masking. Redundancy is an essential technique for most of fault tolerance mechanisms and replication is a software implementation to cooperate with redundancy. The replication can be active and passive as we have explained in a great detail.

A range of existing system architectures was then presented to show how the fault tolerance was achieved in some of the commercial distributed systems by using the combinations of passive and active replication techniques and how the failures were detected by matching operations on replicated processors or masking the faults by using

majority voting. From this review, we know that the most existing fault-tolerant distributed systems were achieved with the expensive cost of extra hardware redundancy and were developed on common media bus communication networks. Furthermore, existing fault tolerance algorithms were mostly designed to favour data transactions and were implemented in hardware, and it is difficult to extend these to a wider arrange of applications.

The previous research efforts and results on distributed diagnosis and checkpointing/restarting in multiprocessor systems were introduced along with the discussion about the thesis contributions. We have shown that the proposed hybrid diagnosis algorithm and (ACRS) algorithm can provide efficient fault-tolerance for parallel and distributed computing on loosely coupled multi-processor architectures, thereby enhancing their general useability. Specifically, we draw the following conclusions about the distributed diagnosis and the checkpointing algorithms and their speed optimisations.

9.1 Hybrid diagnosis algorithm

The hybrid diagnosis algorithm has a significant improvement in efficiency over the existing diagnosis algorithms that were classified in three groups: centralised, truly distributed and processor membership, as we discussed at the beginning of chapter 5. The centralised diagnosis algorithms are restricted by the reliability of the central supervisor processor. The truly distributed diagnosis algorithms impose heavy communication overheads for assessing the status of the system. Processor membership diagnosis algorithms are mostly used to monitor processors departing and joining in the system and it can be used to diagnose the network only in the situation when a processor with fail-silent semantics fails.

The hybrid diagnosis algorithm is formed by integrating the three traditionally separated approaches into one scheme. It improved traditional centralised diagnosis methods in

two ways: (1) when the centralised supervisor is faulty, the system can survive by employing the processor membership mechanism. (2) It reduces diagnosis communication overhead significantly to O(tk+2) rather than O(n) messages as in traditional centralised diagnosis. In the hybrid diagnosis algorithm, the messages required for identifying t faulty processors are a constant $tk+2$ for the worst case. Its efficiency is especially desirable in a large network.

The hybrid diagnosis algorithm uses processor membership methods to ensure correct diagnosing even in the failure situation of the centralised supervisor P_m. The algorithm requires only one diagnosis request message for global state diagnosis in a fault-free network and a maximum of $tk+2$ messages in the worst case. It needs no queries to fault-free PEs and is able to learn the correct diagnosis based on fault report information from t nodes. With a truly distributed diagnosis algorithm, the minimum number of messages required for identifying the states of all processors in the network is O(kn). The diagnosis of t faulty PEs requires a minimum $2t+1$ queries of fault-free PEs to learn the correct diagnosis in the worst case. The comparison shows that the hybrid diagnosis algorithm reduces the operating overhead enormously.

Further to the improvements summarised above, the hybrid diagnosis algorithm parallelises the diagnosis executions over the network. It takes advantage of array architectures and point-to-point message passing protocols that provide wide communication bandwidth to enhance the system performance.

The hybrid diagnosis algorithm was a further development based on the diagnosis software used in the fault-tolerant NWP parallel system. The efficiency and effectiveness of the hybrid diagnosis algorithm has been induced in the implementation of a parallel weather prediction application. The implementation of fault-tolerance strategy in the NWP system was described in chapter 4. It has performed its daily duty correctly for more than three years at the NZ Meteorological Service. This system is

able to diagnose faults and failures that have emerged and automatically recover the system from failures, then resume operations without the need for the user's attention.

9.2 ACRS algorithm

For checkpointing in loosely coupled multiprocessor architectures, we studied three existing algorithms: Sync-and-Stop, Chandy-Lamport, and Network Sweeping. The ACRS algorithm presented a unifying technique in replicating checkpointing data to adjacent processors. In such distribution of error-free state, the system reliability is not limited by any individual devices.

The ACRS is based on consistent checkpointing to provide fault tolerance for multiprocessor failures. In general, it is able to tolerate up to n/2 processor failures except the worst case in which k concurrent failed processors are adjacent to a same node, where network is k regular connected. The advantages of the ACRS algorithm over the network sweeping algorithm proposed by Plank in his Ph.D thesis is that ACRS parallelised the data saving and transmitting for checkpointing and reduced communication congestion. This parallelism minimised the time required for defining a consistent cut and for taking global states, it doesn't need to distinguish cross-cut message. The ACRS algorithm simplified the checkpointing procedure significantly.

The reliability improvement of a multiprocessor system with toroidal mesh topology under the support of the ACRS algorithm was predicted using Markov chain mathematical model as described in chapter 7. The modelling results of partial system failure show that reliability is improved. This analysis method can be used to determine the capability of tolerating failure processors to meet the reliability requirement.

The ACRS algorithm imposes a small performance overhead during the failure free situation. Performance degradation after recovery from 1 to n/2 processor failures is up to 50% of original performance at the worst case where every processor is heavily

loaded. However this degradation can be avoided by using a few shared spare redundant processors to replace failures so that the system will have no performance degradation after recovery from failures. The performance analysis was conducted with discrete event simulation technique and it demonstrated that the ACRS algorithm is desirable both for the situation of failure free and for the situation of the presence of failures.

9.3 Limitations and Future Work

The ACRS algorithm replicates error-free process states on each node to on-chip memory of adjacent processors rather than a central memory space for keeping checkpointing data for all processors. The algorithm assumed that there is always sufficient memory space on each processor module as the storage for checkpointing data. This assumption might be difficult when error-free states are needed to be transmitted to many adjacent processors to meet the reliability requirement and when the size and multiple versions of error-free states increasing.

The ACRS algorithm can be further optimised if checkpointing uses compression to decrease the number of bytes that get transmitted to adjacent processors, thereby reducing the checkpointing time as well as storage spaces. A compression algorithm defined by Wheeller and called "Algorithm 1" [BW92] and some traditional compression algorithms, such as the "LZC" algorithm used in the Unix compress utility [BWC89], can be suitable algorithms of choice for on-line data compression in this fault-tolerant system. A concern of compression optimisation is the speed of the compression algorithm versus the amount of the checkpoint that gets compressed. If the compression algorithm is several times slower than a memory-to-memory copy, and it does not shrink the size of the checkpoint data considerably, then it is likely to increase both the checkpoint time and overhead rather than decrease them. Such tradeoff can be investigated in further advancement of the algorithm.

As all consistent checkpointing and recovery algorithms, the ACRS algorithm also concentrated on the efficiency of checkpointing and consistent recovery and it uses a centralised processor to synchronize the fault tolerance and recovery activates. The algorithm assumed that the central synchronization processor is very reliable and it does not cover the situation of the central synchronization processor failure. This defect can be improved by making the synchronising process distributed among more than one site. This improvement has been left to future research work.

Appendix:

A1: data for ACRS performance overhead

The data produced from the simulation program of ACRS fault-tolerant system when failure is absence.

check intervals	check size	total CPU	performance overhead	check intervals	check size	total CPU	performance overhead
5	64	264.9719	15.08409	15	4096	258.4484	5.183891
5	128	264.9721	15.08420	15	8192	258.4769	5.226735
5	512	264.9875	15.10738	15	16384	258.5334	5.311415
5	1024	265.0188	15.15428	15	32768	258.6469	5.481700
5	2048	265.039	15.18525	15	65535	258.8815	5.833659
5	4096	265.0804	15.24662	20	64	257.5400	3.807373
5	8192	265.1643	15.37256	20	128	257.5515	3.824619
5	16384	265.3304	15.62157	20		257.6203	3.927964
5	32768	265.6643	16.12246	20	1024	257.6285	3.940182
5	65535	266.354	17.15758	20	2048	257.6339	3.948214
10	64	259.8772	7.341562	20	4096	257.6445	3.964161
10	128	259.9744	7.452155	20	8192	257.6664	3.996979
10	512	260.0097	7.553740	20	16384	257.7095	4.061684
10	1024	260.0253	7.577167	20	32768	257.7963	4.191897
10	2048	260.0357	7.592670	20	65535	257.9757	4.460974
10	4096	260.0561	7.623365	25	64	256.9612	2.928780
10	8192	260.0981	7.686300	25	128	256.9700	2.942066
10	16384	260.1812	7.810864	25	512	257.0230	3.021519
10	32768	260.3481	8.061295	25	1024	257.0293	3.030918
10	65535	260.6932	8.578882	25	2048	257.0334	3.037120
15	64	258.3117	4.978878	25	4096	257.0416	3.049374
15	128	258.3267	5.001388	25	8192	257.0584	3.074581
15	512	258.4168	5.136584	25	16384	257.0916	3.124386
15	1024	258.4274	5.152481	25	32768	257.1584	3.224566
15	2048	258.4345	5.163043	25	65535	257.2964	3.431632

A2: data for ACRS performance degradation

The data produced from the simulation program of ACRS fault-tolerant system at the presence of one processor failure after 13 seconds. All timings are in ten millisecond units, the period of the transputer low priority clock. The performance of non-ACRS algorithm system is a constant: 255.031909 clock periods.

Check intervals	Check size	percentage of time increased	performance over head	Total CPU	Non-ACRS performance
5	64	94.316949	240.538315	495.570224	255.031909
5	128	94.342221	240.602768	495.634677	255.031909
5	512	94.493964	240.989761	496.021671	255.031909
5	1024	94.511854	241.035385	496.067294	255.031909
5	2048	94.523641	241.065445	496.097354	255.031909
5	4096	94.547058	241.125168	496.157077	255.031909
5	8192	94.595138	241.247787	496.279697	255.031909
5	16384	94.710995	241.543259	496.575168	255.031909
5	32768	94.893671	242.00914	497.041049	255.031909
5	65535	95.292468	243.026199	498.058109	255.031909
10	64	92.687155	236.381821	491.41373	255.031909
10	128	92.699971	236.414505	491.446414	255.031909
10	512	92.776877	236.610641	491.642551	255.031909
10	1024	92.785936	236.633743	491.665652	255.031909
10	2048	92.791931	236.649033	491.680942	255.031909
10	4096	92.803801	236.679306	491.711215	255.031909
10	8192	92.828165	236.74144	491.773349	255.031909
10	16384	92.886739	236.890823	491.922733	255.031909
10	32768	92.979393	237.127121	492.15903	255.031909
10	65535	93.18149	237.642532	492.674442	255.031909
15	64	92.099532	234.883194	489.915103	255.031909
15	128	92.107836	234.904373	489.936282	255.031909
15	512	92.157711	235.03157	490.06348	255.031909
15	1024	92.163587	235.046555	490.078464	255.031909
15	2048	92.167476	235.056473	490.088382	255.031909
15	4096	92.175182	235.076126	490.108035	255.031909
15		92.190989	235.11644	490.148349	255.031909
15	16384	92.22885	235.212997	490.244907	255.031909
15	32768	92.288992	235.366379	490.398288	255.031909
15	65535	92.42007	235.700668	490.732578	255.031909
20	64	87.833241	224.002792	479.034702	255.031909
20	128	87.839488	224.018723	479.050632	255.031909
20	512	87.876894	224.11412	479.14603	255.031909
20	1024	87.881309	224.125381	479.157291	255.031909
20	2048	87.884229	224.132828	479.164737	255.031909
20	4096	87.889997	224.147537	479.179446	255.031909
20	8192	87.901844	224.17775	479.209659	255.031909
20	16384	87.930036	224.249649	479.281558	255.031909

20	32768	87.97522	224.364883	479.396793	255.031909
20	65535	88.073487	224.615494	479.647403	255.031909
25	64	83.700123	213.462021	468.49393	255.031909
25	128	83.705316	213.475266	468.507175	255.031909
25	512	83.736488	213.554764	468.586673	255.031909
25	1024	83.740162	213.564133	468.596042	255.031909
25	2048	83.742591	213.570328	468.602237	255.031909
25	4096	83.747401	213.582596	468.614505	255.031909
25	8192	83.757297	213.607834	468.639743	255.031909
25	16384	83.780607	213.667282	468.699191	255.031909
25	32768	83.818325	213.763474	468.795383	255.031909
25	65535	83.900197	213.972275	469.004184	255.031909

References:

[AF89] Artsy, Y. and Finkel, R., 1989, *Designing a Process Migration Facility.* IEEE Computer, September, 47-56

[AKL87] Avizienis, A. Kopetz, H. and Laprie (Eds), 1987, *The Evolution of Fault-Tolerant Computing*, Springer-Verlag/Wien.

[AL81] T. Anderson and P. A. Lee. 1981, *Fault Tolerance Principles and Practice*. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey.

[And85] Anderson, T. (Ed.), 1985, *Resilient Computing Systems*. Collins.

[And89] T. Anderson (Ed.), 1989, *Dependability of Resilient Computers*, Blackwell Scientific Publications, Oxford, 1989.

[Avi84] Avizienis, A., 1984, *Fault Tolerance by Design Diversity: Concepts and Experiments*, Computer, **17**.

[Bar78] Bartlett, J.F., 1978, *A NonStop Operating System*, Proceedings of Eleventh Hawii Int. Conf. of System Sciences. Honolulu, HI 1978, 103-119.

[BBG83] Borg, A. 1983. *A message system supporting fault-tolerance*, In the 18th International IEEE Symposium on Fault-Tolerant Computing. IEEE.

[Ber88] Bernstein, P.A., 1988, *Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing*. IEEE Computer. February. 37-45.

[Ber90] Bernstein, P.A., 1990, *The Sequoia System*. LNCS on Fault-Tolerant Distributed Computing 448. Spector, S. (Eds). pp. 217-223.

[BGN90] Bianchini, R. Goodwin, K., and Nydick, D. 1990. *Practical application*

and implementation of distributed system-level diagnosis theory. In the 20th International IEEE Symposium on Fault-Tolerant Computing. IEEE, New York, 332-339.

[BH91] Bagchi, A., & Hakimi, S. 1991. *An optimal algorithm for distributed system level diagnosis.* In the 21st International IEEE Symposium on Fault-Tolerant Computing. IEEE, New York, 214-221.

[Bir79] Birtwistle, G.M., 1979, *Discrete Event Modelling on Simula,* Macmillan, Basingstoke.

[Bir85] Birman, K.P. 1985. *Replication and fault tolerance in the ISIS system.* Proceedings of 10th ACM symposium on Operating System Principles, Washington, December. 79-86.

[BW92] Burrows, M. & Wheeler, D. J., 1992, *A Block-sorting lossless data compression algorithm,* Technical Report, Digital Equipment Corporation, Polo, Alto, California.

[BKL87] Brilliant, S.S., Knight, J.C. and Leveson, N.G., 1987, *The consistent comparison problem in N-Version software.* IEEE Trans. Software Eng. **15**, 1481-1485.

[BKL90] Brilliant, S.S., Knight, J. C. and Leveson, N.G. 1990, *Analysis of Faults in an N-version software Experiment.* IEEE Trans. Software Eng., **16**.

[BWC89] T.Bell, I.H. Witten, and J. G. Cleary. 1989, *Modelling for text compression.* ACM Computing Surveys, **21(4),** 557-589.

[CGS81] Ciompi, P., Grandoni, F., and Simoncini, L. 1981. *Distributed diagnosis in multiprocessor systems: The MuTeam approach.* In the 11th International IEEE Symposium on Fault-Tolerant Computing. IEEE, New York, 25-29.

[CJ91] Flaviu Cristian and Farnam Jahanain. 1991, *A timestamp-based check-pointing protocol for long-lived distributed computations.* In Proceedings of the 10th Symposium on Reliable Distributed Systems, 12-20, October 1991.

[CL85] K.Mani Chandy and Leslie Lamport. 1985, *Distributed snapshots: Determining global states of distributed systems.* ACM Transactions on Computer Systems, **3(1),** 3-75.

[Cri91] Cristian,F., 1991, *Understanding Fault Tolerant Distributed Systems.* Communications of the ACM, **34,** 57-78.

[Dah88] A. Dahbura, 1988. *System-level diagnosis: A perspective for the third decade.* AT & T Bell Laboratories Report. Concurrent Computations: Algorithms, Architecture, and Technology. Plenum Press, New York.

[DSK87] Dahbura, A., Sabnani, K., and King, L. 1987. *The comparison approach to multiprocessor fault diagnosis.* IEEE Trans. Comput., **C-36**, 373-378.

[EST89] P.D.Ezhilchelvan, S.K. Shrivastava and A. Tully, 1989, *Constructing Replicated Systems Using Processors With Point-to-point Communication Links*, Proceedings of the 16th Annual Symposium on Computer Architecture, Jerusalem, June 1989, 177-184.

[Fis90] Fischer, M.J., 1990, *A Theoretician's View of Fault Tolerant Distributed Computing.* LNCS on Fault-Tolerant Distributed Computing 448. Spector, S. (Ed), 1-9.

[FS80] Friedman, A., and Simoncini, L. 1980. System-level fault diagnosis. Computer (Mar.), 47-53.

[Fuj90] R. Fujimoto, *Discrete Event Simulation*, Communications of the ACM, **33**, 31-53.

[Gor78] Gordon, G., 1978, *The Development of the General Purpose Simulation System (GPSS)*, ACM Sigplan Notices **13(8).** 183-198.

[GT90] Graham, I. and Tian, Y. 1990, *A study of the reliability of a multi-transputer system for numerical weather prediction.* Report submitted to the New Zealand Meteorological Service.

[GTP90] Graham, I., Tian, Y. andPurnell, D., 1990, *A transputer system for numerical weather prediction.* Proceedings of the Australian Transputer and occam Users' Group Conference, Sydney, June.

[GTP91] Graham, I., Tian, Y.and Purnell, D. 1991, *A fault tolerant transputer system for numerical weather prediction.* in Transputer '91, Proceedings of the World Transputer User Group Conference, eds P.Welch, D. Styles, T. Kunii and A. Bakkers, 476-496, IOS Press, Amsterdam.

[HA74] Hakimi, S., and Amin, A. 1974. *Characterization of connection aasignment of diagnosable systems.* IEEE Trans. comput. **C-23**, 86-88.

[HC85] R.Horst, and T.Chou, 1985, *The Hardware Architecture and Linear Expansion of Tandem Non-stop Systems*, Proceedings of the 12th Annual Symposium on Computer Architecture, June 1985.

[HKR84] Hosseini, S. Kuhl, J. & Ready, S. 1984. *A diagnosis algorithm for distributed computing systems with dynamic failure and repair.* IEEE Trans. Comput. **C-33**, 223-233.

[HKR85] Hosseini, S. Kuhl, J.and Ready, S. 1985. *On self fault-diagnosis of the distributed systems.* In the 15th International IEEE Symposium on Fault-Tolerant Computing. IEEE, New York, 30-35.

[HLD88] R.E.Harper, J.H.Lala and J.J.Deyst, 1988, *Fault tolerant processor architecture overview*, Digest of papers, FTCS-18, Tokyo, Japan, June 1988, 252-257.

[HOS89] Perihelion Software Ltd. 1989, *The Helios Operating System.* Prentice Hall International.

[HSL78] Hopkins, A. L. Smith, T. B. and Lala, J. H. 1978. *FTMP - A highly Reliable Fault-Tolerant Multiprocessor for Aircraft.* Proceedings of IEEE, **66**, 1221-1239.

[Inm90] INMOS Ltd, 1990, *Transputer Reference Manual*, Prentice Hall, Hemel Hempstead.

[Joh89] Johnson, Barry. W., 1989, *Design and Analysis of Fault-Tolerant Digital system.* Addison-Wesley.

[JZ90] D.B. Johnson & W. Zwaenepoel, 1990, *Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,* J. Algorithms, **11**, 462-491.

[KH87] Kerutzer, S., and Hakimi, S. 1987. *System-level fault diagnosis: A survey*. Microprocess. Micro-program. **20**, 323-330.

[KH88] Kerutzer, S., and Hakimi, S. 1988. *Distributed diagnosis and the system user.* IEEE Trans. Comput. **37**, 71-78.

[Kim78] Kim, K.H., 1978, *An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and its Effcient Implememtation Rules*. Proceedings of International Conference on Parallel Processing, Detroit (MI), 58-68.

[KM85] Kopetz, H. and Merker, W. 1985, *The architecture of Mars*, Digest of papers, FTCS-15, Ann Arbor, Michigan, June, 274-276.

[Koh81] Kohler, W.H., 1981. *A Survey of Techniques for Synchronization and Recovery*. Computing Surv., **13**, 149-183.

[Kop89] H.Kopetz, et al, 1989, *Distributed Fault Tolerant Real-Time Systems: The Mars Approach.* IEEE Micro., **9**, 25-40.

[Kop89] Kopetz, H. et al, (1989). Distributed Fault Tolerant Real-Time Systems: The Mars Approach. IEEE Micro. Vol.9, No. 1, February. pp. 25-40.

[KR80] Kuhl, J. G. and Reddy, S. M. ,1980, *Distributed fault-tolerance for large multiprocessor systems.* In proceedings of the 7th Annual Symposium on Computer Architecture, 23-30.

[KR81] Kuhl, J. G. and Reddy, S. M., 1981, *Fault-diagnosis in fully distributed systems.* In the 11th International IEEE Symposium on Fault-Tolerant Computing. IEEE, New York, 100-105.

[KR86] Kuhl, J. G. & Reddy, S. M., 1986, *Fault-Tolerance considerations in Large, Multiple-Processor Systems.* IEEE Computer, **19.**

[KT87] Richard Koo and Sam Toueg. *Checkpointing and rollback-recovery for distributed systems.* IEEE Transactions on Software Engineering, SE-

13(1), 23-31.

[KWF+88] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, The *MAFT Architecture for Distributed Fault-Tolerance*, IEEE Transactions on Computers, **37**, 398-404.

[KY86] Kim, K., and Yang, S. 1986, *Fault tolerance mechanisms in real-time distributed operating systems: An overview.* In the Pacific Computer Communications' 85. Elsevier Science Publishers, 239-248.

[LAB+90] Laprie, J.C., Arlat, J., Beounes, C., Kanoun, K., 1990, *Definition and Analysis of Hardware and Software Fault Tolerant Architectures.* IEEE Computer.

[Lal86] J.Lala, , 1986, *A Byzantine resilient fault tolerant computer for nuclear power plant applications*, Digest of Papers, FTCS-16, Vienna, July 1986, 338-343.

[LNP92] Kai Li, Jeffrey F. Naughton, and James S. Plank. *An efficient checkpointing method for multicomputers with wormhole routing.* International Journal of Parallel Processing, **20(3).**

[LS92] Michael Litzkow and Marvin Solomon. 1992, *Supporting checkpointing and process igration outside the Unix kernel.* In Conference Proceedings, Usenix Winter 1992 Technical Conference, 283-290.

[LSP82] Lamport, L., Shostak, R. and Pease, M., 1982, *The Byzantine Generals problem.* ACM Transactions on Programming Languages and Systems **4(2),** 382-401.

[Man86] L. Mancini, 1986, Modular redundancy in a message passing system. IEEE Transactions on Software Engineering. 79-86.

[MHK63] Markowitz, H.M., Hausner, B. and Karr, H.W., 1963, *SIMSCRIPT: a Simulation Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.

[ML80] Malek, M. and Liu, K. 1980. *Graph theory models in fault diagnosis and Fault-Tolerance.* In Design Automation and Fault-tolerant Computing, vol. 3, 3, 4. Computer Science Press, 155-169.

[MM81] Maeng, J., and Malek, M. 1981. *A comparison connection assignment for self-dianosis of multiprocessor systems.* In the 11th International IEEE Symposium on Fault-Tolerant Computing. IEEE, New York, 173-175.

[MR78] Merlin, P.M. and Randell, B., 1978, *State Restoration in distributed Systems.* Proc. 8th Int'l Symp on Fault Tolerant Computing. 129-134.

[Nai78] Nair, R. 1978. *Diagnosis, self-diagnosis and roving diagnosis.* Dept. of Computer Science Rep. R-823, University of Illinois, Urbana, I11.

[Nel90] Victor, P. Nelson, 1990, *Fault-Tolerant Computing: Fundamental Concepts.* IEEE, Computer **23.**

[Pla93] James S. Plank. 1993, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University.

[PM83] Powell M.L. and Miller B.P., 1983, *Process Migration in DEMOS/MP.* In Proceedings of the Ninth ACM Symposium on Operating System Priciples, Bretton Woods, New Hampshire, 110-119.

[PMC67] Preparata, F., Metze, G., and Chien, R., 1967. *On the connection assignment problem of diagnosable systems.* IEEE Trans. Elect. Comput. **EC-16**, 848-854.

[Ran75] B. Randell., 1975, *System Structure for Software Fault Tolerance.* IEEE Transactions on Software Engineering, **SE-1(2),** 220-232.

[Ran77] Randell, B. 1977, *System Structure for Software Fault Tolerance* in Current Trends in Programming Methodology (eds R.T. Yeh), **1,** Prentice-Hall, Engle-wood Cliffs (NJ).

[Rus80] Russell, D.L., 1980, *State Restoration in Systems of Communicating Processes.* IEEE Transactions on Software Engineering **SE-6(2),** 183-194.

[SB89] Speirs, N. A. and Barrett, P.A., 1989, *Using Pasive Replicates in Delta-4 to Provide Dependable Distributed Computing.* Digest of

Papers, FTCS-19, Chicago, Illinois, 184-190.

[SBB92] Stahl, M., Buckens, R., and Bianchini, R. 1992. *On-line diagnosis in general topology networks.* In the IEEE workshop on Fault-Tolerant Parallel and Distributed Systems. IEEE, New York, 114-121.

[Sch90] J. Scheurich, 1990. *A distributed loader for transputer systems.* M.Sc thesis, Computer Science Department, Univeristy of Waikato.

[SCL89] Standeven, J., Colley, M. J. and Lyons, D. M., 1989. *Hardware voter for fault-tolerant transputer systems.* Microprocessors and Microsystems. **13(9),** 588-595.

[Shr87] S.K. Shrivastava, 1987, *Replicated distributed processing*, Lecture notes in Computer Science, **248**, 325-337, Springer Verlag.

[Shr91] Shrivastava, S. et al, 1991. *An overview of the ARJUNA distributed programming system.* IEEE Software, **8**, 66-73.

[Sie90] Siewiorek, D. P., 1990. *Fault tolerance in Commercial Computers*", IEEE. Computer, **23.**

[Smi79] Smith, J. 1979. *Universal system diagnosis algorithms.* IEEE Trans. Comput. **C-28**, 374-378.

[SR81] Suk, D.S. and Reddy, S.M., 1981, *A March Test for Functional Faults in Semiconductor Random Access Memories*, IEEE Trans Comp, **C-30**.

[SS82] Siewiorek, D. P. and Swarz, R.S., 1982. *The theory and Practice of Reliable System Design.* Digital Press.

[SS83] R.D.Schlichting and F.B.Schneider, 1983, *Fail-stop Processors: an approach to designing fault-tolerant computing systems*, ACM Transactions on Computing Systems, 1(3), 222-238.

[SY85] R.E. Strom and S. Yemini, *Optimistic recovery in distributed systems*, ACM Trans. Comput. Syst., **3**, 204-226.

[TD92] Tian, Y. Q. and Day, M., 1992. *Mutual redundancy and dynamic migration for crash recovery in distributed systems.* Proceedings of

1992 New Zealand computer research student conference, Univeristy of Waikato.

[TGS91] Tian, Y., Graham, I. and Smith, I. 1991. *A Tool for Fault Prevention in Transputers.* in ATOUG 4, 173-176.

[Tul91] Tully, A. 1991, *Preventing State Divergence in Replicated Distributed Systems*, Ph.D thesis, The computing Laboratory, The University of Newcastle Upon Tyne.

[VV90] Van De Goor, A.J. & Verruijt, C.A, 1990. *An overview of Deterministic Functional RAM Chip Testing*, ACM Computing Surveys, **22**

[Wei80] C.B. Weinstock, 1980, *SIFT: System Design and Implementation*, Digest of papers, FTCS-10, Kyoto, Japan, June 1980, 75-77.

[Wen78] Wensley, J. H. et al., 1978. SIFT: *Design and Analysis of a fault-tolerant computer for aircraft control,* Proceedings of IEEE, **66**, 1240-1255.

[WKF85] C.J. Walter, R.M. Kieckhafer, A.M. Finn, *MAFT: A Multicomputer Architecture for Fault-Tolerant in Real-time Control Systems*, Proceedings of IEEE Real-Time Systems Symposium, Dec. 1985.