Fault Injection Testing of Software Implemented Fault Tolerance Mechanisms of Distributed Systems

Sha Tao

Ph.D. Thesis

October 1996

University of Newcastle upon Tyne

Department of Computing Science

NEWCASTLE UNIVERSITY LIBRARY

096 51028 9

Thesis L5770

Abstract

One way of gaining confidence in the adequacy of fault tolerance mechanisms of a system is to test the system by injecting faults and see how the system performs under faulty conditions. This thesis investigates the issues of testing software-implemented fault tolerance mechanisms of distributed systems through fault injection.

A fault injection method has been developed. The method requires that the target software system be structured as a collection of objects interacting via messages. This enables easy insertion of fault injection objects into the target system to emulate incorrect behaviour of faulty processors by manipulating messages. This approach allows one to inject specific classes of faults while not requiring any significant changes to the target system. The method differs from the previous work in that it exploits an object oriented approach of software implementation to support the injection of specific classes of faults at the system level.

The proposed fault injection method has been applied to test software-implemented reliable node systems: a TMR (triple modular redundant) node and a fail-silent node. The nodes have integrated fault tolerance mechanisms and are expected to exhibit certain behaviour in the presence of a failure. The thesis describes how various such mechanisms (for example, clock synchronisation protocol, and atomic broadcast protocol) were tested. The testing revealed flaws in implementation that had not been discovered before, thereby demonstrating the usefulness of the method. Application of the approach to other distributed systems is also described in the thesis.

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Santosh Shrivastava for his constant support and constructive advice. I am grateful to Professor Shrivastava for his comments and criticisms on the preliminary drafts of the work.

I would also like to thank my colleagues Dr Paul Ezhilchelvan, Dr Neil Speirs, and Dr Francisco Brasileiro for the many fruitful discussions I had with them during the course of the work.

The support and encouragement offered by my family during my studies are also greatly acknowledged.

The work reported in this thesis was financially supported by grants from the CEC ESPRIT programme and the UK Engineering and Physical Sciences Research Council (EPSRC).

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Fault Injection Techniques and Software Testing 5

2.1. Introduction 5

2.2. Simulated Fault Injection 7

2.3. Hardware-Implemented Fault Injection 11

2.3.1. Pin Level Fault Injection 11

2.3.2. Heavy-ion Radiation Injection 15

2.3.3. Other Techniques of Hardware-Implemented Fault Injection 17

2.4. Software-Implemented Fault Injection 17

2.4.1. FIAT Fault Injection Tool 18

2.4.2. FERRARI Fault Injection Tool 20

2.4.3. SFI Fault Injection Tool 22

2.4.4. FINE Fault Injection Tool 23

2.4.5. Simulation-Assisted Fault Injection 26

2.4.6. Other Work on Software-Implemented Fault Injection 27

2.5. Fault Injection for Fault Removal 28

2.5.1. Testing of Distributed Fault Tolerant Algorithms 28

2.5.2. Fault Tolerance Testing of AAS 30

2.6. Overview of Software Testing Techniques 33

2.6.1. Structural Testing 33

2.6.2. Functional Testing 34

2.6.3. Random Testing 35
2.6.4. Equivalence Partitioning Testing 35
2.6.5. Cause-effect Testing 36
2.6.6. Mutation Testing 38
2.6.7. Assertion Testing 39
2.6.8. Comments 40
2.7. Fault Tolerance Testing Strategies 41
2.7.1. Structural Testing 42
2.7.2. Functional Testing 44
2.8. Summary 46

Chapter 3 Focused Fault Injection Method 50
3.1. Introduction 50
3.2. Modelling Faulty Behaviour 51
3.2.1. Simple Responses 52
3.2.2. Replicated Responses 53
3.3. Software Structure Permitting Focused Fault Injection 55
3.4. Injection of Replicated Responses 59
3.5. Injection of Multiple Processes 61
3.6. Injection of Multiple Processors 63
3.7. Ordering Arrivals of Erroneous Messages 67
3.8. Software-Implemented Fault Tolerance in Distributed Systems 70
3.8.1. Node Level Fault Tolerance 70
3.8.2. Distribution Level Fault Tolerance 71
3.8.3. Application Level Fault Tolerance 72
3.9. Summary 73

Chapter 4 Focused Fault Injection on Voltan TMR Node 75

4.1. Introduction 75

4.2. Voltan TMR Node Architecture 76

4.2.1. System Model and Assumptions 77

4.2.2. Node Software Architecture 79

4.3. Implementation 81

4.3.1. Voting Module 83

4.3.2. Clock Synchronisation Module 84

4.3.2.1. The Protocol 84

4.3.2.2. The Implementation 89

4.3.3. Ordering Module 90

4.3.3.1. The Protocol 91

4.3.3.2. The Implementation 95

4.3.4. Communications Layer 96

4.4. Fault Injection Implementation 97

4.5. Experiments and Results 100

4.5.1. Voting Module 101

4.5.2. Clock Synchronisation Module 105

4.5.3. Ordering Module 111

4.5.4. Comments 115

4.6. Summary 115

Chapter 5 Focused Fault Injection on A Fail-Silent Node 117

5.1. Introduction 117

5.2. Fail-Silent Node Architecture 118

5.2.1. System Model and Assumptions 119
5.2.2. Basic Node Architecture 120
5.2.3. Node Failure Semantics 121
5.2.4. The Symmetric Node Design 123
5.2.5. The Leader-Follower Node Design 125
5.3. Leader-Follower Node Implementation 128
5.3.1. Communications Layer 130
5.3.2. Ordering Modules 131
5.3.3. Comparison Modules 133
5.4. Implementing Focused Fault Injection 135
5.5. Experiments and Results 137
5.5.1. Experimental Set-Up 138
5.5.2. Comparison Module of Follower 140
5.5.3. Comparison Module of Leader 141
5.5.4. Time-Monitoring Mechanism of Follower 142
5.6. Summary 143

Chapter 6 Applying Focused Fault Injection at Higher Levels of a Distributed System 144
6.1. Introduction 144
6.2. Distribution Level Fault Tolerance 146
6.2.1. Focused Fault Injection Scheme 146
6.2.2. The ISIS Example 149
6.3. Application Level Fault Tolerance 153
6.3.1. The Manetho Example 154
6.4. Summary 155

Chapter 7 Conclusions 157

7.1. Contributions 157

7.2. Future Directions 159

7.2.1. Limitations of the Work 159

7.2.2. Further Work 160

References 163

Chapter 1: Introduction

Very high reliability is required from computing systems that are used in life- and mission-critical applications. Enormous effort is put into the design and implementation of such systems. Various types of fault tolerance mechanisms are employed to achieve that required level of system reliability. A major problem related to the development of fault tolerant computing systems is their validation.

Fault tolerant systems must handle an 'extra class' of inputs, i.e., failure events, which they are designed to cope with. For systems intended for critical applications, failure probabilities in the range of 10^{-6} to 10^{-10} per hour are often specified [Wensl78]. It is simply not possible to take the conventional approach of running a system for long periods (so failure events do occur) to collect failure data for evaluating the reliability properties of a system. Other ways of system validation must be employed to examine the behaviour of the system in the presence of failures.

Fault injection based testing is recognised as an effective means of validating reliability properties of systems. It has been used to examine reliability mechanisms, such as error detection mechanisms, error recovery schemes, and other aspects of fault tolerance. Fault injection can also be used to study the behaviour of large systems under faulty conditions. Over the years, various fault injection tools and methods have been developed and implemented in both hardware and software.

Fault injection is the general term used to describe a wide range of activities which create the effects of fault occurrences. Especially, 'software-implemented fault injection' actually refers to software based approaches to the injection of errors (manifestations of faults).

This thesis describes a fault injection method that we have developed for testing software-implemented fault tolerance mechanisms of distributed systems. The method differs from the previous work in that it exploits an object oriented approach of software implementation to support the injection of specific classes of faults at the system level. The method requires that the target software system be structured as a collection of objects interacting via messages so that fault injection objects can be easily inserted into the target system to emulate incorrect behaviour of faulty processors by manipulating messages. This approach allows one to inject specific classes of faults without requiring any significant changes to the target system. The method has been applied to test the implementation of redundancy management protocols of a TMR (triple modular redundant) node and a fail-silent node.

The need for the injection of specific classes of faults at the system level is exemplified by the so called 'Byzantine Generals problem'. The problem refers to a situation in which a failed processor of a system exhibits 'two-faced' behaviour, telling one processor one thing and another processor a different thing, thereby 'confusing' the correct processors of the system. It is therefore necessary that the correct processors execute an 'agreement' protocol to prevent any disagreement about the disseminated information. Such Byzantine agreement protocols do exist. These protocols have been validated by formal correctness proofs. However, faults can still be introduced at the

implementation stage. Testing-based validation of the implementation is therefore required. Effective testing of software modules implementing these protocols can only be achieved by the injection of specific types of faults that can create required failure scenarios, such as 'two-faced' Generals. We show in chapter 3 how such software modules can be effectively tested using the method presented in the thesis.

The rest of the thesis is organised as follows.

Chapter 2 gives a survey of the existing work on fault injection. Various fault injection tools and methods and their applications are analysed. Their merits and shortcomings are discussed. A brief discussion on software testing techniques in general is also presented.

In chapter 3, we present our fault injection method. We first describe a fault model for distributed systems. Then we discuss in detail how the target software should be structured to support fault injection and how various failure scenarios can be created through fault injection. In this chapter we also discuss the various levels at which fault tolerance can be provided in a distributed system.

Chapter 4 describes the fault injection experiments conducted to test the soundness of the design and implementation of Voltan TMR node. The Voltan TMR node is implemented entirely in software using standard transputer hardware. A TMR node consists of three processors and is capable of masking the failure of a single processor through replicated processing. The key to the correct functioning of a TMR node is redundancy management. In a software-implemented TMR node, this is provided by the

implementation of redundancy management protocols which forms the 'hard-core' of system software of a node. The 'hard-core' must function correctly in the presence of a single failure for the node to be correct. We tested the 'hard-core' consisting of the voting, clock synchronisation and ordering modules of the Voltan TMR node software.

Our fault injection method has also been applied to test the fault tolerance mechanisms of a software-implemented fail-silent node. Unlike a TMR node which is designed to mask the failure of a single processor and continue to provide a required service, a fail-silent node is only required to exhibit certain fail-silence properties in the presence of a single processor failure. In chapter 5 we describe the fault injection experiments performed on a fail-silent node.

In chapter 6 we discuss the application of our fault injection method to distributed systems where fault tolerance is provided at distribution level or application level. In such systems, message exchanges among the processors of the system are often based upon the use of a set of primitives provided by the underlying communication layer. The target system modules have direct access to the primitives for sending and receiving messages, rather than make use of link handling objects as suggested in chapter 3. It is not possible to insert an injection object to intercept and manipulate output messages. In chapter 6 we describe how our fault injection method can be adapted to be used in the testing of such systems. Thus we show that our approach can be used to test a wide class of distributed computing systems.

Chapter 7 concludes the thesis. Limitations of our work are discussed and plans for future work are outlined.

Chapter 2: Fault Injection Techniques and Software Testing

2.1. Introduction

For systems intended for critical applications, failure probabilities in the range of 10^{-6} to 10^{-10} per hour [Wensl78] and system down times in the range of 3 to 156 seconds per year [Crist90] are often specified. It is simply not feasible to take the conventional approach of running a system for long periods (so failure events do occur) to collect failure data for evaluating the reliability properties of such systems. A more direct means of feeding the target system with a special category of inputs, i.e. failure events, through fault injection, is therefore required. Fault injection based experiments have increasingly been recognised as a very useful way of validating system reliability. Fault injection can be employed for two different objectives with regard to system reliability validation: *fault forecasting* and *fault removal* [Arlat91, Lapri92].

Fault forecasting is not about forecasting the occurrence of faults. It is about the impact of faults on the target system, that is, forecasting the *consequences of faults*. Fault forecasting handles issues such as the likelihood of a fault being detected, how long it takes to detect a fault, and how the target system would behave under the influence of faults, etc. Fault injection has been widely used to examine coverage and detection latency of various error detection mechanisms, and to study system behaviour under

faulty conditions.

Within the context of fault tolerant systems, fault removal deals with the uncovering of flaws and deficiencies in the design and implementation of fault tolerance mechanisms, to make sure that fault tolerance mechanisms do cope with the faults which they are designed to handle. In other words, fault removal is concerned with the removal of *fault tolerance deficiency faults*. In this sense, the process of fault tolerance testing for the objective of fault removal is similar to that of conventional software testing, only that the inputs are of a special category (faults).

The fault injection techniques and methods used in fault injection experiments for the two different objectives are quite different. In the experiments for fault forecasting, the essential requirement is to emulate the occurrence of faults in the real world as closely as possible. The techniques are usually geared towards supporting fault injection of random nature at low levels. While in the experiments for fault removal, the requirement is to be able to inject specific classes of faults so that the fault tolerance mechanisms under test can be checked adequately. Such experiments allow testers to find out whether the fault tolerance mechanisms can indeed tolerate the faults which they are supposed to tolerate.

There is a wide range of techniques which are used to implement fault injection for the purpose of fault forecasting at various stages of the development process of reliable systems. These techniques can be classified into three categories: simulated fault injection, hardware-implemented fault injection, and software-implemented fault injection. Simulated fault injection is typically employed at the design stage, so

that different architectural design ideas can be tested and evaluated, and potential reliability deficiencies can be identified. Hardware- and software-implemented fault injection approaches are suitable for prototype testing and evaluation; these direct approaches avoid the task of constructing complex simulation models. In sections 2.2 - 2.4 we will discuss and analyse the three categories of fault injection techniques.

In section 2.5 we will discuss some existing work on fault injection with the objective of fault removal. Such work is similar in nature to software testing, that is the aim is to test that a system does what it is expected to do. Section 2.6 gives an overview of software testing techniques in general. Section 2.7 presents a brief discussion on the issue of testing strategies that are used in fault tolerance testing. Section 2.8 summarises the chapter.

2.2. Simulated Fault Injection

Simulated fault injection is carried out by injecting faults into the simulation model of the target system under study. It constitutes an important means for performance and reliability evaluation [Iyer93]. Such evaluation is highly useful in comparing alternative design ideas and analysing reliability characteristics. Another obvious advantage of simulated fault injection over hardware/software-implemented fault injection is that there is no restriction in accessing internal parts of components of a processor, and very low level faults can be simulated (see below). Simulated fault injection has been used in evaluating fault tolerant processor architectures during the design stage.

Various levels of system abstraction can be considered for simulated fault injection. In

practice, three levels of abstraction are often used for injection based analysis. They are: transistor level, gate level, and function level.

A wide range of research work has been carried out concerning simulated fault injection at all three levels of system abstraction. Some of them deal mainly with simulation tools, while others concentrate on studying reliability characteristics of some specific systems.

Transistor level faults are simulated by changing the electric voltage and current inside a circuit, which is in fact a form of circuit simulation. FOCUS [Choi92] is a transistor level simulation tool, it adopts a *mixed-mode* approach of simulation. The non-faulty parts of the circuit are simulated at the logical level while the injected parts are simulated at the electrical (analogical) level. Logical level simulation of the non-faulty parts helps reduce the complexity of the simulation model while electrical level simulation of the faulty parts enables a more realistic emulation of real world faults.

FOCUS has been used to study error propagation within a microprocessor [Choi92]. In the study, a total of 2100 simulations was performed to obtain stable results. The study found: 71.9% of the faults injected never caused an error (a faulty signal has to be 'latched' to become an error); 16.4% of the faults injected caused errors that propagated to a pin of an IC chip; and 9.2% of the faults injected caused errors that propagated to the functional output of the microprocessor.

Gate level fault simulation adopts fault models at a higher level of system abstraction. It simulates logical faults, such as stuck-at-0, stuck-at-1, or inverted logic value faults. A

number of studies have used gate level fault simulation to analyse error propagation in IC chips and to characterise the impact of gate level faults on program behaviour. Lomelino [Lomel86] investigated error propagation from the gate level to the pin level. Czeck and Siewiorek [Czeck90] injected faults in a gate level simulation model of the IBM RT PC to investigate the impact of transient gate level faults on program behaviour.

Simulated fault injection can also be carried out at function level to study the reliability of complete computer systems or even distributed systems. In such simulations, components of the system of either hardware (e.g. cpu, memory) or software (e.g. workload) are modelled and their interaction considered.

DEPEND [Goswa90] is a typical example of a function level fault simulation environment. It takes an object oriented approach and provides a collection of objects representing hardware and software components of systems. Users can use these objects to build simulation models rapidly. DEPEND has been used to simulate the UNIX-based Tandem Integrity S2, a TMR node system [Jewet91]. Faults were injected into the simulation model of the system to evaluate the impact on the system MTBF (mean time between failure) by correlated errors, latent errors, memory scrubbing, and repair times [Goswa91]. The results show that correlated errors (i.e., errors affecting two or three processors) with no latency cause enormous degradation to the system MTBF. However, errors typically have latencies and when error latency is taken into account the reduction in the MTBF is not as pronounced. The results also show that there is no relationship between the size of error latency and the system MTBF. For systems designed to tolerate

single faults, repair time is a window of vulnerability. The study shows that reduced repair time improves the system MTBF as long as there are no correlated errors. Memory scrubbing, which is used to detect and remove memory errors, is found to be extremely effective at eliminating errors with large latencies.

Jenn et al [Jenn94, Rimen93] reported work on a simulated fault injection tool (MEFISTO) based on a widely used hardware description language (VHDL). The work differs from previous research work in this area in two aspects. Firstly, MEFISTO is based on VHDL [IEEE88], an existing hardware description language with a wide spectrum of applications. Target system simulation models written in VHDL can be injected directly. This removes the burden of having to learn a new simulation language and construct a simulation model in this language. Secondly, MEFISTO allows fault injection at multiple levels of abstraction. This feature is supported by VHDL's ability to describe both structure and behaviour of a target system. MEFISTO has been used to fault inject a processor. The main objective of the experiment is to analyse the impact of the choice of the injection method and the model description level on the error outcome.

Another example of function level fault simulation tool is React [Clark95]. React is specifically designed to assess reliability properties of multi-processor architectures. It can be used to study different fault tolerant architectures such as, N modular redundancy, duplication and comparison, and standby sparing.

Some of the network simulation tools, such as NEST [Dupuy90], which were not originally developed for fault simulation, can also be used to simulate node and link

failures to model faults in distributed systems.

2.3. Hardware-Implemented Fault Injection

Hardware-implemented fault injection is also known as physical fault injection and involves the physical introduction of faults into the target system either by applying voltage levels to the pins of IC chips, radiating IC chips with heavy ion, or some other forms of physical interference. Pin level fault injection changes the external behaviour of an IC chip by having some of its pins stuck-at-1, stuck-at-0, or inverted. Heavy ion radiation allows faults to be injected inside a chip and so changes the external behaviour of the chip in an indirect way. The behaviour of IC chips can also be modified through power disturbance or electro-magnetic interference.

2.3.1. Pin Level Fault Injection

Pin level fault injection is the most widely used hardware fault injection method. It is especially useful in evaluating error detection mechanisms for detection latency and coverage. There are two different techniques used for implementing pin level fault injection, known as *forcing* and *insertion* respectively.

With the forcing technique, probes are attached to the IC pins (injection points) directly. The current/voltage levels of the injected pins can then be altered to emulate erroneous logic values. Using the forcing technique, the fault types allowed are limited to stuck-at-0 and stuck-at-1.

The insertion technique requires some physical modification of the target

hardware. The selected IC chip is extracted from the circuit board and an extra piece of hardware, called *socket*, is then inserted between the IC chip and the circuit board. Through the socket, logic values of the pins can be manipulated. Apart from the stuck-at-0 and stuck-at-1 fault types, more complex fault types, such as inverted faults where logic values are inverted (0 to 1, or 1 to 0) and open faults where pins are 'open' (disconnected), can also be injected. Most pin level fault injection tools use insertion technique.

For an early example of pin level fault injection method, see the papers by Decouty and Crouzet [Decou80, Crouz82]. The aim was to evaluate the coverage of error detection mechanisms attached to various modules of a self-checking microcomputer [Morei76]. Faults were injected into the target microcomputer and results were monitored through the use of a purpose-built tool. The tool consisted of a fault injector and a hardware monitor. The hardware monitor observed whether the errors caused by the injected faults were detected by the error detection mechanisms and whether erroneous outputs were emitted by the microcomputer. The study found error detection rates to be very high: 96% CPU errors, 99% ROM errors, and 99% RAM errors were detected. Another significant observation of the experiments was that no erroneous outputs were emitted by the microcomputer. In other words, either the microcomputer was stopped when an error was detected or the error didn't result in the microcomputer emitting erroneous outputs.

The fault recovery mechanism of the FTMP computer [Hopki78] was evaluated using pin level fault injection [Finel87]. The main objective was to collect data on fault

recovery times and establish their statistical distribution. This information is of vital importance for reliability estimation of the FTMP. The fault injection set-up has a hardware fault injector, interface hardware, and support software. Experiments were controlled from a host computer on which the support software was run. Fault injection instructions were issued from the host computer to the injector; the results were read by the FTMP itself and sent back to the host computer through the hardware interface. The results of the experiments were very interesting. While no single distribution of fault recovery times was shown to be the best fit for all the data sets, the exponential distribution, which is often assumed in reliability modelling, was a bad fit for all data sets.

The MESSALINE fault injection tool [Arlat89, Arlat90a, Arlat90b] developed at LAAS has been used to test both centralised and distributed target systems with reliability mechanisms through pin level fault injection. In the case of the centralised target system [Arlat90a], the target system was a subsystem of a computerised interlocking system for railway control applications. The mechanisms examined were a self-test program and the hardware error detection mechanism of the system. Stuck-at-1, stuck-at-0, and open faults were injected. The experiments showed that the test program was far more efficient in detecting errors than the hardware error detection mechanism.

In the second exercise [Arlat90a], the Multicast Communication System (MCS) of Delta-4 distributed system [Powel91] was examined. The MCS provides multicast services that are built using an atomic multicast protocol (AMP). The MCS is implemented within Network Attachment Controllers (NACs) which connect the host

machines of a distributed fault tolerant system to a local area network. The NACs have self-checking capabilities and are assumed to be fail-silent (i.e., they fail by stopping and becoming silent). Fault tolerance of the MCS depends on two levels of coverage: local coverage, provided the self-checking capabilities of the NACs, and distributed coverage provided by the defensive properties of the AMP. The defensive properties of the AMP refer to its ability to provide continued fault tolerance in the event that the fail-silence property of NACs is broken and erroneous messages are sent by NACs. The experimental results showed that 67.47% of the errors caused by injected faults were detected and the NAC was subsequently extracted from the network (become silent). The results also showed that 23.79% of the errors, though not detected, did not result in any erroneous behaviour of the MCS. This was due to the distributed fault tolerance coverage provided by the AMP. Thus, in 91.26% of the injections, the MCS was able to handle the error correctly.

In a more recent study, Madeira and Silva [Madei94] investigated the effectiveness of error detection mechanisms in guaranteeing fail-silent behaviour by using pin level fault injection. Two target computers, one Z80 based and the other 68000 based, were evaluated. A number of error detection mechanisms were tested in the contexts of the two computers. These error detection mechanisms check program control flow, memory access behaviour, and illegal instructions. The results showed that by using a combination of such error detection mechanisms, very high error detection rates could be achieved. The Z80 based computer achieved a combined error detection rate of 97.8%; while the 68000 based computer achieved 90.4%. As expected, not all undetected errors would cause a violation of fail-silence. The actual fail-silence

coverages achieved were higher than the error detection rates. The Z80 based computer achieved a fail-silence coverage of 99.6%; while the 68000 based computer achieved 98.1%.

Fault injection experiments were also conducted on the Z80 and 68000 based computers without any added error detection mechanism. The results were not surprising. 16.7% of errors caused by injected faults resulted in a fail-silence violation in the 68000 based computer; while 45.6% of such errors resulted in a fail-silence violation in the Z80 based computer.

Other examples of pin level fault injection include [Shin86] for measuring error detection latency and [Schue86] for rating the coverage of error detection mechanisms.

2.3.2. Heavy-ion Radiation Injection

Heavy-ion radiation of IC chips [Gunne89, Mirem92] allows faults to be injected within IC chips which is not possible with pin level fault injection. Another difference between pin level injection and heavy-ion radiation injection concerns the certainty of fault injection. With pin level fault injection, one has control over the faults being injected with regard to the types of faults and the location of injection. However, when a IC chip is radiated with heavy-ion, there is no such certainty of control. As a result, heavy-ion radiation based fault injection experiments are conducted in ways different from those of pin level fault injection. An experimental set-up consisting of two CPUs is normally required; one is subject to fault injection and the other acts as a reference. The two CPUs operate in synchrony using the same main clock and are connected to a

comparator. When a fault which actually causes an error is injected, it will be detected by the comparator and the output signals of both CPUs are recorded.

Heavy-ion radiation testing was used when a MC6809E microprocessor was fault injected to generate error data [Gunne89]. The error data were later used as input of programs which simulated error detection schemes suitable for a watchdog processor. A watchdog processor is a small processor that checks the behaviour of the main processor on the external bus [Mahmo88]. Each error detection scheme consisted of a number of individual error detection mechanisms for checking program control flow, memory access behaviour, and illegal instructions. All error detection mechanisms were individually evaluated and then a number of combinations (schemes) were also evaluated. The results showed that, while the individual error detection mechanisms typically have detection rates at around 25% - 60%, the error detection schemes have much higher coverage. The best scheme detected 79% of errors and 99% of the errors that caused execution to diverge were detected by the scheme.

Heavy-ion radiation has also been used to examine two software error detection techniques directly [Mirem92]. These two error detection techniques are known as Block Signature Self-Checking (BSSC) and Error Capturing Instructions (ECI), respectively [Mirem92]. They are intended for checking program control flow. In the experiments, the two error detection techniques were evaluated under different workloads and detection coverage and latency were measured.

2.3.3. Other Techniques of Hardware-Implemented Fault Injection

Other techniques of hardware-implemented fault injection include electro-magnetic interference [Leber93] and power supply disturbance [Damm86, Mirem92]. These techniques of fault injection are similar in nature to that of heavy-ion fault injection. They allow faults to be injected within IC chips but they suffer from the same lack of experimental control in terms of fault injection location and types of faults to be injected. However, electro-magnetic interference and power supply disturbance do emulate faults in the real world most closely.

2.4. Software-Implemented Fault Injection

Software-implemented fault injection introduces errors into the target system by software means. Compared with hardware-implemented fault injection, the software approach does not require special hardware equipment, therefore it offers reduced cost, more flexibility and better experimental control. With the software approach, high level fault injection becomes possible, which opens the way for effective testing of some software implemented fault tolerance mechanisms. For these reasons, software-implemented fault injection has become increasingly popular in recent years.

Software-implemented fault injection is typically carried out by changing memory content in either data or program code sections, changing register content, or triggering some built-in hardware error detection mechanism. The injection techniques used mostly involve modifying the memory image of processes at compile time. In this way, the program control flow can be altered and fault injection routines can be incorporated

into the target program which will later carry out fault injection work when the program is run.

2.4.1. FIAT Fault Injection Tool

Segall et al [Segal88] developed the FIAT fault injection environment for evaluating reliability properties of distributed real-time fault tolerant systems. FIAT works by manipulating the target software systems, known as *workloads*, at symbolic level (see the explanation in the following paragraph). A target software system is a collection of communicating processes, with each process consisting of a code segment and a data segment.

The target system is first analysed and the symbolic names (known as *attributes*) are extracted. These symbolic names identify individual processes, and code and data segments within individual processes of the target software system. Using the extracted symbolic names, the tester can express what faults are to be injected at what objects identified by the symbolic names. These fault injection intentions consisting of type of fault to inject and location of injection are expressed in the form of *fault classes*. A fault class is like an abstract data type. The tester specifies the characteristics of the faults to be injected in a fault class, and later the fault class will be used by the fault instance generator to generate a list of faults to be injected.

The fault classes provided by FIAT are:

- Memory fault;

- Register fault;
- Communication fault;
- Error detection mechanism triggering fault.

A number of program attachments are linked to the target system at link time. These program attachments monitor the target system, carry out actual fault injection, and report high level abnormal events. The specially linked target software systems are executed on Fault Injection Receptacle (FIRE) machines and the experiment is controlled from a Fault Injection Manager (FIM) machine. The FIREs and the FIM are connected by a local area network.

FIAT has been used to examine a real-time distributed checkpointing fault tolerant system [Segal88]. The target system consisted of two computational engines: the primary and the secondary. The primary, on receiving a request for real-time computation, informs the secondary of the request as well as the time for next interaction. The primary then executes the request. The secondary waits for the next interaction. If the next interaction has exceeded the time bounds (i.e., primary failure), the secondary then initiates a recovery action and becomes the primary. If the primary detects that no secondary exists (i.e., secondary failure), it creates a secondary. In the experiments, the primary was fault-injected to examine the failure detection coverage and detection latency of the secondary.

The FIAT fault injection environment demonstrated the viability of emulating hardware

faults through software-implemented fault injection.

2.4.2. FERRARI Fault Injection Tool

The FERRARI fault injection tool [Kanaw92] made improvements in software-implemented fault injection in terms of better controllability. It allows faults to be injected in specific physical locations instead of only those mapped by symbolic names. The time of fault injection during the execution of the target system can also be controlled. FERRARI can also inject transient faults which cause errors of limited durations.

In FERRARI, the target system is first analysed and executed. The purpose of this analysis and execution phase is to determine the starting address and size of the text (code) and data segments of the executable file, and to extract the execution behaviour of a fault free run, such as the execution time, the output, and the addresses used by the program. Fault injection instructions of a user are expressed through experiment parameters. These parameters include:

- Experiment modes (user specified or automatic selection of fault location, time and duration);
- Fault types (bit XOR, bit set, bit reset, byte set, or byte reset);
- Fault class (data, control flow, or user defined);

- Type of reliability measurements (coverage, or coverage and latency).

Using the results of the target system analysis and the experiment parameters provided by the user, the target system is modified and injection points (software traps) are set up. When the modified target system is run, it will be trapped at the injection points, where selected faults are injected.

FERRARI supports the injection of both transient faults (called transient errors in [Kanaw92]) and permanent faults. When the execution reaches a specified address, the program is trapped. For transient faults, a selected fault is injected and the current instruction is executed, and then the error caused by the injected fault is removed and the program is allowed to resume execution. For permanent faults, the error caused by the injected fault is not removed. The program is trapped for the next n instructions, where n is the duration of the fault.

In FERRARI, faults are modelled on bus line faults (both address line faults and data line faults) and faults in condition code flags, though the actual manipulations (fault injection) are applied to memory cells and registers. For example, an "address line fault while the processor is fetching an instruction" is said to have been injected when the processor is forced to fetch a different instruction. This is achieved by modifying the program counter. For transient faults, the modified program counter will be restored to its correct value after the execution of the incorrect instruction. While for permanent faults, the program counter will be modified repeatedly for several instructions, or the entire execution interval of the target application.

FERRARI has been implemented on a SUN SPARC station and used to measure the effectiveness of several redundancy based error detection techniques that were built into application programs [Kanaw92]. As the experimental results showed, most of the errors caused by injected faults were detected by the built-in error detection mechanisms of the SPARC system before the application level error detection techniques had a chance. However, most of the errors that slipped through the detection of the SPARC system were either detected by the application level detection techniques or caused a program crash.

2.4.3. SFI Fault Injection Tool

The Software Fault Injector (SFI) developed by Rosenberg and Shin [Rosen93] allows fault injection at various levels for different purposes. Low level faults can be injected to create memory errors and CPU failures (such as the failure of the adder or multiplier) for testing reliability mechanisms implemented on single nodes. Injection at message level facilitates the testing of distributed reliability mechanisms. Messages from the injected node can be omitted, delayed, or altered.

Compared with other software-implemented fault injection tools, SFI also offers better timing control. With SFI, faults can be injected as transient, intermittent, and permanent faults, and the timing parameters of all these types can be specified by the user. A transient fault is injected only once, at a given time after the start of an experiment run. An intermittent fault is injected repeatedly at the same location. For an intermittent fault, the tester can specify the distribution of the interval time between injections. The

interval time can be deterministic, with a set time between injections, or can follow an exponential distribution with a given mean. When the interval time between injections is small, the injected fault will behave like a permanent fault.

Three methods are employed in implementing fault injection in SFI. They are active injection, control flow alteration, and code replacement. Active injection is performed by a process that runs concurrently with the executing workload. Active injection is used to inject memory faults. Control flow alteration can be used to modify the functional behaviour of the system. It is used to inject communication faults. Code replacement can be used to emulate faults in areas which are otherwise not accessible, such as the adder or the multiplier of the processor.

SFI has been used to examine the effect of intermittent communication failures (message omissions) on the message delivery time between two adjacent nodes in the HARTS distributed real-time system [Shin91], and it has also been used to evaluate a number of routing algorithms for distributed systems by injecting omission faults in selected nodes.

2.4.4. FINE Fault Injection Tool

While most fault injection based studies concentrate on the final impact of faults on the target system with emphasis on latency and coverage issues, Kao et al [Kao93] looked into the issue of how errors propagate in a software system. A software tool for fault injection and monitoring (FINE) was developed and used to trace UNIX system behaviour under the influence of faults. FINE is made up of four major components

(fault injector, software monitor, workload generator, and controller) and some analysis utilities.

The fault injector supports the injection of both hardware faults and software faults. Since the application programs do not have the privilege to modify the kernel, the fault injector is implemented in two parts, client and server. The server part is implemented in the UNIX kernel. It provides an interface for the client part to specify the faults to be injected into the kernel.

The software monitor traces the execution flow (by using *probes*) and key variables of the kernel, and writes trace data to a file. The probes are inserted into most of the significant functions to keep track of the execution flow and arguments. The software instrumentation is embedded in the kernel to monitor system behaviour. While the functionality of the software monitor is implemented in the kernel, an interface (in the form of a system call) is provided so that a user program can specify the probes and key variables to trace.

The workload generator generates synthetic workload of system calls according to user specification. The controller specifies the fault for the fault injector, the key variables for the software monitor, and the workload specification for the workload generator; it then starts the experiment.

Both hardware faults and software faults (software bugs) can be injected with FINE. The hardware fault types are:

- Memory faults;
- CPU faults (register faults);
- Bus faults;
- I/O faults.

The software faults modelled in FINE are:

- Initialisation faults;
- Assignment faults;
- Checking faults;
- Function faults.

Initialisation faults include uninitialised variables and wrongly initialised variables. Assignment faults can be missing assignment or incorrect assignment. Checking faults include missing condition checks and incorrect condition checks. Function faults are those which involve multiple incorrect statements.

Experiments on SunOS 4.1.2 were conducted to investigate error propagation and to evaluate the impact of various types of faults. Based on the results of the experiments, error propagation models were built for both hardware and software faults. The experimental results also revealed that memory faults and software faults usually have a

very long latency while bus faults and CPU faults tend to crash the system immediately.

2.4.5. Simulation-Assisted Fault Injection

Software-implemented fault injection is restricted to the parts of the target system that are accessible to software. Sub-instruction level faults, such as the omission of a micro-instruction in a RISC (Reduced Instruction Set Computer) processor can not be directly injected using conventional software-implemented fault injection methods. Guthoff and Volkmar [Gutho95] proposed a simulation-assisted software-implemented fault injection method.

Under this method, fault injection is carried out in three steps. In the first step, the target system starts off normally and then is interrupted when a fault is to be injected. In the second step, the state of the target system is transferred to the simulator and the resulting state after fault injection is calculated. In the final step, the resulting state is transferred back to the target system and execution is resumed. This method has been used to investigate the impact of the omission of a single micro-instruction on the behaviour of a Motorola MC88100 RISC processor. The experiments were carried out while the processor was running a benchmark program. The experiments revealed that the omission of a single micro-instruction can cause segmentation violation, bus error, and division by zero error. The detection latency of these errors by the processor's built-in error detection mechanisms was also measured.

This combined method offers the benefits of software-implemented fault injection while allows access to parts of the target system which are not accessible using conventional

software approach. Because the portion of the target system which is simulated is only the part of the target system which is not accessible by software, the effort required for the construction of the simulation model is kept to a minimum.

2.4.6. Other Work on Software-Implemented Fault Injection

Software-implemented fault injection provides an important means for studying systems behaviour under faulty conditions. Chillarege and Iyer [Chill87] investigated error latency in systems by injecting memory faults in a VAX 11/780 system using data gathered through hardware instrumentation. The workload is that of a typical multi-user time-sharing system, which consisted of a variety of scientific and miscellaneous word and data processing applications. The study finds that the mean error latency in the memory containing the operating system varies by a factor of 10 to 1 (in hours) between the low and high workloads. The study also finds that most errors were discovered within a day of fault injection.

Chillarege and Bowen introduced the idea of *failure acceleration*, and studied the failure behaviours of a large commercial transaction processing system using memory fault injection [Chill89]. The work revealed: only 16% of faults actually cause a total loss of primary service; some errors do not affect short term system availability but would cause a catastrophic failure following a change in operating state, where a change of operating state refers to a substantial change in workload or a change in system configuration; some errors are potential candidates for repair before total failure.

Though the emphasis of the two papers has been on the methodology and design of such

experiments rather than on the fault injection techniques, it does show the value of software-implemented fault injection in such investigations.

2.5. Fault Injection for Fault Removal

Unlike fault injection for the purpose of fault forecasting (i.e., issues of error latency, error detection coverage and error propagation, etc.), fault injection for the purpose of fault removal inevitably requires the injection of specific classes of faults in order to uncover flaws in the design/implementation of fault tolerance mechanisms. The issues of injecting specific classes of faults (also known as *deterministic fault injection* [Echtl91]) for the objective of fault removal have been looked at in a number of contexts.

2.5.1. Testing of Distributed Fault Tolerant Algorithms

The issue of fault injection based testing for the removal of design faults of distributed fault tolerant algorithms has been discussed in [Echtl91], and a *structural testing* approach suggested. It is observed that the distributed fault tolerant algorithm under test can be represented by a *structure graph*. The goal of structural testing is to feed the software implementation of the algorithm with carefully selected inputs so that some (or all) structural parts of the algorithm are executed and results (outputs) are monitored. It is hoped that design faults can be revealed through this execution.

For distributed fault tolerant algorithms, the inputs also include faults. In distributed systems, the behaviour of a faulty processor is exhibited by the erroneous messages the

faulty processor sends. So message level fault injection was adopted in [Echtl91] for the testing of distributed fault tolerant algorithms. The internal conditions of a faulty processor is of no significance for the distributed fault tolerant algorithm under test.

One important issue in structural testing is the selection of faults in order to cover certain structural parts of the algorithm. Echtle et al [Echtl91] suggested the use of a special heuristics which is based on the typical characteristics of distributed fault tolerant algorithms.

EFA [Echtl92], a distributed testbed system for testing the fault tolerance capabilities of distributed algorithms, was developed to support the implementation of deterministic fault injection. This distributed testbed system provides a number of facilities, including special communication primitives. These special communication primitives allow the transmission and receipt of messages to be intercepted and monitored. This provides the basis for deterministic fault injection. The intercepted messages can be manipulated in a number of ways. Injection of the following faults is supported:

- Falsification of message contents;
- Multiple transmission of messages;
- Falsification of message transmission times (delays);
- Re-ordering of message transmissions;
- Generating spontaneous messages;

- Combinations of the fault types listed above.

The fault cases to be injected are expressed by the testers in the form of a program module which will be called by the testbed system software when the experiment is executed.

The distributed fault tolerant algorithms to be tested must be implemented using the special facilities provided by EFA. The main idea here is to modify messages in a manner that will force the algorithm under test to take specific execution paths. Distributed fault tolerant target systems implemented using the usual communication facilities provided by a communication subsystem can not be tested in this testbed. In other words, this testbed is for algorithms, not for implementations.

Avresky et al [Avres92] also investigated the issue of structural testing of fault tolerant algorithms through deterministic fault injection, and the Inter-Replica protocol (IRp) of Delta-4 distributed fault tolerant architecture [Powel91] was partially tested on a simulator. The IRp provides co-ordination functions necessary to handle communications between replicated application processes. A small part of the code implementing the IRp was tested on a simulator. The simulator simulated three stations (three replicas). Two faults were discovered: one was an implementation fault and the other was a protocol design fault.

2.5.2. Fault Tolerance Testing of AAS

The Advanced Automation System (AAS) is a distributed real-time system developed

for the US Federal Aviation Administration to provide future air traffic control services for the US [Avizi87, Benel89, Crist90]. It is a large and complex system with very high reliability requirements. To verify its reliability properties prior to commissioning, it becomes necessary to conduct a systematic testing of the fault tolerance capabilities of AAS at all levels through various forms of fault injection [Dilen91].

Fault tolerance in AAS is provided in a hierarchy of fault handling facilities, at both local level and distributed systems level. These fault handling facilities include error detection, error reporting, and error recovery. The AAS fault tolerance testing is intended for both reliability assessment (fault forecasting) and validation of fault tolerant software. As a result, two approaches were taken: specific testing and selective sample testing. With specific testing, specific error conditions are created to verify whether the system can operate correctly and cope with the expected failures. With selective sample testing, large numbers of faults of random nature are injected in the system under various operational conditions in order to identify any previously unknown failure modes and to establish a statistical basis for the evaluation of system reliability.

The method adopted in the AAS fault tolerance testing is to integrate the fault injection provisions into the AAS architecture. Each Ada (the AAS implementation language) program address space contains a fault injection subsystem which interprets and co-ordinates the execution of fault injection instructions in that address space. At the local level, the faults which can be injected include:

- Ada exceptions raised;

- Memory corruption;
- Timer manipulation;
- Processes delayed or terminated;
- Operating system failures.

At the distributed systems level, communication messages can be manipulated to emulate:

- Message loss;
- Message delay;
- Message corruption;
- Message duplication.

These fault injection capabilities allow a wide range of AAS modules to be tested for fault tolerance. A good example of how specific testing can help uncover software bugs is the testing of the implementation of a group membership protocol in AAS. After the initial analysis showed possible flaws in the implementation, a specific test case was constructed to test the software. The experiment revealed a missing piece of exception handling code.

Although the AAS fault tolerance testing mainly centres on specific testing for the

removal of fault tolerance deficiency faults, the fault injection capabilities developed are also used in selective sample testing for assessing system reliability.

2.6. Overview of Software Testing Techniques

The testing activities that occur during a software project can be classified into unit testing, integration testing and system and acceptance testing [Ince89]. Unit testing is the process of checking a program unit (subroutine or procedure) with test data. The main aim of unit testing is to ensure that a program unit meets its specification. Integration testing is the process of testing a partial version of the system while small chunks of the system are added. The aim is to ensure that the interface between the chunk that has been integrated and the system is correct. Finally, system and acceptance testing is conducted to ensure that a software system meets its system specification. System testing is carried out by the developers of the system while acceptance testing is carried out by the users though the aim of the testing and the techniques used are the same.

In this section we present an overview of some of the established software testing techniques and discuss their application in different testing activities.

2.6.1. Structural Testing

Structural testing [Ince93] involves testing a program so that some structural metric is satisfied or a particular path is traversed. The latter is often referred to as path testing. Examples of program metric include percentage rate of statements or conditional

branches being executed. While in path testing, a program path can be any execution path from the beginning to the end of the program being tested.

The test data for structural testing can be generated by analysing the source code of the program. Whether particular parts of a program have been executed can be monitored by inserting software probes into the program under test.

Structural testing is normally employed during unit testing where specific structural metrics are set, for example, all statements are to be executed and 90% of conditional branches are to be traversed.

The advantage of structural testing is that test data can be derived systematically and test coverage measured. In the next section we will discuss structural testing further in the context of fault tolerance testing.

2.6.2. Functional Testing

Functional testing is a testing technique where the specification of the program under test is used to derive test data and then the test data is used to check whether the program behaves as specified. In functional testing, the tester is not concerned with the internal structure of the program being tested.

The key issue here for the tester whose aim is to discover program defects is to select test data which have a high probability of exposing program defects. Effective selection of test data is dependent on the skills and experience of the tester but there are some structured techniques (see sub-sections 2.6.4 and 2.6.5) which can be used to guide the

selection.

Functional testing can be employed in unit testing, integration testing, and system testing.

2.6.3. Random Testing

Random testing is a technique of randomly generating test data. It involves identifying the input data space for a program and randomly generating test data inside that space.

Random testing is cheap to conduct in terms of tool support as all that required is some form of random number generator. Another advantage of random testing is that it is extremely good at producing data which a human tester would not think of. The main disadvantage of random testing is that for large programs the amount of data needs to be generated is prohibitively high. And this ensures that random testing can only be used for unit testing.

Due to the random nature of this approach to testing, it should only be employed as a useful adjunct to other testing techniques.

2.6.4. Equivalence Partitioning Testing

Equivalence partitioning [Somme92] is a test data selection technique whereby the input data is divided into classes (*equivalence partitions*) of common properties. A program should behave in a comparable way for all members of an equivalence partition.

The equivalence partitions may be identified by using design or functional specification and by the tester using experience to predict classes of input data which may lead to different execution paths. For example, a program processing temperatures may process temperature readings in different ranges (below zero, at zero, and above zero) differently. Then the input data space can be divided into three equivalence partitions. Test data will be selected from the three equivalence partitions.

This test data selection technique is useful during unit testing, integration testing, and system testing.

2.6.5. Cause-effect Testing

Cause-effect testing [Ince93] is another test data selection technique. This technique involves examining the program output and analysing it to establish the relationship between input events (causes) and output events (effects). The main advantage of cause-effect testing is in that it is possible to consider combinations of events that occur in a test.

The basic elements of the cause-effect testing notation are shown in Fig. 2.1. On the left-hand side of the graphs are the causes which give rise to events in a system. Typical causes might be an operator typing a command, or a valve closing. On the right-hand side of the graphs are the events that occur because of causes, for example, an alarm being sounded, or an error message being displayed on a screen.

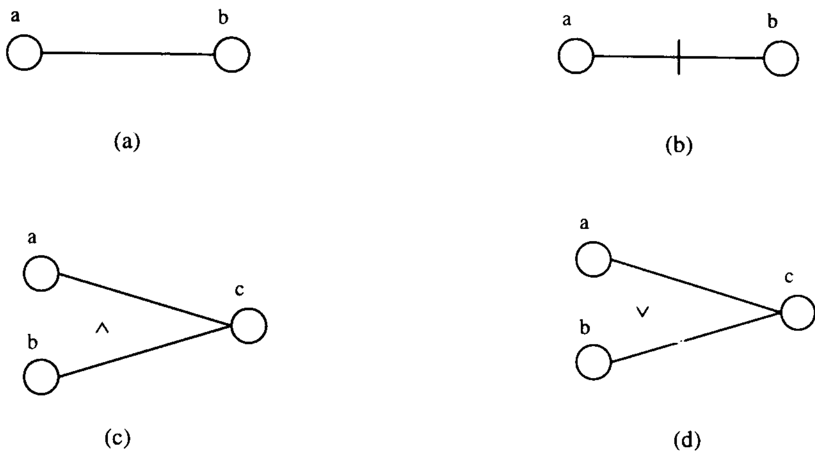

Fig. 2.1

The first graph in Fig. 2.1 states that event b will occur when event a occurs. The second graph states that event b will occur when event a does not occur. The third graph states that event c will occur when events a and b occur, and the fourth graph states that event c will occur when event a or event b occurs. A simple example cause-effect graph is shown in Fig. 2.2. The graph shows that event e will occur when either event a occurs or events b and c both occur.

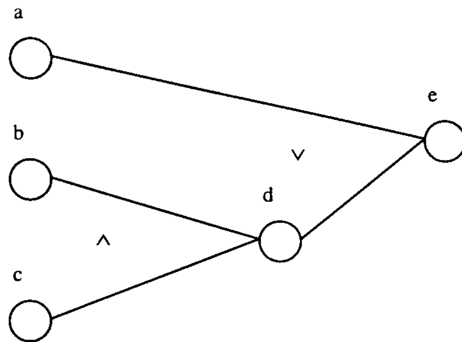

Fig. 2.2

The cause-effect graph of a software system is derived from the functional specification of the system. This process involves a number of steps. First the system is partitioned into manageable chunks so that each chunk can be analysed on a piece of paper or on a computer screen. Then the causes and effects are identified. The cause-effect graph is then built up. The resulting graph is used to guide the selection of test data.

The cause-effect testing technique is mainly employed during system testing.

2.6.6. Mutation Testing

Mutation testing [Mathu94] is a technique which is used to examine the adequacy or effectiveness of test data. Once a series of tests have been conducted, a collection of test data will have accumulated. One important question a tester may ask is: is the test data used adequate in terms of test effectiveness? Mutation testing is a technique aimed at answering this question.

Mutation testing is based on creating *mutants* of a program. A mutant is a

modified version of the original program in which a small error is inserted. A typical error would be to replace an operator with a different one, for example, replacing an addition operator with a multiplication operator. A large number of such mutants are created, and are then executed using the test data which was employed in testing the original program. If the mutant gives a result which is different from the result of the test of the original program then the mutant is said to have been killed. This means that the test data is able to distinguish between the original program and the mutant.

The percentage rate of the mutants having been killed reflects the level of adequacy of the test data. If the tests of the mutants result in all the mutants being killed then the test data is adequate. However, if any mutants are still living after the tests then it is clear that the test data is incapable of exposing these bugs. Further test data will be required to kill off any living mutants.

Though mutation testing can be used during integration testing and system and acceptance testing, it is best employed during unit testing. This is mainly due to the fact that in general there are a massive number of mutants that can be created.

2.6.7. Assertion Testing

An assertion is a predicate which relates the values of variables in a program and describes a condition which must be true during the execution of a program. Assertion testing [Ince93] is a technique which checks such properties of a program during execution. If a program is correct then such properties must hold during program execution. As a simple example, the following predicate is an assertion: $a > b + c$. It

states that during program execution variable a must be greater than the sum of variable b and variable c.

Assertions can be inserted into a program under test either by hand or by means of a software tool [Ince93]. Assertion testing can be employed during the whole testing process up to system testing.

It should be pointed out that assertion testing only checks certain properties of a program. Such properties must hold if the program is correct. However, such properties may still hold even if the program contains bugs. Hence assertion testing should only be used in addition to other testing techniques.

2.6.8. Comments

In general there are two broad approaches to software testing: structural testing and functional testing [Somme92]. In the structural approach, the aim is to achieve certain structural metrics or to traverse a particular path in the testing. The selection of test data is clearly guided by this aim.

In the functional approach, the tester is not concerned with the internal structure of the program under test. The key issue here for the tester is to select test data which has a high probability of exposing program defects. In this section we have examined a number of test data selection techniques: random testing, equivalence partitioning, and cause-effect testing.

The adequacy or effectiveness of test data in terms of its defect revealing power can be

checked through mutation testing. The assertion testing technique is aimed at testing a program from a different perspective by checking certain run-time properties of a program which must be true if the program is correct.

In this section we only discussed generic software testing techniques which are generally independent of application domains. Some application domain specific testing techniques have also been proposed which take into account the characteristics of application. For example, the testing of telecommunications software [Avrit95], where the arrival distribution of input (telephone calls) is of great importance in revealing software defects.

2.7. Fault Tolerance Testing Strategies

When considering fault injection based testing for the purpose of removing fault tolerance deficiency faults, one is faced with two related issues: fault injection techniques and testing strategies. While fault injection techniques deal with the question of how to inject required faults effectively, testing strategies deal with the issue of what faults to inject in order to achieve an adequate test of the target system. In this chapter we have already described the techniques for fault injection, now we discuss the related issue of testing strategies.

As in conventional software testing, there are two broad testing strategies: *structural testing* and *functional testing*. In the structural approach, the idea is to select faults such that these faults will cause certain structural parts of the target software system to be executed. The overall aim is to inject a set of faults so that all parts of the target system

are executed at least once, with the hope that fault tolerance deficiency faults in the target system will be exposed once the system is fully exercised. Functional testing takes a more direct approach. Faults are injected to create specific failure scenarios to ascertain that the target system can indeed tolerate such faulty conditions. These two contrasting approaches are discussed and their merits and shortcomings are analysed in the following sub-sections.

2.7.1. Structural Testing

The first step in structural testing involves the construction of the ***structure graph*** [Echtl91] (or *execution tree* [Avres92]) of the program under test. The structure graph models a program by characterising it as forks leading to branches. A node in the structure graph represents a sequence of statements, while an edge represents a branch of a conditional statement. An example of a structure graph is shown in Fig. 2.3. When constructing the structure graph of a program, conditional loops are modelled as a chain of forks if the number of iterations is finite. With the structure graph of a program constructed, faults can be selected and injected in 'faulty' processor(s) with the aim of having the program running on correct processor(s) execute certain branch(es) of the structure graph.

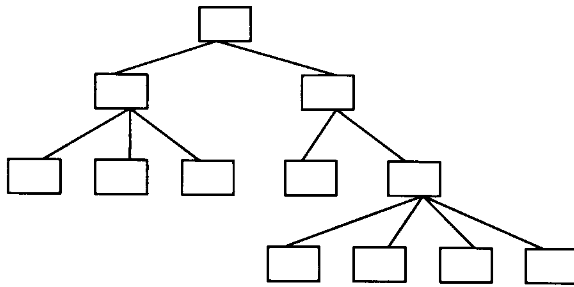

Fig. 2.3 Program Structure Graph

The structural approach towards fault tolerance testing has the theoretical nicety of completeness, in the sense that a complete test of the target program can be carried out by covering all the branches of the structure graph. The level of testing granularity can also be determined according to the amount of resources allocated to the testing efforts. In a 'large grain' testing, the leaves of the structure graph may be large pieces of program that contain conditional statements; while in a 'fine grain' testing, the leaves may contain only sequential statements (non-conditional statements). Given enough resources, it is possible to have each and every one of the statements of the program covered (executed) in a complete test.

The structural approach has some serious problems. The first question one would ask is "what is the relationship between the correctness of a program under test and a complete coverage of its structure graph". Obviously achieving a complete coverage of the structure graph does not necessarily mean that a program is correct. The execution of a part of the structure graph may give correct results for some input data but incorrect results for some other input data. Another shortcoming with the structural approach has to do with the way the structure graph is generated. A structure graph is generated from

the design of the target software system to be tested. So if the design is incorrect then the structure graph, which is used as a reference framework for testing, may be incorrect as well. For example, if a branch of the structure graph is missing due to a system design fault (i.e. a certain failure scenario is not handled by the program), the bug would not be revealed by a testing based on structural coverage.

There are also some practical difficulties in implementing structural testing of fault tolerance. In order to cover the structure graph with a limited number of test runs, a mechanism for recording information regarding which parts of the graph have been covered will be needed. It was suggested in [Echtl91] that software implemented *probes* be planted in the program code to record the actual program flow. It might be feasible for testing fault tolerant algorithms implemented in a purpose-built testbed system with facilities supporting the use of probes, but it is likely to be difficult to plant probes in a target system implemented in its own environment. Structural testing is generally a tedious process and is not scaleable. Thus it is often impractical to conduct on complex target systems.

2.7.2. Functional Testing

Functional testing of fault tolerance takes a more direct approach. Faults are selected from the domain of all possible faults and then are injected in the 'faulty' processor(s) to create specific failure scenarios. The selection of faults is based on an analysis of the target system which looks into the possible failure scenarios.

In functional testing, the selection of faults is of crucial importance. Because the domain

of all possible faults is often large, it is impossible in practice to inject all of them. With limited time and resources, one must be vary careful in fault selection. The corresponding failure scenarios created through fault injection should be representative of possible failures and include those known as *malicious* failures.

In order to be able to selected the appropriate faults to inject, a tester must fully understand the algorithm employed by the software system under test. Only with a solid understanding of the algorithm, the tester can determine what failure scenarios to create for the testing. The tester also needs to know the implementation structure of the target system in terms of the processes that make up the system and how they interact with one another. This outline knowledge of implementation is needed for inserting fault injection objects. The testing of the clock synchronisation module of Voltan TMR node described in chapter 4 offers a good example. The tester must know the clock synchronisation algorithm well to be able to select sensitive faults and the tester also needs to know the implementation structure of the module to be able to insert the fault injection object.

Most distributed fault tolerant systems and algorithms reported in the literature assume processors are fail-silent (i.e., a processor either works correctly or fails by crashing). Therefore, in such systems, the only failure processors can suffer is stop sending messages (permanent omission fault). This assumption greatly simplifies the efforts of fault selection; fault selection becomes a matter of deciding at what points during the execution of the target system the message flow should be cut off to emulate processor failure. Although in theory there are numerous points during the execution of the target

system processors can fail, an analysis of the system may show that there are only limited number of logical points which are representative of various failure scenarios. What is important in distributed systems is the sequence of events, not absolute timings of events.

Functional testing based on initial analysis was adopted in the fault tolerance testing of the Advanced Automation System [Dilen91]. A number of errors in system requirements, system design, and system implementation were uncovered through the fault tolerance testing.

2.8. Summary

As stated at the beginning of the chapter, fault injection can be used for two objectives with regard to system reliability validation: fault forecasting and fault removal. The fault injection techniques required for the two different objectives of system reliability validation are quite different.

The essence of fault forecasting is that of *understanding*, to understand how various reliability mechanisms perform and to understand how systems behave under faulty conditions [Stein95]. The measurements taken in fault injection experiments include coverage and detection latency of various error detection mechanisms, and recovery latency of error recovery mechanisms. Such measurements can be used to estimate the actual performance of the reliability mechanisms in field use [Powel95]. With regard to systems behaviour under faulty conditions, work has been carried out to investigate error propagation, error latency (the time it takes for an error to cause a system failure),

and the various ways in which errors can cause system failures.

The basic requirement for fault injection techniques used for fault forecasting is to emulate the occurrence of faults in the real world as closely as possible. The techniques are geared towards supporting fault injection of a random nature. In this chapter we discussed various techniques and their applications; a number of examples in each category were examined in some detail. As we can see from this discussion, hardware-implemented fault injection requires special hardware equipment and is generally difficult to conduct. The experiment personnel must have detailed knowledge of the target system's hardware implementation to be able to carry out experiments. And in many cases, access to registers or memory addresses in chips is impossible. Software-implemented fault injection generally allows more flexibility and controllability. A wide variety of faults can be injected by software means. However, software-implemented fault injection is not without shortcomings. First, the fault injection software integrated with the target system may affect the running of the target system, though careful experimental design can alleviate the problem. Second, poor time resolution of the software approach can be a problem in some experiments, e.g., when measuring error detection latency.

The nature of fault removal through fault injection based testing is rather similar to that of conventional software testing. The aim is to uncover any fault tolerance deficiency faults in the design and/or implementation of fault tolerance mechanisms. Here the faults injected need not necessarily represent closely the occurrence of faults in the real world, rather faults should be selected for their potential for exposing flaws in the fault

tolerance mechanisms under test. This implies the injection of specific classes of faults at levels appropriate to the target systems under test.

Compared with the research work carried out on fault injection for fault forecasting, existing work on fault injection for fault removal has been of a rather limited nature. The issue of structural testing of fault tolerant algorithms through the injection of specific classes of faults has been investigated by some researchers, and a distributed testbed system (EFA) has been developed for such testing. EFA allows the injection of specifically altered messages, including those emulating malicious faults. However this tool is designed to test algorithms instead of target systems (implementations), the algorithms must be implemented using the primitives provided by the tool to allow it to be tested.

As we all know, implementing distributed fault tolerant algorithms is not a trivial task. All sorts of errors can be introduced at the implementation stage. It is very important to be able to test a fully implemented target system. This point is well illustrated by the experience with the fault tolerance testing of AAS. In the fault tolerance testing of AAS, fault injection provisions were made in the target system by the system developers. The testers can later make use of these provisions to conduct fault injection experiments.

In this chapter, we also discussed software testing techniques in general. As it is clear, in conventional software testing, the ability to feed the target system with required input is not an issue; the research work has centred around the issue of test data selection. While in fault tolerance testing one has to face the additional issue of how to inject a specific

fault once it has been selected though the issue of fault selection is still there.

Testing fault tolerance mechanisms of distributed systems is generally a difficult task. In this thesis, we develop a fault injection method which exploits the object oriented approach of software implementation to support the injection of specific classes of faults at the distributed systems level. This method does not require fault injection provisions to be made in the target system, only that the target system be structured in an object oriented way.

Chapter 3: Focused Fault Injection Method

3.1. Introduction

Our fault injection method is intended for testing software-implemented fault tolerance mechanisms of distributed systems. It requires that the target software be structured in a modular fashion of objects interacting via messages so that messages can be manipulated to emulate incorrect behaviour of faulty processors [Tao93].

In distributed systems where processors interact by message exchanges, the failure of a processor will be exhibited by its external behaviour which is entirely represented by the messages the processor sends (or fails to send). Thus the failure of a processor can be emulated by altering the messages a processor is supposed to send. There is no need to be concerned about the internal conditions of the failed processor. This helps explain the suitability of message level fault injection for testing fault tolerance mechanisms of distributed systems.

Such an approach should ideally achieve the following two objectives: (1) The task of instrumenting the system in order to perform fault injection based testing should be simplified as much as possible. (2) A tester should be able to test the system without having to ask the target system designers to make explicit provisions in the target system in order to support such testing. The benefits of easing the task of testing are obvious. Separating the tasks of implementing and testing a target system is highly

desirable, because it allows testers to test the target system independently without being influenced by the target system designers.

The focused fault injection method supports the injection of specific classes of faults at specially selected points within the target software system. To allow focused fault injection, the target software system must be structured in a modular fashion of objects interacting via messages. In such a system, fault injection objects can easily be inserted into the target software system to carry out fault injection work and no other provisions for fault injection are required in the target system. The target system needs only minimal changes in order to run the experiments.

This chapter is organised as follows. In section 3.2 we present a fault model which characterises the faulty behaviour of a processor when a simple response of one message or a replicated response of multiple message replicas is expected from the processor. In sections 3.3 - 3.6 we describe how various failure scenarios can be created using the fault injection method. Section 3.7 discusses controlling the arrival order of erroneous messages at a correct processor. In section 3.8 we present a brief discussion on software-implemented fault tolerance in distributed systems and put our fault injection method in this context. Section 3.9 concludes the chapter.

3.2. Modelling Faulty Behaviour

Faults are causes of failures. In this section we consider various forms of faults in distributed systems. We assume the components in our systems to be processors and communication links connecting them. We can model link failures by the failures of the

processors associated with the links. We therefore restrict our discussion of faults in distributed systems to that of processor faults. We start with the simplest case, that is when a processor is expected to produce a response consisting of only a single message; later on we consider more complicated situations.

3.2.1. Simple Responses

Processor faults can be classified into the value and time domains as *omission fault*, *value fault*, *timing fault*, and *arbitrary fault* according to the types of failures caused by the faults [Ezhil86, Shriv90]. When a processor is expected to produce a *simple response* consisting of a single message, a correctly functioning processor will produce a message with the correct value and within the correct time frame; whilst a faulty processor's behaviour could be any violation of this correct behaviour.

An *omission fault* would cause an expected message not to be produced at all; the corresponding failure is called an *omission failure*.

A *value fault* would cause a message to be produced within the specified time frame but with its content corrupted; the corresponding failure is called a *value failure*.

A *timing fault* would cause a message with correct content to be produced outside the specified time frame, either early or late; the corresponding failure is called a *timing failure*.

An *arbitrary fault* would cause any violation from the specified behaviour in terms of timing and/or value; the corresponding failure is called an *arbitrary failure*. A fault

which causes a processor to produce an unexpected message is also classified as arbitrary fault.

An arbitrary fault (failure) subsumes all other three classes of faults (failures). The relationships among these four fault (failure) classes can be expressed by the fault (failure) lattice in Fig. 3.1, where an arrow from A to B, A $\rightarrow$ B, indicates that fault (failure) class A is a special case of fault (failure) class B. Omission fault can be treated as either a (very) late timing fault or a fault causing no value to be produced (a special case of corrupted value).

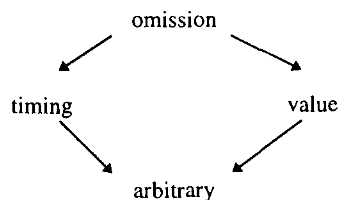

Fig. 3.1 Fault/Failure Lattice (simple response)

3.2.2. Replicated Responses

In distributed fault tolerant systems, replicated processing is often employed to increase system reliability. In such systems, a processor is often required to generate a *replicated response*. A replicated response consists of a number of message replicas. The fault model outlined in Fig. 3.1 can be extended to deal with the various ways a replicated response may differ from the correct one [Ezhil86, Shriv90].

A correct replicated response will be the one in which all individual message replicas have identical and correct values and are produced within the required time frame.

An incorrect replicated response can take many forms. One specific type of fault can be defined as a *consistent fault* which would cause the individual messages of a replicated response to violate the specified behaviour in an identical way, such as having identical but wrong values (*consistent value fault*), while with the general case of a value fault, the individual message values could be wrong and need not be identical, or only some values could be wrong and others correct.

In a similar manner, *consistent omission fault* and *consistent timing fault* can be seen as special cases of omission fault and timing fault respectively.

A consistent omission fault would cause all messages in a replicated response not to be produced at all, while with the general case of an omission fault, it could be that only some messages are not produced.

A consistent timing fault would cause all messages in a replicated response to be produced either early or late, while with the general case of a timing fault, it could be that only some messages are produced late or early.

We use the fault (failure) lattice in Fig. 3.2 to summarise the relationship among the fault (failure) classes in the extended fault model [Ezhil86, Shriv90].

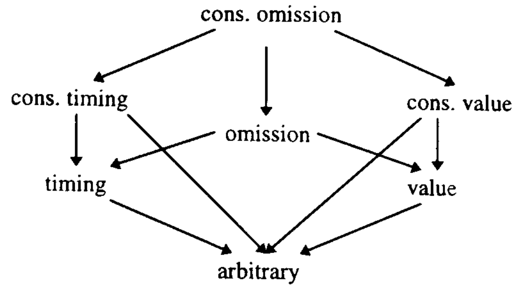

Fig. 3.2 Fault/Failure Lattice (replicated response)

The fault model discussed here has similarities to other fault models proposed in the literature [Crist91, Powel92]. Essentially addressing faults which cause incorrect simple responses and incorrect replicated responses, these models are mainly used as the basis for the design of fault tolerant system architectures and algorithms [Crist91] and for system reliability analysis [Powel92].

3.3. Software Structure Permitting Focused Fault Injection

The key of focused fault injection is the way the target software system is structured. To support focused fault injection, the software within a processor must be structured out of a collection of *active objects* representing processes, which communicate with one another by exchanging messages through *message queues.*

An example of such systems is illustrated in Fig. 3.3. In this system, there are six active objects (processes), two of which (L_{in} and L_{out}) are link handling objects. For each physical link of the processor, there will be a link handling object (L_{in}) which receives messages and distributes them to their destination processes by depositing the messages

in the message queues associated with the processes; and there will be another link handling object (L_{out}) which actually sends messages down the link. Other processes wishing to send messages to destinations outside the local processor would deposit messages in the message queue associated with the link handling object (L_{out}).

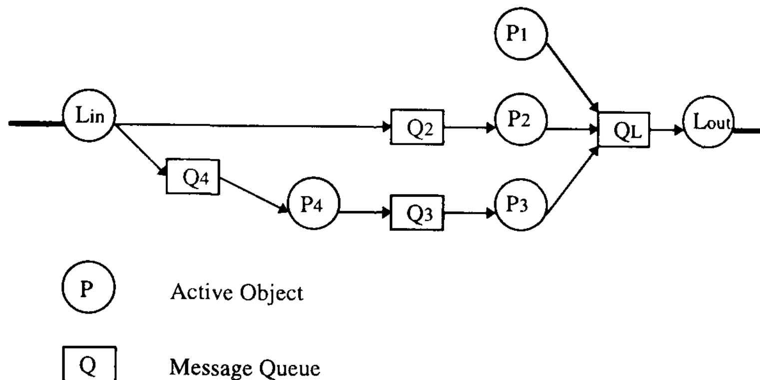

Fig. 3.3 Software System Structure

This system structuring approach makes it possible for a simple and effective way of injecting faults. Fault injection is carried out by *fault injection objects*, which are active objects.

A fault injection object (FO) with its own input message queue (FQ) is inserted between two normal active objects (P, L) which are connected by a message queue (Q) (see Fig. 3.4(a) and Fig. 3.4(b)). P represents a functional process while L represents a link handling (output) process which actually sends outgoing messages down a physical link of the processor hosting P and L. Thus, the faulty behaviour of this processor can be emulated by modifying the messages being output by L. P puts its output messages on FQ. FO picks up messages from FQ, does the fault injection work by modifying the

messages, and puts the output messages on Q which is used by L as its input queue. In the normal operation mode, P is started with Q as one of its parameters, but in fault injection mode, P is started with FQ instead of Q as one of its parameters. The fault injection object is started with FQ and Q as its parameters. The active object L is unchanged.

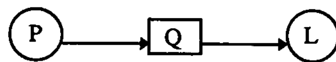

Fig. 3.4(a) Normal Software Structure

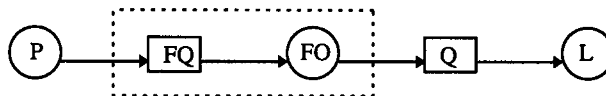

Fig. 3.4(b) Software Structure with Fault Injection Object

Various classes of faults can be injected by the fault injection object. The net effect is the processor hosting P and L producing erroneous messages. Thus we can tamper with messages produced by specific processes within a processor, so as to be able to create required failure scenarios.

An omission fault in P can be injected by having the injection object delete a message produced by P. A value fault in P can be injected by having the injection object change the content of a message produced by P. A late timing fault in P can be injected if the

injection object holds a message produced by P for a period of time before depositing the message in its output queue (Q).

Unfortunately there is no equivalent way of injecting an early timing fault, though it is possible to achieve this in a target-system-dependent way. For example, an early timing fault in the ordering module of Voltan TMR node (see chapter 4) can be emulated by changing the value of the timestamp of a broadcast message.

The injection of an arbitrary fault can be done either by the injection object injecting both timing and value faults, or by having two pipelined injection objects, one injecting timing fault and the other injecting value fault. A faulty processor may also produce a message unexpectedly. This failure scenario can be created by having the fault injection object send a self-made message.

It should be noted that although a late timing fault can be injected by delaying the message for a certain mount of time which can be specified by the tester, the tester has no control over either the time when the message actually arrives at its destination process or the relative ordering position of the message with regard to other messages received by the destination process. This is due to variable message transmission delays which are inherent to distributed systems. A different way of fault injection will be required to achieve the desired behaviour at the destination. We discuss this issue and the solution in section 3.7.

Inserting a fault injection object into the target system is simple, it needs only minimal changes to the target system. This can be illustrated by the implementation of focused fault injection on a Voltan TMR node (see chapter 4).

3.4. Injection of Replicated Responses

We have described the injection of simple responses involving only one single message in the last section. Now we describe how failure scenarios of replicated responses can be created with the flexible use of fault injection objects. When a processor is expected to produce a replicated response and has failed, the possible failures are classified in the fault lattice for replicated response (Fig. 3.2).

If the processor only has one link through which messages are sent out, one can use a single fault injection object to inject faults and create incorrect responses the same way as for simple responses. The injection object has full control over the erroneous messages injected in terms of value and timing of the individual messages. Omission fault, consistent omission fault, timing fault, consistent timing fault, value fault, consistent value fault, and arbitrary fault can all be injected. Desired failure scenarios can thus be created according to experimental requirements.

However, if the processor has multiple links and the message replicas of a replicated response need to be sent down the multiple links, the situation becomes complicated (see Fig. 3.5(a)). Consider, for example, that a consistent value fault is to be injected and the incorrect value is dynamically generated at run time. Such a fault would be difficult to inject simply by inserting multiple injection objects between the functional process P

and the multiple link handling processes (L_1, ..., L_n), due to the co-ordination required among the multiple injection objects.

Though it is possible for the fault injection objects to co-ordinate their injection activities, a straightforward solution would be to use a single injection object with multiple input and output queues (see Fig. 3.5(b)). The injection object FO has full control over the erroneous messages injected. This structure is particularly suitable when the effects of a processor behaving like a 'two-faced General' [Lampo82] are to be emulated.

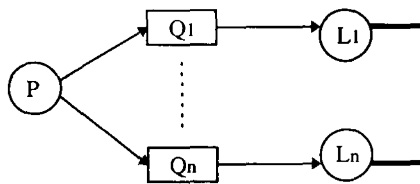

Fig. 3.5(a) Normal Software Structure

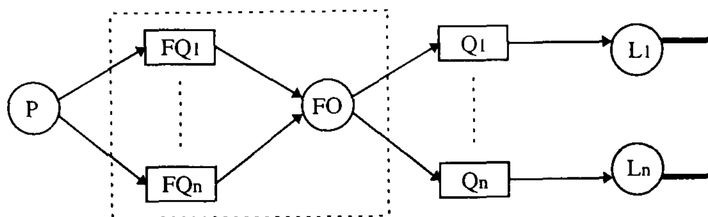

Fig. 3.5(b) Software Structure with Fault Injection Object

The fault injection object FO can manipulate individual message replicas of the response in the value and time domains using the techniques described in the previous section. Extra messages can also be generated by the fault injection object.

3.5. Injection of Multiple Processes

In this section we discuss the injection of multiple processes, that is, to emulate the faulty behaviour of a processor which has a number of processes running on it. Apart from the manipulation of the individual messages produced by the processes, it may also require the order of the messages be manipulated in some fault injection experiments.

One possible approach of injecting multiple processes is to use multiple fault injection objects to intercept and manipulate individual messages of the processes. The fault injection objects can also co-ordinate their injection activities to change the order of the messages involved. However, as we suggested in the previous section, a straightforward solution would be to use a single injection object with multiple input and output queues to centralise the co-ordination work involved.

Let us consider another example for which the use of a single injection object will be highly suitable. In this example, a processor with multiple output links hosts a number of processes and a crash-failure (permanent omission failure) of the processor is to be emulated. There are large numbers of distributed fault tolerant services that are designed on the assumption that the processor fails by crashing (becoming silent). These services can be rigorously tested by having a participating processor fail at a critical moment, for example, in the midst of an atomic transaction.

Fig. 3.6(a) shows the software structure within a processor before the injection object is inserted. There are three functional processes (P_1, P_2, and P_3) and two link handling (output) processes (L_1 and L_2) which actually send the outgoing messages down the two links respectively. The functional processes send messages on the links by putting the messages on the respective queues (Q_1 and Q_2). Fig. 3.6(b) shows the software structure with the injection object (FO) inserted.

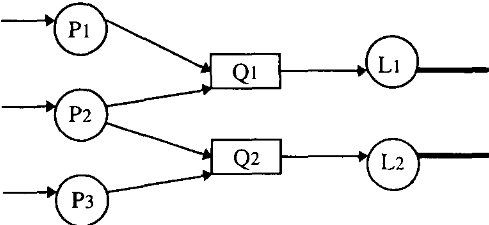

Fig. 3.6(a) Normal Software Structure

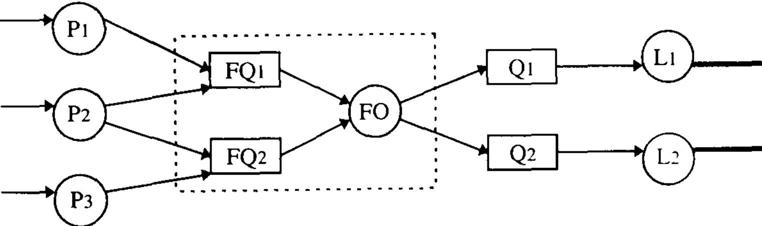

Fig. 3.6(b) Software Structure with Fault Injection Object

The injection object has two input queues and two output queues. Its essential role is to inspect the incoming messages and decide whether to pass them on or cut off the message flow.

3.6. Injection of Multiple Processors

In a more generalised situation, more than one processor of a distributed system can fail and their failures can be related. Many distributed fault tolerant algorithms are designed to handle such multiple failures, the testing of their implementation would require the injection of multiple processors to emulate such failure scenarios.

The injection of multiple processors generally requires a certain amount of co-ordination among the fault injection objects running on the injected processors. The exact pattern of co-ordination depends on the failure scenario being created and the nature of the target system under test. We first consider a number of cases in which the only co-ordination required is among the fault injection objects of the injected processors. In the following section we will discuss the situation in which a fault injection object may need to be inserted into the software running on a non-faulty processor.

Here we discuss some of the examples we have looked at.

Example 1: Voting protocol for a five-processor NMR node.

A five-processor NMR (N-Modular Redundant) node is designed to tolerate up to two failures of its constituent processors. This is achieved through replicated processing on the individual processors and voting on their outputs. As long as there are at least three

processors functioning correctly, a correct majority can be chosen from the outputs of the five processors. When testing the implementation of the voting protocol of such a node, two selected processors are subject to fault injection, creating double failures.

A malicious double failure scenario will be one in which the two injected processors produce incorrect but identical output messages. This has the effect of two faulty processors colluding with each other.

Such a failure scenario can be easily created by programming the fault injection objects on the two injected processors to manipulate output messages in an identical manner, for example, by adding the same values to the intercepted output messages.

Example 2: A clock synchronisation algorithm.

We consider the testing of the implementation of a clock synchronisation algorithm. To be specific, we consider the algorithm by Srikanth et al [Srika87] for a five-processor system. The algorithm can tolerate up to f processor failures in a system of n processors, where n=2f+1. In a five-processor system, it tolerates up to two processor failures.

Assuming the clocks on the processors are initialized correctly, they are synchronised on a periodic basis using the clock synchronisation algorithm. A processor will sign and broadcast a clock synchronisation message to all other processors when the expected synchronisation time is up according to its local clock. When a processor gathers f+1 messages it will synchronise its local clock and relay the f+1 messages to other processors. The idea is that when f+1 messages are received, at least one correct processor is ready to synchronise.

A situation of collusion by two faulty processors would be one in which the two faulty processors both send a clock synchronisation message to a correct processor in a pre-determined synchronisation round earlier then a non-faulty processor would, with the aim of having the correct processor synchronise earlier then it should.

The fault injection objects on the two injected processors can be programmed to send such an early synchronisation message in the pre-determined synchronisation round.

Example 3: Byzantine agreement protocol.

Being able to reach agreement in the presence of faults is of fundamental importance. The agreement problem can be stated simply as follows. Assume there are n processors. Each non-faulty processor produces a value and the value must be communicated to each other non-faulty processor. Non-faulty processors always communicate 'honestly', whereas faulty processors may 'lie'. An agreement protocol, in which processors communicate their own values and relay values received from others, allows each non-faulty processor to infer a value for each processor. Such a protocol satisfies the following two conditions:

(1) *Validity*: The value inferred for a non-faulty processor must be the value produced by that processor.

(2) *Unanimity*: The value inferred for a faulty one must be consistent with the corresponding value inferred by each other non-faulty processor.

The authenticated agreement protocol by Pease et al [Pease80] is such an agreement protocol. A message authentication mechanism is assumed to make sure that modification of messages relayed by faulty processors can be detected by non-faulty processors, though faulty processors may lie about their own values and decide not to relay certain messages. This protocol can tolerate up to n faulty processors, though it will be vacuous if fewer than two processors are non-faulty.

The essence of the protocol is that it ensures each non-faulty processor receives an identical set of values produced by other processors (non-faulty processors or faulty processors), knowing that a faulty processor may send one value to a non-faulty processor and a different value to another non-faulty processor. Once all non-faulty processors have the identical set of values, they can derive an identical value for each processor, faulty or non-faulty.

In the testing of the implementation of this protocol, a sensitive failure scenario which stretches the protocol to its limits would be one in which the faulty processors co-operate in not sending/relaying a value to a non-faulty processor.

We consider a system of four processors, where two of them are faulty. The fault injection object in one injected processor can be programmed such that a value (message) is not sent to a non-faulty processor, while the injection object on the other injected processor can be correspondingly programmed not to relay this value (message) to that non-faulty processor.

3.7. Ordering Arrivals of Erroneous Messages

We have investigated the faulty behaviour of processors in distributed systems and described how such behaviour can be emulated using the focused fault injection method. The perspective taken is that of a faulty processor or a set of faulty processors.

Due to the inherent variation in message transmission delays in distributed systems, the exact impact of such faulty behaviour on correct processors may not be deterministic. In other words, though we are able to control the faulty behaviour of a processor through fault injection we may not be able to control the way a correct processor is affected by such faulty behaviour of the injected processor.

Let us consider a simple example. In a distributed fault tolerant system of four processors, three processors (P1, P2, and P3) will each send a message (msg1, msg2, and msg3 respectively) to the fourth processor (P4). When the processors are fault free, the three messages will arrive at P4 in the order of msg1 msg2 msg3. Now we want to test whether a late timing fault of P1 can indeed be tolerated by the system as it is designed to. The message msg1 from processor P1 is delayed by a pre-determined amount of time before it is sent to P4, emulating a late timing fault suffered by P1. Apart from msg1 sent by P1, the correct processor P4 also receives two other messages (msg2 and msg3) from P2 and P3 respectively. Because of the unpredictable message transmission delays, there are three possible orders of message arrival: (1) msg1 msg2 msg3; (2) msg2 msg1 msg3; (3) msg2 msg3 msg1. The three different message arrival orders may represent three different failure scenarios as far as the correct receiving processor P4 is concerned. The fault injection based testing of the software running on

the correct processor P4 may require the failure scenario in which the erroneous message msg1 arrives between the two correct messages (msg2, msg3).

From a practical point of view, it would be very useful to be able to control the arrival order of erroneous messages relative to the correct messages and the arrival order among the erroneous messages themselves. This would allow the creation of failure scenarios as perceived by a correct processor. Here the perspective taken is that of a correct processor.

Fault injection techniques similar to those described in previous sections for manipulating output messages from 'faulty' processors can be employed here to manipulate the arrival order of input messages which originated from 'faulty' processors.

It should be emphasized that, this means planting fault injection objects in the software running on *correct processors*. One must be very careful so that the semantics of the software which is under test is not altered by the introduction of fault injection objects. This is quite different from inserting a fault injection object into the software running on a 'faulty' processor, in which case one's only concern is the creation of a certain failure scenario.

Fig. 3.7(b) shows the insertion of an injection object between a link handling (input) process L and a functional process P. The software structure before the insertion of the injection object is shown in Fig. 3.7(a). The fault injection object FO can control and

manipulate the arrival order of the erroneous messages relative to the correct messages and the arrival order among the erroneous messages themselves.

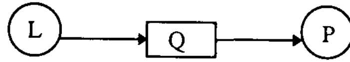

Fig. 3.7(a) Normal Software Structure

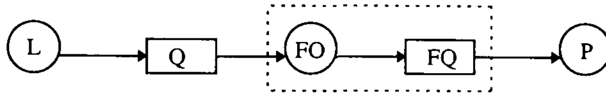

Fig. 3.7(b) Software Structure with Fault Injection Object

When an erroneous message destined for P arrives at the correct processor, it is received by L and is deposited in Q. FO picks up the erroneous message and can decide whether to deliver the message to P immediately or hold the message until certain correct messages have been delivered to P.

The fault injection object FO can also control and manipulate the message order among the erroneous messages themselves instead of just the order of erroneous messages relative to the correct messages.

3.8. Software-Implemented Fault Tolerance in Distributed Systems

The central objective of implementing fault tolerance in distributed systems is to achieve systems reliability so that the services provided by the system will still be available in the presence of component failure(s). Fault tolerance can be applied at three different levels in distributed systems to achieve systems reliability as shown in Fig. 3.8. They are node level, distribution level, and application level.

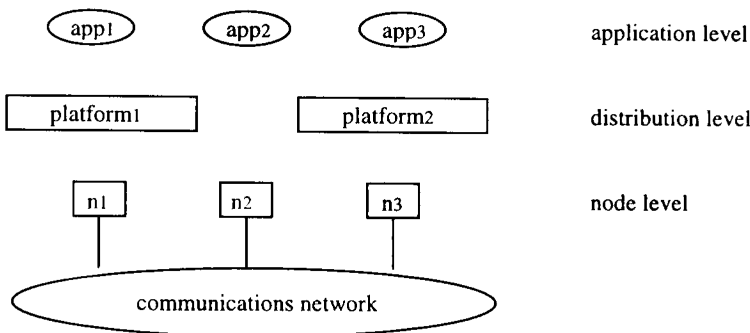

Fig. 3.8 Levels of Fault Tolerance in Distributed Systems

3.8.1. Node Level Fault Tolerance

At the node level, fault tolerance can be implemented so that the nodes are capable of masking internal component failures or exhibiting fail-silent behaviour. If the nodes of a distributed system are capable of masking their internal component failures, then no fault tolerance for component failures will be required at higher levels. Conventional application programs should run on such nodes with little or no change. If the nodes are

only capable of exhibiting fail-silent behaviour then failure-masking capabilities will need to be provided at a higher level.

A failure-masking node consisting of multiple conventional processors can be implemented either in hardware [Jewet91] or in software [Shriv92]. A software-implemented failure-masking node is effectively a distributed system itself. This is due to the fact that the multiple processors of a node communicate with one another through message exchanges using fault tolerant protocols. The same comments apply to fail-silent node implemented in software.

Focused fault injection method can be used in fault injection testing of failure-masking and fail-silent nodes implemented in software. The ability of the method to inject arbitrary faults is especially useful since the processors of such nodes can fail in any arbitrary way.

3.8.2. Distribution Level Fault Tolerance

Fault tolerance can also be provided at the distribution level. This level is typically responsible for implementing distribution transparency, permitting application level programs to manipulate local and remote objects in a uniform manner. At this level, mechanisms are required for providing continued service in the presence of failures. Normally, the nodes of the distributed system are assumed to be fail-silent. The distribution layer software, sometimes called *fault tolerant platform*, is responsible for redundancy management in the presence of failures. Applications developed on top of

this layer are shielded from the complex techniques required for redundancy management.

Arjuna [Shriv91, Parri95] and ISIS [Birma93] are two good examples in this category. Arjuna adopts an object oriented and provides atomic transaction facilities for manipulating persistent objects. Objects can be replicated for availability. While in ISIS the central structuring concept is fault tolerant *process groups*. ISIS provides reliable, ordered multicast protocols for managing process groups.

Our fault injection method can be employed in the testing of such platforms; for example, by injecting permanent omission faults to emulate node crashes.

3.8.3. Application Level Fault Tolerance

Finally, fault tolerance can also be built into the application directly. In this approach, fault tolerance techniques are used in the development of applications and the application programmers are directly involved in using fault tolerance techniques.

The most well known type of application level fault tolerance are those using checkpointing algorithms[Elnoz92, Plank95]. A checkpointing system itself does not provide continued service in the presence of node failures. What it provides is a series of consistent states of the distributed system so that when a node failure is detected a back-up node can be activated. The previously stored consistent states of the system can be retrieved and the replacement node can join remaining nodes. In a distributed application employing a checkpointing algorithm, it is the responsibility of the application programmers to implement error recovery so that services can be restored.

Checkpointing algorithms typically assume that the nodes are fail-silent. Fault injection testing of a checkpointing system would involve injecting permanent omission faults in the selected node(s).

3.9. Summary

The focused fault injection method is most suitable when the target software system is structured in a modular fashion of active objects communicating with one another by sending messages via message queues. This software structuring approach allows easy insertion of fault injection objects into the target software system. The inserted fault injection objects can be programmed to manipulate the output messages to emulate incorrect behaviour of faulty processors. No other provisions for fault injection are required in the target system under test. The method not only simplifies the task of fault injection based testing but also separates the task of system testing from system development. Details of how various faults can be injected were described in this chapter.

Due to the inherent non-determinism in message transmission delays in distributed systems, the arrival order of a late message (resulting from the injection of a late timing fault) relative to other messages at a correct receiving processor may be arbitrary, though the injected late timing fault is itself well defined. The testing of a target software system may well require specific arrival orders of erroneous messages at a correct processor. This problem can be addressed by inserting fault injection objects in the software running on the correct processor.

In this chapter we also discussed the different architectural levels of distributed systems at which various software-implemented fault tolerance mechanisms can be applied to achieve system reliability. We mentioned briefly how at each of the three levels focused fault injection can be applied for fault tolerance testing. In chapter 4 and chapter 5, we describe in great detail how we have applied focused fault injection technique at node level. Here a node (a collection of processors) is treated as a distributed system. In chapter 6 we describe how our approach can also be applied at higher levels.

Chapter 4: Focused Fault Injection on Voltan TMR Node

4.1. Introduction

Reliable nodes capable of tolerating individual processor failures can be constructed by adopting replicated processing on distinct processors, whereby outputs from faulty processors can be prevented from appearing at application level. This is achieved by voting the outputs produced by the processors. Processors of a reliable node need to execute special protocols to carry out replicated processing to achieve node level fault tolerance. Such a reliable node is commonly known as N-Modular Redundant (NMR) node; it is capable of tolerating up to m individual processor failures, where N=2m+1. When the degree of replication is three, it is called Triple Modular Redundant (TMR) node. A TMR node can tolerate the failure of a single constituent processor.

We have implemented a family of fault tolerant nodes called Voltan [Shriv92, Speir93]. One of the members of the Voltan family is a three-processor TMR node capable of masking the failure of one processor. The special protocols employed for replicated processing on Voltan TMR nodes are all implemented in software while only standard off-the-shelf processors are used in the construction of Voltan TMR nodes. Since the processors of a Voltan TMR node communicate with one another only through message exchanges using fault tolerant protocols, the node is effectively a distributed system on its own.

Our objective in focused fault injection experiments on a Voltan TMR node is to test the soundness of the implementation of the protocols used in Voltan TMR nodes and to demonstrate how focused fault injection can be easily applied to a practical target software system [Tao95a]. These protocols are themselves quite well known, but their implementation is no trivial task. It should be emphasised that, since the Voltan TMR node is implemented entirely in software, its correct implementation relies on the correctness of its system software. We will test the major modules of the system software of the node through fault injection.

We will first briefly introduce the architecture and implementation of Voltan TMR nodes, because knowing the target software structure is essential for focused fault injection. Then the implementation of focused fault injection in a Voltan TMR node is explained in detail. Finally we present the experiments and the results obtained.

4.2. Voltan TMR Node Architecture

A Voltan TMR node is constructed out of three interconnected conventional processors on which application level processes are replicated to achieve fault tolerance. By voting the outputs from the individual processors of the node, erroneous output from a faulty processor can be prevented from appearing at application level, and so providing fault tolerance. The basic idea behind replicated processing is conceptually simple: a node is built out of a number of processors which execute special protocols to carry out replicated processing of computations to achieve fault tolerance. The three-processor TMR node is capable of tolerating the failure of a single processor by masking the faulty

processor's output. Such reliable TMR nodes can be used as building blocks for constructing fault tolerant distributed systems [Ezhil89].

4.2.1. System Model and Assumptions

It is assumed that a computation consists of a number of processes residing potentially on a number of processors and the processes of the computation communicate with one another through message exchanges. As an example, the function of a typical 'server' process is to pick up an input message from one of its input ports, process it and if required, output one or more messages on its output ports. It is also assumed that if a process with multiple input ports has input message pending on those ports then any of these messages is chosen *non-deterministically* for processing. Message selection is however assumed to be fair, that is, the process will eventually select a message present on a port. Here is such a process that picks up a pending message, processes the message, and sends a result message (output message):

```
process S:   /* a typical server process */
  cycle
     receive(msg);
     process msg;
     send(result_msg);
  end
end S
```

The model presented here is based on the well known state machine model (where a state machine is a process) for which the precise requirements for supporting replicated processing are known [Schne90]. Basically, in the replicated version of a process, multiple input ports of the non-replicated process are merged into a single port and the replica selects the message at the head of its port queue for processing. It is also necessary to assume that the computation performed by a process on a selected message is *deterministic*. This assumption is fundamental to active replication.

Given such a model of computation, replication of a process (with a replica, one each running on the underlying processors of a node) will require the following two conditions to be met:

1. *Agreement*: all the non-faulty replicas of a process receive identical input messages.

2. *Order*: all the non-faulty replicas process the messages in an identical order.

So, if all the non-faulty replicas of a process of a node have identical initial states then identical output messages in an identical order will be produced by them. This is the underlying principle of active replication [Schne90].

Practical distributed programs often require additional functionality such as the use of time-outs when waiting for messages. Time-outs (and other asynchronous events), high priority messages etc. are potential sources of non-determinism during input message selection, making such programs difficult to replicate. However, it is possible to transform some of these non-deterministic programs into deterministic ones [Tully90, Shriv92].

It will be assumed throughout that a message can be signed by its originator such that any modification of the message can be detected by a non-faulty receiver through a process of authentication. The implementation of such a mechanism in Voltan nodes is discussed in next section.

4.2.2. Node Software Architecture

The TMR node has the following two properties: 1) it functions correctly as long as there is no more than one processor failure; and 2) any spurious messages emitted by the failed processor of a correctly functioning node can be detected and rejected by all the correctly functioning receiver nodes.

As stated early, it is necessary that the replicas of computational processes on non-faulty processors within a node select identical messages for processing, to ensure that they produce identical outputs. This can be guaranteed by presenting a single input queue, referred to as a *delivered message queue* (DMQ), to a process and ensuring that a process picks up the message at the head of its DMQ for processing. An atomic broadcast protocol, designed to tolerate Byzantine failures, meeting both the agreement and order property is then used to ensure that identical messages are enqueued in an identical order at all the non-faulty replicas of a node. The broadcast mechanism itself requires that the clocks of all non-faulty processors of a node be synchronised such that the measurable difference between readings of the clocks at any instant is bounded by a known constant. The application output messages are voted to prevent erroneous messages from appearing at the application level.

The Voltan system software running on each processor of a TMR node has three major fault tolerant software modules: voting module, clock synchronisation module, and ordering module. These modules are supported by some communications software.

1. Voting Module: the voting module is responsible for voting the messages produced by the application and thus preventing erroneous output from appearing at the application level. This module consists of two processes.

2. Clock synchronisation module: the clock synchronisation module maintains the local clocks on the non-faulty processors of the node synchronised such that at any instant of time the difference between the local clock readings of any two non-faulty processors is within a certain bound. This service is required by the ordering module. This module consists of two processes.

3. Ordering Module: the ordering module orders messages by atomically broadcasting authentic messages received to all the order modules of that node (including itself). This permits order modules to construct identical queues of authentic messages (DMQs) for application processes. The ordering module requires the clocks on the non-faulty processors be synchronised. The ordering module consists of four processes which will be explained in detail in next section.

The communications software provides services for both inter-node and intra-node communications. The communications software is quite different from the three major modules of Voltan software in the sense it is conventional non-fault tolerant software.

While the three major modules are fault tolerant software; they are required to deliver a specified service in the presence of faults.

4.3. Implementation

Fig 4.1 shows the node hardware organisation of the present implementation which uses T800 transputers [Inmos88]. A transputer has four communication links. We use two of them for intra-node communication and the other two for inter-node communication. The TMR node masks the failure of one component which may be a processor and/or its links. Since a link failure can be seen as the failure of the processor associated with the link, we will only be concerned with processor failures. A link failure that prevents a message sent from a processor to be received by its neighbour in the node will be considered as a failure of the sender processor.

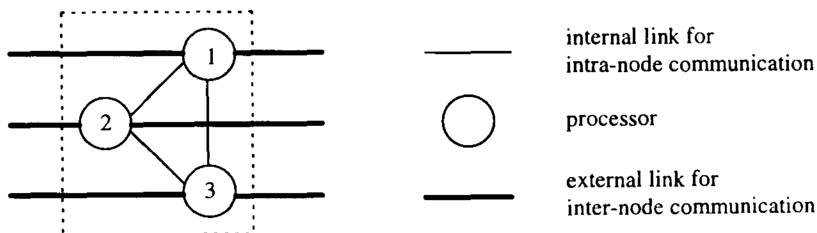

Fig. 4.1 Voltan TMR Node Hardware Organisation

As described in the last section, the system software of Voltan TMR node consists of three major modules and some communications software. A copy of the system software runs on each processor of a node. Fig. 4.2 shows how the three major modules of Voltan system software relate to a given application process replica S.

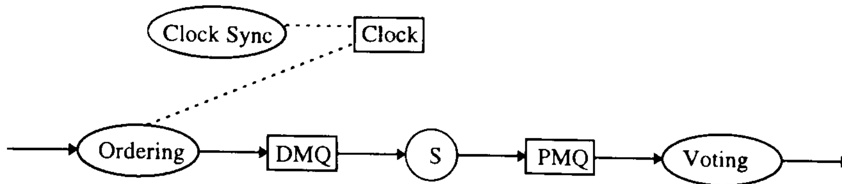

Fig. 4.2. Voltan TMR Node Software Organisation

The application process replica S has access to two message queues: *delivered message queue* (DMQ) and *processed message queue* (PMQ). When messages destined for the application process S arrive at a processor, they are ordered by the ordering module. The ordering module makes use of the local clock (Clock) which is kept in synchronisation with clocks on other correct processors by the clock synchronisation module. The ordered messages are made available to S via the DMQ. When the application process generates an output message, it is deposited in the PMQ. These deposited messages are voted by the voting module before being sent to their destinations.

The three modules of Voltan software all require the use of a message authentication mechanism - both for creating digital signatures and authenticating them. A message authentication mechanism allows a message to be signed and the signature of a received message to be verified. As a result, any alteration to a signed message can be detected by a recipient. The simplest form of digital signature is a checksum; checksums are adequate if it can be assumed that a processor would not deliberately forge signatures. More sophisticated forms of digital signature could be developed based on the techniques proposed in [Rives78]. In Voltan TMR software, a simple checksum based authentication mechanism is implemented.

4.3.1. Voting Module

The voting module (Fig. 4.3) consists of two processes: diffuser process and voter process. The diffuser picks up a message from the PMQ, signs the message, and puts a copy of it in the IMQ (internal message queue) and sends one copy to each of its two neighbouring processors. Each message contains a sequence number assigned to it by the application process. The sequence numbers are unique to each application process. Non-faulty replicas of a given application process will assign identical sequence numbers to message replicas. At the neighbouring processor, the authenticity of the incoming signed message is verified; if found authentic, the message is deposited at the local EMQ (external message queue).

The job of the voter is to vote the matching messages in the IMQ and EMQ. Messages from IMQ and EMQ are matched by using their sequence numbers. The voted message from the EMQ is counter-signed (the local processor signature is added to the original signature). Such a double-signed message is then sent to its destination node. At a destination node, only double-signed and authentic messages will be accepted for processing (such messages will be termed *valid*).

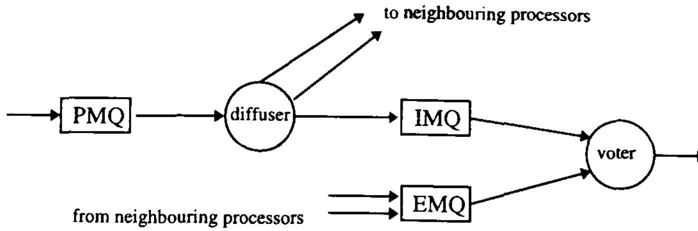

Fig. 4.3. The Voting Module of a Processor

4.3.2. Clock Synchronisation Module

The ordering protocol used in Voltan TMR node assumes that the clocks of the non-faulty processors of the node are synchronised, so that the difference of the readings of clocks at any instant is bounded by a known constant ε. The clock synchronisation module is implemented using the protocol by Halpern et al [Halpe84]. We first briefly describe the clock synchronisation protocol and then details our implementation of the protocol for Voltan TMR node.

4.3.2.1. The Protocol

The clock synchronisation protocol [Halpe84] makes the following assumptions.

(1) A correct clock's rate of drift from the real time is bounded by a known constant $\rho > 0$. Formally, between real time u and v the clock's readings C(u) and C(v) satisfies the condition: $(1+\rho)^{-1}(v-u) < C(v)-C(u) < (1+\rho)(v-u)$. Based on this assumption, it follows that the relative drift rate between two correct clocks will be bounded by dr, $dr=\rho(2+\rho)/(1+\rho)$.

(2) Processors are connected by a point-to-point network and faults do not cause network partition.

(3) A message sent between two adjacent correct processors is delivered within δ time units.

(3) Messages can be signed and subsequently authenticated. Any modification of a message while the message is being relayed can be detected by the destination processor.

Clock synchronisation is modelled by starting a new clock. After the kth round of synchronisation, a processor has clock C^k running as its *current* clock, or, put it another way, the current clock C is C^k. The beginnings (*beg*) and ends (*end*) of a synchronisation round is defined as follows: beg^k is the (real) time that the first correct processor starts its kth clock; end^k is the (real) time the last correct processor starts its kth clock. Between k^{th} and $k+1^{st}$ synchronisations, a processor will consider C^k its current clock.

The clock synchronisation protocol maintains the following three properties for all correct processors Pi and Pj in the presence of arbitrary failures. In the following expressions, C_i^k and C_j^k are the kth clocks of processors Pi and Pj respectively.

(1) There is an upper bound on the difference between correct processors' k^{th} clocks. More precisely, there is a constant DMAX such that

$\forall t \in [end^k, end^{k+1}]$,

$|C_i^k(t) - C_j^k(t)| < DMAX$

(2) If k>1, then the time the kth clock of Pi reads is no less than that of C_i^{k-1} (i.e., clocks are never set back) and can differ from C_i^{k-1} by at most a bounded amount. Formally, there is a small constant ADJ and a time $t \in [beg^k, end^k]$ such that C_i^k is started at t and if k>1

$0 \leq C_i^k(t) - C_i^{k-1}(t) < ADJ$

(3) The length of a synchronisation round is small, that is, there exists a small constant dmin such that

$0 \leq end^k - beg^k \leq dmin$

The exact values for DMAX, ADJ, and dmin are to be discussed later in this sub-section after the presentation of the protocol itself.

Now we describe the protocol. The protocol consists of two tasks (TIME_MONITOR and MESSAGE_MANAGER) which run continuously on each correct processor. The clock of a processor can be synchronised by either of the two tasks. There are two parameters of the protocol: PER and D. PER is the time between synchronisations, while D is an upper bound on the difference between correct clocks. The selection of values for PER and D needs to satisfy certain conditions which we will discuss later in this sub-section. There are three global variables shared between the two tasks: ET (the expected time of the next synchronisation), CURRENT (the current clock being used, e.g., if CURRENT=5 then the current clock C is C^5), and C (the clock).

When a processor is started, ET=PER, CURRENT=0, and $C^0=0$. It is also assumed that all processors in the network are started within dmin of each other. The two tasks are presented below with comments.

```
task TIME_MONITOR
var m: message;
global var C, ET: time; CURRENT: integer;
cycle
    if (C==ET) then {                // it is time to synchronise
        m="The time is ET";          // generate synchronisation message
        sign(m);                     // sign the synchronisation message
        send_on_all_links(m);        // send it to all neighbours
        CURRENT=CURRENT+1;           // update to the next clock
        C=ET;
        ET=ET+PER;                   // set the next synchronisation time
    }
endcycle
endtask

task MESSAGE_MANAGER
var m: message; s: integer; T: time;
global var C, ET: time; CURRENT: integer;
cycle
    receive(m);                      // receive a synchronisation message
    if (m is authentic saying "The time is T") then {
        s=no_of_signatures(m);       // extract the number of signatures of m
        if ((T==ET) and (ET-sD)<C)) then {  // it is the expected
                                     // synchronisation and it is not too early
            sign(m);                 // add the processor's own signature
            send_on_all_links(m);    // send it to all neighbours
            CURRENT=CURRENT+1;  // update to the next clock
            C=ET;
            ET=ET+PER;               // set the next synchronisation time
```

```
        }
    }
endcycle
endtask
```

The clock synchronisation protocol maintains three properties as stated earlier in this sub-section. The exact values of the first two bounds (DMAX and ADJ) depend on the protocol parameters (PER and D) and the third bound (dmin). The selection of values for PER and D needs to satisfy certain constraints of which dmin is a factor. Thus the value of dmin needs to be determined first.

For a fully and directly connected TMR node system with at most one processor failure, dmin is easily determined: dmin=δ, where δ is the upper bound of message transmission time between two adjacent correct processors.

The selection of the values for the two protocol parameters needs to satisfy the following two conditions, where f is the number of faulty processors:

$$(1+\rho)dmin + dr(1+\rho)PER \leq D$$

$$PER > (1+\rho)dmin + fD$$

For a TMR node hardware configuration in which $\rho=10^{-6}$ and $\delta=5$ ms (milliseconds), the conditions are very easily satisfied. For example, PER=10 s (seconds) and D=5.1 ms, will satisfy the conditions.

DMAX and ADJ are worked out using the following formulae:

$DMAX=(1+\rho)dmin + dr(1+\rho)PER$

$ADJ= (f+1)D$, where $f=1$ for a TMR node.

Thus for the hardware configuration mentioned above and the protocol parameter values subsequently selected, we have DMAX=5.02 ms, and ADJ=10.2 ms.

In this sub-section we only presented the bounds achieved by the protocol for a TMR node, with an example hardware configuration, more detailed analysis and formal proofs of the protocol can be found in [Halpe84].

4.3.2.2. The Implementation

In our implementation of the protocol, there are two processes (see Fig. 4.4): TM and MSG. They implement the TIME_MONITOR task and MESSAGE_MANAGER task of the protocol respectively.

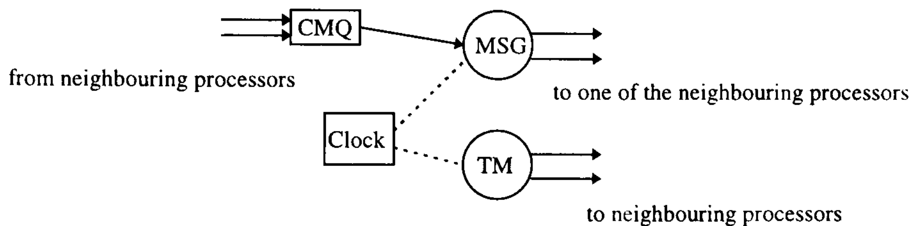

Fig. 4.4. The Clock Synchronisation Module of a Processor

Assuming the clocks of the three processors of a node are initialized correctly, here we briefly describe the functionality of the two processes of the clock synchronisation module as implemented for Voltan TMR node.

The TM process sleeps until the expected synchronisation time has come. It will first check whether the clock has already been synchronised by MSG. If the clock has been synchronised, TM will do nothing and go to sleep till the next expected synchronisation time. If the clock has not been synchronised, TM will broadcast a signed clock synchronisation message to other processors saying "It is time to start k^{th} clock", start the new clock and set the next expected synchronisation time on the local processor, and go to sleep again.

When an authentic clock synchronisation message arrives from a neighbouring processor, the MSG process picks it up from the CMQ (clock message queue). MSG will check whether the clock number (k) carried by the message matches the number it expects and whether the message arrives within the acceptable time frame (for details see the protocol description in sub-section 4.3.2.1). If one of the conditions is not satisfied, MSG will do nothing and wait for the next message. If both conditions are satisfied, MSG will relay the clock synchronisation message (with its own signature added) to the other processor saying "It is time to start k^{th} clock", start the new clock and set the next expected synchronisation time on the local processor, and wait for the next message.

4.3.3. Ordering Module

The ordering module employs the atomic broadcast protocol of [Crist85] adapted for a fully and directly connected three-processor system. It ensures that all non-faulty replicas of an application process receive identical input messages in an identical order.

We first briefly describe the atomic broadcast protocol and then details our implementation of the protocol for Voltan TMR node.

4.3.3.1. The Protocol

The atomic broadcast protocol developed by Cristian *et al* [Crist85] exhibits the following properties in the presence of arbitrary failures:

(1) *Termination*: It delivers every message broadcast by a correct sender to all correct receivers within some known time bound.

(2) *Atomicity*: It ensures that every message whose broadcast is initiated by a sender is either delivered to all correct receivers or to none of them.

(3) *Order*: It guarantees that all delivered messages from all senders are delivered in the same order at all receiving processors.

In order for the protocol to work, the following assumptions are made:

(1) Processors are connected by a point-to-point network.

(2) Faults do not cause network partition.

(3) A message sent between two adjacent correct processors is delivered within δ time units.

(4) The clocks of the correct processors are synchronised to within ε time units.

(5) No correct processor issues the same timestamp twice.

(6) Messages can be signed and subsequently authenticated.

Now we describe the atomic broadcast protocol as adapted for a fully and directly connected three-processor TMR node system, where there is at most one processor failure.

The protocol is based on the following two basic observations. First, to achieve the order property it is sufficient that in every correct processor messages be delivered in the order of their generation times (timestamps), and that messages generated at the same clock time be delivered in increasing order of their sender's identifier. Second, to ensure that any message broadcast at clock time t by some processor s and received by one correct processor p is also received by the other correct processor q, a timeliness check must be carried out when a message is received. If a message is received directly from the original sender then the message will be accepted and relayed to the other processor only if the current clock reading of the local clock t_local satisfies the condition: $t-\varepsilon <$ $t_local < t+\delta+\varepsilon$. If a message is received indirectly through relay then the message will be accepted only if the following condition is satisfied: $t-\varepsilon < t_local < t+2(\delta+\varepsilon)$.

In this way, a message can spend at most $(\delta+\varepsilon)$ clock time units in transit before being accepted by a correct processor. From that moment, it needs at most δ time units to reach the other correct processor. So the message can be delivered at the local clock time $t+2(\delta+\varepsilon)$. The protocol terminates.

The description of the protocol is modelled on three tasks: START task, RELAY task, and END task. There is one global variable, pool, which is shared among the three tasks. The global variable pool holds accepted messages which will later be delivered to the target processes with duplicates discarded. The three tasks are presented below.

```
task START;
var m: message; t: time;
global var pool: message_pool;
const time_delivery=2(d+e);
cycle
      take_a_msg(m);                          // take a message
      t=read_clock();                         // read the local clock
      timestamp(m,t);                         // timestamp the message
      deposit(m, pool);                       // deposit the message in the
                                              // local message pool
      send_on_both_links(m);                  // send the message to the
                                              // other processors
      schedule(END, time_delivery+t);         // arrange for the message's
                                              // delivery
endcycle
endtask

task RELAY
var m: message; t: time;
global var pool: message_pool;
const time_delivery=2(d+e);
cycle
      receive_on_a_link(m);                   // receive a message
      if (authentic(m) and timely(m)) then {  // check for authenticity and
                                              // timeliness
            deposit(m, pool);                 // deposit the message in the
                                              // local message pool
```

```
        if (from_sender(m)) then                // relay the message if it is from
            send_on_other_link(m);              // the original sender
        t=timestamp_of_msg(m);                  // get the timestamp of m
        schedule(END, time_delivery+t);         // arrange for the message's
                                                // delivery
    }
    else discard(m);
endcycle
endtask

task END(t:time)
var m: message;
global var pool: message_pool;
while (messages_ready_to_deliver(pool, t)>0) {
    take_a_ready_msg(m, t);                     // take a ready message
    if (duplicate(m)) then discard(m);          // discard it if it's a duplicate
    else deliver(m);                            // deliver it to the target process
}
endtask
```

The main difference between the protocol presented in this sub-section and the original protocol of [Crist85] is that messages are not signed when they are relayed. In the original protocol, the signatures are used to count the hops a message has been through in order to determine the timeliness of a message. In a fully and directly connected TMR node system, there is no need for this. If a message is received from the original sender then it has been through one hop; if a message is received through relay then it has been through two hops.

4.3.3.2. The Implementation

In our implementation of the protocol, there are four processes (Fig. 4.5): broadcaster, relayer, transferrer, and deliverer. The protocol task START is mapped to broadcaster, RELAY to relayer and transferrer, and END to deliverer. By splitting RELAY into two processes, each process will handle only one type of message. The synchronised clock service required by the protocol is provided by the clock synchronisation module described in section 4.2.

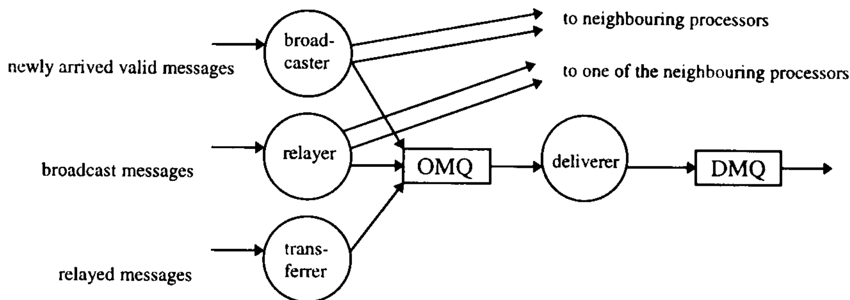

Fig. 4.5. The Ordering Module of a Processor

When a valid (i.e., double-signed and authentic) message is received at a processor, the broadcaster appends the message with the current reading of the local clock as message timestamp, signs the message (this third signature is needed because a timestamp has been added to the message), broadcasts the message to its two neighbouring processors, and also inserts a copy of it in the local OMQ (ordered message queue) where messages are queued in increasing timestamp order.

When a broadcast message arrives at a processor, the relayer will receive it. Note that the message received by the relayer will have three signatures and would have been received from the processor that is the creator of the third signature. The relayer verifies the authenticity and timeliness of the received message (as specified by the atomic broadcast protocol). If the message is authentic and timely, it is relayed to the other (non-signatory) processor and a copy of it is inserted in the local OMQ.

The transferrer process picks up relayed messages, and inserts them in the local OMQ if the messages are found to be valid and timely. The message picked up by the transferrer will also have three signatures, but would have been received directly from the processor who is not the owner of the third signature. This simple way of distinguishing the broadcast messages from the relayed messages eliminates the need (as required by [Crist85]) to sign a message by the relayer.

The deliverer process will be checking the messages in the OMQ regularly to see whether a message has become *stable*, that is whether the delivery time of the message as specified by its timestamp has come. The deliverer process moves stable messages to the DMQ for consumption by the application process. The deliverer process queues messages in the DMQ in increasing timestamp order, while duplicates are discarded.

4.3.4. Communications Layer

The communications layer of the Voltan software contains four software modules for supporting both intra-node and inter-node communications. The intra-node messages are transmitted over raw transputer links. The internal links (see Fig. 4.1) between

constituent processors of a TMR node are configured in such a way that they can be accessed directly by Voltan software. This allows fast intra-node communication. The inter-node messages are transmitted through the use of a message passing service provided by the Helios operating system [Perih89] running on top of each transputer. The external links (see Fig. 4.1) of a transputer are used by the Helios operating system to provide basic operating system services (including the message passing service).

The two modules for intra-node communications are RX and TX. For each internal link of a transputer, there is a pair of RX process and TX process. The RX process listens on the link. When a message arrives through the link the RX receives the message and deposits it in the message queue associated with its destination process. The TX process waits on the message queue associated with the link. When the queue is not empty the TX process picks up a message at the head of the queue and sends it down the link.

The two modules for inter-node communications are Receive and Send. For each processor (transputer) of a TMR node, a pair of Receive and Send are employed to receive messages from other nodes and to send messages to other nodes respectively. The Receive process and Send process work in a way similar to that of RX and TX. Send and Receive are implemented on top of the message passing service supported by the Helios operating system.

4.4. Fault Injection Implementation

The Voltan software has been implemented on top of the Helios operating system [Perih89] which runs on each transputer to provide essential operating system services.

All of the Voltan software is written in C++ [Lippm89], as are the fault injection objects. Each Voltan system service is provided by a system module consisting of one or more processes as described in the previous section.

Messages are instances of a class called Message_Block. Queues are instances of a class called Message_Block_Queue. These passive (data) objects are used for communications between the active objects which represent processes. Active objects are also instances of C++ classes.

The object-oriented implementation of the Voltan TMR node software makes it quite easy and convenient to implement focused fault injection. The overall Voltan software system with an application process has the following form:

```
 .........
/* passive objects for communications between active objects */
     Message_Block_Queue  mo0, mo1, vmp, rmp, omq, dmq, *mp[4], pmq, imq;
/* active objects for link handling */
     RX  rx1(0, mp), rx2(1, mp);
     TX  tx1(0, &mo0), tx2(1, &mo1);
/* passive object representing synchronised clock */
     Clock clock();
/* active objects implementing clock synchronisation algorithm */
     Time_Monitor tm(&clock, &mo0, &mo1);
     Message_Manager msg(&clock, &mo0, &mo1, mp[0]);
/* active objects implementing ordering module */
     Broadcaster broadcaster(&rmp, &omq, &mo0, &mo1, &clock);
     Relayer relayer(mp[2], &omq, &mo0, &mo1, &clock);
     Transferrer transferrer(mp[3], &omq, &clock);
     Deliverer deliverer(&omq, &dmq, &clock);
/* active objects implementing voting module */
```

```
    Diffuser diffuser(&pmq,&imq,&mo0,&mo1);
    Voter voter(&imq, mp[1], &vmp);
/* active objects for inter-node message communications */
    Receive receive(&rmq);
    Send send(&vmq);
/* active objects representing application processes */
    Application application(&dmq, &pmq);
    . . . . . . . . . .
```

With a particular fault injection object inserted, the above program would change slightly to the following form:

```
      . . . . . . . . . .
    Message_Block_Queue  mo0, mo1, vmp, rmp, omq, dmq, *mp[4], pmq, imq, fq;
    RX  rx1(0, mp), RX  rx2(1, mp);
    TX  tx1(0, &mo0), tx2(1, &mo1);
    Clock clock();
    Time_Monitor tm(&clock, &mo0, &mo1);
    Message_Manager msg(&clock, &mo0, &mo1, mp[0]);
    Broadcaster broadcaster(&rmp, &omq, &mo0, &mo1, &clock);
    Relayer relayer(mp[2], &omq, &mo0, &mo1, &clock);
    Transferrer transferrer(mp[3], &omq, &clock);
    Deliverer deliverer(&omq, &dmq, &clock);
    Receive receive(&rmq);
    Send send(&vmq);
/* one of the parameters of the following object is changed */
    Diffuser diffuser(&pmq,&imq,&fq,&mo1);
/* fault injection object */
    Fault_Object fo(&fq, &mo0);
    Voter voter(&imq, mp[1], &vmp);
    Application application(&dmq, &pmq);
    . . . . . . . . . .
```

When this program is run on a selected 'faulty' processor, it will be injecting faults in those output messages which are diffused by the Diffuser and sent to one of the neighbouring processors. This creates the failure scenario in which the 'faulty' processor sends erroneous output messages to one of its neighbouring processors for voting. Thus the voting module running on the processor which receives the erroneous messages is tested for its effectiveness in masking the failure of the injected processor.

Note that there are only two small differences between the original version of the program and the one with the fault injection object: 1) the application object is defined (started) with a different parameter; 2) an extra active object fo (of object class Fault_Object) and a queue it uses are added to the system. The Voltan system software modules do not need to be changed. Thus, the efforts involved in fault injection experiments are kept to a minimum.

4.5. Experiments and Results

According to the design, a Voltan TMR node should continue to function correctly even if one of its three constituent processors has failed. The delivery of this correctness property relies on the correctness of the system software of the node, since the TMR node is implemented entirely in software and only standard off-the-shelf hardware is used. As described earlier in the chapter, there are three fault tolerant modules in the system software. These modules are required to perform certain functions in the presence of faults, so they are subject to fault injection based testing.

Before starting fault injection experiments, we had tested the system software without fault injection and it worked correctly. We assume that the checksum based message authentication mechanism has been implemented correctly. The message authentication service was not subject to fault injection testing.

Our experiments concentrate on injecting faults to test the three fault tolerant modules, namely voting, clock synchronisation, and ordering modules. In particular, we wish to ascertain that a single processor failure does not cause the node to fail, even if the faulty processor behaves in a 'two-faced' manner [Lampo82].

Faults are injected into the software of one of the three TMR node processors and the behaviour of the modules under test are observed in various ways. How faults are injected on the selected processor depends on which fault tolerant software module is being tested and the nature of that module.

4.5.1. Voting Module

In the experiments, the three replicas of the server (S) running on a TMR node provide a reliable service, with clients (C1, C2) running on a separate processor sending requests and receiving replies. The system configuration is shown in Fig. 4.6.

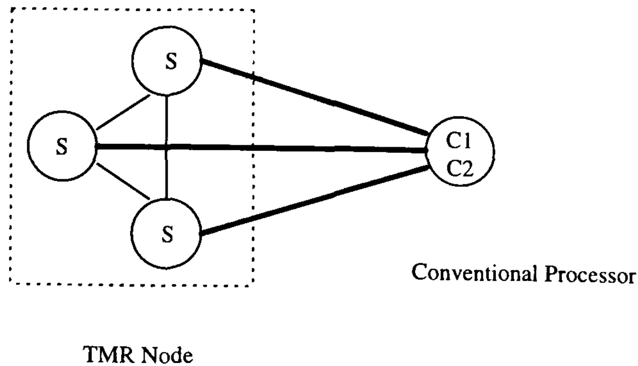

Fig. 4.6. System Configuration for Testing Voting Module

The application server S running on the TMR node provides a positioning service. It holds two sets of co-ordinates for two graphical objects. Each client manoeuvres an object; for this purpose, it needs the positioning service provided by the server S. A client sends a request to the server giving its identity and the next position number. The corresponding reply from the server will contain the co-ordinates for the next position.

To test the voting module, we injected faults to emulate the behaviour of a faulty processor generating erroneous output messages . The correct functioning of the voting module can be observed by the clients from the fact that double-signed and authentic reply messages are still being sent by the TMR node despite the 'failure' of one processor.

The Voting module is a relatively simple module, it consists of two active objects (see Fig. 4.3). However, even such a simple module has been known to contain software bugs [Yang85]. We inserted two injection objects (FO_1 and FO_2) each with its own message queue (FQ_1 or FQ_2) between the diffuser object and the link handling objects

in the software of one processor (see Fig. 4.7). The link handling objects which actually send the messages down the links are not shown in the figure.

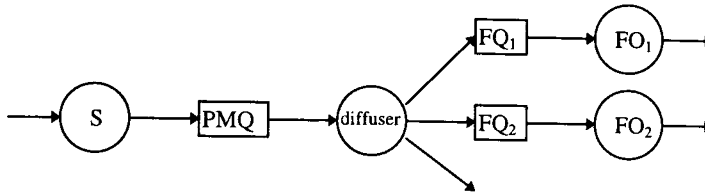

Fig. 4.7. Fault Injection in Diffuser

This created the effects of a faulty processor producing incorrect output (reply) messages. It is the job of the voting modules on the other two correct processors to weed out wrong reply messages and so masking the failure of one processor.

The following classes of faults were injected by the two fault injection objects in our experiments.

Omission Faults: In the experiment, we first injected consistent omission faults by having the two injection objects both delete messages. This simulates a faulty processor which is not producing any message for voting. Despite the silence of the faulty processor, the other two correct processors could still vote and manage to send double-signed reply messages to the clients. We then generalised the case whereby the processor sometimes appeared silent to just one of the remaining two processors. No bugs were discovered.

Value Faults: In the experiment, value faults were injected by replacing one byte of application data with a randomly generated byte or by replacing the sequence number of

the message with a random number. A new signature was also generated to replace the one on the intercepted message, otherwise the injected fault will be easily picked up by the message authentication mechanism. The two injection objects operated independently of each other. This creates the effects that the processor concerned is sending messages with wrong contents and correct signatures. The voting modules on the other two processors (where byte-by-byte comparison is performed) successfully detected and discarded all incorrect messages from the injected processor.

During this experiment, a software bug regarding the data structure of a message was discovered. It was not in the voting module, but in the passive object class Message_Block. This was not expected, so shows the value of fault injection based testing.

Timing Faults: Timing faults of a single failed processor should also not affect voting at the voters of the correct processors. This was the case when we injected late timing faults at the selected processor. Random and independent delays were injected by the two fault injection objects.

Arbitrary Faults: Arbitrary faults which cause the processor to violate expected behaviour in both timing and value domains were injected in the experiment. These faults were introduced by having the fault injection objects injecting both timing and value faults. The intercepted messages were first delayed and then the message values were modified, by the same fault injection objects. No bugs were discovered.

4.5.2. Clock Synchronisation Module

A precise testing of any clock synchronisation module is impossible unless special hardware support, such as the one used in [Palum94], is available for correctly measuring clock differences. The impossibility arises from the fact that a processor cannot 'instantly' read another processor's clock to check whether the clock difference at a given instant of time is within the bound ε. The error or imprecision involved in reading a remote clock is influenced by variation in message transmission and processing delays. The special hardware support of [Palum94] provides each processor with access to a global reference clock. With such a facility, a processor can then indicate to another processor its own time with reference to this globally accessible time base. This enables processors to compute accurately their relative differences at a given instant of the reference time.

In our testing of the clock synchronisation module, no special hardware is used. We however circumvent the impossibility of instant access by exploring the minimum requirement imposed by the ordering module on the clock synchronisation module. This requirement (see below) is weaker than requiring that correct processors' clocks be synchronised within some known bound ε.

Thus the experiments reported here only check whether the clock synchronisation module provides what is required from it by the ordering module, rather than whether processor clocks are synchronised within ε. This is enough for our purpose which is to test the Voltan software in implementing failure masking strategies. We will first

describe the mechanism we have set up to measure the difference in clock readings of two processors.

This mechanism involves two processes (reader and checker), each running on a processor. The reader process on one processor reads its local clock and sends a message containing the clock reading to the checker process on the other processor. Upon receiving the message, the checker process reads its own clock and works out the difference by subtracting the clock reading contained in the message from the local clock reading. The actual message transmission and processing delay involved in taking a measurement varies and is bounded by the known constant δ.

Using this measurement mechanism, we will not test whether the actual difference between two clocks is within the bound ε, but will ascertain whether the measured clock difference, d, is within the range: $-\varepsilon < d < \varepsilon+\delta$. A careful analysis of the correctness reasoning in [Crist85] will indicate that the ordering protocol presented there will be correct so long as $-\varepsilon < d < \varepsilon+\delta$ holds; in fact any ordering protocol that assumes ε-synchronised clocks will only require $-\varepsilon < d < \varepsilon+\delta$. (Note that ε-synchronisation implies that $-\varepsilon < d < \varepsilon+\delta$ holds, but not vice versa.)

The testing of the clock synchronisation module does not require the running of an application. The experimental set-up only involves a TMR node. Let the three processors of a TMR node be designated as P1, P2, and P3. P3 is selected for fault injection while the clock differences between P1 and P2 are measured. We put the

reader process of the measurement mechanism on P2 and the checker process on P1 (see Fig. 4.8).

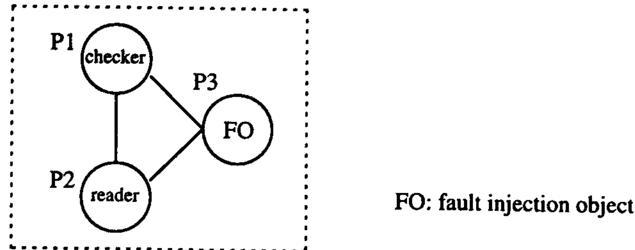

Fig. 4.8. Node Configuration for Testing Clock Synchronisation Module

It is assumed that there is no fault when the clocks of the processors are initialized. A simple non-fault tolerant program is used to initialise the clocks. Due to the way the clocks are initialized, we know that P1 is running ahead of P2 and P3.

Two kinds of clock difference measurements were taken during the experiments. One is when both processors are running the same clock (k^{th} clock or $k+1^{st}$ clock); the other is when one processor is running k^{th} clock while the other processor is already running $k+1^{st}$ clock. It is in the second scenario when one clock has been synchronised while the other has yet to be synchronised, the clock difference is potentially the largest (see the protocol description in sub-section 4.3.2.1).

As shown in Fig. 4.4, the clock synchronisation module consists of two active objects (TM and MSG), either one of them can synchronise the local clock and send synchronisation messages to other processors. We first fault-injected TM of the selected processor P3. Two fault injection objects were used as shown in Fig. 4.9.

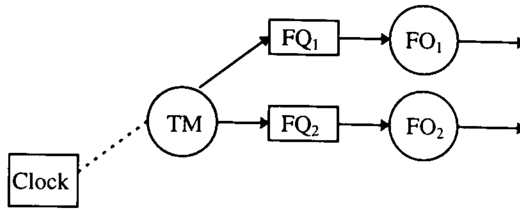

Fig. 4.9. Fault Injection in TM

Omission Faults: In the experiment, we first injected consistent omission faults by having the injection objects delete all clock synchronisation messages from TM. The measurements taken on P1 and P2 indicated the two non-faulty processors remained in synchronisation. We then generalised the case whereby TM appeared silent to just one of the two processors. No bugs were discovered.

Value Faults: In the experiment, value faults were injected by adding a random value to the synchronisation round number k carried by the messages. A new signature was also generated to replace the old one on the intercepted message. This creates the scenario where a faulty processor sends clock synchronisation messages with incorrect round numbers which should be rejected by non-faulty processors. The measurements taken on P1 and P2 indicated the two non-faulty processors remained in synchronisation.

Early Timing Faults: The injection of early timing faults requires an additional fault injection object. The two existing fault injection objects now delete messages as if omission faults were being injected. The third injection object (see Fig. 4.10) will generate and send a clock synchronisation message before the next round of synchronisation is due. The message is only sent to P1, the processor with a fast clock.

The aim here is to create a malicious failure scenario in which the faulty processor tries to push the correct processor with a fast clock even faster so as to cause a violation of the clock synchronisation bound.

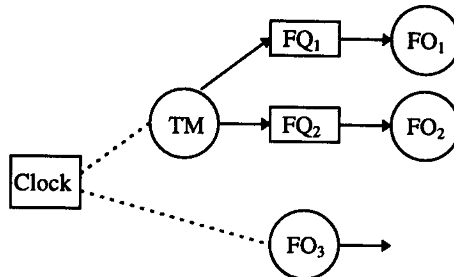

Fig. 4.10. Fault Injection in TM

The experimental results were quite interesting. The measurements taken on P1 and P2 showed that on four occasions the recorded difference of clock readings of the two correct processors exceeded the bound of $\varepsilon+\delta$, other figures were all within the bound. These violations happened during the first four rounds of fault injection, and when P2 was running k^{th} clock while P1 had started $k+1^{st}$ clock. The experiment was repeated several times and this phenomenon recurred. This indicated a bug in the program.

Our subsequent analysis of the source code revealed a subtle bug. The clock initialisation program makes use of the clock synchronisation program. To allow the synchronisation program to be used in this manner, the message timeliness check (which detects synchronisation messages that arrive too early) is disabled during initialisation period. It takes two rounds of synchronisation during the initialisation period to get the clocks initialized within the required initial bound; at each round a processor is expected

to receive *at most* two messages. The timeliness check at each processor is restored after receiving *exactly* four messages. This is incorrect because according to the clock synchronisation protocol, a processor with the fastest running clock does not receive any message from any other non-faulty processor. Due to this bug, P1 - the processor with fastest running clock - thought it was still in initialisation period and did not do timeliness check when the first four erroneous messages arrived, and allowed itself to be pushed exceeding the bound. This bug was later corrected.

Late Timing and Arbitrary Faults: These faults were also injected in TM using fault injection objects of FO_1 and FO_2 (Fig. 4.9). The measurements taken indicated no further bugs.

Having injected TM, we also fault-injected MSG of P3, using the following software structure (Fig. 4.11). During the experiments, the clocks of P1 and P2 remained synchronised.

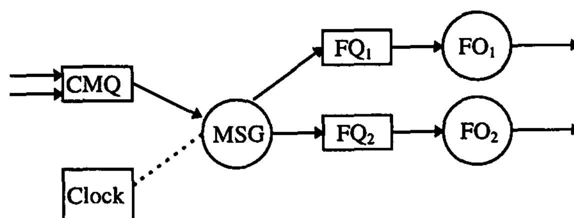

Fig. 4.11. Fault Injection in MSG

4.5.3. Ordering Module

With the voting module and clock synchronisation module tested, we then went on to the testing of the ordering module. The testing of the ordering module relies on the correct functioning of both the voting module and clock synchronisation module.

If the ordering module of a non-faulty processor does not work correctly and as a result replicas of the application process on non-faulty processors end up processing different input messages and producing different output messages, the voting module will be unable to form a majority. So the failure of the ordering module, manifested by the lack of double-signed and authentic reply messages, can be observed by the clients.

The experimental set-up required for testing the ordering module is similar to that used for testing voting module (see Fig. 4.7). The only difference is that the operations of the two clients C1 and C2 need to be carefully co-ordinated.

In order to put the ordering module through its paces, we would need a scenario like this: a processor receives C1's request followed by C2's request while another processor of the node receives C2's request followed by C1's request. To guarantee this scenario, a single process is used to simulate two clients sending independent requests.

Message ordering in the TMR node is achieved by the use of an atomic broadcast protocol [Crist85] as described earlier in this chapter. The protocol achieves the required ordering properties in two stages: 1) broadcast stage, 2) relay stage. A message sent to the TMR node would first be timestamped and broadcast by the processor to its two neighbours, and the other processors (in the second stage) would then relay the message

to each other. The idea behind all this is that every one of the non-faulty processors should receive identical messages while timestamps are used to achieve identical message ordering.

As we have seen, the ordering module handles two types of messages received from other processors: broadcast messages at the broadcast stage and relayed messages at the relay stage. In other words, a faulty processor could only produce two types of message to 'confuse' non-faulty processors. In the experiments we fault-injected the software of one processor so it produced erroneous broadcast and relayed messages.

We first fault-injected the broadcaster of the selected processor by inserting an injection object as shown in Fig 4.12. The injection object has two input and two output channels. The reason we used a single injection object instead of two separate ones is that we need to co-ordinate the injection to emulate a 'two-faced General' [Lampo82]. The effect of this fault injection is that erroneous broadcast messages will be generated by the processor selected for fault injection.

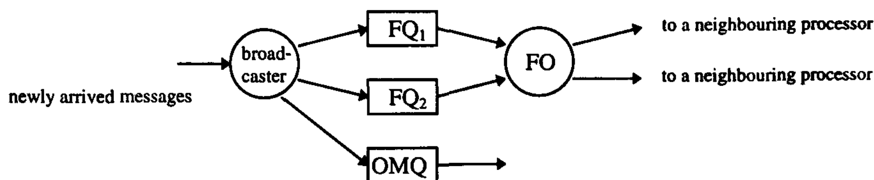

Fig. 4.12. Fault Injection in Broadcaster

When injecting value faults we injected faults at the timestamp field of the messages. This is because the timestamp is the only piece of data appended to the message when a

message is broadcast and it is the value of the timestamp that decides message order. Value faults injected in other parts of a message would be detected by the authentication mechanism which had been assumed to work correctly as stated before.

The experiment results are described below.

Omission Faults at the Broadcast Stage: The fault injection object deleted broadcast messages. For the case of consistent omission faults, no broadcast messages were sent to neighbouring processors; while for the case of inconsistent omission faults, only one of the two neighbouring processors received broadcast messages. The ordering module and the TMR node as a whole were observed to work correctly.

Value Faults at the Broadcast Stage: The injection object would add a random number to the timestamp of a message and generate a new signature for the message to replace the one generated by broadcaster. When we had the injection object add the same random number to the timestamps of the two messages (emulating consistent value fault), experimental results showed that the ordering modules of the two non-faulty processors worked as expected. However, when different random numbers were used by the injection object (emulating inconsistent value fault), creating the scenario of a 'two-faced General', identical message ordering at the server replicas running on the two non-faulty processors was not always achieved.

The cause of the problem was eventually traced to an incorrect optimisation of the broadcast protocol. This was later corrected.

Timing and Arbitrary Faults at the Broadcast Stage: These faults were also injected at the broadcaster, no further bugs were found.

Having injected broadcaster, we then injected relayer. This was also done by inserting a single injection object (see Fig. 4.13). We only injected omission faults and timing faults (as the authentication mechanism will catch corruption of messages, so there is no need to inject value faults). The node was observed to work correctly. No bugs were discovered.

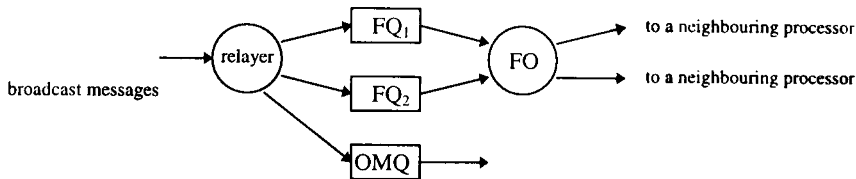

Fig. 4.13. Fault Injection in Relayer

Omission Faults at Relay Stage: Omission faults were introduced by the injection object. The reliable service provided by the TMR node was maintained despite these faults. No bug or unexpected behaviour was observed.

Timing Faults at Relay Stage: Late timing faults were introduced by the injection object. The two streams of messages were subject to random delays. The ordering module functioned correctly despite these faults.

4.5.4. Comments

The experiments described in this section have clearly demonstrated the value of the focused fault injection method. Adopting the method, we were able to inject specific classes of faults and create required failure scenarios quite easily. Only minimal changes are required of the main program of target software system, the modules which implement the various system functions do not even need to be re-compiled.

4.6. Summary

Fault injection based testing is a very useful way of uncovering faults in the implementation of a system. Even if the design has been validated adequately, faults can still be introduced at the implementation stage. In the case of the Voltan TMR node, the basic algorithms that form the core of the node (voting, clock synchronisation, and ordering protocols) are really quite well-known, but their implementation is not a trivial task.

In this chapter we described fault injection based testing of the Voltan TMR node. The Voltan TMR node is implemented entirely in software using only standard off-the-shelf hardware. Whether the node can satisfy the required failure-masking property depends on the correctness of its system software. We tested the three fault tolerant modules of the node software through focused fault injection.

The experiments we carried out on the Voltan TMR node are by no means exhaustive, in fact there can be many combinations and variations of the basic faults we injected. For example, faults can be injected simultaneously into the voting and ordering modules

and the behaviour of the node can be observed. However, the experiments performed by us does test the most essential parts of the node. This is so because the voting sub-system is independent, in that it does not require the services of the ordering or the clock synchronisation module, so can be tested by fault-injecting just within the voting module as described here. The ordering module depends on the services of the clock synchronisation module (but not vice versa), so it is essential to test the clock synchronisation module first, and having satisfied that it functions correctly, test the ordering module.

The experiments performed on the Voltan TMR node have demonstrated the usefulness of our fault injection method. It helped to uncover subtle implementation bugs that had remained undetected.

Chapter 5: Focused Fault Injection on A Fail-Silent Node

5.1. Introduction

Replicated processing on distinct processors whereby outputs from faulty processors can be prevented from appearing at the application level (by employing means such as comparing or voting the outputs produced by the processors), not only provides a practical means of constructing systems capable of masking individual processor failures (e.g., a TMR node) but also provides the basis for the construction of *fail-silent* nodes.

A fail-silent node of f+1 processors either works correctly or stops functioning (becomes silent) soon after an internal failure is detected. This behaviour of a node is guaranteed so long as no more than f processors of the node fail. A two-processor fail-silent node (f=1) provides fail-silence property in the presence of at most one processor failure. In this chapter we concentrate our discussions on software-implemented two-processor fail-silent nodes.

One of the members of the Voltan family of reliable nodes [Shriv92, Speir93] is a two-processor fail-silent node. This node is essentially a 'cut-down' version of the Voltan TMR node. It shares the basic system architecture with the TMR node and employs the same protocol to order input messages. Its comparison module also works in a way similar to the voting module of the TMR node.

Though this fail-silent node delivers the required fail-silence property, careful analysis indicates that the order protocol used in a fail-silent node does not need to be a fault tolerant one as in a failure-masking TMR node. This is because the order protocol is only required to deliver message order when both processors are correct. When there is a failure the fail-silence property is guaranteed by the comparison protocol. This led to the development of a leader-follower fail-silent node [Brasi94] which offers better performance than the original Voltan fail-silent node which uses a fault tolerant order protocol. In this chapter we describe the fault injection based testing of this leader-follower fail-silent node [Tao95b]. Like the TMR node described in chapter 4, the leader-follower fail-silent node is effectively a distributed system on its own. The two processors of the node communicate with each other through message exchanges.

This chapter is structured as follows. In section 5.2 we introduce the architecture of fail-silent node and discuss in particular a node design which is based on the leader-follower technique. Section 5.3 presents the current implementation of the leader-follower node. The implementation of focused fault injection in the node is explained in section 5.4. Section 5.5 presents the experiments and the results obtained. Section 5.6 summarises the chapter.

5.2. Fail-Silent Node Architecture

A software implemented two-processor fail-silent node is a self-checking node composed of two conventional 'fail-uncontrolled' processors that work together to provide the fail-silence property. Such a node achieves the abstraction of fail-silence in the following sense: a node produces either *valid* messages which can be verified as

such by destination nodes, or it ceases to produce new valid messages, in which case destination nodes can detect any messages it may produce as unwanted. A valid message is signed by both correct processors of the node and can be verified as such. The two processors of a software implemented fail-silent node, on which the application processes are replicated, need to execute message order and comparison protocols to 'keep in step' and check each other respectively.

5.2.1. System Model and Assumptions

Like the Voltan TMR node, a fail-silent node adopts active replication to achieve fail-controlled behaviour. The system model and assumptions described in section 4.2.1 of the last chapter also apply to fail-silent node. Here we only present a brief summary.

A distributed computation is assumed to be composed of a number of processes that interact only via message exchanges. The function of a process is to pick up the input message at the head of its only input queue, process it and, if necessary, output one or more messages. The computation performed by a process on an input message is assumed to be deterministic. Given such a computational model, if the non-faulty replicas of a process have identical initial states then identical output messages will be produced by them, provided the input queues of all correct replicas can be guaranteed to contain identical messages in an identical order. This requirement of "identical messages in an identical order" is satisfied by the ordering module of a fail-silent node.

We also assume the sender of a message is able to sign a message which can later be authenticated by the receiver of the message.

5.2.2. Basic Node Architecture

The basic software architecture of a fail-silent node consists of two modules: ordering module and comparison module. Fig. 5.1 shows how these two modules relate to a given application process replica S.

Fig. 5.1. Basic Node Architecture

The role of the ordering module is to guarantee that the application process replicas process identical input messages in an identical order. The input messages are ordered by the module and then delivered in the *delivered message queue* (DMQ) of the application process. The application process picks up an input message at the head of its DMQ, processes it and, if necessary, output one or more messages. The output messages are deposited in the *processed message queue* (PMQ). Each application process has its own DMQ while the PMQ is shared among all application processes running on the processor.

The comparison module compares locally produced messages with their counterparts produced by the neighbour processor. It takes output messages produced by the local application process replica, signs them and sends them to the neighbouring processor for comparison. It also receives signed application output messages from the neighbour

processor, authenticates them, and (if found authentic) compare them with the corresponding messages produced by the local application process replica.

If a message received from the neighbour processor fails authentication or does not match its locally produced counterpart, a failure is detected. Similarly, an absence of a message for comparison (after a node specific time-out interval) also indicates a failure. Once a failure is detected, the processor stops functioning and the node becomes silent. If a received message is found to be authentic and matches its locally produced counterpart, the received message is counter-signed (double-signed) and sent to its destination. A double-signed and authentic message is termed a *valid* message.

Depending on the design of the fail-silent node (either symmetric or asymmetric), the system software running on the two processors of the node may or may not be identical. In a symmetric design, an identical copy of the system software runs on each processor of the node. While in an asymmetric design (leader-follower node), the system software running on one processor is different from the one running on the other processor.

5.2.3. Node Failure Semantics

As stated earlier in the chapter, a fail-silent node either works correctly or stops functioning (becomes silent) *soon after* an internal failure is detected. Let us assume that an application process running on a correctly functioning fail-silent node takes at most t time units to compute the response (output message) to a given input message. If the output from the fail-silent node is produced later than t then the node is said to have suffered a performance failure. A fail-silent node can be in one of the three states:

(1) **Normal State**: In this state, a node produces correct outputs. Detection of an internal failure (by the comparison module) causes the node to irreversibly enter either the failing state or the silent state.

(2) **Failing State**: This is an intermediate state in which the node can suffer at most one performance failure. From this state the node enters the terminal silent state.

(3) **Silent State**: No new valid messages are produced by the node. Any messages produced by the node can only be invalid or copies of previously produced valid messages: any functioning destination node can detect these messages as unwanted. Here we assume the use of monotonically increasing sequence numbers for output messages so that any duplicates can be easily detected.

The relationship among the three states of a fail-silent node is illustrated in Fig. 5.2. The reason for the existence of the failing state is as follows. A faulty processor can contain a message from the correct processor sent for comparison (a message that was sent before the correct processor stopped). The faulty processor can output this as a double-signed valid message at any future time. The comparison module of each processor must therefore incorporate an intra-node message synchronisation mechanism to ensure that each processor of a node at any time contains no more than one message from the neighbour processor for comparison; in this way, the number of performance failures in the failing state is limited to at most one.

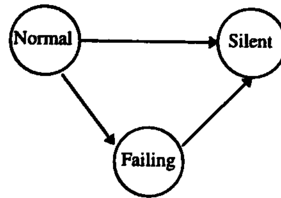

Fig. 5.2. Fail-Silent Node States

In the following two sub-sections, we first present a simple symmetric design which allows multiple performance failures in the failing state and then describes an asymmetric design which not only improves node response time when it functions correctly but also guarantees that the node suffers at most one performance failure in the failing state.

5.2.4. The Symmetric Node Design

This is essentially a 'cut-down' version of the three-processor TMR node. The ordering module of the node employs the same order protocol [Crist85] as the TMR node and much of the source code is shared. Like the TMR node, it requires the clocks of the node processors be synchronised. The comparison module of the node also works in a way similar to that of the voting module of the TMR node.

The ordering module consists of three processes: broadcaster, transferrer, and deliverer (see Fig. 5.3). If we compare this module with the ordering module of the TMR node, we will find the relayer process is missing. This is because there are only two processors in the fail-silent node and a broadcast message does not need to be

relayed. Otherwise it works the same way as the ordering module of the TMR node (see section 4.3).

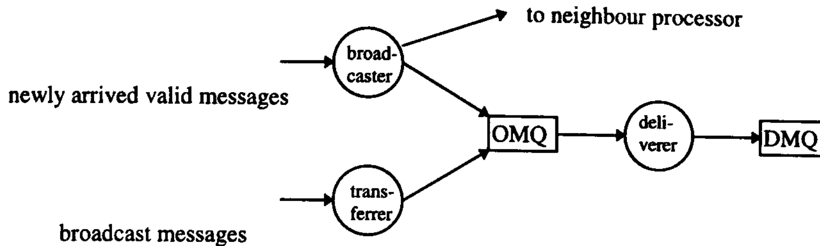

Fig. 5.3. The Ordering Module of a Processor

The comparison module consists of two processes: diffuser and comparator (see Fig. 5.4). The diffuser process picks up a message from the PMQ, signs the message, and deposits a copy of it in the internal message queue (IMQ) and sends another copy to the neighbouring processor. At the neighbouring processor, the authenticity of the incoming signed message is verified; if found authentic, the message is deposited in the external message queue (EMQ). The comparator process compares a locally produced message with its counterpart produced by the neighbour processor. If the two messages match, the copy from the neighbour processor will be counter-signed and sent to its destination. The comparator will stop the processor if one of the following three conditions is satisfied: (1) a message fails authentication; (2) a message mismatch is detected; (3) a message fails to arrive (after a time-out interval).

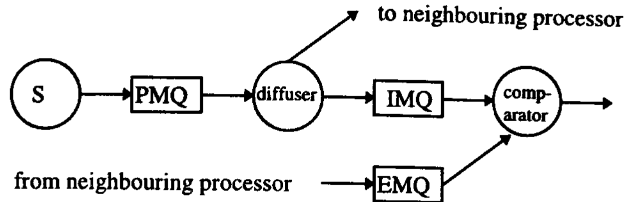

Fig. 5.4. The Comparison Module of a Processor

As we can see in this simple comparison module, the EMQ of a processor is permitted to contain more than one correct messages from the neighbour processor; thus potentially, a faulty processor can emit more than one late valid messages.

5.2.5. The Leader-Follower Node Design

A careful analysis of the ordering requirement of fail-silent node indicates that a fail-silent node is fundamentally different from a failure-masking TMR node though both architectures adopt active replication and require an ordering module to make sure application process replicas 'keep in step'.

A TMR node is a failure-masking node, it is required to produce correct outputs in the presence of a single processor failure. Thus the order protocol used in a TMR node must be a fault tolerant one. Despite the failure of a single processor, application process replicas on the other two correct processors must still be guaranteed to process identical messages in an identical order.

A fail-silent node is not required to produce any correct output in the presence of a failure. The order protocol used in a fail-silent node does not need to be a fault tolerant

one. The order protocol is only required to guarantee that application process replicas process identical messages in an identical order when there is no failure. When a failure occurs, it is up to the comparison module to detect the failure and stop the node.

The leader-follower fail-silent node [Brasi94] employs a simple non-fault tolerant order protocol. The two processors of a fail-silent node are designated as *leader* and *follower* respectively. It is the leader that decides the message order and this order is followed in the follower processor.

When a valid message is received by the leader processor, one copy of the message is deposited in the DMQ of the destination application process and another copy of the message is relayed to the follower processor. At the follower processor, the relayed message is deposited in the DMQ of the destination application process. This is how message ordering is achieved in the leader-follower fail-silent node.

This simple order protocol used in the leader-follower fail-silent node helps reduce substantially the message ordering delay as compared with the original Voltan fail-silent node which uses a fault tolerant order protocol. A detailed comparative performance evaluation is presented in [Brasi94].

Another issue addressed in the leader-follower fail-silent node is the "at most one performance failure" issue. The comparison protocol discussed in the last sub-section permits a node in the failing state to commit more than one performance failures. The only way of preventing this from happening is to use a comparison protocol that guarantees that a processor sends the next message for comparison to its neighbour *only*

after the processor has compared the current one. In order to prevent deadlocks, it is also necessary that the processors agree on the next message to compare. This ordering requirement can be achieved by inserting an order process between the PMQ and the comparison module. This ordering process for output messages adopts an approach similar to the one employed for ordering input messages. Now we describe the asymmetric comparison protocol used in the leader-follower node.

This comparison protocol is also based on the same leader-follower concept. Of the two processors of the node, one is assigned the role of a leader, and the other the follower. In the leader, the messages in the PMQ follow the same path as in the original Voltan node (see Fig. 5.4). However, it is necessary to synchronise the diffuser and the comparator: the diffuser is allowed to send a new message to the neighbour processor for comparison only if permitted by the comparator, and this permission is granted by the comparator after it has finished comparing the current message.

On the follower's side, messages produced by the application processes follow a slightly different path, as shown in Fig. 5.5. The comparator compares a message in the EMQ (sent by the leader) with its locally produced counterpart in the PMQ; if the comparison succeeds, the message received from the leader is counter-signed and this valid message is sent to its destination, and then the locally produced copy is sent to the leader for comparison. This message will arrive in the EMQ of the leader, get compared and, if successful, the comparator of the leader will then permit the next message from the leader to be transmitted to the follower for comparison.

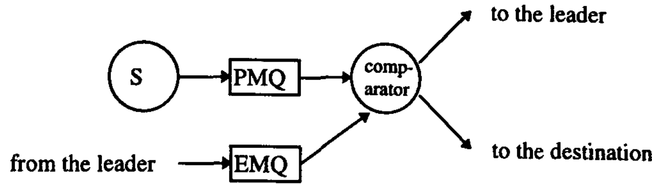

Fig. 5.5. The Comparison Module of the Follower

The message synchronisation mechanism incorporated in the leader's comparison module alone can not guarantee "at most one performance failure" suffered by the node in the failing state. This is due to the asymmetric design of the ordering module of the leader-follower node. If the leader processor is faulty, its ordering module can delay a valid message for a while before ordering it. As a result, the output messages produced by both leader and follower will be late and this delay can not be detected by the time-out mechanism employed in the comparison modules.

To overcome this problem, a time_monitor is employed in the follower processor. Its role is to monitor the arrival times of the valid messages relayed by the leader and compare them with those of their counterparts which the follower receive from the network directly. If a valid message received from the leader is found to be late, the time_monitor will stop the processor.

5.3. Leader-Follower Node Implementation

The leader-follower fail-silent node has been implemented on Inmos T800 transputers [Inmos88]. The two processors of a node are directly connected to each other by a

transputer link (see Fig. 5.6), thereby providing a fast internal path for intra-node communication.

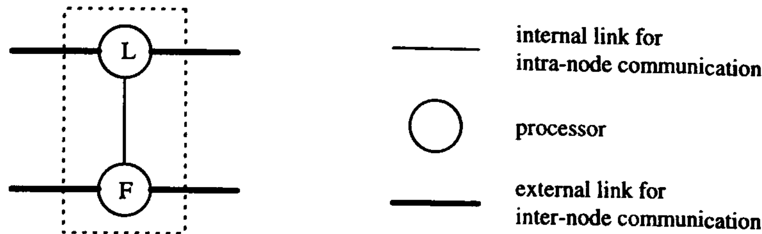

Fig. 5.6. Leader-Follower Node Hardware Connection

The Helios operating system [Perih89] runs on each of the transputers to provide operating system services. The basic implementation environment for the leader-follower fail-silent node is the same as the one for the Voltan TMR node described in chapter 4.

The leader-follower node software is written in C++. Messages are instances of a class called Message_Block. Queues are instances of a class called Message_Block_Queue. Processes are implemented as active objects which are instances of C++ classes. Apart from Message_Block and Message_Block_Queue, a C++ class called Message_List is also defined for the efficient implementation of the leader-follower node.

Message_List supports two important operations (methods) called find_or_add_tm and find_or_add_cmp, they are used by the ordering module of the follower and comparison modules respectively. As we shall see later, the use of Message_List makes the implementation of time-outs simple and efficient.

The following sub-sections describe the implementation of the leader-follower node in detail.

5.3.1. Communications Layer

The communications layer of the node is rather similar to that of the Voltan TMR node. It contains four processes: RX and TX for intra-node communication, and Receive and Send for inter-node communication. The basic function of the four processes are summarised here.

RX is used to receive messages from the neighbour processor through the internal link. TX is used to send messages to the neighbour processor through the internal link. Messages handled by RX and TX are used for either ordering or comparison. RX and TX are implemented on top of the hardware only using the 'raw' link. Receive is used to receive double-signed valid messages from the network. Send is used to send double-signed valid messages to the network. Receive and Send are implemented on top of the message passing system provided by the Helios operating system.

Another important function provided by the communication layer is message authentication. When a message is received, either a double-signed valid message from the network or a single-signed message from the neighbour processor, the message will be authenticated. If a message fails authentication, it will be discarded. Currently a checksum based message authentication mechanism is implemented.

5.3.2. Ordering Modules

The ordering module of the leader processor is made up of a single process: relayer. As shown in Fig. 5.7, the relayer picks up a valid message, relays a copy of it to the follower processor and deposits another copy in the DMQ of the destination application process.

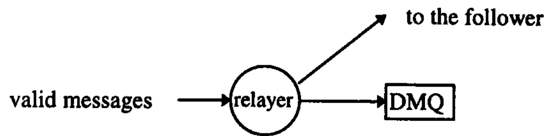

Fig. 5.7. Ordering Module of Leader

The ordering module of the follower processor is made up of two processes: receiver and deliverer (see Fig. 5.8).

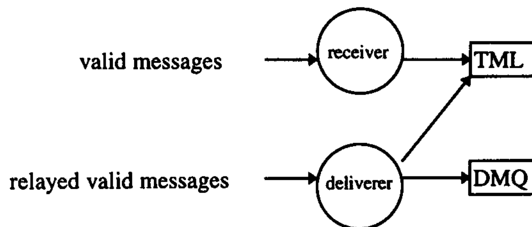

Fig. 5.8. Ordering Module of Follower

Before explaining the operation of the module, we describe the find_or_add_tm method of Message_List since it plays an important part in the implementation of module. The find_or_add_tm method takes a message as its parameter and tries to find its matching

copy on the list. If the matching copy is found, the matching copy's timestamp is checked against the current clock reading to determine whether the original message (supplied as the parameter of the method) is late or not. If the message is late, the processor will be stopped; if the message is not late, the message itself and its matching copy on the Message_List will be discarded. If there isn't a matching copy, the original message is timestamped and added to the tail of the Message_List.

The deliverer handles valid messages relayed by the leader. When a relayed message arrives, it calls, with a copy of the relayed message as the parameter, the find_or_add_tm operation on the temporary message list (TML) which is an instance of Message_List. Then the deliverer will deliver another copy of the relayed valid message to the DMQ of the destination application process. Note that if the relayed valid message is late, the find_or_add_tm method will stop the processor before a copy of it can be delivered to the DMQ.

The operation of the receiver is trivial. It simply calls the find_or_add_tm method of the TML with the valid message it itself receives from the network.

Strictly speaking, for the purpose of message ordering, the ordering module of the follower only needs to deliver the valid messages relayed by the leader to the DMQ of the destination application process. The rest of the functionality described here is really for the purpose of detecting late relayed valid messages which may cause more than one late output messages to be emitted by the node in the failing state.

5.3.3. Comparison Modules

Before explaining the operation of the comparison modules, we describe the find_or_add_cmp method of Message_List, it is vital to the implementation of comparison modules. The find_or_add_cmp method takes a message as its parameter and tries to find its counterpart on the Message_List. If the counterpart message is not on the Message_List, the original message is timestamped and added to the tail of the Message_List, and a null pointer is returned. If the counterpart message is found, the counterpart's timestamp is checked against the current clock reading to determine whether the original message (supplied as the parameter of the method) is late, and the two messages are also compared. If either the message is late or the comparison fails, the processor will be stopped; otherwise the counterpart of the original message is returned.

Fig. 5.9 shows the comparison module of the leader processor. It consists of two processes: diffuser and comparator. The diffuser diffuses a message by sending a signed copy of the message to the follower and calling the find_or_add_cmp method of the candidate message list (CML) with another copy of the message as the parameter. The CML is an instance of Message_List. If the invocation of the find_or_add_cmp method returns the matching copy of the original message, the diffuser will counter-sign the returned message, send it to its destination, and signal a semaphore (see below). The diffuser can only diffuse the next message when permission is granted. This is implemented by using a semaphore. The semaphore is initialized to 1, meaning that the diffuser can diffuse the first message. Then the diffuser will have to wait on the

semaphore before it can diffuse the next message. The semaphore is signalled when a successful comparison is done, either by the diffuser itself or by the comparator.

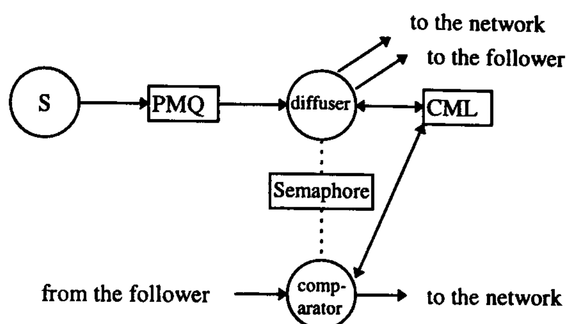

Fig. 5.9. The Comparison Module of the Leader

The comparator process receives a message diffused by the follower and calls the find_or_add_cmp method of the CML with a copy of the message as the parameter. If the call returns the matching copy of the message, the comparator will sign the message it received from the follower and send it to its destination (to the network), and then signals the semaphore so that the diffuser can diffuse the next message. The matching copy of the original message returned by the find_or_add_cmp method is discarded.

The comparison module of the follower is illustrated in Fig. 5.10. It has two processes: diffuser and comparator. The diffuser picks up an output message, signs it, and calls the find_or_add_cmp method of the CML with the message as the parameter. If the call returns the matching copy of the original message, the returned message is counter-signed and sent to its destination (to the network), and then the locally produced copy of the message is sent to the leader.

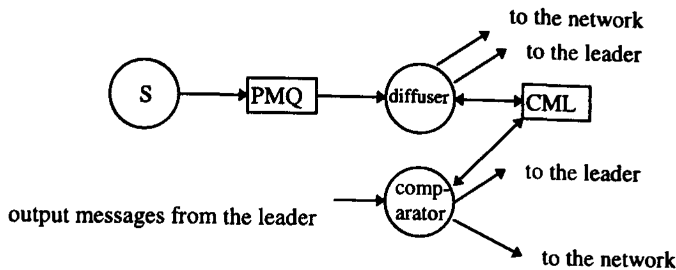

Fig. 5.10. The Comparison Module of the Follower

The comparator receives an output message diffused from the leader and calls the find_or_add_cmp method of the CML with the message as the parameter. If the call returns the matching copy of the message, the message received from the leader is counter-signed and sent to its destination (to the network), and then the returned matching copy of the message is sent to the leader.

5.4. Implementing Focused Fault Injection

The system software of the leader-follower fail-silent node is structured in a way that meets the requirement of the focused fault injection method. Focused fault injection is easily implemented on the node for fault tolerance testing. The software running on the leader processor of the node, including the system software and an application program, has the following form:

```
    .........
/* passive objects for communications between active objects */
    Message_Block_Queue  mo, imq, omq, *mp[4], dmq, pmq;
    Message_List CML;

/* active objects for intra-node communication */
```

```
    RX  rx(0, mp);
    TX  tx(0, &mo0);

/* active objects for inter-node communication */
    Receive  receive(&imq);
    Send  send( &omq);

/* active object implementing ordering module */
    Relayer relayer(&imq, &dmq, &mo);

/* active objects implementing comparison module */
    Diffuser diffuser(&pmq,&CML,&mo);
    Comparator comparator(mp[1], &CML,&omq);

/* active object of the application process */
    Application application(&dmq, &pmq);
    .........
```

With a particular fault injection object, the above program would change slightly to the following form:

```
    .........
/* passive objects for communications between active objects */
    Message_Block_Queue  mo, imq, omq, *mp[4], dmq, pmq, fq;
    Message_List CML;
/* active objects for intra-node communication */
    RX  rx(0, mp);
    TX  tx(0, &mo0);
/* active objects for inter-node communication */
    Receive  receive(&imq);
    Send  send( &omq);
/* one of the parameters of the following object is changed */
    Relayer relayer(&imq, &dmq, &fq);
/* fault injection object */
```

```
    Fault_Object fo(&fq, &mo);
/* active objects implementing comparison module */
    Diffuser diffuser(&pmq,&CML,&mo);
    Comparator comparator(mp[1], &CML,&omq);
/* active object of the application process */
    Application application(&dmq, &pmq);
     . . . . . . . . .
```

This program will be capable of injecting faults (such as late timing faults) in the double-signed valid messages which are being relayed by the relayer of the leader processor to the follower processor, and hence can be used to test the effectiveness of the time-monitoring mechanism of the follower. Other fault tolerant modules of the node can be tested in the same manner.

5.5. Experiments and Results

Our objective in the fault injection experiments on the leader-follower fail-silent node is to ascertain that the node does deliver the fail-silence properties expected from it. Especially, we want to verify, through fault injection based testing, that the node stops when one of the two processors of the node fails and the node suffers at most one performance failure in the failing state.

Since the node is implemented entirely in software using only standard off-the-shelf hardware, the delivery of fail-silence properties relies on the correctness of the system software of the node. We concentrate our efforts on the testing of the fault tolerant modules of the system software by injecting faults in one of processors of the node. It

should be pointed out that the correct functioning of the node in the absence of faults is not our concern, it can be dealt with using conventional software testing techniques.

An analysis of the system software of the node indicates there are three pieces of software in the node that are responsible for implementing the checking mechanisms of fail-silence: comparison module of the leader, comparison module of the follower, and the time-monitoring mechanism of the follower. We tested these software modules through fault injection.

Before fault injection experiments, the node had been tested by its developer with no fault injected and the node worked correctly. We assume that the checksum based message authentication mechanism has been implemented correctly. The message authentication mechanism was not subject to fault injection testing.

5.5.1. Experimental Set-Up

The set-up for our fault injection experiments is shown in Fig. 5.11. The application server S is replicated on the two processors of the node, so that either correct service is delivered or no service is delivered at all. The client runs on a conventional processor which is connected to both processors of the fail-silent node.

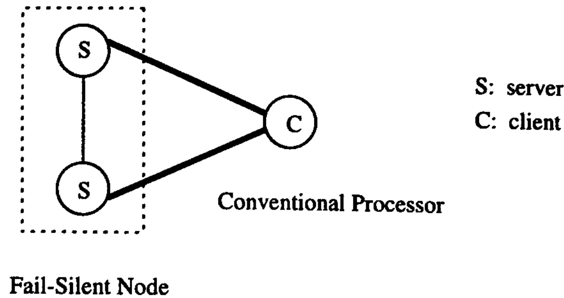

Fig. 5.11. Experimental Set-Up

The service provided by the server is trivial. When a request message which contains a number is received by the server, it sends back a reply message which contains a character string saying the number is even or odd.

The client C sends requests to the server asynchronously in close succession. This is meant to create the condition in which an incorrectly implemented fail-silent node could suffer more than one performance failures in the failing state. The violation of the "at most one performance failure in the failing state" semantics may not happen even if the node is not implemented correctly. For example, if the client sends requests to the server synchronously, i.e., it sends the next request only after the reply to the current request has been received, then the violation will not occur even if the comparison modules of the node do not incorporate a message synchronisation mechanism such as the one used in the leader-follower fail-silent node.

In the following sub-sections we describe the experiments carried out to test the three fault tolerant modules of the system software of the node.

5.5.2. Comparison Module of Follower

The task of the comparison module of the follower processor is to stop the processor when it receives an erroneous message for comparison. To test the module, we fault-injected the leader processor so that the output messages sent by the leader for comparison at the follower were erroneous.

We first injected omission and late timing faults using the software structure shown in Fig. 5.12. The fault injection object FO was inserted into the software running on the leader processor to intercept and manipulate the output messages produced by the server S.

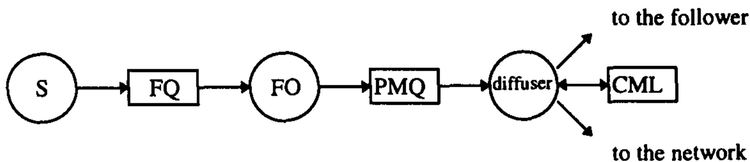

Fig. 5.12. Fault Injection in S

The experiments revealed no faults in the comparison module. The node stopped successfully and there was no violation of the "at most one performance failure" semantics in the failing state.

We then injected value faults. In this experiment, we emulated a faulty situation in which incorrect output is produced by S and this output is diffused by the diffuser. As a result, not only the message sent to the follower is erroneous, the local copy used for comparison is also erroneous.

In the experiment, we injected value faults by modifying a single byte of the character string carried by the output message. To our surprise, the node did not stop and incorrect reply messages were sent to the client.

This phenomenon was reported to the implementer of the node. A subsequent analysis of the code by the implementer revealed that a wrong function was called to compare two messages. The function only compares the control sections of the messages while the data sections are not compared at all. This function was written for a different purpose but was mistakenly used. This bug was also present in the comparison module of the leader processor. This bug was later corrected.

We also injected arbitrary faults by both modifying the content of the message and delaying the message for a while. No further bug was detected.

5.5.3. Comparison Module of Leader

To test the comparison module of the leader processor, we fault-injected the follower processor so that the output messages sent by the follower for comparison at the leader were erroneous. The experiments carried out were identical to those carried out to test the comparison module of the follower.

Using the software structure shown in Fig. 5.12, we injected omission, late timing, value, and arbitrary faults. The node stopped successfully and there was no violation of the "at most one performance failure" semantics in the failing state either.

Note that the bug uncovered in the comparison module of the follower had also been present in the comparison module of the leader, but it was corrected before the experiments reported here were carried out.

5.5.4. Time-Monitoring Mechanism of Follower

The time-monitoring mechanism of the follower is integrated in the ordering module. Its sole function is to detect a late relayed valid message and stop the processor.

We injected late timing faults in the leader processor using the following software structure (Fig. 5.13). The valid messages relayed by the leader to the follower were delayed by the fault injection object FO.

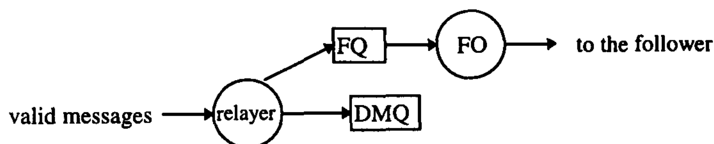

Fig. 5.13. Fault Injection in Leader

The time-monitoring mechanism of the follower successfully detected the first late relayed message. As a result, the processor was stopped and the node became silent, and there was no violation of the "at most one performance failure" semantics in the failing state.

5.6. Summary

In this chapter we described the fault tolerance testing of the leader-follower fail-silent node, using focused fault injection. Like the Voltan TMR node, the leader-follower fail-silent node is also implemented entirely in software. Its ability to fulfil the fail-silence properties depends on the correctness of the system software of the node. The fault tolerant modules of the system software must perform their specified functions in the presence of failures.

The fault injection experiments carried out on the node again demonstrated the usefulness of the focused fault injection method in uncovering fault tolerance deficiency faults in systems. One bug in the comparison module of the follower processor was detected.

Chapter 6: Applying Focused Fault Injection at Higher Levels of a Distributed System

6.1. Introduction

In chapter 3 we discussed the three levels of a distributed system at which fault tolerance can be applied to achieve system reliability. They are node level, distribution level, and application level. Here we reproduce Fig. 3.8 to illustrate the point.

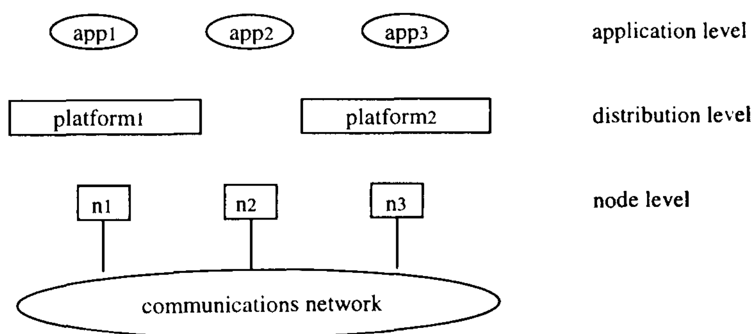

Fig. 3.8 Levels of Fault Tolerance in Distributed Systems

We have shown in previous chapters how focused fault injection method can be used for fault tolerance testing at node level when the fault tolerant node concerned is implemented in software. The purpose of this chapter is to show that the same approach

can also be used for fault tolerance testing at distribution level and application level.

In Voltan nodes, access to communication software for insertion of fault injection objects was straightforward. However, in many distributed systems where fault tolerance is applied at higher levels, message exchanges among the processors of the system are based upon the use of a set of primitives provided by the underlying communication layer. Examples include the Arjuna distributed programming system [Shriv91, Parri95] and ISIS system [Birma93]. The target system modules have direct access to the primitives for sending and receiving messages, rather than make use of link handling objects as in Voltan software architecture. For these systems it is not possible to insert an injection object to intercept and manipulate output messages.

The focused fault injection method described in chapter 3 obviously can not be employed directly in such distributed systems. The essence of the focused fault injection method is the transparent implementation of fault injection. Transparent implementation means not having to go through the source code of the target system and make considerable changes to accommodate fault injection activities. This essential point must be maintained when the focused fault injection method is modified and applied to such systems.

In order to conduct focused fault injection in such systems, we will need to have the ability to intercept communication messages in a transparent way. Once communication messages are intercepted, it will be possible to manipulate them to emulate the faulty behaviour of processors.

In this chapter we show, through examples, the application of focused fault injection at higher levels of distributed systems. Section 6.2 discusses focused fault injection at distribution level. Application level fault injection is described in section 6.3. Section 6.4 summarises this chapter.

6.2. Distribution Level Fault Tolerance

Distribution level fault tolerance is typically provided as a separate layer of software (fault tolerance platform software) between the 'raw' distributed system (hosts connected through a communications subsystem) and application software. It shields the application developers from the complexities of transparent access to remote objects and redundancy management.

In this section, we first introduce a generic scheme for applying fault injection and discuss its implementation using a known technique. And then we describe how the scheme can be usefully applied to the ISIS system [Birma93] for testing its atomic broadcast protocol.

6.2.1. Focused Fault Injection Scheme

The first step in focused fault injection involves the interception of communication messages which characterise the external behaviour of processors in distributed systems. Once messages are intercepted, they can be manipulated to emulate the faulty behaviour of processors.

We take the approach of structuring the fault injection software into two logical

layers for message interception and message manipulation, respectively. Communication messages intercepted by the message interception functions will be passed on to the fault injection functions for manipulation. On the injected processor, there will be a fault injection synchronisation object (FISO) through which injection activities on the processor can be co-ordinated if required. The software structure is shown in Fig. 6.1. The target system processes P1 and P2 are linked to the fault injection software. The individual fault injection functions communicate with the FISO for the necessary fault injection control information. The FISO is an independent process dedicated to co-ordinating injection activities on the injected processor.

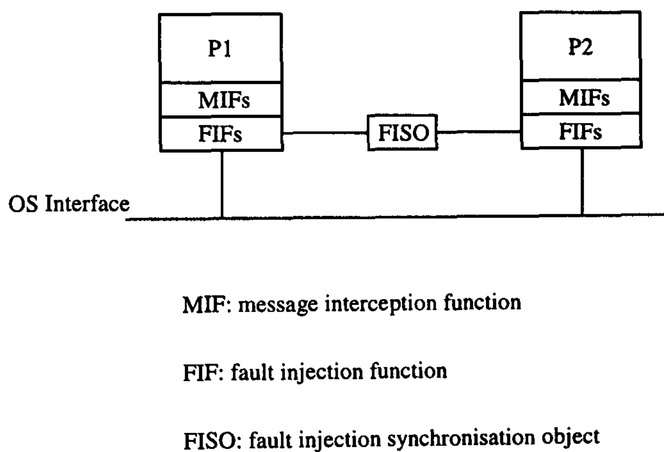

Fig. 6.1. Fault Injection of Multi-Process Target System

An omission fault (message loss) is injected, if the fault injection function concerned does not send the message at all and simply returns. A value fault is injected by having the fault injection function concerned change the value of the message. A late timing fault can be injected by delaying the delivery of the intercepted message. Other more

complex failure scenarios can be created in ways similar to those described in chapter 3.

The injected process can either be a management process which only contains platform software or an application process which contains both platform software and application software.

Techniques for transparent message interception do exist. For example, a tool, Delayline [Ingha94], has been developed, originally to simulate wide area network characteristics over a local area network for UNIX based distributed systems. It has the capability of intercepting communication messages transparently; neither the source code nor the operating system needs to be changed in any way to accommodate message interception. Intercepted communications messages are manipulated to simulate the characteristics of a wide area network. Message interception is achieved by using compile time switches to force the application program to use a set of alternative header files in preference to the standard ones. As a result, Delayline versions of the communication primitives are called. In this way, messages are intercepted and manipulated.

Another issue we come across when we consider fault injection based testing of distributed systems that manage persistent, long lived data is that these systems assume the existence of stable storage. Stable storage is used to support persistence and crash recovery properties. The general assumption is that data written to the stable storage will survive processor failures. In many practical distributed systems, the disk based file system is used as stable storage.

Because of the use of stable storage in distributed systems for achieving fault tolerance,

fault injection based testing would inevitably have to take into account the operations on stable storage. For example, a processor failure just before writing a piece of important information to the stable storage and one just after the write operation may well have different consequences. In other words, they are two different failure scenarios.

The focused fault injection method should allow these different failure scenarios to be emulated. This means operations on stable storage must be monitored so that one is able to determine when to fault-inject the processor. Fortunately, file system operations are invoked through the use of system calls. These system calls can be intercepted using standard Delayline like techniques.

Once intercepted, system calls for disk operations are inspected to monitor computational progress only and no manipulation will be required. The information extracted from inspecting the parameters of the system call will be communicated to the FISO on the local processor. This information can later be used by the FISO in making fault injection decisions, such as when to inject and what to inject.

6.2.2. The ISIS Example

Here we discuss the ISIS system [Birma93] as an example of distribution level fault tolerance. ISIS has been widely used in the financial sectors as an alternative to the traditional fault tolerant computers. ISIS supports the development of fault tolerant application systems out of conventional UNIX machines connected by a network. It does not require any specialised hardware. Redundancy management required for

replicated processing is provided by the ISIS software.

The central idea to the ISIS approach of distributed fault tolerant computing is *process group*. An application system is modelled as a collection of communicating processes. To achieve required system reliability, these processes are replicated among a number of hosts, in the form of process groups. Each application process is now represented by a replicated process group. The member processes of a process group must receive and process identical messages in identical order to 'keep in step'. This is a basic requirement for replicated processing as discussed in chapter 4. In ISIS, this ordering requirement is satisfied by an atomic broadcast protocol [Birma87]. This protocol guarantees that all members of a process group receive identical messages in identical order even if the sender process fails while sending the message. This protocol is one of the key components underpinning the ISIS architecture. We consider the fault tolerance testing of the implementation of this protocol using focused fault injection.

We first present the essence of the protocol (the insignificant details are omitted here, for a full description of the protocol see [Birma87]) and then describe how failure scenarios can be created using focused fault injection method to test the implementation of the protocol.

The atomic broadcast protocol is a two-phase protocol. The protocol assumes that the hosts (called *sites* in ISIS) on which the protocol executes are fail-silent. For each application process, the protocol maintains two separate queues: *temporary queue* and *delivery queue*. The temporary queue is used as a buffer for messages to be ordered. The delivery queue contains ordered messages for the process. Messages on the

temporary queue are assigned *priority values*. Priority values are integers with a process ID appended as a suffix to disambiguate the priority values assigned for different processes. Each message in the temporary queue is tagged *deliverable* or *undeliverable*. The protocol works as follows:

1. The sender transmits the message to its destinations.

2. Each recipient adds the message to the temporary queue, tagging it as undeliverable. It assigns this message a priority value lager than the priority value of any message on the queue, with the process ID of the application process as a suffix. It then informs the sender of the priority value that it assigned to the message.

3. The sender collects responses from recipients that remain operational. It then computes the maximum value of all the priority values it received, and sends this value back to the recipients.

4. The recipients change the priority value of the message to the value they receive from the sender, tag the message as deliverable, and re-sort their temporary queue. They then transfer messages from the temporary queue to the delivery queue in order of increasing priority value, until the temporary queue becomes empty or the message with the lowest priority value is undeliverable. In the latter case no more messages are transferred until the message at the head of the queue becomes deliverable.

If a sender failure occurs, any site that has a message tagged undeliverable detects this using the monitoring mechanism and can then take over as the new co-ordinator to

complete the protocol. It does so by interrogating participants about the status of the message. A participant being interrogated either has never received the message or responds with the priority value and tag. The new co-ordinator collects responses. If any process has marked the message deliverable, the new co-ordinator distributes the corresponding priority value to other processes; if the message is marked as undeliverable by all participants, it computes the maximum priority value and distributes it (step 3). Otherwise, it resumes from step 1.

From the description of the protocol, we can see a number of rather difficult failure scenarios the protocol must cope with. These are the sender failures at the various points of the protocol execution. The first one is when the sender fails while sending the message to its destinations; the second one is when the sender fails just after it finished sending the message to its destinations; the third one is when the sender fails while sending the maximum priority value. Each of the three failure scenarios requires the site which takes over the control of the protocol execution from the sender to respond in a different way.

In the implementation of the protocol, the sender part is implemented in a library. This library is linked to any application process which sends messages to a process group. This library can be re-compiled using an alternative set of header files such that all operating system calls for message communications are replaced by calls to our message interception functions (see Fig. 6.1). The message interception functions will then call fault injection functions for the actual fault injection. In this case, the fault to be injected is simple. All fault injection functions need to do is to monitor the outgoing messages

and stop the process at the required point of protocol execution.

The above-mentioned three failure scenarios can be created using this technique. Such failure scenarios can be used to test the fault tolerance capabilities of the protocol implementation.

6.3. Application Level Fault Tolerance

Fault tolerance can also be built into distributed applications using fault tolerance techniques such as checkpointing based recovery protocols. A checkpointing based recovery protocol supports fault tolerance through non-replicated processing. It provides facilities which allow a process to recover the state of a previously crashed process and resume the computation. This approach to fault tolerance is suitable for long-running distributed applications without real-time requirements.

In such systems, a distributed application is modelled as a collection of processes communicating with one another through message exchanges. The processes also have access to stable storage for saving *checkpoints* – process states. Messages exchanged and other information may also be saved on stable storage to assist recovery. The processes are assumed to be fail-silent.

Recovery protocols are typically designed to allow arbitrary number of process failures, especially process failures can occur when a previously crashed process is being recovered. To effectively test the recovery capabilities of a protocol implementation, faults must be injected to create such failure scenarios.

The generic fault injection scheme described in section 6.2 for distribution level fault tolerance can also be used for application level fault tolerance. This is because fault injection is achieved by intercepting and manipulating the messages sent by the processes running on the injected processor. The internal structure of a injected process is not important. Within a process, the application software can be built on top of a fault tolerance layer as in the case of distribution level fault tolerance, or fault tolerance mechanisms can be built into the application software as in the case of checkpointing based recovery systems.

6.3.1. The Manetho Example

Manetho [Elnoz92] is a recovery protocol which employs *antecedence graphs* to record 'happened before' relations [Lampo78] between events of the system. An event can be the receipt of a message or an internal state change triggered by the operating system. In Manetho, each process of the system maintains its own growing antecedence graph as the computation progresses. When a message is transmitted from one process to another, the sender (conceptually) attaches its current antecedence graph to the message so that the receiver knows the events that have happened before the receipt of the message.

When a process is recovered, it first retrieves its previously saved state (checkpoint) from stable storage and then it queries the surviving processes for their antecedence graphs. By conducting the query, the process is effectively asking the surviving processes for information on the state to which it had progressed before crashing. Based on the information gathered, in the form of antecedence graphs, it can 'replay' the

computation between the time the last checkpoint was saved and the time it crashed. During the replay the recovering process may ask the surviving processes to send some messages they sent before the crash.

While a process is recovering, another process can crash and subsequently needs to recover. The antecedence graph or the messages needed for the replay by the first recovering process may not be available. A deadlock situation may potentially arise.

Focused fault injection method would be suitable for creating various required failure scenarios to test the implementation of the protocol. By intercepting messages (see sub-section 6.2.1), either messages sent to other processes or information saved on the stable storage, a process can be crashed at any desired point during the execution and hence creating the required failure scenario.

For example, during a recovery, a surviving process can be crashed after it has sent its antecedence graph to the recovering process but before it has sent the messages required for the replay by the recovering process. This may cause a potential deadlock since the messages required by the (first) recovering process will not be available until the second crashed process recovers successfully.

6.4. Summary

In this chapter we discussed the application of focused fault injection to distributed systems where fault tolerance is provided at either distribution level or application level. In the implementation of such systems, typically messages are sent by invoking

primitives provided by the operating system instead of depositing messages in an output message queue. This makes it impossible to intercept and manipulate communication messages as described in chapter 3. To apply focused fault injection in these systems, different techniques are required.

We propose the use of a transparent message interception technique which has been developed and used elsewhere for a different purpose. Intercepted massages can then be passed to fault injection functions for manipulation to create failure scenarios.

In this way, our focused fault injection method can be used for a wide variety of distributed systems while the essence of the method is maintained, that is, supporting transparent implementation of fault injection.

In a recent paper by Dawson and Jahanian [Dawso95], a different approach of testing error handling capabilities of distributed systems was proposed. The essence of the approach is to re-implement the target software in one or more testing (injected) processors in a layered architecture such that a fault injection layer can be easily inserted. This approach is suitable for the testing of implementations of standard protocols, e.g., TCP [Postel81]. This is because once the protocol has been implemented in a testing host it can then be used to test any implementation of the protocol. However, for distributed fault tolerant systems, it typically means one re-implementation for each system to be tested, which requires substantial efforts.

Chapter 7: Conclusions

7.1. Contributions

Fault tolerant computing systems are designed to perform specified functions even in the presence of specified types of faults. Testing fault tolerance capabilities of such systems therefore requires creation of faulty conditions the system is supposed to tolerate. In this thesis we presented a fault injection method which is essentially intended for testing software-implemented fault tolerance mechanisms of distributed systems.

In distributed systems where processors communicate with one another through message exchanges, messages provide a natural and convenient way of injecting faults into the system. The focused fault injection method described in the thesis is based on an object oriented approach of software implementation. It requires that the target system be structured as a collection of objects interacting via message exchanges. In such a system, fault injection objects can be easily inserted into the system to intercept and manipulate output messages so that incorrect behaviour of faulty processors can be emulated. We described, in a systematic manner, how various failure scenarios can be created using the fault injection method.

The central objective of implementing fault tolerance in distributed systems is to achieve system reliability so that the services provided by the system will still be

available in the presence of component failure(s). Fault tolerance can be applied at three different levels in distributed systems to achieve system reliability. They are node level, distribution level, and application level.

The method has been applied to test two different reliable node systems constructed out of conventional processors using software-implemented fault tolerance. In the first case, the target system is a three-processor TMR node which is required to mask the failure of a single processor. Two fault tolerance deficiency faults - software bugs that compromise the system's fault tolerance capabilities, were uncovered in the experiments.

In the second case study, a leader-follower fail-silent node was subject to fault injection based testing. The fail-silent node is not required to mask a processor failure, it is required just to stop outputting valid messages. The aim of the testing was to check whether this and other related properties were maintained when one of the processors of the node was fault-injected. One fault tolerance deficiency fault was uncovered.

In both case studies, the focused fault injection method has been shown to be easy to use and allow us to inject specific classes of faults to create failure scenarios required by the experiments.

From our experience of fault tolerance testing using focused fault injection, we made the following two observations. First, in order to select appropriate faults to inject, the tester must fully understand the algorithm employed by the software system under test. Only with a solid understanding of the algorithm, the tester can determine what failure

scenarios to create for the testing, especially when creating 'stress conditions'. Second, the tester also needs to know the implementation structure of the target system in terms of the processes that make up the system and how they interact with one another. This outline knowledge of implementation is needed for inserting fault injection objects. However, the tester does not need to know the internal details of the implementation of individual processes.

In the thesis, we also described the application of our fault injection method to distributed systems where fault tolerance is provided at either distribution level or application level.

7.2. Future Directions

7.2.1. Limitations of the Work

Software testing involves two separate issues: the ability to test and the selection of test data. In conventional software testing, the ability to test is not normally regarded as a real issue; once the test data is selected, the testers are assumed to know how to feed the test data to the target system. As a result, software testing research has concentrated on test data selection. The software testing techniques reported in the literature are mostly test data selection techniques.

Fault tolerant computing systems must handle an additional class of inputs: failures. The ability to test is a real issue here. This is especially true for complex distributed fault tolerant systems which are designed to cope with various failure scenarios. This is the issue tackled in this thesis. However, the issue of test data selection, i.e., the selection of

failure scenarios, is still with us. This issue is largely untouched in the thesis. The experiments reported in the thesis were mainly intended to demonstrate the usefulness of focused fault injection method. The results of the experiments must be interpreted cautiously since the failure scenarios selected have not been examined for their fault revealing power using techniques such as mutation testing.

Using focused fault injection method, a fault injection object is inserted into the target system to intercept and manipulate output messages. This obviously causes delays to the messages intercepted. The amount of delay added is implementation dependent and its impact on the experiments also depends on the nature of the target system. The important issue is whether the added delay is within an acceptable bound. This delay may be unacceptable for some real-time systems. For example, when a value fault is to be injected and the added delay would make the fault appear like a value and timing fault (arbitrary fault).

7.2.2. Further Work

As discussed in the previous section, fault selection is an important issue in fault tolerance testing. More research is needed in this area so that fault selection has a sound theoretical basis and one can be more confident in interpreting test results. There are two potential approaches. Since faults are an extra class of inputs for fault tolerant systems, one may treat faults as any other inputs. In this approach, the established software testing techniques can be used to test fault tolerant systems. However, faults are not ordinary inputs; system reliability requirements and functional requirements are often specified separately. In the second approach, faults and functional inputs are

categorised separately. One can first test the system under fault-free conditions and then carry out fault tolerance testing with faults and functional inputs selected using certain methods.

Powell *et al* [Powel95] investigated the problem of estimating the coverage of a fault tolerance mechanism through statistical processing of observations collected in fault injection experiments. A framework which clearly characterises the activity set (functional inputs) and fault set (faults) was used to model the experiments. Though this work was aimed at estimating fault tolerance coverage, a similar approach can also be adopted in software testing for the purpose of removing fault tolerance deficiency faults.

In chapter 6 we discussed the possibility of adapting the focused fault injection method and applying it to existing message based fault tolerant systems. Further detailed investigations are needed in this area and the approach should be evaluated with practical examples.

We have exploited the software implementation approach of structuring a target system as a collection of objects interacting via messages for fault injection by inserting fault injection objects into the target system to intercept and manipulate output messages. We realise that this system structuring approach may be exploited more generally for distributed systems testing. Due to the inherent non-determinism in message transmission delays in distributed systems, repeatability of certain operational scenarios may be difficult. The message manipulation techniques described in chapter 3 could be used to create required operational scenarios, provided that the target system semantics

are not violated. It would be quite interesting to investigate how such techniques can be applied to distributed systems testing.

References

[Arlat89]
J. Arlat, Y. Crouzet and J.-C. Laprie, "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems", Proc. 19th International Symposium on Fault-Tolerant Computing(FTCS-19), pp. 348-355, Chicago, IL, USA, June 1989.

[Arlat90a]
J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications", IEEE Transactions on Software Engineering, 16(2), pp. 166-182, February 1990.

[Arlat90b]
J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell, "Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System", IEEE Transactions on Reliability, 39(4), pp. 455-467, October 1990.

[Arlat91]
J. Arlat, Y. Crouzet and J.-C. Laprie, "Fault Injection for the Experimental Evaluation of Fault Tolerance", Proc. Esprit'91 Conference, pp. 791-805, Brussels, November 1991.

[Avizi87]
A. Avizienis and D. Ball, "On the Achievement of a Highly Dependable and Fault Tolerant Air Traffic Control System", IEEE Computer, 20(2), pp. 84-90, February 1987.

[Avres92]
D. Avresky, J. Arlat, J.-C. Laprie and Y. Crouzet, "Fault Injection for the Formal Testing of Fault Tolerance", Proc. 22nd International Symposium on Fault-Tolerant Computing(FTCS-22), pp. 345-354, Boston, MA, USA, July 1992.

[Avrit95]
A. Avritzer and E. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software", IEEE Transactions on Software Engineering, 21(9), pp. 705-716, September 1995.

[Barton90]
J. Barton, E. Czeck, Z. Segall and D. Siewiorek, "Fault Injection Experiments Using FIAT", IEEE Transactions on Computers, 39(4), pp. 575-582, April 1990.

[Benel89]
R. Benel, R. Dancey, J. Dehn, J. Gutmann, and D. Smith, "Advanced Automation System Design", Proceedings of the IEEE, 77(11), pp. 1653-1660, November 1989.

[Birma87]
K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures", ACM Transactions on Computer Systems, 5(1), pp. 47-76, February 1987.

[Birma93]
K. P. Birman, "The Process Group Approach to Reliable Distributed Computing", Communications of the ACM, 36(12), pp. 37-53, December 1993.

[Brasi94]
F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs and S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems", Technical Report, Department of Computing Science, University of Newcastle upon Tyne, January 1994.

[Budd81]
T. A. Budd, "Mutation Analysis: Ideas, Examples, Problems, and Prospects", in Computer Program Testing (Eds., B. Chadrasekaran and S. Radicchi), North-Holland, pp. 129-148,1981.

[Chill89]
R. Chillarege and N. S. Bowen, "Understanding Large System Failures - A Fault Injection Experiment", Proc. 19th International Symposium on Fault-Tolerant Computing(FTCS-19), pp. 356-363, Chicago, IL, USA, June 1989.

[Chill87]
R. Chillarege and R. K. Iyer, "Measurement-Based Analysis of Error Latency", IEEE Transactions on Computers, 36(5), pp. 529-537, May 1987.

[Choi90]
G. S. Choi, R. K. Iyer, and V. A. Carreno, "Simulated Fault Injection: A Methodology to Evaluate Fault Tolerant Microprocessor Architectures", IEEE Transactions on Reliability, 39(4), pp. 486-491, October 1990.

[Choi92]
G. S. Choi and R. K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis", IEEE Transactions on Computers, 41(12), pp. 1515-1526, December 1992.

[Clark95]
J. A. Clark and D. K. Pradhan, "Fault Injection: A Method for Validating Computer-System Dependability", IEEE Computer, 28(6), pp. 47-56, June 1995.

[Crist85]
F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement", Proc. 15th International Symposium on Fault-Tolerant Computing(FTCS-15), pp. 200-206, Ann Arbor, MI, USA, June 1985.

[Crist90]
F. Cristian, B. Dancey, and J. Dehn, "Fault Tolerance in the Advanced Automation System", Proc. 20th International Symposium on Fault-Tolerant Computing(FTCS-20), pp. 6-17, Newcastle upon Tyne, UK, June 1990.

[Crouz82]
Y. Crouzet and B. Decouty, "Measurement of Fault Detection Mechanisms Efficiency: Results", Proc. 12th International Symposium on Fault-Tolerant Computing(FTCS-12), pp. 373-376, Santa Monica, CA, USA, June 1982.

[Czeck90]
E. W. Czeck, and D. P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behaviour", Proc. 20th International Symposium on Fault-Tolerant Computing(FTCS-20), pp. 236-243, Newcastle upon Tyne, UK, June 1990.

[Damm86]
A. Damm, "The Effectiveness of Software Error-Detection Mechanisms in Real-Time Operating Systems", Proc. 16th International Symposium on Fault-Tolerant Computing(FTCS-16), pp. 171-176, Vienna, Austria, July 1986.

[Dawso95]
S. Dawson and F. Jahanian, "Probing and Fault Injection of Protocol Implementations", Proc. 15th International Conference on Distributed Computing Systems, pp. 351-359, Vancouver, Canada, May 1995.

[Decou80]
B. Decouty, G. Michel, and C. Wagner, "An Evaluation Tool of Fault Detection Mechanisms Efficiency", Proc. 10th International Symposium on Fault-Tolerant Computing(FTCS-10), pp. 225-227, Kyoto, Japan, October 1980.

[Dilen91]
T. R. Dilenno, D. A. Yaskin and J. H. Barton, "Fault Tolerance Testing in the Advanced Automation System", Proc. 21st International Symposium on Fault-Tolerant Computing(FTCS-21), pp. 18-25, Montreal, Canada, June 1991.

[Dupuy90]
A. Dupuy, J. Schwartz, Y. Yemini and D. Bacon, "NEST: A Network Simulation and Prototyping Testbed", Communications of the ACM, 33(10), pp. 64-74, October 1990.

[Dyer89]
M. Dyer, "The Clean-Room-Software Development Process", in Measurement for Software Control and Assuraance (eds. B. A. Kitchenham and B. Littlewood), Elsevier Applied Science, pp.1-62, 1989

[Echtl91]

K. Echtle and Y. Chen, "Evaluation of Deterministic fault Injection for Fault-Tolerant Protocol Testing", Proc. 21st International Symposium on Fault-Tolerant Computing(FTCS-21), pp. 418-425, Montreal, Canada, June 1991

[Echtl92]
K. Echtle and M. Leu, "The EFA Fault Injector for Fault-Tolerant Distributed System Testing", Proc. 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 28-35, Amherst, MA, USA, July 1992.

[Elnoz92]
E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit", IEEE Transactions on Computers, 41(5), pp. 526-531, May 1992.

[Ezhil86]
P. D. Ezhilchelvan and S. K. Shrivastava, "A Characterisation of Faults in Systems", Proc. 5th Symposium on Reliability in Distributed Software and Database Systems, pp. 215-222, Los Angeles, CA, USA, January 1986.

[Ezhil89]
P. D. Ezhilchelvan, S. K. Shrivastava and A. Tully, "Constructing Replicated Systems Using Processors with Point to Point Communication Links", Proc. 16th Annual Symposium on Computer Architecture, pp. 177-184, Jerusalem, Israel, June 1989.

[Finel87]
G. B. Finelli, "Characterization of Fault Recovery through Fault Injection on FTMP", IEEE Transactions on Reliability, 36(2), pp. 164-170, June 1987.

[Goswa90]
K. K. Goswami and R. K. Iyer, "DEPEND: A Design Environment for Prediction and Evaluation of System Dependability", Proc. 9th Digital Avionics Systems Conference, pp. 87-92, Virginia Beach, VA, USA, October 1990.

[Goswa91]
K. K. Goswami and R. K. Iyer, "A Simulation-Based Study of a Triple Modular Redundant System Using DEPEND", Proc. 5th International Conference on Fault Tolerant Computing Systems: Tests, Diagnosis, Fault Treatment, pp. 300-311, Nuremberg, Germany, September 1991.

[Gunne89]
U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", Proc. 19th International Symposium on Fault-Tolerant Computing(FTCS-19), pp. 340-347, Chicago, IL, USA, June 1989.

[Gutho95]
J. Guthoff and V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method", Proc. 25th International Symposium on Fault-Tolerant Computing(FTCS-25), pp. 196-206, Pasadena, CA, USA, June 1995.

[Halpe84]
J. Y. Halpern, B. Simons, H. R. Strong, and D. Dolev, "Fault-tolerant Clock Synchronization", Proc. 3rd ACM Symposium on Principles of Distributed Computing, pp. 89-102, Vancouver, B.C., Canada, August 1984.

[Hopki78]
A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP - A Highly Reliable Fault Tolerant Multiprocessor for Aircraft", Proceedings of IEEE, 66(10), pp. 1221-1239, October 1978.

[IEEE88]
IEEE Standard VHDL Language Reference Manual, 1988.

[Ingha94]
D. B. Ingham and G. D. Parrington, "Delayline: A Wide-Area Network Emulation Tool", USENIX Computing Systems, 7(3), pp. 313-332, Summer 1994.

[Inmos88]
Inmos Limited, "Transputer Reference Manual", Prentice Hall International, 1988.

[Iyer93]
R. K. Iyer and D. Tang, "Experimental Analysis of Computer System Dependability", Technical Report, CRHC-93-15, Center for Reliability and High Performance Computing, University of Illinois at Urbana-Champaign, September 1993.

[Jenn94]
E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", Proc. 24th International Symposium on Fault-Tolerant Computing(FTCS-24), pp. 66-75, Austin, Texas, USA, June 1994.

[Jewet91]
D. Jewett, "Integrity S2: A Fault-Tolerant UNIX Platform", Proc. 21st International Symposium on Fault-Tolerant Computing(FTCS-21), pp. 512-519, Montreal, Canada, June 1991.

[Kanaw92]
G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", Proc. 22nd International Symposium on Fault-Tolerant Computing(FTCS-22), pp. 336-344, Boston, MA, USA, July 1992.

[Kao93]

W. Kao, R. K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behaviour under Faults", IEEE Transactions on Software Engineering, 19(11), pp. 1105-1118, November 1993.

[Lampo82]
L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, 4(3), pp. 382-401, July 1982.

[Lapri92]
J.-C. Laprie (ed.), "Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese", Springer-Verlag, 1992.

[Leber93]
G. Leber, "Preliminary Results of the Validation of the MARS System by EMI Fault Injection", Proc. IEEE International Workshop on Fault/Error Injection for Dependability Validation of Computer System, Gothenburg, Sweden, June 1993.

[Lippm89]
S. B. Lippman, "C++ Primer", Addison-Wesley Publishing Company, 1989.

[Lomel86]
D. Lomelino and R. K. Iyer, "Error Propagation in a Digital Avionic Processor: A Simulation-Based Study", Proc. Real-time Systems Symposium, pp. 218-225, New Orleans, Louisiana, USA, December 1986.

[Madei94]
H. Madeira and J. G. Silva, "Experimental Evaluation of the Fail-Silent Behaviour in Computers Without Error Masking", Proc. 24th International Symposium on Fault-Tolerant Computing(FTCS-24), pp. 350-359, Austin, Texas, USA, June 1994.

[Mahmo88]
A. Mahmood and E. J. McClusky, "Concurrent Error Detection Using Watchdog Processors -- A Survey", IEEE Transactions on Computers, 37(2), pp. 160-174, February 1988.

[Mirem92]
G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin, "Two Software Techniques for On-line Error Detection", Proc. 22nd International Symposium on Fault-Tolerant Computing(FTCS-22), pp. 328-335, Boston, MA, USA, July 1992.

[Morei76]
J. Moreira de Souza, E. Peixoto Paz, and C. Landrault, "A Research Oriented Microcomputer with Built-in Auto-diagnosis", Proc. 6th International Symposium on Fault-Tolerant Computing(FTCS-6), pp. 151-157, Pittsburg, USA, June 1976.

[Palum94]

D. L. Palumbo, "The Derivation and Experimental Verification of Clock Synchronisation Theory", IEEE Transactions on Computers, 43(6), pp. 676-686, June 1994.

[Parri95]
G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little, "The Design and Implementation of Arjuna", USENIX Computer Systems Journal, 8(2), Spring 1995.

[Perih89]
Perihelion Software Ltd, "the Helios Operating System", Prentice Hall International, 1989.

[Plank95]
J. S. Plank, Y. Kim, and J. J. Dongarrat, "Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations", Proc. 25th International Symposium on Fault-Tolerant Computing(FTCS-25), pp. 351-360, Pasadena, CA, USA, June 1995.

[Postel81]
J. Postel, "Transmission Control Protocol", RFC-793, Network Information Center, September 1981.

[Powel91]
D. Powell (ed.), "Delta-4: A Generic Architecture for Dependable Distributed Computing", Springer-Verlag, 1991.

[Powel92]
D. Powell, "Failure Mode Assumptions and Assumption Coverage", Proc. 22nd International Symposium on Fault-Tolerant Computing(FTCS-22), pp. 386-395, Boston, MA, USA, July 1992.

[Powel95]
D. Powell, E. Martins, J. Arlat, and Y. Crouzet, "Estimators for Fault Tolerance Coverage Evaluation", in Predictably Dependable Computing Systems (ed. B. Randell, J. C. Laprie, H. Kopetz, and B. Littlewood), Springer-Verlag, pp. 347-366, 1995.

[Rimen93]
M. Rimen, J. Ohlsson, J. Karlsson, E. Jenn, and J. Arlat, "Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance", Proc. 1st ESPRIT Basic Research Project PDCS-2 Open Workshop, pp. 461-483, LAAS-CNRS, Toulouse, France, September 1993.

[Rives78]
R. Rivest, A. Shamir and L. Adleman, "A Method of Obtaining Digital Signature and Public-Key Cryptosystems", Communications of the ACM, 21(2), pp. 120-126, February 1978.

[Rosen93]

H. A. Rosenberg and K. G. Shin, "Software Fault Injection and Its Application in Distributed Systems", Proc. 23rd International Symposium on Fault-Tolerant Computing(FTCS-23), pp. 208-217, Toulouse, France, June 1993.

[Schne90]
F. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), pp. 299-319, December 1990.

[Schue86]
M. A. Schuette, J. P. Shen, D. P. Siewiorek, and Y. X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes", Proc. 16th International Symposium on Fault-Tolerant Computing(FTCS-16), pp. 138-143, Vienna, Austria, July 1986.

[Schue87]
M. A. Schuette and J. P. Shen, "Processor Control Flow Monitoring Using Signature Instruction Streams", IEEE Transactions on Computers, 36(3), pp. 264-275, March 1987.

[Segal88]
Z. Segall, D.Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT - Fault Injection Based Automated Testing Environment", Proc. 18th International Symposium on Fault-Tolerant Computing(FTCS-18), pp. 102-107, Tokyo, Japan, June 1988.

[Shin86]
K. G. Shin and Y. H. Lee, "Measurement and Application of Fault Latency", IEEE Transactions on Computers, 35(4), pp. 370-375, April 1986.

[Shin91]
K. G. Shin, "HARTS: A Distributed Real-Time Architecture", IEEE Computer, 24(5), pp. 25-35, May 1991.

[Shriv90]
S. K. Shrivastava, P. Ezhilchelvan, and M. Little, "Understanding Component Failures and Replication in Distributed Systems", ISA Project Report (UNT/TR1), Department of Computing Science, University of Newcastle upon Tyne, May 1990.

[Shriv91]
S. K. Shrivastava, G. N. Dixon and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", IEEE Software, pp. 66-73, January 1991.

[Shriv92]
S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully, "Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems", IEEE Transactions on Computers, 41(5), pp. 542-549, May 1992.

[Somme92]

I. Sommerville, "Software Engineering" (4th Edition), Addison-Wesley, 1992.

[Speir93]
N. A. Speirs, S. Tao, F. V. Brasileiro, P. D. Ezhilchelvan and S. K. Shrivastava, "The Design and Implementation of VOLTAN Fault-Tolerant Nodes for Distributed Systems", Transputer Communications, 1(2), pp. 93-109, November 1993.

[Srika87]
T. K. Srikanth and S. Toueg, "Optimal Clock Synchronisation", Journal of the ACM, 34(3), pp. 626-645, July 1987.

[Stein95]
A. Steininger and H. Schweinzer, "A Model for the Analysis of the Fault Injection Process", Proc. 25th International Symposium on Fault-Tolerant Computing(FTCS-25), pp. 186-195, Pasadena, CA, USA, June 1995.

[Tao93]
S. Tao, P. D. Ezhilchelvan, N. A. Speirs, and S. K. Shrivastava, "Fault Injection for Fault Tolerance Validation: an Object-Oriented Approach", Proc. IEEE International Workshop on Fault/Error Injection for Dependability Validation of Computer System, Gothenburg, Sweden, June 1993.

[Tao95a]
S. Tao, P. D. Ezhilchelvan, and S. K. Shrivastava, "Focused Fault Injection Testing of Software Implemented Fault Tolerance Mechanisms of Voltan TMR Nodes", Distributed Systems Engineering Journal, 2(1), pp. 39-49, March 1995.

[Tao95b]
S. Tao, P. D. Ezhilchelvan, and S. K. Shrivastava, "Fault Injection Based Testing of Software Implemented Fail-Silent Node", Technical Report, Department of Computing Science, University of Newcastle upon Tyne, February 1995.

[Theve91]
P. Thevenod-Fosse, H. Waeselynck and Y. Crouzet, "An Experimental Study of Software Structural Testing: Deterministic versus Random Input Generation", Proc. 21st International Symposium on Fault-Tolerant Computing(FTCS-21), pp. 410-417, Montreal, Canada, June 1991.

[Tully90]
A. Tully and S. K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs", Proc. 9th IEEE Symposium on Reliable Distributed Systems, pp. 104-113, Huntsville, AL, USA, October 1990.

[Wensl78]
J. H. Wensley et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proceedings of IEEE, 66(10), pp. 1240-1255, October 1978.

[Yang85]
X.-Z. Yang and G. York, "Fault Recovery of Triplicated Software on the Intel iAPX 432", Proc. 5th International Conference on Distributed Computing Systems, pp. 438-443, Denver, Colorado, USA, May 1985.