Constructing Fail–Controlled Nodes for Distributed Systems: A Software Approach

Francisco Vilar Brasileiro

Ph.D. Thesis

University of Newcastle upon Tyne
Computing Science Department

May/1995

Abstract

Designing and implementing distributed systems which continue to provide specified services in the presence of processing site and communication failures is a difficult task. To facilitate their development, distributed systems have been built assuming that their underlying hardware components are *fail–controlled*, i.e. present a well defined failure mode. However, if conventional hardware cannot provide the assumed failure mode, there is a need to build processing sites or *nodes*, and communication infra–structure that present the fail–controlled behaviour assumed.

Coupling a number of redundant processors within a replicated node is a well known way of constructing fail–controlled nodes. Computation is replicated and executed simultaneously at each processor, and by employing suitable validation techniques to the outputs generated by processors (e.g. majority voting, comparison), outputs from faulty processors can be prevented from appearing at the application level.

One way of constructing replicated nodes is by introducing hardwired mechanisms to couple replicated processors with specialised validation hardware circuits. Processors are tightly synchronised at the clock cycle level, and have their outputs validated by a reliable validation hardware. Another approach is to use software mechanisms to perform synchronisation of processors and validation of the outputs. The main advantage of hardware based nodes is the minimum performance overhead incurred. However, the introduction of special circuits may increase the complexity of the design tremendously. Further, every new microprocessor architecture requires considerable redesign overhead. Software based nodes do not present these problems, on the other hand, they introduce much bigger performance overheads to the system.

In this thesis we investigate alternative ways of constructing efficient fail–controlled, software based replicated nodes. In particular, we present much more efficient order protocols, which are necessary for the implementation of these nodes. Our protocols, unlike others published to date, do not require processors' physical clocks to be explicitly synchronised. The main contribution of this thesis is the precise definition of the semantics of a software based *fail–silent* node, along with its efficient design, implementation and performance evaluation.

Acknowledgements

"*Mais foi tanto dos vaquêro*
qui rênô no meu sertão,
qui cantano um dia intêro
num menajo todos não."

Excerpt from *História de Vaqueiros* by Elomar.

There are many people who, in one way or another, have helped me during the time this work has been developed, and to whom I shall be forever grateful.

First of all, I would like to thank my supervisor, Dr. Neil Speirs, for the many technical discussions we have had over the past three and a half years and for his comments on earlier drafts of this thesis. A special dept of gratitude is due to Dr. Paul Ezhilchelvan, who gave me invaluable guidance for the work on *order protocols*. I am also grateful for Paul's careful reading of some important parts of this thesis. Further, I would like to express my gratitude to Professor Santosh Shrivastava for his constructive criticisms and continuous stimulus throughout the development of this work. It is my pleasure to acknowledge that part of the work on *fail–silent nodes* was developed in collaboration with Dr. Speirs, Dr. Ezhilchelvan and Professor Shrivastava.

Further thanks go to Sha Tao, Steve Caughey and David Black – fellow members of the Voltan project; and to the staff of the Computing Science Department of Newcastle University; particularly to Shirley Craig, for the tremendous work she does as the Department's librarian.

I am indebted to my colleagues at the Systems and Computing Department of the Federal University of Paraíba (DSC/UFPb) for coping with the extra workload during my absence. I can only hope that in the years to come we will all profit from my experience at Newcastle University.

I should not forget to thank the support and encouragement I have received from my family and friends in Brazil, and from the many new friends I have made whilst in Newcastle. Thanks very much folks!

Last, but not least, I am sincerely grateful to the Federal University of Paraíba which has provided me with institutional support, and to the Brazilian National Research Council (CNPq/Brasil) for their financial support (grant number 201601/91–5).

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Since their invention, computer systems have been designed to tolerate faults that may occur on their components. Nevertheless, the reasons that have made computer system designers introduce mechanisms for fault tolerance into their designs have changed considerably throughout the years. Changes in the objectives and amount of fault tolerance mechanisms added to computer systems have been driven mainly by two factors: i) the evolution of the technology of components; and ii) the introduction of new applications with new specification requirements to be met.

High failure rates of components such as relays and vacuum tubes, used to build the first generation computers in the nineteen forties and early fifties, made fault tolerance an overwhelming concern to the designers of those systems. However, the rapid evolution of components technology (e.g. transistors/core), which marked the beginning of the second generation of computers, was responsible for a change in the direction of the evolution of fault–tolerant computer systems. The reliability of components was considered to be sufficiently high for the requirements of existing applications, hence the need for any built–in fault tolerance mechanisms was not sensibly felt [Avizienis 78]. Throughout the nineteen sixties and seventies, the specification requirements of applications began to influence the design of fault–tolerant computer systems. Applications such as control of life–support hospital equipment, flight control on commercial airlines, control systems of high–speed trains, safety processes monitoring in industrial installations, launch of vehicles for space exploration projects, electronic telephone switching control and satellite support systems, on which computer failure could place human life in danger or

cause heavy economic penalties, demanded a degree of *dependability*[1] [Laprie 89] which could not be met by existent computer systems. Thus, parallel to the development of general purpose computers, highly dependable specific purpose computer systems started to be developed [Hopkins et al. 78, Ihara et al. 78, Katsuki et al. 78, Rennels 78, Siewiorek et al. 78a, Siewiorek et al. 78b, Toy 78, Wensley et al. 78]. In the past sixteen years, mainly due to the ever increasing number of applications whose specifications incorporate dependability requirements, an increasing number of dependable computer systems have been designed and implemented [FTCS 71–94]. Recently, with the development of parallel and distributed computer systems, dependability has become a major concern not only in the design of specific purpose computer systems, but also in the design of general purpose ones.

1.1. Designing Dependable Computer Systems

Fault avoidance and fault tolerance are two complementary approaches for improving the dependability of a computer system. In the first approach, the aim is to avoid the presence of faults in the system. Fault avoidance techniques are mainly concerned with the manufacturing process of the system, and the environmental conditions to which the system is exposed. The utilisation of conservative design methodologies and high quality components in the manufacturing process can considerably reduce the probabilities that the system will fail. Further, when the system is operative, a close control over environmental variables such as temperature, power supply fluctuation and human interference, can reduce even more the probabilities of a fault. Unfortunately,

---

1. Dependability as defined in [Laprie 89] is "that property of a computing system which allows reliance to be justifiably placed on the services it delivers." It can also be understood as a global concept that encloses a number of attributes such as reliability, availability, maintainability, safety, integrity and security. Each of these can be seen as a different perception of the same concept. For instance, in the context of the control system of a nuclear power plant, where a computer failure could cause devastating social and economic losses, dependability is better associated with system reliability, whilst in the context of an electronic telephone switching control system, where failures which cause the system to become unavailable for a long period of time are unacceptable, dependability is better associated with system availability.

extensive use of high quality components can impact the cost of the system immensely. Also, for some computer systems, it is not possible to successfully control or even anticipate the operating environment of the system. These two constraints reduce the feasibility of using fault avoidance techniques as the only means of providing dependability for a large number of systems.

For a more general class of systems, a more flexible approach is to forecast possible faults, and to introduce redundancy into the system in such a way that anticipated faults can be tolerated. The goal of fault tolerance techniques is therefore to guarantee that the system provides its service, with the required dependability, despite the occurrence of faults. Faults are tolerated mainly by the introduction of redundant components that can take over the functionality of faulty ones. Redundancy can also be introduced in the time domain, where faults are tolerated via redundant computation on non–redundant components (e.g. mechanisms for tolerating transient faults [Kopetz et al. 90]).

Design decisions on which fault avoidance techniques to use and which fault tolerance mechanisms to introduce into a computer system, so that it delivers its service with the required degree of dependability, are subject to cost considerations. It is necessary to weigh the cost of using such techniques and mechanisms against the cost of system malfunction, where system malfunction is normally translated as incorrect computation and/or system downtime. As will be seen in Chapter 2, the purpose of a dependable computer system is an important factor when deciding which techniques and mechanisms are appropriate. The more information that is available about the applications, the more accurate is the choice of techniques and mechanisms to be utilised. Thus, one can state that designing dependable, specific purpose systems, where the characteristics of the applications are likely to be known in advance, allows the evaluation of the dependability cost/benefit relation of the system in a much simpler and more precise fashion than when evaluating the same relation for a general purpose architecture, where little is known about the comportment of applications.

In this research we present ways of providing dependable distributed systems with a fault–tolerant processing infra–structure on top of which they can be more easily implemented. Hence, in the next chapter, when discussing the design of various dependable computer systems described in the literature, and studying how fault avoidance techniques and fault tolerance mechan-

isms have been used to implement such systems, we pay special attention to the case of dependable general purpose distributed systems.

1.2. Fault Tolerance in Distributed Systems

There are many reasons for the success of distributed computer systems. With the current development level of microprocessor and communication technologies, a distributed computer system composed of a collection of processing sites interconnected by a high–speed network can offer a much better price/performance relation than a traditional centralised mainframe. In some cases, a distributed system can even out–perform a mainframe, and becomes the only feasible solution for applications which require high degree of performance. Also, applications which involve spatially separated machines possess inherent distributed characteristics which can be captured more easily when those applications are to be developed within the framework of a distributed system. Finally, a carefully designed distributed system can potentially be more dependable than a centralised one. To achieve greater dependability, the failure of an isolated processing site, or a network connection, should have small impact, if any, on the operation of the system as a whole.

Of course, distributed systems have also got their drawbacks. The difficulty in designing and programming those systems can be regarded as one of their main problems. Ideally, programming a distributed system should be no more difficult than programming a traditional centralised computer system. This notion is reflected in the following definition of a distributed system from [Tanembaum 92]: "A distributed system is one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer."

Aiming to achieve this desirable level of transparency, new programming paradigms such as *remote procedure call* [Birrell–Nelson 84], *atomic transaction* [Lampson 81] and *group communication* [Birman et al. 91] have been proposed. Fault tolerance is one of the many issues that must be faced when designing such programming concepts. Supported by these new programming paradigms, a number of distributed systems, ranging from distributed operating systems [Kopetz–Merker 85, Accetta et al. 86, Mullender et al. 90] to distributed programming toolkits

[Shrivastava 89, Birman et al. 91], have been developed. All these systems incorporate, to a great or a lesser extent, the notion of fault tolerance.

Experiences in constructing distributed systems which continue to provide specified services in the presence of processing site and communication failures, have shown that designing and implementing such systems is a difficult task. In a perfect world, one would like to construct a distributed system using hardware components which are guaranteed to be either failure–free or to have well defined failure modes. However, all hardware components must fail eventually, possibly in an unpredictable manner. A sensible approach, taken by the designers of a considerable number of dependable distributed systems reported in the literature (e.g. [Bartlett 81, Kopetz–Merker 85, Shrivastava 89, Powell 92, Birman et al. 91]), is to build their systems assuming that the underlying hardware components of a distributed system are *fail–controlled* [Laprie 89], i.e. present a well defined failure mode, and then build processing sites or *nodes* and communication infra–structure that do indeed present the fail–controlled behaviour assumed.

The complexity of the fault tolerance mechanisms implemented at an upper software level of a dependable distributed system reflects the assumptions made upon the underlying hardware where the software is going to be executed. Many dependable systems reported in the literature have been built under the assumption that they execute on nodes that fail safely [Laprie 89], i.e. a faulty node will halt, rather than perform an unspecified transition. Nodes with this failure semantics are referred as *fail–safe nodes*. *Fail–stop nodes* [Schlichting–Schneider 83, Schneider 84] and *fail–silent nodes* [Bernstein 88, Shrivastava et al. 91, Shrivastava et al. 92, Webber–Beirne 91, Brasileiro et al. 92, Reisinger–Steininger 93] are two representatives of this class of fail–controlled nodes. Other systems assume that nodes are failure–free, hence they must execute on nodes that are able to guarantee proper functioning for the entire duration of the mission of the applications in question. *Failure–masking nodes* [Hopkins et al. 78, Wensley et al. 78, Siewiorek et al. 78a, Smith 84, Lala 86, Theuretzbacher 86, Shrivastava et al. 91, Shrivastava et al. 92, Powell 92, Speirs et al. 93] form another class of fail–controlled nodes, which are able to mask failures and therefore deliver proper service with arbitrary high probability.

On the other hand, the assumption of which failure semantics to use for the underlying hardware depends not only on the hardware characteristics itself, but also, among other para-

meters, on the dependability requirements and the mission lifetime of the applications to be executed. If conventional hardware cannot provide, with sufficiently high probability, the failure semantics assumed (for the duration of the mission of the applications they are executing), there is a need to construct nodes which guarantee the required failure semantics. It has been observed that 'off–the–shelf' processors do fail in an arbitrary way [Lala 86, Harper et al. 88]. Furthermore, for an increasing number of applications, the probability with which conventional processors fail in an arbitrary way is high enough to rule out their utilisation as fail–controlled underlying nodes for dependable computer systems.

In this thesis we present ways of using 'off–the–shelf' processors to construct nodes with different fail–controlled behaviour. We discuss the design of both failure–masking nodes (Chapter 4) and fail–silent nodes (Chapters 5), together with their implementation and performance analysis (Chapter 6). Then, in Chapter 7, we discuss the design of reconfigurable fail–controlled nodes which possess characteristics of both failure–masking and fail–silent nodes.

1.2.1. Constructing Fail–Controlled Nodes

Constructing a fail–controlled node from components that can fail in an arbitrary way involves necessarily the introduction of redundant components. One way to do this is by adding to the design self–checking logic such as error detection codes, watch-dog timers, power supply monitors, temperature monitors, etc. A problem with this approach is that achieving high error detection coverage is normally difficult. Furthermore, there is an interval of time between the occurrence of a failure and its detection by a checking circuit, during which the node can potentially present an undesirable behaviour.

Both the degree of error detection coverage and the length of the detection delay depend on the amount of redundancy introduced in the form of checking circuits. A great amount of redundant components reduces the detection delay, and increases error detection coverage, but at the same time increases the complexity and the cost of the node. In [Reisinger–Steininger 93] a fail–controlled node based on the introduction of self–checking logic is presented, whose error detection coverage is estimated to be better than 99% [Kopetz et al. 90]. The choice of the amount

and the kind of self–checking mechanisms introduced in that node is simplified by taking advantage of an a priori knowledge of specific characteristics of the system.

Another approach is to couple redundant processors within a *replicated node*. A fail–controlled replicated node is composed of a number of processors which fail independently. Computation is replicated and executed simultaneously at each processor. By employing a suitable validation technique to the outputs generated by the replicated processors (e.g. majority voting, comparison), outputs from faulty processors can be prevented from appearing at the application level. Error detection coverage in a replicated node is extremely high, since it depends exclusively on the design of a simple validation mechanism. Further, if the communication mechanisms of the system ensures that information is validated before being used at the upper application level, then damage is confined to the lower node level even when there is a substantial delay in the detection of errors.

Distributed applications are normally structured as a set of tasks executing in parallel, which do not share memory, and communicate only via messages exchange. Thus, replicated nodes can provide an attractive solution for the problem of constructing fail–controlled nodes for dependable distributed systems. A number of ways of constructing replicated nodes have been reported in the literature [Hopkins et al. 78, Toy 78, Smith 84, Lala 86, Bernstein 88, Webber–Beirne 91]. Those nodes are based on hardwired mechanisms to couple replicated processors with specialised validation hardware circuits (e.g. comparator, voter). Processors are tightly synchronised at the clock cycle level, and have their outputs validated at appropriate times by a reliable validation hardware. Another strategy, pioneered by the designers of the SIFT system [Wensley et al. 78], is to implement fail–controlled replicated nodes by using software mechanisms to perform both synchronisation of redundant processors and validation of the outputs of the node. Later in this thesis we discuss in more detail how replicated nodes following each of these approaches have been designed and implemented.

The main advantages of the hardware based nodes are the minimum performance overhead incurred, and the small impact that the architecture imposes on the software design process. However, there are also some problems with this approach. Firstly, individual processors must be built in such a way that they have a deterministic behaviour at each clock cycle. This can rule out the

utilisation of 'off–the–shelf' processors, whose reliability is normally higher than specially designed processors [Siewiorek–Swarz 92]. Secondly, the introduction of special circuits such as reliable comparator/voter and synchronisation mechanisms increases the complexity of the design, which at an extreme can result in a decrease in the overall node reliability. Finally, every new microprocessor architecture requires a considerable redesign overhead.

The advantages of the software based nodes are very much the converse of the drawbacks discussed above. The absence of tight synchronism allows the utilisation of 'off–the–shelf' processors. Further, by employing different types of processors within a node, there is a possibility that a measure of tolerance against design faults in processors can be obtained, without recourse to any specialised hardware assistance. Another advantage is that software protocols are much more flexible than their hardware counterparts. Also, the fact that the redundancy management protocols are implemented in software, allows the design of the underlying hardware to be made much simpler and possible to scale. Finally, technology upgrades appear to be easy, as the principles behind the protocols do not change, and software protocols can be ported relatively easily to any type of processor (including those expected to be available in the future).

Unfortunately, these advantages do not come for free. The synchronisation strategy of software based nodes normally imposes rules in the way that applications must be programmed, which might increase the complexity and/or introduce limitations on the development of applications. Furthermore, the extra overhead in performance imposed by the execution of the redundancy management protocols can be substantial. The first problem is less critical within the framework of some of the solutions presented in the literature [Schneider 84, Schneider 90, Shrivastava et al. 91, Shrivastava et al. 92]. There is however, a major concern over the performance overhead incurred by the redundancy management protocols. In SIFT, for instance, the overhead associated with redundancy management can consume as much as 80% of the processor throughput [Palumbo–Butler 85].

Hybrid solutions, which incorporate both tight synchronisation and software synchronisation mechanisms, have been proposed to reduce this overhead. MAFT [Kieckhafer et al. 88], FTP–AP [Lala–Alger 88] and Delta–4 [Powell 92] are hybrid architectures which share the same general structure. These architectures are structured around a tight synchronised hard core, on

top of which conventional processors are replicated. The tight synchronised hard core is responsible for executing management functions, whilst application processes are executed at the upper level replicated processors. The extra computational power delivered by the replicated processors increases the throughput of the system, and provides all the advantages of the software synchronisation approach; however, the underlying hard core re–introduces the problems associated with tight synchronisation.

The work in this thesis investigates alternative ways of constructing efficient fail–controlled replicated nodes based solely on the utilisation of 'off–the–shelf' processors (which can fail in an arbitrary way, although restricted by authentication capabilities [Strong et al. 90]) and software protocols to control system redundancy, without recourse to any specialised hardware. In particular, we present much more efficient order protocols, which are necessary for the implementation of both failure–masking and fail–silent nodes. The precise definition of the semantics of a software based fail–silent node, along with its efficient design, implementation and performance evaluation, are the main contributions of this thesis. Other contributions are:

i) design and implementation of efficient protocols for the construction of software based failure–masking nodes;

ii) performance evaluation of software based fail–controlled nodes, which indicates the feasibility of the utilisation of such nodes in a wide range of applications; and

iii) proof of the possibility of constructing software based *failure–masking before stopping nodes* composed solely of conventional processors, without recourse to specialised hardware. (This proves false a conjecture in [Shrivastava et al. 91], which argued the impossibility of constructing such nodes.)

1.3. Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2 we study several fault–tolerant computer systems, emphasising the impact of the purpose of the system on the decision

of the amount and kind of fault tolerance mechanisms used to provide systems with their required dependability. We show how the design of fault–tolerant distributed systems can be simplified, by assuming that the underlying processing and communication hardware possesses a well defined failure mode. Then, in Chapter 3, we discuss how replicated nodes can be used as a means of providing fail–controlled behaviour for the underlying processing hardware of a distributed computer system. We analyse how redundant processors are used, and how redundancy is controlled, in order to build nodes with different fail–controlled behaviour. Then, we compare the hardware and software based approaches to the control of redundancy when constructing fail–controlled nodes, highlighting the advantages and drawbacks of each approach. Finally, a general model for the construction of software based fail–controlled nodes is presented.

The model presented in Chapter 3 indicates that the key issues to be tackled when designing a software based replicated node are the provision of efficient protocols for ordering the messages to be input by the node, and for validating the messages to be output by the node. In Chapter 4, the design of failure–masking nodes is presented. Order protocols are studied in detail. [Shrivastava et al. 92] presents a software based failure–masking node, which incorporates an order protocol based on having the replicated clocks of the node synchronised within a known bound. We present new order protocols that do not need physical clocks to be explicitly synchronised. We also take into account the characteristics of the nodes we are constructing to present order protocols which are more efficient than the ones reported in the literature. The protocols presented are then applied to the important case of a Triple Modular Redundant (TMR) node, where the performance of the order protocols can be further improved. The correctness proof of the protocols is sketched, and their performance is analysed. An extended correctness proof of the protocols presented can be found in Appendix A.

[Shrivastava et al. 92] also indicates how a software based fail–silent node can be derived from the failure–masking node presented in that paper. In Chapter 5 we show that the reduced redundancy of fail–silent nodes introduces additional complications, particularly in the validation protocol (a comparison protocol), which, if not treated with due care, can lead the node to present an unpredictable behaviour. We then define the precise semantics of a software based fail–silent node [Brasileiro et al. 92]. A two–processor fail–silent node is the cheapest fail–con-

trolled node that can be constructed using 'off–the–shelf' replicated processors, thus we concentrate on the design of a two–processor fail–silent node, and introduce two new order protocols for this node. Based on the insights gained whilst developing order protocols, we have designed a comparison protocol that guarantees, with efficiency, the correct semantics of a fail–silent node. The chapter is concluded with a discussion of this comparison protocol.

Chapter 6 is devoted to the discussion of the implementations of three–processor failure–masking nodes (TMR nodes) and two–processor fail–silent nodes. Several versions of the nodes have been implemented using the different protocols presented in Chapter 4 and Chapter 5. A set of experimental distributed applications was executed using the fail–controlled nodes as platform, and the performance figures obtained for each implementation were analysed. The results indicate the feasibility of using the nodes implemented for a wide class of applications.

Fail–silent nodes take the conservative approach of stopping as soon as a failure is detected. On the other hand, failure–masking nodes are designed to survive a bounded number of failures. However, this masking property can also hide the fact that failures have occurred and have reduced the fault tolerance capabilities of the node. When the masking capabilities of the node have been totally degraded, a further failure can lead to catastrophic results. Thus, it is desirable to design a fail–controlled node which could incorporate the safety property of fail–silent nodes, as well as the survivability property of failure–masking nodes. In [Shrivastava et al. 91] the notion of a *failure–masking before stopping node* (FMS node) is presented. Such a node should be able to mask a number of failures up to a point where a further failure would potentially have catastrophic effects. At this point the node should fail safely, thus avoiding the potential catastrophic effects of a further failure. In [Shrivastava et al. 91] the authors conjecture that it is not possible to build such nodes from 'off–the–shelf' processors, which can potentially fail in an arbitrary way. In Chapter 7 we discuss reconfigurable replicated nodes. We study how dependable systems reported in the literature have incorporated reconfiguration as a means to improve system dependability. We then show that the conjecture in [Shrivastava et al. 91] is false, by presenting the design of an FMS node composed merely of 'off–the–shelf' processors, without recourse to any specialised hardware. Finally, in Chapter 8, we present our conclusions and directions for further research.

Chapter 2
Dependable Computer Systems

2.1. Introduction

Fault avoidance and *fault tolerance* are the two complementary approaches that can be taken in order to provide a computer system with the ability of dependably delivering its specified service. Fault avoidance techniques try to minimise the probability with which faults manifest themselves within the system. On the other hand, the goal of fault tolerance is to guarantee that the system provides its service with the required dependability, despite the occurrence of faults.

Fault avoidance techniques are mainly concerned with the manufacturing process of the system, and the environment conditions to which the system is exposed. During the manufacturing process, the dependability of the system is increased by means of conservative design methodologies and the utilisation of high quality components. When the system is operative, close control must be exercised over environment variables such as temperature, power supply fluctuation and human interference, so that failure rates can be kept extremely low.

The basic principle of fault tolerance is the introduction of redundancy into the system. Redundancy can be introduced in either time or resources. For instance, a processor which tolerates transient faults by simply computing the same operation several times in succession (e.g. [Kopetz et al. 90]) is an example of fault tolerance attained through time redundancy. On the other hand, a system that tolerates faults occurring on a bus by having a duplicated bus configuration (e.g. [Bartlett 81]) is an example of fault tolerance achieved by means of resource redundancy.

Extensive use of high quality components has an immense impact on system cost. Furthermore, since most of the time designers cannot control or even anticipate the operating environment of the system, fault avoidance alone is not suitable for attaining the dependability requirements of a large number of computer systems. Thus, in the discussion to follow, we concentrate on fault tolerance mechanisms, rather than fault avoidance techniques, as the main means of achieving dependability for a more general class of systems.

2.2. Designing Fault–Tolerant Computer Systems

A computer system is composed of a number of components which are bounded together, and interact in a well defined way to provide a specified service. The behaviour of a component can be specified as a function of its internal state, the state transitions that it should perform, and possible outputs that it should produce in response to specified input stimuli. A component failure occurs when the behaviour of the component first deviates from its specification. A failure, therefore, can be attributed to either an erroneous transition, or to a sequence of valid transitions initiated from an erroneous state. In both cases the system has experienced the manifestation of a fault [Laprie 89]. To summarise, there is a cause/effect relationship that is represented by the chain **fault→error→failure**, and can be interpreted in the following way: the manifestation of a fault can give rise to errors in the internal state of a component, which can lead to the failure of the component. The ultimate intent of fault tolerance is to prevent failures in internal components causing the system to fail in delivering its service. A fault–tolerant computer system is designed to tolerate a finite and bounded number of failures that its components may suffer. Hence, fault–tolerant systems must provide mechanisms to deal with both faults and errors, so that a finite number of failures of the components of the system do not cause the system to deviate from its specified behaviour.

In [Lee–Anderson 90] four constituent phases of fault tolerance are identified. The first three phases – error detection, damage confinement and assessment, and error recovery – are related to error treatment mechanisms, whilst the last phase is related to fault treatment mechanisms, and encompasses fault diagnosis and location, system repair, and continued system service. It is worth noting that there is no strict order on the execution of actions associated with each of

the phases listed above, furthermore, in many fault–tolerant systems it is not always possible to match a particular mechanism with a particular phase. Nevertheless the discussion of each phase in isolation is useful to indicate to the reader how fault–tolerant computer systems are structured.

i) *Error detection*: fault–tolerant systems must first learn of the manifestation of faults before attempting to tolerate any fault it is designed to tolerate. Hence, fault tolerance mechanisms are usually triggered by the detection of errors in the internal state of the system. This implies that error detection mechanisms are a crucial issue in the design of many fault–tolerant system. Replication, timing checks, reversal checks and coding, form the set of most popular techniques to implement error detection mechanisms (see [Lee–Anderson 90] for more details on these techniques).

ii) *Damage confinement and assessment*: error detection mechanisms try to attain the maximum possible coverage of error detection. There is however a *detection latency* delay, which corresponds to the interval of time since the manifestation of a fault until the detection of an error caused by that fault, during which, damage can be spread throughout the system. The longer the detection latency, the greater the possibility and the extent of damage proliferation. Therefore, although error detection mechanisms are capable of identifying the manifestation of a fault, they cannot guarantee that all of the unwanted consequences of a fault are detected. It is necessary to adopt strategies for damage assessment, before any error recovery can take place. Damage is normally spread via the flow of information (data and control signals), therefore damage assessment must be based on assumptions about the structure of the system. In order to have damage proliferation minimised, the system must be structured in such a way that the flow of information is well defined and controlled.

iii) *Error recovery*: mechanisms for error detection and damage confinement do not change the system in any way. Thus, these mechanisms alone

do not allow systems to actually tolerate faults and their consequences. Hence, there is a need to introduce active mechanisms which modify the system after an error is detected, such that the system is still able to deliver its service despite the manifestation of faults. Mechanisms to implement error recovery are concerned with the elimination of errors in the system state. The basic idea behind error recovery is to transform an erroneous state of the system into a valid one, from which the system can continue to provide its service. There are two basic strategies to implement error recovery mechanisms, namely *forward error recovery* and *backward error recovery.* Forward error recovery tries to undo the damage introduced by a fault into the system state, so that after recovery the system state is the same as it would have been had the manifestation of a fault not happened. Contrarily, backward error recovery mechanisms are based on the restoration of the state of the system to a previously known valid state.

iv) *Fault treatment and continued system service*: error detection, damage assessment and error recovery mechanisms have the objective of ensuring that any error introduced to the system due to the manifestation of a fault is removed. This prevents the immediate danger of a failure. However, this may not be enough to ensure dependability, since those mechanisms only deal with the symptoms produced by the manifestation of a fault, rather than with the source of the fault. Therefore, although in some situations error treatment mechanisms can cope successfully with faults (e.g. when the recovery mechanisms are powerful enough to deal with recurring faults; or the future operation of the system avoids the fault; or the fault is transient), in many other situations the recurrence of a fault can cause the system to fail. This happens either because the fault becomes more and more serious, or because the recovery operations demand so much work to be carried out, that the system is unable to deliver

its proper service. Fault treatment mechanisms try to eradicate faults from the system so that its service can be sustained, despite the manifestation of faults. Fault location, components repair and system reconfiguration are the main procedures associated with fault treatment.

Design decisions on the amount and type of fault tolerance mechanisms to introduce into a fault–tolerant system are subject to cost considerations. It is necessary to weigh the cost of these mechanisms against the cost of system malfunction, where system malfunction is translated as incorrect computation and/or system downtime. In conventional mainframe systems, where only a small number of applications have dependability requirements, the introduction of fault tolerance mechanisms is seldom cost effective. These systems normally rely on simple fault avoidance techniques and only a very limited amount of fault tolerance mechanisms to increase their dependability. On the other hand, in many safety–critical systems for instance, incorrect computation can lead to immense social and/or economical penalties, therefore these systems normally incorporate a great deal of fault tolerance mechanisms. Moreover, as will be shown later in this chapter, the actual mechanisms used by different systems vary considerably, depending on the requirements of the applications executing on each particular system.

In this way, the purpose of a fault–tolerant computer system dictates the amount and the goals of fault tolerance that must be introduced, such that the system can provide application programs with the degree of dependability they require. Also, the more information that is available about the applications, the more accurate is the choice of which fault tolerance mechanisms to use. In other words, designing specific purpose fault–tolerant systems, where the characteristics of the applications are likely to be known in advance, allows the evaluation of the cost/benefit relation associated with fault tolerance to be carried out in a much simpler and more precise fashion than when evaluating the same relation for a general purpose fault–tolerant architecture, where little is known about the comportment of applications.

Next we discuss the design of various fault–tolerant computer systems described in the literature, and study how fault tolerance mechanisms have been used to implement both specific and general purpose fault–tolerant systems.

2.2.1. Specific Purpose Systems

Specific purpose fault–tolerant computer systems have been designed to cope with a diversity of applications such as spacecraft control [Rennels 78, Larman 83], commercial aircraft flight control systems [Hopkins et al. 78, Wensley et al. 78], process monitoring in industrial installations [Siewiorek et al. 78a, Siewiorek et al. 78b, Smith 84, Kopetz–Merker 85, Lala 86], telephone electronic switching systems [Toy 78, Toy–Gallaher 83], on–line transaction processing [Bartlett 81, Bernstein 88, Webber–Beirne 91] and many others. It is possible, however, to divide these systems into three distinct categories, each one possessing well defined goals.

The first category is composed of those systems whose applications require system's downtime to be kept to a minimum, although a certain degree of incorrect operation is acceptable. For instance, a typical requirement for a telephone electronic switching control system is that system's downtime do not exceed 2 hours during a period of 40 years of system's operation, though the interruption of a small number of on–going calls can be accepted. Those systems are therefore classified as *highly–available* systems.

On the other hand, *long–life* systems form another category of system, whose applications can sustain longer periods of downtime, provided that the system eventually delivers its correct service. A main requirement of long–life applications is that they must survive their mission lifetime without the need of system maintenance. Unmanned spacecrafts control and submarine prospecting, are examples of long–life applications where maintenance is either impossible or too expensive, and where the correctness of the service delivered is crucial, although intervals of unavailableness (e.g. due to system reconfiguration) are acceptable.

Safety–critical systems comprise the last category of special purpose systems. Safety–critical applications require that the system deliver its correct service with high degree of reliability, since the effect of a malfunction can be catastrophic. A great many safety–critical systems are associated with real–time applications, therefore, for those systems, recovery must be achieved with little, if any, degradation on system performance. Commercial aircraft flight control, railway switching control and industrial processes monitoring are examples of applications whose requirements can only be met by safety–critical fault–tolerant computer systems.

2.2.1.1. Highly–Available Systems

This class of systems is represented mainly by telephone electronic switching control systems and on–line transaction processing systems such as airlines reservations and bank accounting. Those systems are normally structured around a basic error detection mechanism at the processing unit, which can be realised either by self–checking logic based mechanisms or duplication and matching mechanisms. Following the detection of an error, indicated by an interruption for instance, the processing unit is disabled. Error recovery is attained by a variety of mechanisms which differ considerably from system to system. On the other hand, continued service is generally achieved via the use of standby spares. Since there are no strict requirements on the timeliness with which the service is provided, a performance decrease whilst recovery and reconfiguration are executed is acceptable; therefore, there is no need to keep spare modules in close synchronism.

One of the first highly–available systems to be developed was the AT&T ESS family of telephone electronic switching control systems [Toy 78]. These systems were first introduced in 1965, and since then, three generations have been built. The differences between each generation is very much a function of the advances on the technology of components, and the variations on system goal. The 3B20D architecture [Toy–Gallaher 83] for instance, was designed for a broad range of AT&T applications, and marked the beginning of the third generation of ESS highly–available systems.

The 3B20D is a duplex processor architecture with extensive use of self–checking hardware to detect errors. The promptness of error detection guaranteed by the massive utilisation of self–checking logic minimises the possibility of damage proliferation. At each time, only one processor is operational, however, special fault–tolerant memory update circuits are used to guarantee that write operations are executed simultaneously in both memories. If an error is detected, a switch circuit is activated, which transfers control from the faulty processor to the spare, making the latter operational. Since the processors are not executing in synchronism, an initiation sequence to load the on–board data of the processor (e.g. registers) with correct information is required, so that service is continued properly. Additional fault tolerance mechanisms include the

use of 4 parity bits for the 32–bit data paths, error correction codes for memory units, mirrored disk units equipped with self–correcting cyclic redundant code and a timing check (sanity check) for processing units, which cause automatic reset of operational processors in the event of time–out expiration (correct operational processors are designed to reset the timing check at predefined intervals of time).

A similar design is found on the Tandem family of NonStop highly–available systems [Bartlett 81]. These systems are targeted to on–line transactions processing (OLTP) applications. Fault confinement is achieved through the use of a loose synchronised multicomputer, where individual computers are connected to a dual bus system through carefully designed bus interfaces. The interface to the bus incorporates self–checking logic, and is implemented in such a way that a faulty computer cannot inject spurious signals into the bus through the interface. Processing units also use self–checking logic to detect errors. Recovery is implemented via backward error recovery mechanisms which restore valid states previously saved in well defined checkpoints. Application processes are duplicated and executed at two different computers. One of the processes in the pair is assigned to be active, whilst the spare process receives periodic checkpoints from the active process. If the spare process detects the failure of its active copy, then service is resumed from the last checkpoint saved.

Two other highly–available systems providing support for OLTP applications followed Tandem's NonStop systems. The Stratus architecture [Webber–Beirne 91] is a loosely coupled multicomputer system. Each Stratus system consists of up to 32 modules connected via a proprietary inter–module link. Each module is formed by a number of processing, memory and I/O channel units, connected via a reliable bus. Each processing unit is composed of two conventional processors which are driven by a common clocking source. The two processors execute in *lock–step* and have their output compared by a reliable comparison circuit. This first level of duplexing provides error detection. Once a mismatch is detected by the comparison mechanism, the faulty processing unit is disabled, resulting in no more signals being output to the bus. Two processing units are then coupled together in tight synchronism to provide continued service despite the failure of one of the processing units.

The Sequoia system [Bernstein 88], on the other hand, is a tightly coupled multiprocessor system. Duplicated processing elements (PEs), duplicated I/O elements (IOEs) and memory elements (ME) are connected by a duplicated bus through master/slave bus interfaces. PEs and IOEs execute in lock–step, accomplishing error detection in the usual duplication and matching way, via a special comparator circuit at the bus interfaces. Once a mismatch is detected, the faulty PE is isolated from the system (a similar mechanism is used for the IOEs). MEs incorporate a powerful error correction mechanism based on an extended Hamming SEC–DEC code [Siewiorek–Swarz 92], which is also able to tolerate failures of the encode/decode circuit itself. Error recovery is attained through software protocols based on process pairs and checkpoints, similar to those used in the Tandem architecture. Mirrored disks are used to provide fault–tolerant I/O operations to the file system.

2.2.1.2. Long–Life Systems

A typical long–life system is encountered in space exploration missions. In those applications it is common to have most of the useful collection of data being realised near the end of the system's mission, thus it is imperative to have the system operative at that stage. Since component repair is not easily accomplished remotely, the duration of the mission dictates the amount of spare units with which the system must be equipped, so that it will survive the mission lifetime. It is important to notice that the designers of such systems must take into account all aspects of the system, and not only the computational part. In a spacecraft for instance, the craft structure, weight, power supply sources and data communication channels are other aspects where reliability constraints must also be imposed.

The Galileo spacecraft [Larman 83], for instance, has been designed to perform a two stage mission to collect data from Jupiter. The spacecraft is composed of an orbiter vehicle and a probe vehicle. The first part of the mission initiates when the spacecraft is around 150 days away from Jupiter, and consists of the release of the probe vehicle into Jupiter's atmosphere. The orbiter vehicle then starts acquiring data from the probe. In the second part of the mission, the orbiter vehicle goes into orbit about Jupiter for a period of 20 months, during which it explores the sur-

rounding space environment and stores data for future analysis. (The same data is also transmitted to the Earth base.)

The basic mechanism for fault tolerance in the Galileo spacecraft is the use of duplicated modules equipped with extensive self–checking logic. The main processing unit is actively redundant, i.e. processing is carried out on both processors, executing in tight synchronism. Once a failure is detected (via the self checking logic), the faulty module is isolated, whilst the functioning module can continue operation without disruption of the system service.

Characteristics of the application software are used to implement reasonableness checks. Also, parity checks are heavily used, as well as periodic diagnostic checks. Due to the unknown, and possibly hostile, characteristics of the environment, persistent checks based on retries are carried out consistently. A component is only considered to have failed if a failure is observed throughout a predefined interval of time, or alternatively, after a predefined number of unsuccessful retries have been attempted. There is also the possibility of a remote intervention of the maintenance personal at the Earth base to deal with unanticipated conditions.

System operation is divided into critical and non–critical operations. If an error is detected whilst the system is executing a non–critical operation then the operation in hand is aborted. The system is then placed into a safe state with minimum power consumption and thermal stability. (This includes internal temperature control, as well as the use of shades to protect external equipment from sun rays.) Finally recovery is attempted. Most of the cases recovery consists of simply retrying the erroneous operation, or skipping to the next operation. Detection of errors during the execution of critical operations is performed in a different way. The active redundancy of the processing unit guarantees the provision of services after the failure of one of the processors. Recovery from a second failure is attained through rollback techniques followed by retry.

2.2.1.3. Safety–Critical Systems

The strict time requirements of most safety–critical applications demand that fault tolerance must be attained with no disruption of system service. Hence, safety–critical systems are generally built with failure–masking mechanisms. A classical example of this kind of systems is the C.vmp system [Siewiorek et al. 78a, Siewiorek et al. 78b]. It consists of a triplicated as-

sembling of conventional processors executing in lock–step, and a hardware voting mechanism which controls access to a replicated bus. All information retrieved (stored) from (into) memory is voted, and the reliable voting mechanism masks the failure of one processor and one memory module. In fact, since the voting is performed in a bit–by–bit fashion, there are more combinations of faulty components that can be tolerated. For instance, if memory is organised in such a way that every bit of a particular word is stored in a different chip, the system can tolerate the simultaneous failure of memory chips in the three computers, provided that failures in different computers do not affect the same bit of a particular word.

Among the first safety–critical architectures to be implemented were FTMP, from the C.S. Draper Laboratory [Hopkins et al. 78], and SIFT, developed by the SRI Laboratory [Wensley et al. 78]. Their design goals are very similar – both were designed to be used by commercial transport aircraft, nevertheless, their designers have followed two very distinct approaches. Both FTMP and SIFT are multiprocessors of arbitrary size. Processing modules are organised in triads to perform tripled redundant functions and mask the failure of one processor. The basic difference between the two systems is on the way the replicated computation performed by each of the processors forming a triad is synchronised, and the way replicated outputs are voted, so that failures can be masked.

In FTMP voting is achieved via a hardware circuit which implements a reliable voted access to the bus. Processors forming a triad must therefore execute in lock–step. A fault–tolerant clock synchronisation algorithm implemented by a hardware circuit provides a common time frame for the processors forming a triad. On the other hand, in SIFT processors do not need to maintain tight synchronism, further, voting is achieved via a software mechanism. Each processor in a triad is able to read (but not write) the memory of the other processors and vote on relevant information deposit in predefined locations. Processes need only maintain a loose synchronism, which allows the executive software to guarantee that information upon which voting is performed is available at the required time at each correct processor. SIFT voting mechanism is highly dependent on the cyclic nature of SIFT's application programs. Loose synchronisation is attained through a software fault–tolerant clock synchronization protocol based on message exchange. The voting mechanism in both systems is able to detect a faulty processor. If a failure

is detected then reconfiguration is performed to restore the failure–masking property of the system. (In FTMP there are 14 processors, which are arranged in 4 triads, leaving 2 processors as spares.) The interval of time between error detection and system reconfiguration is short, reducing the probability of the manifestation of a subsequent non–maskable fault, whilst reconfiguration is being performed.

Other examples of safety–critical systems are the 'fly–by–wire' system control of the airbuses A320, A330 and A340 [Brière–Traverse 93]. These systems have very rigourous requirements in terms of both reliability and availability. Their design is based on special processing nodes, which are composed of two separate processing channels. One of the processing channels is responsible for executing the system functions, whilst the other is responsible for monitoring the correct operation of the former. Each channel, including its design and software, is completely independent from the other. These nodes are replicated in order to achieve the required availability. Spare nodes are kept in close synchronism with the active nodes, so that reconfiguration is realised with almost no delay. Flight control computers must be especially robust, thus, their design incorporate protection against over–voltages, under–voltages, electromagnetic aggressions and indirect effects of lightning.

2.2.2. General Purpose Systems

The variety of applications that can be executed on general purpose computer systems rules out the utilisation of application attributes in order to choose the appropriate fault tolerance mechanisms to be used. Therefore, designers must be much more careful when introducing fault tolerance mechanisms into general purpose computer systems. Ideally, applications should be able to receive dependable services on an application–by–application basis, so that an application that does not require any degree of dependability would not incur any extra penalty, either in performance or in system cost, whilst at the same time other applications with dependability requirements would be able to co–exist, receiving dependable services, and with an overhead corresponding to their dependability demands.

The difficulty in achieving this idealised scenario has lead to less ambitious, compromising solutions. Traditional mainframes, for instance, possess only a restricted number of built–in fault

tolerance mechanisms, and indeed represent the minimal standard to which any fault–tolerant system should aspire. On the other hand, the natural replication of resources found in parallel, and distributed systems can be exploited to provide much better solutions to applications with dependability requirements, raising only a limited overhead to the remaining applications not requiring dependable services.

Next, we analyse fault tolerance mechanisms used by several general purpose fault–tolerant systems described in the literature.

2.2.2.1. Mainframes

Unlike the systems presented before, mainframes such as the VAX 8600 from DEC and the 3090 from IBM, primarily use fault avoidance techniques to attain better reliability levels. Fault avoidance techniques, as discussed before, try to avoid system failure, hence they are mainly concerned with the manufacturing process of the system, and the environmental conditions to which the system is exposed. Thus, most mainframes are constructed with high quality, low intrinsic failure components, and must operate in a friendly and well controlled environment, where power fluctuations are avoided, and temperature is kept within specified limits. Apart from this, only a small number of fault tolerance mechanisms are introduced.

A mainframe can be divided into three main sections: central processor unit, memory (possibly divided into main memory and cache memory) and I/O channels. Since each section has its own attributes, fault tolerance mechanisms added to the system vary in accordance with the section where they are inserted. Table 2–1 below summarises the most common error detection mechanisms used in each section of a mainframe.

Mainframe section	Error detection mechanism
main memory	double–error–detection code on data; parity on addresses and control
cache memory	parity on data, address and control
central processor	parity on data paths and control store; duplication and matching on control logic
I/O processors	parity on data and control

Table 2–1: Common error detection mechanisms for a mainframe

The design goal is to tolerate transient faults, hence the extensive use of coding. The basic recovery mechanisms is retry after the detection of an error. If an error has been caused by a transient fault, a small number of retries inter–spaced by a predefined delay is sufficient to bring the system back to normal operation. Memory units are generally equipped with error correction codes for data.

Usually there is no mechanism to allow continued service after the manifestation of permanent faults. System service is resumed only after maintenance is performed. The information collected by error detection mechanisms is logged, and can be used to minimise the systems' mean time to repair. Further, diagnostic programs can be automatically executed to anticipate system breakdown. Also, remote access can be granted to maintenance personal to allow the remote execution of diagnostic programs. Based on data gathered by the diagnostic programs, a field service engineer can then be sent with appropriate information and spare equipment, to rapidly substitute the faulty component and allow the resumption of system's operation.

2.2.2.2. Parallel and Distributed Systems

In general purpose systems, replication of components for the purpose of fault tolerance is not cost effective. As discussed before, fault tolerance mechanisms incorporated within general purpose centralised systems can cope with transient faults, but have only a very limited impact upon the manifestation of permanent faults, especially when those faults affect processors. On the other hand, in parallel and distributed systems, multiple processing units are a natural part of the architecture design. A number of fault–tolerant systems have been designed taking advantage of this inherent property of parallel and distributed systems.

FTPP [Harper et al. 88], for instance, is a fault–tolerant parallel architecture developed at the C.S. Draper Laboratory, which can provide high availability and high throughput, on an application–by–application basis. The system is composed of a number of processors connected to each other via communication paths. High throughput is achieved by structuring application programs as a set of co–operative processes which execute in parallel, and communicate via message passing. Reliability is achieved through *active replication* of processes. Active replication consists of having replicas of processes executing simultaneously at different processors, which fail

independently. Replicas of a particular process must be provided with identical input messages upon which they perform identical computation, and produce identical output messages. Output messages are subjected to a voting mechanism in order to mask failures. Once a failure is detected, the process replicas executing at the faulty processor are re–allocated to another correct processor, and brought back into synchronism with the remaining replicas. This operation allows system reliability to be re–established. Faulty processors are isolated and subjected to fault diagnosis checks. Depending on the outcome of these checks, they are either re–integrated into the system (e.g. when the fault was a transient), or reported as faulty, becoming non–operational until repair is performed.

Fault–tolerant distributed systems are also organised in a very similar way. In most distributed system architectures, distributed applications are composed of a set of co–operative processes normally executing at different hosts, and communicating via messages. An example of fault–tolerant distributed system can be found in the Delta–4 project [Powell et al. 88, Powell 92], part of the European Strategic Programme for Research in Information Technology (ESPRIT). The aim of this project is to define an open, fault–tolerant, distributed computer architecture, where existing proprietary heterogeneous computer systems can co–operate, and use dependable services on an application–by–application basis. A Delta–4 system consists of a number of, possibly heterogeneous, host computers interconnected by a dependable communication system. Distributed applications are implemented by a collection of processes executing at possibly distinct hosts, and which communicate exclusively through message passing. Again, fault tolerance is achieved through the replication of application processes in separated hosts, which fail independently. In Delta–4, replication can be either *active* [Chérèque et al. 92], *semi–active* [Barrett et al. 90] or *passive* [Speirs–Barrett 89].

Replicas executing in active replication have their output subject to majority voting, thus there is no need to make any assumption on the failure mode of hosts, i.e. they can fail arbitrary. At least $2\pi+1$ replicas are required to mask up to π failures. All replicas receive the same input messages, process them, and produce output messages which are voted before being made available to other applications. As discussed before, it is necessary to guarantee that replica processes receive input messages in the same order, so that they can produce comparable output messages. In

Delta–4, processors communicate with each other through a dependable communication system. Special hardware components called Network Attachment Controllers (NAC) are used to connect individual hosts to the communication system. NACs have *fail–silent* semantics, i.e. in the event of a failure they simply stop, rather than perform an arbitrary transition, thus providing fault isolation. A multicast protocol is implemented on top of the NACs and the dependable communication system, which guarantees the ordering requirements for active replication.

When replicas are executing in either semi–active or passive replication, there is only one replica producing output at each time. Therefore, replicas in both semi–active and passive replication are assumed to execute in fail–silent hosts, and at least $\pi+1$ hosts are required in order to achieve resilience of up to π failures (i.e. replication is used only for the purpose of increasing the availability of the service). Passive replication is based on the utilisation of checkpoints. One of the replicas is designated to be the primary active copy, whilst the other replicas are passive secondary copies. The primary periodically sends checkpoint messages to the secondaries, so that in the event of a failure of the primary, one of the secondary copies can be designated to assume the primary role and start processing from the last checkpoint received. In semi–active replication there is also the notion of primary and secondary replicas; the difference is that secondary replicas also receive and process input messages (but do not produce output), thus keeping in synchronism with the primary without the need of processing checkpoints.

The decentralised characteristics of a distributed system make designing these systems a much more difficult task than that of designing centralised systems, especially when the distributed system must provide dependable services in the presence of processing sites and communication failures. Ideally, one would like to construct a distributed system using hardware components which are guaranteed to be either failure–free or to have well defined failure modes. Indeed, in order to facilitate their design, a large number of distributed systems reported in the literature (e.g. [Bartlett 81, Kopetz–Merker 85, Shrivastava 89, Powell 92, Birman et al. 91]) assume that the system's underlying hardware has a fail–controlled behaviour [Laprie 89]. Higher level system software can then take advantage of these assumptions to implement simpler fault tolerance mechanisms.

In the Delta–4 system discussed above, for instance, hosts executing applications in either passive or semi–active replication, as well as the network interfaces (NACs), are assumed to have fail–silent semantics. The MARS system [Kopetz–Merker 85], a distributed system for real–time applications, is another example of fault–tolerant distributed system whose design has been simplified by assuming that the underlying hardware is fail–controlled. In MARS processors are assumed to have fail–silent semantics, whilst the communication medium is assumed to provide a timely and reliable service.

The complexity of the fault tolerance mechanisms implemented at an upper software level of a dependable distributed system reflects the assumptions made upon the underlying hardware where the software is going to be executed. On the other hand, the assumption of which failure semantics to use for the underlying hardware depends not only on the hardware characteristics itself, but also, among other parameters, on the dependability requirements and the mission lifetime of the applications to be executed.

All hardware components must fail eventually, possibly in an unpredictable manner, therefore, if conventional hardware cannot provide, with sufficiently high probability, the failure semantics assumed (for the duration of the mission of the applications they are executing), there is a need to construct processing sites or *nodes*, and communication infra–structure, that do indeed present the fail–controlled behaviour assumed.

Constructing a fail–controlled node from components that can fail in an arbitrary way involves necessarily the introduction of redundant components. One way to do this is by adding to the design self–checking logic such as error detection codes, watch-dog timers, power supply monitors, temperature monitors, etc. A problem with this approach is that achieving high error detection coverage is normally difficult. Furthermore, there is an interval of time between the occurrence of a failure and its detection by a checking circuit, during which the node can potentially present an undesirable behaviour. Both the error detection coverage and the length of the detection delay depend on the amount of redundancy introduced in the form of checking circuits. A great amount of redundant components reduces the detection delay and increases error detection coverage, but at the same time increases the complexity and the cost of the node. In [Reisinger–Steininger 93] a fail–controlled node based on the introduction of self–checking logic is

presented, whose coverage is estimated to be better than 99% [Kopetz et al. 90]. The choice of the amount and the kind of self–checking mechanisms introduced in that node is simplified by taking advantage of a priori knowledge of specific characteristics of the system.

Another approach is to couple redundant processors within a *replicated node*. A fail–controlled replicated node is composed of a number of processors which fail independently. Computation is replicated and executed simultaneously at each processor. By employing a suitable validation technique to the outputs generated by the replicated processors (e.g. majority voting, comparison), outputs from faulty processors can be prevented from appearing at the application level. Error detection coverage in a replicated node is extremely high, since it depends exclusively on the design of a simple validation mechanism. Further, if the communication mechanisms of the system ensures that information is validated before being used at the application level, then damage is confined to the node level even when there is a substantial delay in the detection of errors.

As discussed before, distributed applications are normally structured as a collection of processes which communicate exclusively via message passing. Thus, replicated nodes (where messages are validated before being output) can provide an attractive solution for the problem of constructing fail–controlled nodes for dependable distributed systems. In the next chapter we discuss in more details how to build fail–controlled replicated nodes.

2.3. Concluding Remarks

We have studied the main fault tolerance mechanisms used by some of the fault–tolerant systems described in the literature. We have classified these systems by their purpose, and we have analysed the relation between the system's purpose and the fault tolerance mechanisms used to achieve different criteria of dependability. It was shown that the knowledge of the application characteristics can be used to simplify the design of specific purpose systems. This is not the case for general purpose systems, which normally cannot make assumptions on the characteristics of the applications it will execute.

We have also discussed the design of dependable distributed systems. It was shown that the design of these systems can be simplified by assuming that the underlying processing sites

(nodes) and communication services present a fail–controlled behaviour. Finally, we have presented a brief discussion on how to build fail–controlled nodes. In the next chapter we discuss how replicated nodes can be used to implement a fail–controlled processing infra–structure for fault–tolerant systems.

Chapter 3
Fail–Controlled Replicated Nodes for Distributed Systems

3.1. Introduction

A replicated node is a processing site composed of a number of redundant processors which fail independently. A fail–controlled replicated node offers a service that can be characterised by its operational semantics and its failure semantics (failure mode). The operational semantics corresponds to the standard specification of the node's service, whilst the failure semantics describes the behaviour of the node when up to a bounded number of components failures, which the node is able to tolerate, have occurred. Any behaviour that the node may present, which is not specified by either its operational semantics or its failure semantics, is considered to be exceptional behaviour. Further, the node presents an exceptional behaviour only if the number of components failures that the node experiences exceeds the number of components failures that the node is designed to tolerate, i.e. the bound specified on its failure semantics.

An upper level system can then assess the suitability of using the services of a particular underlying fail–controlled replicated node by analysing the operational and failure semantics of the latter, and by estimating the likelihood of the occurrence of exceptional behaviour. Alternatively, one can build a fail–controlled replicated node whose probability of presenting excep-

tional behaviour is sufficiently small, so that the node delivers, with the necessary probability, its particular fail–controlled behaviour.

In this context, we highlight two groups of fail–controlled replicated nodes. The first group is composed of *failure–masking nodes*. Failure–masking nodes possess failure semantics which is equivalent to their operational semantics, that is, the node still delivers its standard service despite the occurrence of a bounded number of components failures, which are masked. The second group of fail–controlled replicated nodes is formed by nodes whose failure semantics is considered to be *safe* [Laprie 89], that is, after the detection of the failure of any node component, the node neither delivers its standard service, nor unspecified ones, and simply halts. These nodes are therefore referred as *fail–safe nodes*.

Failure–masking nodes can be built based on the concept of *N–Modular Redundancy* (NMR) processing. The basic idea is to build a replicated node with N processors, and to execute the intended task on each processor in parallel. Outputs generated by each task replica are exchanged among the N processors and subjected to a majority voting. The voted result is then output. This technique is termed *active replication*, and the failure–masking node built in this way is named an *NMR node*. So long as no more than a minority of the processors within an NMR node fail, the majority voting mechanism is able to mask the potential failures of a minority of processors forming the node. The design and implementation of this kind of failure–masking nodes have been reported in [Hopkins et al. 78, Wensley et al. 78, Siewiorek et al. 78a, Smith 84, Lala 86, Theuretzbacher 86, Shrivastava et al. 91, Shrivastava et al. 92, Powell 92, Speirs et al. 93].

A *fail–signal node* [Shrivastava et al. 91] is another example of a failure–masking node. The failure semantics of a fail–signal node is similar to the failure semantics of an NMR node, except that in the case of a fail–signal node, as soon as a failure is detected (and masked), the node is able to signal the detection of this failure through a failure exception. If processors cannot be repaired whilst the node is operative, the signaling of a failure indicates a decrease on the nodes actual reliability. Thus, in a conservative design, such a signal can be used for initiating preventive procedures. (For instance, critical computations could be migrated to other nodes which are believed to be more reliable at that particular moment.)

Fail–silent nodes [Bernstein 88, Shrivastava et al. 91, Shrivastava et al. 92, Webber–Beirne 91, Brasileiro et al. 92], on the other hand, are representatives of the fail–safe group of fail–controlled nodes. Fail–silent nodes must satisfy the property of *halt on failure*, which states that whenever a failure is detected, the node will halt instead of performing an erroneous state transformation that will be visible outside the node. This property guarantees their safe failure semantics.

Active replication can also be used to implement fail–silent nodes. Again, the node is composed of N processors where the intended task is executed in parallel, with the outputs generated by each task replica being exchanged among the N processors. In this case, instead of voting the results collected, a fail–silent node simply compares them. A disagreement on any of the outputs forces the node to halt. A replicated node constructed in this way, has a fail–silent behaviour provided that no more than N–1 processors fail.

Fail–stop nodes [Schlichting–Schneider 83, Schneider 84] form another class of replicated nodes which also possess the halt on failure property, and therefore can also be categorised as fail–safe nodes. Apart from the halt on failure property, a fail–stop node presents two extra properties which differentiate them from fail–silent nodes. A distributed system composed of fail–stop nodes must present *failure status* and *stable storage* properties. To satisfy the failure status property, it must be possible for any functioning node in the system to detect the failure of any fail–stop node that has halted. The storage of a fail–stop node is partitioned into stable storage and volatile storage. To satisfy the stable storage property, it must be possible for a functioning fail–stop node to access the data in the stable storage associated with any other node in the system, including those that have halted.

Clearly, implementing a fail–stop node is more difficult than implementing fail–silent nodes. In [Schneider 84] the design of a system of fail–stop nodes is presented. The system is composed of two types of nodes, namely *p–nodes* and *s–nodes*. The p–nodes are fail–safe nodes responsible for task execution, whilst the s–nodes are failure–masking nodes responsible for providing stable storage for the p–nodes. Each p–node has a special location in its stable storage which records its current status. Thus, a p–node can detect the failure of another p–node in the system by reading the latter status indicator stored in stable storage. However, since tasks execut-

ing on p–nodes communicate with each other via the stable storage kept by the s–nodes, fail–stop nodes are not an attractive solution for implementing fault–tolerant distributed systems. Thus, in this chapter we concentrate our discussion on the design of failure–masking and fail–silent replicated nodes.

3.2. Constructing Fail–Controlled Replicated Nodes

As discussed before, the construction of replicated nodes is based on active replication. Maintaining the synchronisation of the replicas of a task is the underlying principle of active replication. Up to the point where outputs are going to be validated, non–faulty replicas (those executing on non–faulty processors) must have executed equivalent steps, so that the validation can be carried out over consistent outputs. Provided that all correct replicas start execution from the same initial state, an active replicated node must incorporate mechanisms to keep replicas synchronised, avoiding replica divergence between two subsequent validation points. Inconsistent input data, asynchronous events (e.g. interruptions, timeouts) and intrinsic non–deterministic behaviour are some of the potential causes of replica divergence.

When replicas of a task input data from a faulty input source, it is possible that each replica may input a different data value. This problem is referred to in the literature as the *input problem* [Krol–van Gils 85]. Inconsistent input data can lead to a situation where replicas, although executing correctly, output different results. The masking/detection mechanism of a replicated node obviously has no way to differentiate this situation from one where a failure has occurred. Solutions to this problem depend on the type of input source, and whether the input source is replicated or not.

Let us take first the case where a single source is providing data for all the replicas. There are two fault situations here. In the simpler case, a faulty input source can feed replicas with erroneous data, but the data input by the replicas is consistent amongst all non–faulty replicas. In this case, since all replicas receive the same input data, they do not diverge. There is however another fault scenario which requires more elaborated solutions. In this case, apart from producing erroneous data, a faulty source can also broadcast inconsistent data to different replicas. For instance, take a replicated node where replicas independently receive input data from a common non–repli-

cated input device. If each replica samples the input device at different instants, it is possible that replicas input different values. Even if the input device broadcast data simultaneously, it is possible that ambiguous values are broadcast by a failing bus driver. (Note that no two sampling instants, nor two logic discrimination levels are exactly the same.)

To solve the input problem, replicas need to exchange information about the data they have input, and achieve a consensus on which input value to use. A solution to the input problem must guarantee that: i) each non–faulty replica agrees on the same data value to input; and ii) if the input source is non–faulty, the data value input by all non–faulty replicas is the one produced by the input source. This problem is an instance of the widely studied *interactive consistency* problem [Pease et al. 80] (also known as the *Byzantine Generals* problem [Lamport et al. 82]). An algorithm that solves the Byzantine Generals problem is said to have reached *Byzantine agreement.* Hence, to solve the input problem, replicas must reach Byzantine agreement on the data value to input.

Another problem that can potentially cause replica divergence is that of selecting input messages for processing in a distributed system. In these systems, application programs are normally structured as a number of co–operative processes communicating via messages. In order to avoid replicas from diverging, a replicated node must incorporate mechanisms to guarantee that all correct replicas of a particular process input the same set of messages in the same order. As will be studied later in this thesis, a solution for this problem can also be attained by using protocols which implement Byzantine agreement.

A number of theoretical results, within a variety of system environment assumptions, have been derived for algorithms which provide Byzantine agreement. In asynchronous environments, for instance, it has been proved that, provided that at least one processor can fail, there is no deterministic algorithm which achieves Byzantine agreement [Fischer et al. 85][1]. On the other hand, within synchronous environments, where there is a known bound on both the difference between

1. The result in [Fischer et al. 85] assumes that processors have a fail–stop failure semantics. Since our assumptions on the processors' failure semantics are weaker than fail–stop, the result also applies to our work.

the execution rate of processors and the communication delay within which information is exchanged between any two non–faulty processors, it is possible to have deterministic algorithms that achieve Byzantine agreement. A considerable number of such algorithms have been reported in the literature (see [Barborak–Malek 93] for a survey).

Any deterministic algorithm which achieves Byzantine agreement shall perform a number of rounds of information exchange, after which agreement is achieved. The following theoretical lower bounds have been derived, where N is the number of processor replicas, and π is the maximum number of processor failures that can be tolerated:

degree of fault tolerance: a Byzantine agreement algorithm must involve more than three times the number of potential faulty processors, i.e. $N \geq 3\pi+1$ [Pease et al. 80];

replicas connectivity: to achieve Byzantine agreement, replicas must be able to exchange information via at least $2\pi+1$ disjoint paths [Dolev 82]; and

number of rounds: to achieve Byzantine agreement, there must be at least $\pi+1$ rounds of information exchange [Fischer–Lynch 82].

The considerably large values of the above bounds are mainly due to the ability of a faulty processor to 'impersonate' a non–faulty one. These bounds can be reduced if this particular behaviour of faulty processors can be somehow constrained. A well known way to do this is by using authentication techniques when exchanging information. With the help of appropriate techniques, a non–faulty processor is able to generate a unique, unforgeable, information dependent signature which is attached to the information it transmits to the other processors of the node. Further, every non–faulty processor is able to verify the authenticity of the information transmitted to them. Fail–arbitrary processors with such capabilities have an *authentication–detectable fail–arbitrary* failure semantics [Strong et al. 90]. That is to say, if authentication is used, processors may fail in an arbitrary way, except that they are not able to forge (without being detected) information that have been generated by a non–faulty processor, which they relay to other processors. The use of digital signatures techniques [Rivest et al. 78, Okamoto 88] provide the means to achieve such a failure mode from originally fail–arbitrary processors.

The utilisation of this stronger failure assumption allows much simpler and cheaper algorithms to be implemented. For authentication–detectable fail–arbitrary processors, the bounds on the degree of fault tolerance and replicas connectivity, are reduced to $N \geq \pi+1$ and $\pi+1$, respectively [Lamport et al. 82], whilst the bound on the number of rounds remains the same [Dolev–Strong 83].

Another potential cause of replica divergence is related with the way asynchronous events are handled by replicas. For instance, imagine a situation where a task must wait for some event to happen, and then take some action. Further, if the event does not happen within some time, then another action must be taken. In this situation, unless there is some sort of synchronisation between the replicas executing at each processor, it is possible that replicas take different actions. This is also the case when there is a possibility of non–deterministic behaviour within the execution of a task. Non–determinism can be introduced in several ways. It can appear at the application layer, where the computation sequence of a particular application is decided at random when the task is being executed; it can be introduced at the system layer, say through asynchronous interactions with the system environment; or it can even happen at the physical layer, when the hardware design allows non–deterministic transitions to be performed. For all cases above, it is clear that if a task can be correctly executed in different ways, it is possible that task replicas produce different, although correct, results for the same computation. Again, this situation cannot be differentiated from one where a failure has occurred.

To solve this problem, replicas need to be synchronised. Replicated nodes can be divided into two types, according to the way replicas are synchronised. In the first type of nodes, replicas are synchronised at the micro–instruction level, with hardware mechanisms being employed to implement active replication. Hence, we term this type of nodes *hard*[2] nodes. In the other type of nodes, termed *soft* nodes, replicas are synchronised at a higher level (e.g. at the task level), and software mechanisms are used to implement active replication.

---

2. This terminology is in accordance with that used in [Shrivastava et al. 91] to classify fail–silent nodes.

3.2.1. Hard Nodes

The majority of the replicated nodes described in the literature follow a hardware based approach. Synchronisation at the micro–instruction level is the most straightforward way to achieve replica synchronism. In this approach processors are normally driven by a common clock source which guarantees that they execute the same steps at each clock pulse. Outputs are validated by a (possibly replicated) hardware component at appropriate times (e.g. at each bus access). Solutions to the input problem, when applicable, vary from architecture to architecture, but normally involve implementation of Byzantine agreement through hardwired circuits. Interruptions and other asynchronous events must be distributed to the replicated processors through special circuits which guarantee that all replicas perceive the event at the same point of their instruction stream. Since every correct replica executes the same instruction stream, process replicas may incorporate non–deterministic behaviour. Note however that the individual processors must follow a deterministic design, so that they execute the same transitions at each clock pulse [Webber–Beirne 91]. Thus, processors must be carefully designed. In particular, 'don't care' states where bits can apparently harmlessly assume either the value zero or one, are not allowed, since this can trigger replica divergency. Next we discuss some systems which use hard nodes within their design.

Stratus and Sequoia

Highly–available systems like Stratus [Webber–Beirne 91] and Sequoia [Bernstein 88] employ a very simple and cheap design for implementing a hard fail–silent node. In both systems, a fail–silent node is obtained by coupling two conventional processors, which execute in lock–step synchronisation, and whose outputs are compared by a reliable comparator circuit before being made available. To keep processors in tight synchronism, both processors are driven by a common reliable clock. Figure 3–1 shows the block diagram of a Stratus fail–silent node (Sequoia uses a similar approach).

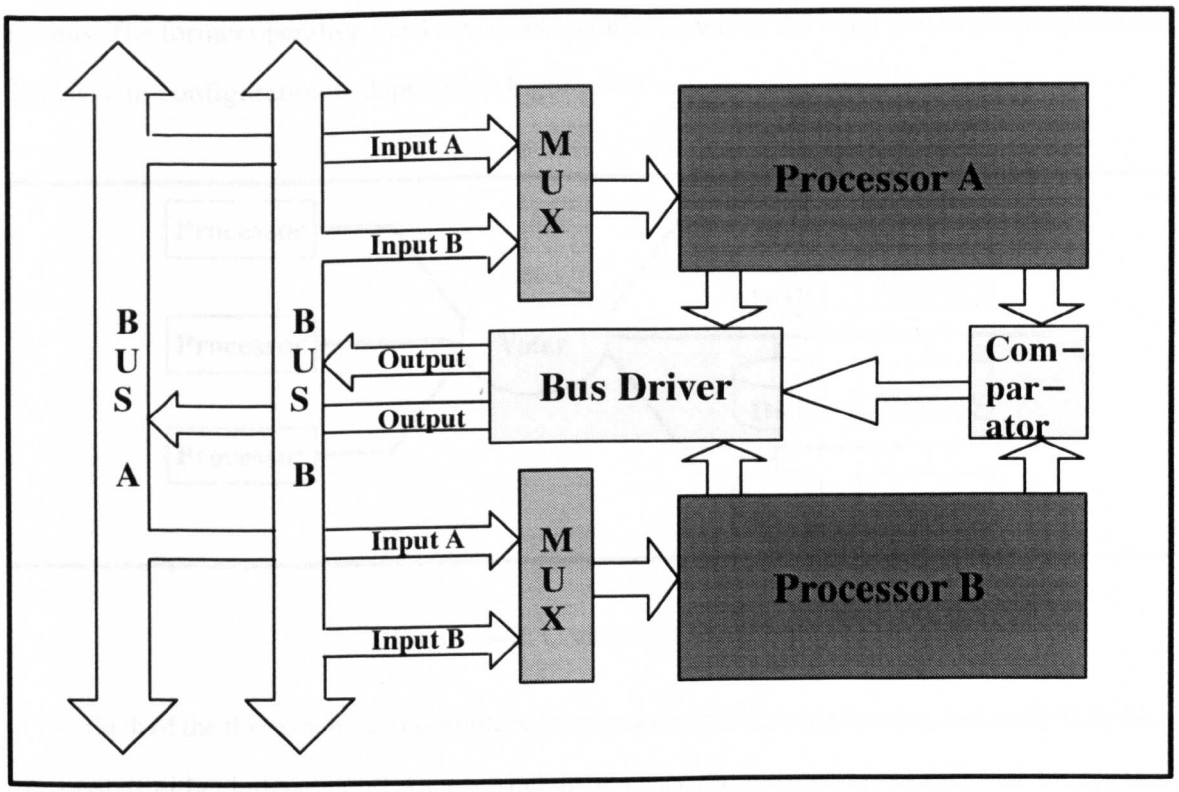

Figure 3–1: Stratus fail–silent node

At every clock cycle, the output generated to the bus driver by each processor is compared, and the bus driver is only enabled to transmit information to the duplicated buses when the result of the comparison operation is successful. When a mismatch occurs, the bus driver is disabled, and no information is sent over the buses. At this point the operating system is informed about the failure (possibly through an interruption), and takes the necessary measures in order to provide system recovery.

C.vmp

The same approach taken to implement hard fail–silent nodes can be taken to implement hard failure–masking nodes. Clearly, in this case, there must be at least three coupled processors, to allow the masking of one faulty processor. The failure–masking mechanism of the C.vmp multiprocessor [Siewiorek et al. 78a] is a triplicated version of the fail–silent node previously described. A C.vmp module is composed of three identical computers executing in lock–step. A single reliable voter circuit executes voting at the bus level. In C.vmp voting is a bidirectional operation, performed both when processors write to the bus, as well as when processors read from

the bus. The former operation masks processors failures, whilst the latter masks memory failures. The system configuration is depicted in Figure 3–2.

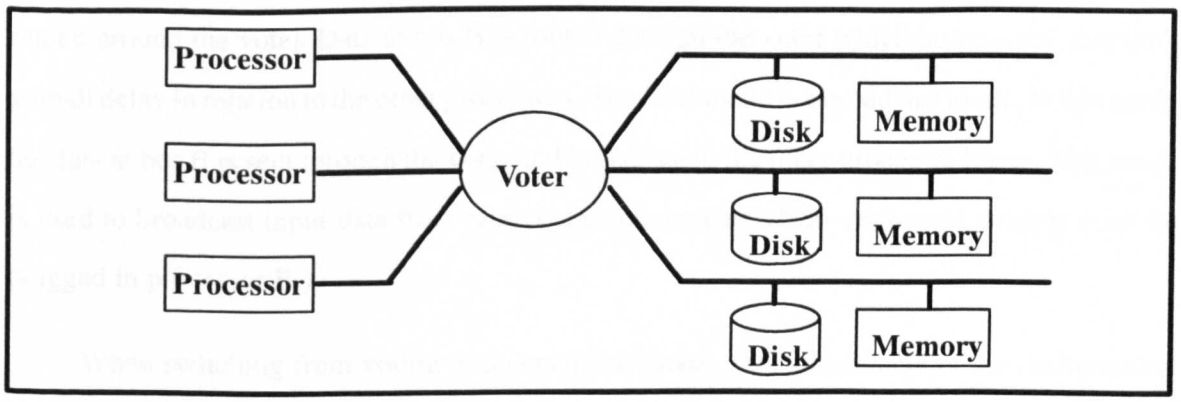

Figure 3–2: C.vmp configuration

Each of the three identical computers, composed of processor, memory, bus and I/O devices can operate either independently or as a triad. In this way, the architecture can provide the applications with a choice between reliability and throughput. When operating at the voting mode, all accesses to memory are directed through the voter circuit (I/O devices based on direct memory access – DMA, must incorporate a similar voter circuit), whilst when operating at independent mode, access to the bus by-passes the voter circuit. C.vmp uses a multiplexing circuit like the one shown in Figure 3–3 to switch between one mode and the other.

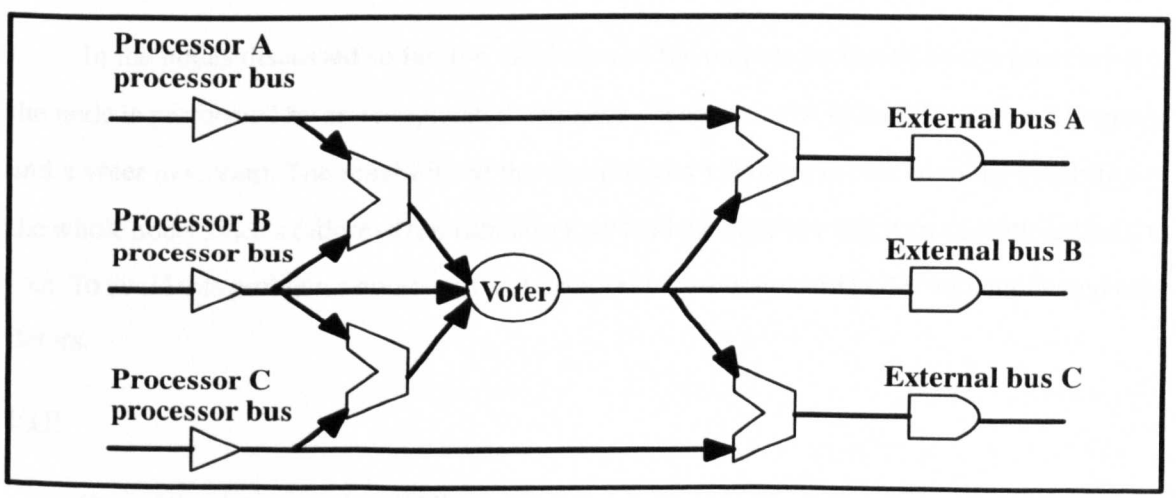

Figure 3–3: C.vmp voter multiplexing circuit

The multiplexing circuit in Figure 3–3 allows C.vmp to operate in three different modes. In the voting mode the data at each individual bus is routed into the voter. The voted result is then routed out to the buses of each processor. In the independent mode the data at buses A and C are routed around the voter. Data at bus B is routed through the voter which outputs the data with a small delay in relation to the other processors. The final mode is a broadcast mode. In this mode the data at bus B is sent through the voter and broadcast to the other processor buses. This mode is used to broadcast input data from non–replicated devices. (Non–replicated devices must be plugged in processor B.)

When switching from voting to independent mode, a simple change in the multiplexing control signals cause the outputs not to be directed through the voter. By providing each processor with a suitable state vector, the operating system can initiate three different processes in each computer. When switching back to voting mode, it is not enough to change the signals in the multiplexing units, so that access to the bus is directed through the voter. Special care must be taken to bring the three processors back in synchronism. Basically, each processor restores the vector state for voting operation, forces an interruption to be executed after some predefined amount of time (large enough to guarantee that the other processors are ready to synchronise), and executes a wait operation. The wait operation halts the processor until an interruption is received. After the interruption, the three processors are back in synchronism, and executing in voting mode.

In the nodes discussed so far, the validation of the outputs produced by the processors of the node is performed by an unreplicated validator – a comparator in both Stratus and Sequoia, and a voter in C.vmp. The reliability of the unreplicated validator is crucial to the reliability of the whole node, since a failure of the validator may lead the node to exhibit an exceptional behaviour. To avoid this problem a number of architectures incorporate hard nodes with replicated validators.

FTP

One of the design goals of FTP [Smith 84] was to make the fault tolerance mechanisms at the physical layer largely transparent to the programs executing at the system and application

layers. Both the application programs and the majority of the system software should execute in much the same way as they would do, if they were executing on a non–replicated architecture. Thus, the utilisation of micro–instruction synchronisation was the obvious design choice. Fault tolerance in FTP is also achieved through triplicating and voting.

FTP was designed for safety–critical applications, and incorporates special treatment to tolerate failures on the data acquiring input devices characteristic of those applications (e.g. aircraft sensors). FTP uses a hardwired circuit to solve the input problem. Figure 3–4 gives an schematic illustration of the circuit used.

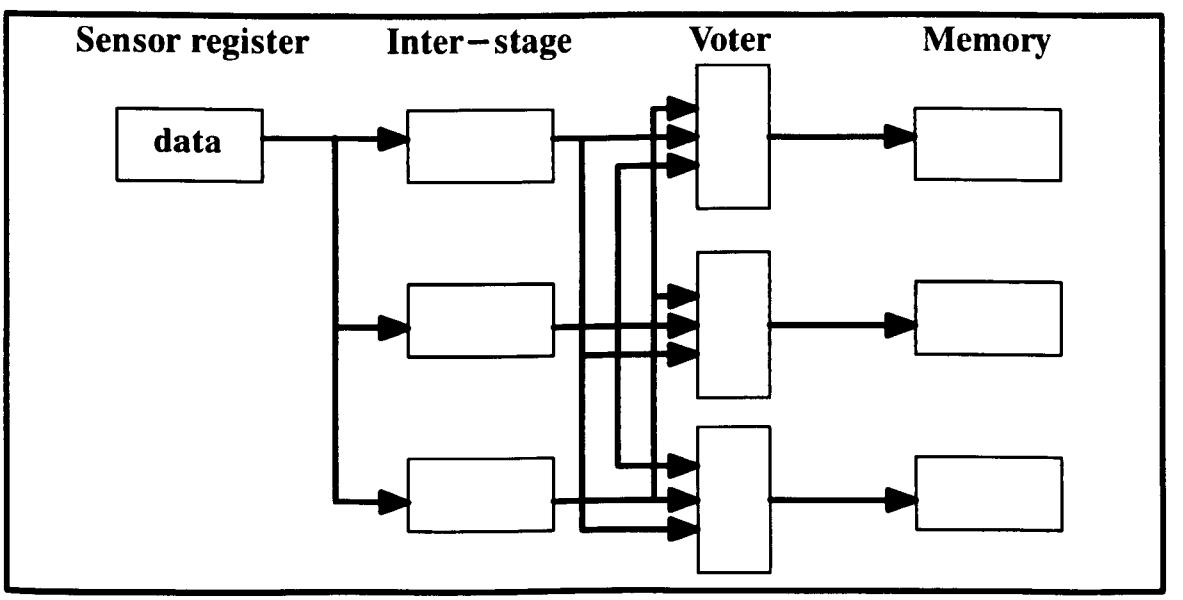

Figure 3–4: FTP input dissemination circuit

Each processor is divided in two fault containment regions. The extra fault containment region in each processor is an inter–stage of trivial implementation and reduced complexity. Data input is a two–phased process. First, the original data available at one of the I/O devices is replicated and distributed to the three inter–stages; then the data available at each inter–stage is passed to each processor, where a vote takes place; finally the result of the vote is used as the input value. This mechanism can tolerate any single fault occurring at either a processor or at an inter–stage. Note that a faulty I/O device, or an error in the transmission of the input value to the inter–stages could generate an incorrect input value. As discussed before, this should be treated at a higher level (e.g. use of replicated I/O devices). The purpose of the mechanism described is merely to

guarantee that all correct processors receive the same (possibly incorrect) input values at the same point of execution.

In a replicated architecture where computers execute in lock–step, each individual processor must be interrupted at the same point in their instruction sequence execution. The FTP architecture guarantees this behaviour through a dedicated hardware distribution circuit. When a processor receives an interruption, this circuit distributes the interruption to all other computers in the system. The circuit, which is also based on the utilisation of inter–stages and voting of control signals, assures that all correctly non–faulty processors are interrupted at the end of the same micro–frame. Spurious interruptions from a single faulty source are masked by the two other sources. Therefore, all correct computers handle the interruption in the same way, and therefore, do not diverge.

FTP differs from C.vmp in many ways. Firstly, unlike C.vmp's unreplicated voting mechanism, the voting mechanism in FTP is also triplicated; secondly the common time base is provided by a fault–tolerant clocking system, as opposed to the single clock source in C.vmp; thirdly rather than having several operating modes, the FTP's architecture is always in voting mode; and finally, in FTP voting only takes place at particular times, explicitly chosen by the executive software (typically embedded within I/O driver routines just before an output operation is executed), whilst in C.vmp voting is performed at each access to the bus (when the system is operating in voting mode).

Other Architectures

Other architectures use more than three processors, in order to implement hard replicated nodes, and provide even greater dependability at the physical layer. Another version of FTP, presented in [Lala 86], uses a quadruplicated architecture, which can also tolerate a single failure. However, after the detection of a permanent failure in one of the computers, the system can be reconfigured, having the faulty processor purged from the voting process, and the remaining three operational computers can tolerate another failure of the surviving computers. Achieving this property implies adding extra complexity to the voting circuit. Also, the complexity of the inter–stage mechanisms used to distributed input values from I/O devices is proportional to the

number of processors used in the assembly. In [Lala 86], alternative approaches to reduce the complexity of the inter–stage mechanism are presented.

Both versions of FTP present a throughput compatible with the throughput of their corresponding simplex architecture. FTMP [Hopkins et al. 78], on the other hand, is a multiprocessor architecture which can provide arbitrary high throughput. Reliability is achieved by coupling processors to build internal replicated nodes. The multiprocessor architecture is arranged in such a way that any three processors can be configured as a triad, and operate in lock–step with their outputs subjected to a voting circuit. This flexibility is only achieved subjected to a considerable increase in the complexity of the system design.

3.2.2. Soft Nodes

Attempting to avoid the complexity involved in designing the special circuits necessary for the implementation of hard nodes, a number of architectures have been designed with the redundancy being controlled via software mechanisms. The idea is to maintain replica synchronism at a higher level, for instance at the process or task level, rather than at the micro–instruction level. The resulting looser synchronisation allows the utilisation of much simpler hardware designs, and more flexible software mechanisms for output validation and replica divergence avoidance.

The input problem is normally solved by the utilisation of software implementations of Byzantine agreement algorithms. Software realisations of Byzantine agreement are generally simpler, and more flexible than their hardwired equivalent. On the other hand, the absence of the tight synchronisation characteristic of hard nodes implies that handling asynchronous events and non–deterministic behaviour of the system is a more difficult task within soft nodes. In this section we study how soft nodes have been incorporated into architectures described in the literature.

SIFT

The SIFT project [Wensley et al. 78] was the pioneer on the utilisation of soft replicated nodes. Like FTMP, SIFT is a fault–tolerant multiprocessor. However, since the control of redundancy is achieved via software protocols, a great deal of flexibility can be incorporated to the architecture without adding substantially to the complexity of the hardware design. Furthermore, the flexibility provided by having voting realised by software protocols, makes it much simpler to

offer a dependable service in an application–by–application basis. The architecture of SIFT is depicted in Figure 3–5.

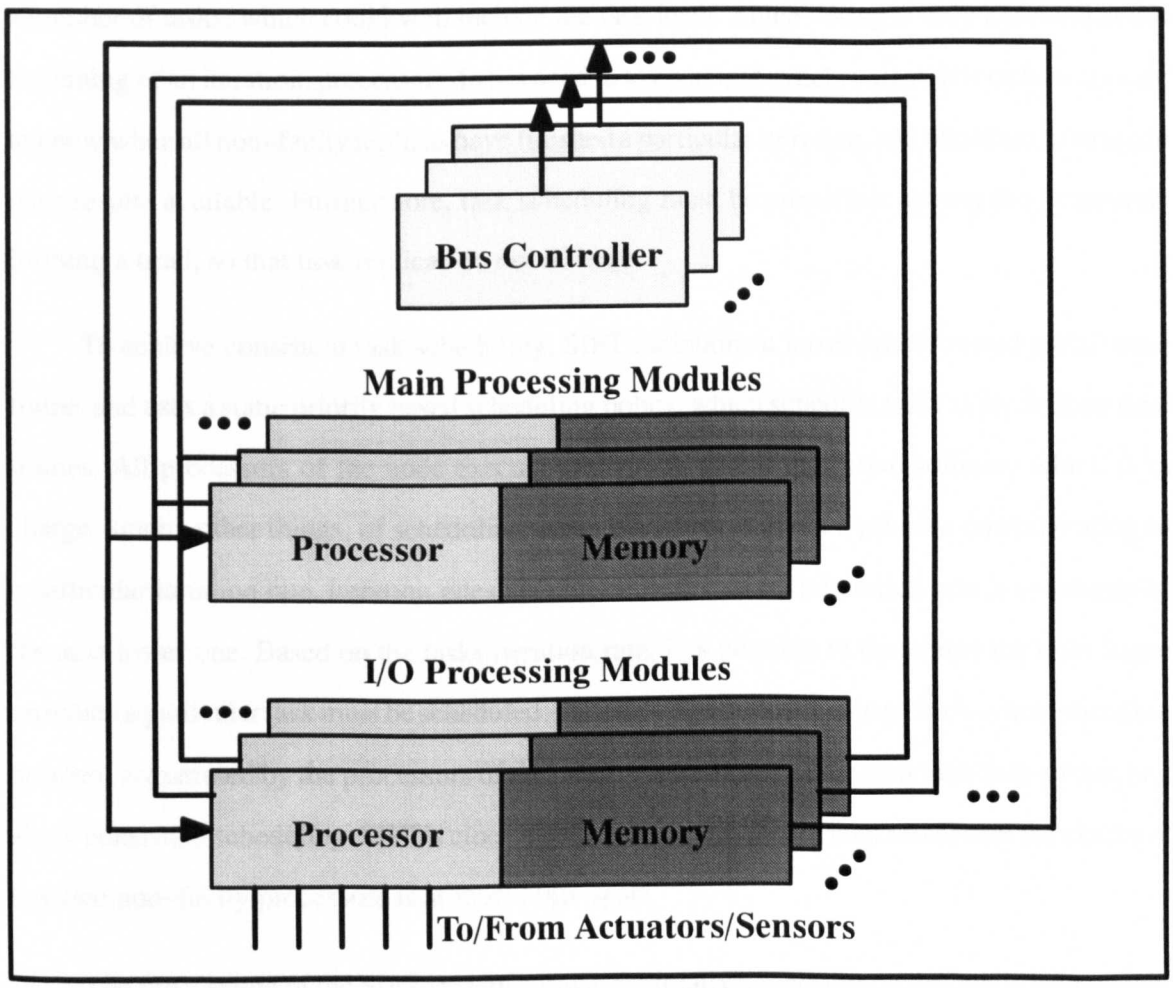

Figure 3–5: SIFT architecture

The system consists of a number of processors and I/O channels, each with its own memory unit, which are interconnected by a replicated bus. Each processor can read and write its own memory, and can also read the memory of any other processor. The system software is responsible for configuring the system in triads, where critical computation is performed. The system software also provides primitives for voting critical information. The voting mechanisms is very simple; the system software executing on each of the processors forming a triad reads the relevant information on the memory of the other two processors, and performs a 2–out–of–3 vote with its local data. The result of the majority vote is then saved in an appropriate location.

Applications in SIFT are structured as a set of co–operative cyclic tasks. At each iteration, a task typically inputs some data, performs some computation and outputs the results. A task inputs data for a particular iteration by voting on the output produced by the previous iteration of a number of tasks, which could well include the task itself. Since voting is only executed at the beginning of an iteration, processors do not need to keep a tight synchronism. It is only necessary to know when all non–faulty replicas have finished a particular iteration, and therefore have made their results available. Furthermore, task scheduling must be consistent among the processors forming a triad, so that task replicas do not diverge.

To achieve consistent task scheduling, SIFT maintains a loose synchronised global time frame, and uses a static priority based scheduling policy, which schedule tasks at predefined time frames. All processors of the node execute a common global executive software which is in charge, among other things, of scheduling tasks. Each task is given a priority, corresponding to a particular iteration rate. Iteration rates are chosen such that each iteration rate is a multiple of the next lower one. Based on the tasks iteration rate, it is possible to determine the time frame on which a particular task must be scheduled. Further, a Byzantine resilient clock synchronisation protocol is executed by the processors of the node to implement a common time frame base, and allow consistent scheduling. SIFT's clock synchronisation protocol guarantees that the clocks of any two non–faulty processors is at most 50μs apart.

The functioning of the SIFT system software is totally dependent on the cyclic characteristic of its applications. Application tasks in SIFT cannot introduce non–deterministic behaviour, although some non–determinism can be partially introduced by synchronising asynchronous events to time frame boundaries. On the other hand, SIFT's loose synchronism provides a potential higher resilience to common failures due to transients.

Voltan Nodes

Because of its application dependent design, the SIFT architecture can only be applied to a very restricted range of applications. This is also the case in the VOTRICS system [Theuretzbacher 86] which follows the design principles of SIFT to provide fault tolerance to a different, but still specific, class of applications (railway signalling systems). In [Shrivastava et al. 92], the

Voltan family of replicated nodes is presented. Their design also follows the approach pioneered by SIFT, but are applicable to a wider range of applications. Using Voltan nodes, it is possible to provide fault tolerance to general purpose applications and yet have the advantages of a task synchronised system.

The design of Voltan nodes follows the state machine approach (where a state machine is a process), for which the precise requirements to achieve active replication are known [Schneider 90]. In this approach, applications are structured as a set of processes which do not share memory, and communicate only through message passing. Asynchronous events, such as timeouts and interruptions are also mapped into messages. Furthermore, processes are assumed to be deterministic. Process replicas exchange the messages they output, and fault tolerance is achieved by validating these messages. To achieve replica synchronism it suffices to provide all correct replicas with the same set of input messages, and in the same order.

Voltan nodes are composed of 'off–the–shelf' processors connected via communication links. The replicated processors execute agreement and order protocols to guarantee that correct replicas of application processes receive and process input messages in the same order. Process replicas exchange output messages produced, which are then validated at each processor by either a comparator software mechanism, in the case of fail–silent nodes, or a voter software mechanism, in the case of failure–masking nodes.

3.2.3. Hard Nodes versus Soft Nodes

In the above sections we have mentioned several advantages and drawbacks associated with both hard and soft nodes. The main advantages of hard nodes are:

i) *minimum performance overhead*: although the hardwired circuits inherent to the implementation of hard nodes may sometimes be relatively complex, they are responsible for only a very small overhead on system throughput (in an FTP prototype reported in [Smith 84] the burden in the throughput due to replication is about 6%); and

ii) *minimum impact on software design*: provided that processors are designed in such a way to avoid non–deterministic transitions, basically

any application or system software which can be executed by a non–redundant machine, can also be executed by a hard node (even those which incorporate non–deterministic behaviour).

However, there are some problems with the micro–instruction synchronisation approach of hard nodes. First, individual processors must be built in such a way that they have a deterministic behaviour at each clock pulse, so that they produce comparable outputs. This can rule out the utilisation of 'off–the–shelf' processors, whose reliability is normally higher than specially designed processors [Siewiorek–Swarz 92]. Second, the introduction of special circuits such as reliable comparator/voter, reliable clock, asynchronous event handlers and bus interfaces, increases the complexity of the design, which at an extreme can result in a decrease on the overall node reliability. Finally, every new microprocessor architecture requires a considerable redesign overhead. In fact, even with the same components technology, substantial redesign is needed when certain parameters of the architecture, for example the level of redundancy, are changed (e.g. [Lala 86]).

The design of soft nodes aims to overcome the problems associated with hard nodes. Thus, the advantages of soft nodes are very much the converse of the drawbacks of hard nodes discussed above. They are:

i) *design diversity*: the absence of micro–instruction synchronism allows the utilisation of 'off–the–shelf' processors; also, by employing different types of processors within a node, there is a possibility that a measure of tolerance against design faults in processors can be obtained, without recourse to any specialised hardware assistance;

ii) *simpler hardware design*: all redundancy management protocols are implemented in software, thus the underlying hardware design can be made very simple; furthermore, software protocols are much more flexible than their hardware counterparts;

iii) *ease in upgrading*: technology upgrades appear to be easy, since the principles behind the protocols do not change; the software protocols can be

ported relatively easily to any type of processor (including the ones expected to be available in the future); and

iv) *common failures robustness*: since replicated computations do not execute in lock–step, a node is likely to be more robust against transient failures [Kopetz et al. 90] (this is because transients are less likely to affect loosely synchronised computations on the processors in an identical fashion).

Unfortunately, these advantages do not come for free. The two main drawbacks of soft nodes are the necessity of programming applications in accordance with the requirements imposed by each particular synchronisation strategy, and the extra overhead in performance imposed by the redundancy management protocols. The first problem is less critical within soft nodes based on the state machine approach, since it is fairly easy to program applications in accordance with that model. Distributed application programs in particular are often build as a set of co–operative processes which communicate only via message passing. There is, however, a major concern over the performance overhead incurred by the redundancy management protocols.

In SIFT, for instance, the overhead associated with redundancy management can consume as much as 80% of the processor throughput [Palumbo–Butler 85]. Hybrid solutions, which incorporate both micro–instruction synchronisation and task synchronisation mechanisms, have been proposed to reduce this overhead. MAFT [Kieckhafer et al. 88], FTP–AP [Lala–Alger 88] and Delta–4 [Powell 92] are hybrid architectures which share the same general structure. These architectures are structured around a micro–instruction synchronised hard core, on top of which conventional processors are replicated. The micro–instruction synchronised hard core is responsible for executing management functions, whilst application processes are executed at the upper level replicated processors. The extra computational power delivered by the replicated processors increases the throughput of the system, and provides all the advantages of the task synchronisation approach, however, the underlying hard core re–introduces the problems associated with micro–instruction synchronisation.

Our approach to the construction of fail–controlled nodes is to follow the state machine model adopted by the Voltan family of nodes, earlier described. We have sought means of optimising the performance of those nodes by developing more efficient redundancy management protocols. The work in this thesis extends that in [Shrivastava et al. 92] by introducing means of implementing efficient, yet suitable for general purpose systems, soft replicated nodes. In particular we present new protocols to manage the redundancy of soft failure–masking replicated nodes which are much more efficient than the ones previously known. We also present the precise definition of the semantics of soft fail–silent nodes, exposing dissimilarities between these nodes and failure–masking nodes, which have not been studied before. We then explore these dissimilarities to develop more efficient protocols for the construction of soft fail–silent nodes.

We have implemented several instances of soft replicated nodes using the different protocols developed. The results obtained after executing a variety of experiments using the nodes implemented indicated the feasibility of using such nodes for a large class of dependable applications. In the next two chapters we describe efficient protocols suitable for the implementation of both failure–masking and fail–silent nodes. Then, in a later chapter, we discuss the implementation details of such nodes, and we analyse the performance of the nodes implemented. First we study the general architecture of the Voltan family of nodes.

3.3. Voltan Architecture

The Voltan family of fail–controlled soft replicated nodes have been reported in [Shrivastava et al. 91, Shrivastava et al. 92]. It includes NMR, fail–signal and fail–silent nodes. The first two nodes are failure–masking nodes, whilst the last is a fail–safe node. Since designing a fail–signal node by extending the design of an NMR node is a trivial task, throughout this thesis we only detail the design of NMR nodes. However, we use the more generic term failure–masking nodes when referring to NMR nodes.

3.3.1. System Model and Assumptions

We assume that non–replicated distributed applications are composed of a number of processes that do not share memory, and interact only via messages. As an example, the function of

a typical 'server' process is to cycle by selecting an input message from any one of its input ports, process it and, if necessary, output one or more messages on its output ports. We also assume that if a process with multiple input ports has input messages available at several of these ports, then any one of these messages is chosen *non–deterministically* for processing. Message selection is however assumed to be *fair*, that is, the process eventually selects a message present on a port. Figure 3–6 shows the basic structure of an unreplicated 'server'.

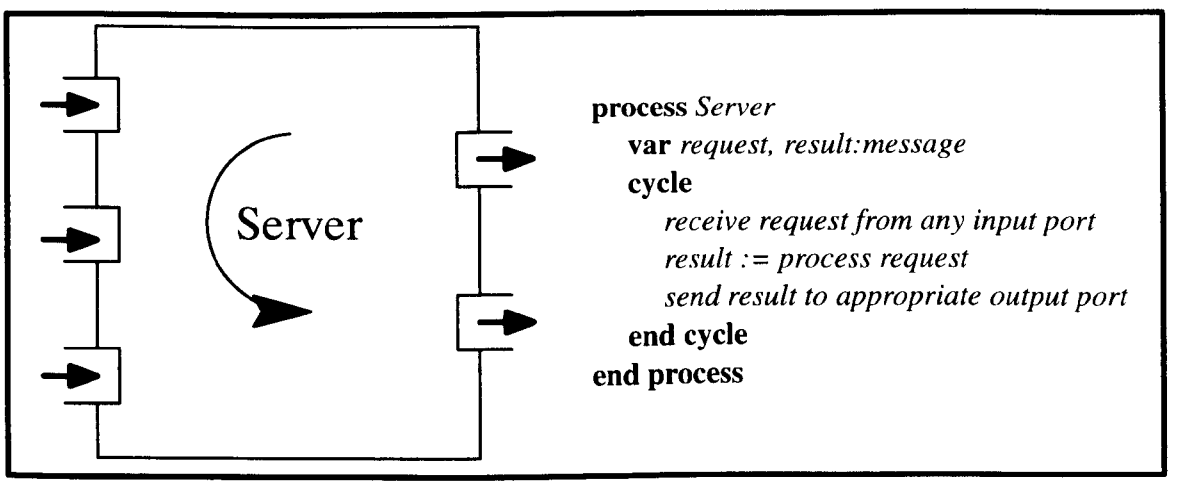

Figure 3–6: Unreplicated 'server'

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. Hence, provided the queues of all correct replicas can be guaranteed to contain identical messages in an identical order, and all the non–faulty replicas have identical initial states, then identical output messages are produced by them. Validation protocols can then be applied to the outputs generated by replica processes. Thus, replication of a process requires the following two conditions to be met:

agreement: all non–faulty replicas of a process receive identical input messages; and

order: all non–faulty replicas process the messages in an identical order.

Note that output messages are identical only if the computation performed by a process on a selected input message is deterministic. So, if processes present this deterministic behaviour, active replication is achieved by providing suitable protocols for achieving agreement and order

of input messages, and for validating output messages (e.g compare, majority vote). The basic structure for the replicated version of the 'server' in Figure 3–6 is shown in Figure 3–7.

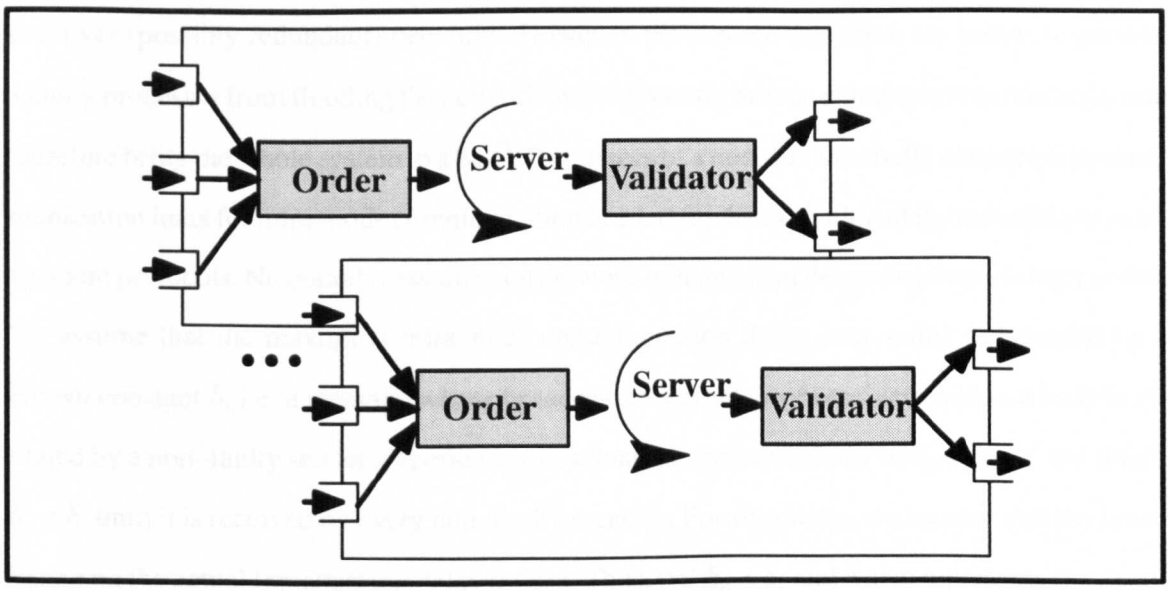

Figure 3–7: Replicated 'server'

Practical distributed programs often require some additional features such as using 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. In [Tully–Shrivastava 90], Voltan nodes are enhanced with the necessary functionality for dealing with such cases. In this thesis, we assume the simple state machine model discussed above, where processes are assumed to be deterministic.

We assume that mechanisms exist for generating and verifying digital signatures, which provides an authentication facility with arbitrary high probability [Rivest et al. 78]. Each non–faulty processor has a mechanism which generates a unique, message dependent, unforgeable signature, which is attached to any message it sends to other processors of the node. Furthermore, every non–faulty processor is also assumed to have an authentication function for verifying the authenticity of a message signature contained in a received message. We assume that a faulty processor (and therefore the processes running on that processor) can fail in an authentication–detectable arbitrary way [Strong et al. 90], where the class of authenticated–detectable arbitrary

failures is defined as the class of arbitrary failures that do not corrupt the authentication mechanisms described above.

Each processor of a node is assumed to have network interfaces for inter–node communication over (possibly redundant) networks. However, we assume that there are means to prevent a faulty processor from flooding the network by consistently broadcasting spurious messages and therefore bring the whole system to a halt[3]. Processors of a node are internally connected by communication links for intra–node communication needed for the execution of the redundancy management protocols. No bound is assumed on network transmission delays between distinct nodes. We assume that the maximum intra–node communication delay over a link is bounded by a known constant δ, i.e. a message whose broadcast to a number of receivers within a node is initiated by a non–faulty sender, experiences an actual transmission delay of δ_a units of real time[4], $\delta_a < \delta$, until it is received by every non–faulty receiver. For simplicity, we assume that the lower bound on the actual transmission delay is zero. Thus, $0 \leq \delta_a < \delta$, and δ also represents the maximum variation in message transmission delays within a node. Link failures are categorised as processor failures, that is, a link failure that prevents a message sent from a processor to be received by other processors of the node is considered as a failure of the sender processor.

Note that the transmission delay is composed of not only the time the messages spent in the communication medium, but also of all the processing time associated with a broadcast, both at the sender's and at the receivers' side. Thus, the transmission delay incorporates the differences in the relative speed of two non–faulty processors.

A processor measures the passage of real time via its physical clock. We further assume that the clock of a non–faulty processor can drift from real time by a bounded and known rate ρ_a, $|\rho_a| \leq \rho$.

---

3. This can be achieved either by using point–to–point connections to connect processors of two different nodes, or by using specially designed bus guardian circuits. Since our approach is to use only 'off–the–shelf' components, we will assume that inter–node processor are connected through point–to–point connections.

4. Real time is assumed to be the mathematical Newtonian time that cannot be directly observed.

In this thesis we adopt the style of writing real time values in Greek or italicised upper case Roman letters, and clock time values in italicised lower case Roman letters; the term 'clock' normally refers to a processor's hardware clock. We summarise below the main assumptions made.

Assumption 1: In a failure–masking node, at least $\pi+1$ out of N processors are non–faulty and never fail, where $N = 2\pi+1$; whilst in a fail–silent node, at least one out of N processors is non–faulty and never fails, where $N = \pi+1$. Processors are assigned a unique numbering which is known to all non–faulty processors.

Assumption 2: A non–faulty processor's signature for a given message is unique and cannot be generated by any other processor. Furthermore, any attempt to alter the contents of a non–faulty processor's signed message is detected by any other non–faulty processor.

Assumption 3: Processors within a node are connected to processors of another node through point–to–point connections.

Assumption 4: When a non–faulty processor sends a message to a subset of processors of the node at real time *SENT*, every non–faulty destination receives the message at real time *RECEIVED*, $SENT \leq RECEIVED < (SENT+\delta)$, where δ, $\delta > 0$, is known.

Assumption 5: A non–faulty processor's clock measures an interval of time x in a real time interval $x(1+\rho_a)$, where $|\rho_a| \leq \rho$ and ρ is a known positive constant.

Note that assumptions 4 and 5 make the environment synchronous, and are essential to guarantee that the agreement necessary for ordering input messages is reached in finite time in the presence of failures [Fischer et al. 85], and thus to ensure the liveliness of processing activities within the replicated node.

3.3.2. Node Architecture

We consider failure–masking nodes and fail–silent nodes comprised of N processors, $P_1, P_2, \ldots P_N$, where $N = 2\pi+1$ in the case of failure–masking nodes, $N = \pi+1$ in the case of fail–silent nodes, and π, $\pi > 0$, is the upper bound on the number of processors of a node that may fail. Each node in the system, each processor of a node, and each group of replicated application pro-

cesses possess unique identifiers (numbers). There is also a sequence number counter associated with each group of replicated application processes. These identifiers and counters are used to uniquely identify each message generated by any application process. Messages generated by an application process are encapsulated with control information which contains the message's sequence number, process identifier, node identifier, and authentication information (processor identifier and message signature). This information is used by the redundancy management protocols to match correlated messages, and to detect and remove duplicated and corrupted messages.

In addition to application processes ($Service_i$ processes), each processor of a node executes five system processes. Figure 3–8 shows the inter–relation between such processes. The function of each system process is described below.

Sender process: this process takes the messages produced by the application processes of that processor, signs them and sends them to the other processors of the node for validation, i.e. voting in failure–masking nodes, and comparison in fail–silent nodes. (In fail–silent nodes, the Sender process needs also to perform some flow control over messages sent for comparison. In chapter 5 we discuss this issue in detail.)

Validator process: the function of this process depends on the type of the node. In failure–masking nodes, the Validator processor is a *Voter* process. It compares authentic messages which have been signed and sent by other processors with their counterparts produced locally. If the comparison is not successful, the message is discarded. Otherwise, the message is countersigned (by considering the existing signatures in the message as part of the message), and if there are now $\pi+1$ signatures in the message, the message, termed a *valid* message, is handed over to the local Transmitter process for network delivery to destination nodes. If there are less than $\pi+1$ signatures, then the message is sent to the other processors of the node that have not signed the message yet. In fail–silent nodes the Validator process is a *Comparator* process. It also compares authentic messages which have been signed and sent by other processors with their counterparts produced locally. If the comparison is successful, the message is countersigned, and if there are $\pi+1$ signatures in the message,

the valid message is handed over to the local Transmitter process. If there are less than π+1 signatures, then the message is sent to the other processors of the node that have not signed the message yet. A comparison that detects a disagreement indicates a failure. Similarly, an absence of a π–signed message for comparison (after a node specific time–out interval) also indicates a failure. Once a failure is detected, the Comparator process terminates its activities, and so does the Sender process. Hence, no new π+1–signed messages are ever output by the node. It follows that, in both failure–masking and fail–silent nodes, all valid messages issuing from a node contain π+1 signatures.

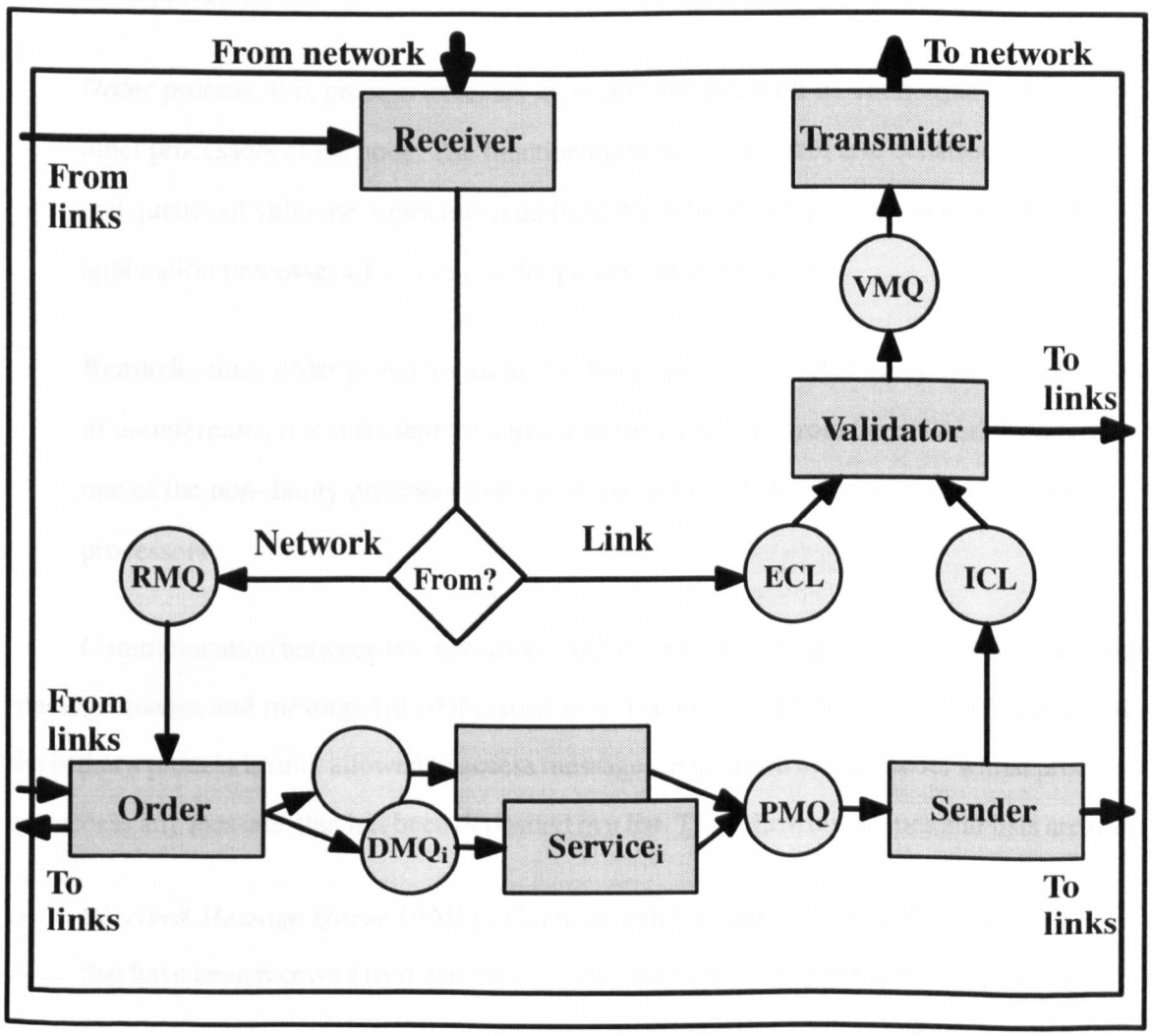

Figure 3–8: Voltan node

Transmitter process: this process is responsible for sending the $\pi+1$–signed messages over the network to destination nodes. As each processor has a Transmitter process, when all processors are non–faulty, failure–masking nodes and fail–silent nodes generate respectively $2\pi+1$ and $\pi+1$ copies of its output messages.

Receiver process: this process authenticates messages received from the network or from the internal links and discards any message which fails authentication or any duplicated message received. Authenticated messages from the network (valid messages) are sent to the local Order process. Authenticated messages received from other processors of the node, which carry less than $\pi+1$ signatures, are sent to the local Validator process.

Order process: this process executes an order protocol with its counterparts in the other processors of the node. The function of the order protocol is to construct identical queues of valid messages received from the network for processing by the local application processes of all non–faulty processors of the node.

Remark: since order protocols entail the Order process to relay valid messages to its counterparts, it is sufficient for a message to be received from the network by any one of the non–faulty processors of a node for it to be ordered at all the non–faulty processors.

Communication between two processes executing at the same processor is realised through message queues and message lists data structures. The basic difference between a queue and a list is that a process is only allowed to access messages in the front of the queue, whilst processes can access any message that has been deposited in a list. The following queues and lists are used:

Received Message Queue (RMQ): Contains valid messages intended for ordering, that have been received from the network and authenticated by the Receiver process.

Delivered Message Queue$_i$ (DMQ_i): Contains ordered messages to be consumed by the application process $Service_i$.

Processed Message Queue (PMQ): Contains unsigned output messages produced by local application processes. These messages must be validated by the Validator process before transmission to the final destination.

External Candidate Message List (ECL): Contains authenticated signed messages that have been received from other processors for validation.

Internal Candidate Message List (ICL): Contains unsigned messages, each waiting for matching signed messages to arrive in ECL.

Validated Message Queue (VMQ): Contains $\pi+1$–signed, valid messages ready to be transmitted over the network.

From the system processes discussed above, the Receiver, Sender and Transmitter processes are of trivial implementation. Thus, we discuss the implementation of these processes in this chapter, whilst we leave the discussion of the implementation of the Order and Validator processes to the next chapters.

```
process Receiver
   process Network
      var m:message
      cycle
         receive m from the network
         if m is not authentic then discard m else deposit m into RMQ end if
      end cycle
   end process
   ||
   process Link
      var m:message
      cycle
         receive m from an internal link
         if m is not authentic then discard m else deposit m into ECL end if
      end cycle
   end process
end process
```

Figure 3–9: Receiver process

The Receiver process (see Figure 3–9 above) is composed of two cyclic processes that execute in parallel. The *Network* process receives messages from the network, and performs an authentication check on the messages, which detects any spurious message that has been sent by

the faulty processors of other replicated nodes. Authentic messages are deposit into the local RMQ. The *Link* process on the other hand, receives messages from the internal links. These messages are also subjected to authentication, and if found to be authentic are deposit into the local ECL. Authentication in this case must detect both internal errors, i.e. any attempt of a faulty processor of the node to corrupt a message sent through an internal link, as well as external errors that have been propagated by a faulty processor that has failed to detect a corrupted message received from the network.

The Sender process (see Figure 3–10) simply cycles removing messages from the local PMQ, depositing them into the local ICL, signing and diffusing them to the other processors of the node that have not received that message yet (i.e. those whose signature are not present in the message).

```
process Sender
   var m:message
   cycle
      remove m from PMQ
      deposit m into ICL
      sign and diffuse m
   end cycle
end process
```

Figure 3–10: Sender process

Finally, the Transmitter process (see Figure 3–11) cycles removing messages from the local VMQ and transmitting them to the appropriate destination nodes.

```
process Transmitter
   var m:message
   cycle
      remove m from VMQ
      transmit m to destination
   end cycle
end process
```

Figure 3–11: Transmitter process

Remark: the failure–masking node described above has the semantics of an NMR node. To attain the semantics of a fail–signal node from the above description, it is

necessary to introduce a minor modification on the Voter process (Validator process for failure–masking nodes) previously described. The Voter process must be enhanced with a facility for generating failure signals, whenever a comparison detects a mismatch. Also, it might be necessary to introduce a *Handler* process to monitor the failure of other nodes in the system and initiate preventive procedures after receiving a failure signal. (See [Shrivastava et al. 91].)

The main difference between a distributed system composed of replicated nodes such the ones presented above, and another composed of unreplicated nodes, is that in the former, nodes are required to produce $\pi+1$–signed messages and use authentication to distinguish between valid and spurious messages. Further, under normal conditions, inter–node traffic is increased $2\pi+1$ times when the distributed system is composed of failure–masking nodes, and $\pi+1$ times when the nodes are fail–silent.

Provided that the number of faulty processors does not exceed π, the semantics of the failure–masking and fail–silent nodes presented above can be expressed in the following way:

failure–masking nodes: when all processors of a failure–masking node are non–faulty, the node produces $2\pi+1$ copies of valid output messages, that can be verified as such by any non–faulty processor of the destination nodes. When up to π processors are faulty, a failure–masking node outputs at least $\pi+1$ copies of valid output messages, and may also output spurious messages. Any spurious message output by the faulty processors of a failure–masking node can be detected and rejected by all non–faulty processors of any receiver node;

fail–silent nodes: when all processors of a fail–silent node are non–faulty, the node produces $\pi+1$ copies of valid output messages, that can be verified as such by any non–faulty processor of the destination nodes. When any number of processors, up to π, are detected as faulty, the node ceases to produce new valid output messages, in which case non–faulty processors of receiver nodes can detect any message it may produce as unwanted.

In both nodes, spurious messages are detected by examining the signatures attached to the messages received, whilst the sequencing information of messages is used to detect duplicates.

3.4. Concluding Remarks

The design of replicated nodes has been discussed. We have differentiated between two types of nodes, namely hard nodes, where redundancy is controlled through hardware mechanisms, and soft nodes, where software protocols are used to manage node redundancy. We have discussed the drawbacks and advantages of each type of nodes, and justified our choice for using soft nodes as building blocks of fault–tolerant distributed systems.

We have finished by presenting the architecture of the failure–masking and fail–silent nodes of the Voltan family of soft replicated nodes. In these nodes, apart from the application processes, each non–faulty processor of the node executes five system processes, namely Receiver, Sender, Transmitter, Order and Validator processes. These processes are responsible for controlling the node redundancy, so that the node presents its correct semantics. Among these processes, the Receiver, Sender and Transmitter processes are of trivial implementation. Therefore, the difficulty in implementing replicated nodes lies in the implementation of the Order and Validator processes. In the next two chapters we discuss protocols for implementing such processes for both failure–masking and fail–silent nodes.

Chapter 4
Soft Failure–Masking Nodes

4.1. Introduction

In the previous chapter, the general structure of the architecture of a soft failure–masking node was described. There are five system processes that must be executed by each non–faulty processor forming the node to provide the node's masking functionality. These processes are the Sender, Receiver, Transmitter, Validator and Order processes. We have already described in Section 3.3.2 how the first three processes listed can be implemented. The other two processes have not been discussed in detail yet. The Validator process is responsible for validating the output messages generated by the node, whilst the Order process is responsible for guaranteeing that application process replicas executing at each processor input the same set of messages, in the same order. In this chapter we discuss protocols which can be used to efficiently implement these two processes.

The performance of a failure–masking node depends on how quickly messages can be validated and ordered. In a failure–masking node, the Validator process is a Voter process whose functioning is relatively straightforward. The delay imposed by the voting protocol is mostly made up by the time spent in the unavoidable exchange of output messages among the processors prior to the execution of voting. On the other hand, ordering is not as straightforward as voting, and can be achieved in several ways [Schneider 90]. Thus, we have sought ways to improve the performance of a failure–masking node by designing more efficient order protocols.

We start the chapter by presenting a reference design of a failure–masking node. The reference design includes the description of simple Voter and Order processes. The Order process in-

corporates an order protocol which requires that the participant processors have *synchronised clocks*, i.e. at any given instance of real time the readings of the clocks of any two non–faulty processors are guaranteed to differ only within a known bound. We then derive more efficient order protocols which do not require clocks to be explicitly synchronised. Different designs of failure–masking nodes can be attained by associating the Voter process of the reference implementation with an Order process which incorporates any of the efficient order protocols presented. Thus, the only difference between the various node designs presented in this chapter is in the way input messages are ordered. We finish the chapter with an analysis of the overhead imposed by the protocols presented.

4.2. Reference Design

4.2.1. Standard Voter Protocol

The voter protocol is very simple. It collects output messages generated by the processors in the node, and performs a majority voting upon the messages collected. The result of the majority voting is then output. Figure 4–1 shows the pseudo–code for a Voter process.

The Voter process has access to two lists of messages (see Figure 3–8, page 56), namely the Internal Candidate Message List (ICL), which contains messages produced locally by the application processes, and the External Candidate Message List (ECL), which contains authenticated signed messages with up to π signatures, which have been produced by the application processes executing at the other processors of the node. Messages in ECL may have been subjected to evaluation at other processors of the node; essentially, a message which contains s signatures has been successfully compared against the locally produced messages of s–1 other processors. The function of the Voter process is to scan ECL looking for signed counterparts of the local messages present in ICL. When such a pair is found, the Voter process compares the contents of the messages, and if there is a mismatch, the message in ECL is discarded. Otherwise, the message in ECL is countersigned and either enqueued in the Validated Message Queue (VMQ) for later transmission to the destination by the Transmitter process, or sent to all other processors of the node that have not signed the message yet. Messages signed by a majority of processors, i.e.

π+1–signed messages, are valid messages, and thus are enqueued in VMQ. Messages with less than π+1 signatures are sent through the internal links to the other processors of the node.

```
process Voter
  var internal, external:message
      found:Boolean
  cycle
    do
      internal := a message from ICL
      found := is there a counterpart of internal in ECL?
    while not found
    external := the counterpart of internal in ECL
    if internal = external then
      generate and append the signature for external
      if number of signatures in external = (π+1) then deposit external into VMQ
      else
        for all P_i, 1 ≤ i ≤ N do
          if P_i has not signed external then transmit external to P_i end if
        end for all
      end if
    else discard external
    end if
  end cycle
end process
```

Figure 4–1: Voter process

4.2.2. Order Protocol with Synchronised Clocks

The function of the Order process is to ensure that authentic valid messages received by the non–faulty processors of the node, i.e. messages enqueued in RMQ (see Figure 3–8, page 56), are enqueued in the same order in the appropriate DMQ_i of every non–faulty processor of the node. Ordering can be achieved in several ways. The basic idea is to have an agreement protocol which guarantees that all non–faulty replicas receive the same set of messages and then accomplish ordering by assigning monotonically increasing sequence numbers to messages. It is also necessary to devise a method to establish when a message becomes *stable*, i.e. when it is guaranteed that no authentic valid messages with sequence numbers less than a certain value, say *sn*, will ever be received, so that all messages with sequence numbers less than *sn* can be processed in a consistent order among all non–faulty replicas. Since the authentication mechanisms of non–faulty pro-

cessors can detect any message corrupted by a faulty processor, an order protocol has to implement a message diffusion mechanism, which guarantees that a message received by a non–faulty processor is also received by all other non–faulty processors of the node, and timeliness checks mechanisms to discard unwanted messages diffused by faulty processors at inappropriate times.

In a synchronous system, Byzantine resilient *atomic broadcast* protocols [Cristian et al. 85, Gopal et al. 90] can be used to achieve ordering. An atomic broadcast protocol presents the following properties:

termination: it delivers every message broadcast by a non–faulty sender within some known time bound Δ, as measured by the clock of any non–faulty processor;

atomicity: it ensures that every message whose broadcast is initiated by a sender is either delivered to all non–faulty receivers, or to none of them; and

order: messages are delivered in the same order at all non–faulty receivers.

To achieve ordering of input messages in a soft failure–masking node, it suffices that every message in the RMQ of a non–faulty processor is atomically broadcast to all other processors of the node. The atomicity property of the atomic broadcast guarantees that non–faulty processors will input the same set of messages, whilst the order property guarantees that messages are delivered in the same order by all non–faulty processors. Non–faulty processors then enqueue delivered messages in the delivered order at the appropriate DMQ_i. Since a particular input message may appear in the RMQ of several non–faulty processors, processors may receive multiple copies of the same input message. Thus, the Order process must make sure that duplicate messages are discarded, rather than enqueued in the DMQ_is.

The Order process of our reference design is based on the authenticated Byzantine resilient atomic broadcast protocol reported in [Cristian et al. 85], which is resilient to any number π of faulty processors, provided $\pi < N$. Since the voting mechanism of a failure–masking node requires at least $N = 2\pi+1$, this atomic broadcast protocol is suitable for implementing the Order process of a failure–masking node. The atomic broadcast protocol also requires that the clocks of all non–faulty processors of the node are synchronised such that the measurable difference between readings of clocks at any instant is bounded by a known constant, say ε.

The ordering of input messages using this atomic broadcast protocol works as follows. The Order process of a processor time–stamps a message to be ordered with its local clock reading. A copy of the time–stamped message is signed and sent over the link to the Order process of the other processors of the node. If t is the time–stamp of a message received from or sent to the Order process of the other processors, then this message becomes stable at local clock time $t+\Delta$, where Δ, the termination bound of the protocol is given by: $\Delta = (\pi+1)(d_\Delta+\varepsilon)$. We define the clock time interval d_Δ, $d_\Delta = \delta/(1-\rho)$, to be the minimum interval that the clock of any non–faulty processor must advance to measure the real time interval δ, i.e. the real time interval elapsed between two readings t, and $t+d_\Delta$ of the clock of any non–faulty processor is at least δ. Once a message with time–stamp t becomes stable, no authentic valid messages with time–stamp $t' < t$ can be received by an Order process (if such messages are received they are found untimely, and are discarded). Stable messages are enqueued in the appropriate DMQ_i in increasing time–stamp order, with the action being taken to discard, rather than to enqueue a stable message, if its replica has already been enqueued. (The identifier of the processor which has initiated the broadcast of a particular message is used as a tie–breaker in the case when there are more than one stable messages with the same time–stamp.)

The Order process is composed of three cyclic processes, namely *Broadcast*, *Diffuse* and *Deliver* processes, which execute in parallel. This three processors communicate with each other through a shared message list data structure called the *Ordered Message List* (OML). Figure 4–2 shows the structure of the Order process.

The Broadcast process picks up a message from the RMQ, time–stamps it with its current clock reading, signs it with a unique, unforgeable signature and then sends it to the other processors of the node. It also inserts a copy of the message into the OML.

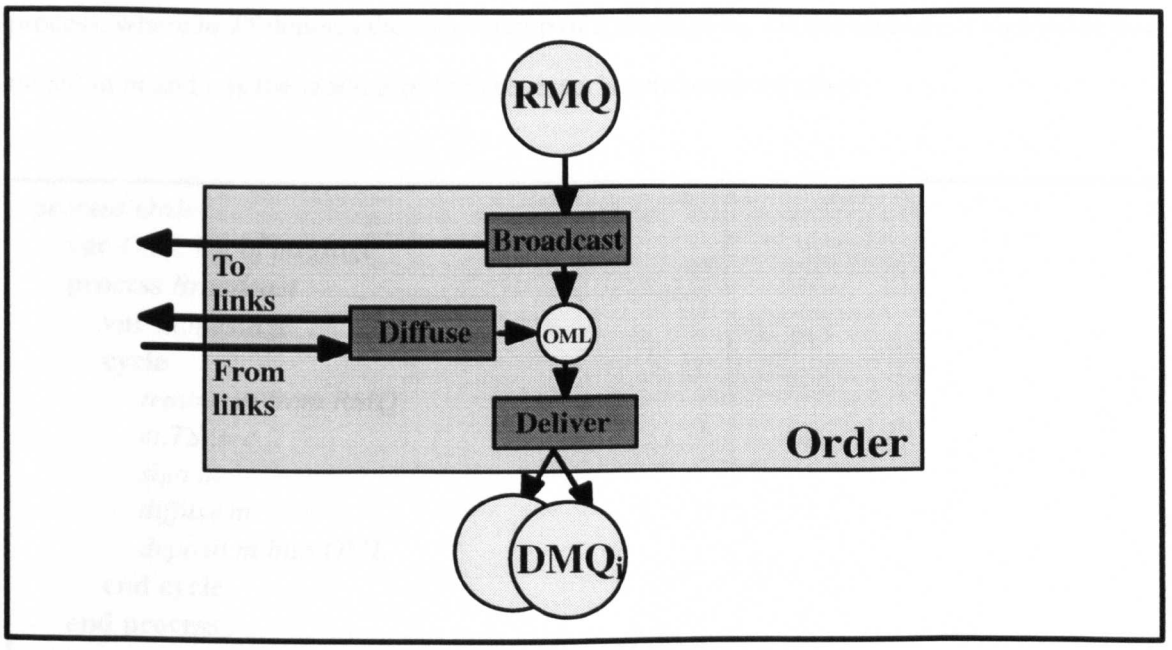

Figure 4–2: Order process structure

The Diffuse process receives diffused messages from the other processors. After checking the authenticity of a message received from another processor, the Diffuse process performs a timeliness check that allows it to discard any message received too early (messages with time–stamp greater than $c+(s\varepsilon)$, where c is the current reading of the processor's clock, and s is the number of signatures contained in the message, i.e. the number of processors that have diffused the message), or any message received too late, i.e. messages with time–stamp less than $c-s(d_\Delta+\varepsilon)$. Authentic and timely messages are accepted and inserted into the OML. Corrupted and untimely messages are discarded. The Diffuse process also checks if the message received is a copy of a previously received message, in which case the message is also discarded. If the message received has not been received before, the Diffuse process signs and diffuses it to the other processors in the node which have not received that message yet.

The Deliver process takes stable messages (messages with time–stamp less than $c-\Delta$) from the OML, filters duplicate messages (i.e. removes all the duplicated copies of a particular message, but one), removes spurious messages (messages diffused by the same source, with the same time–stamps, but with different contents) and enqueues the remaining messages in the appropriate DMQ_is in increasing order of time–stamps. Figure 4–3 shows the pseudo–code for this Order

process, where *m.TS* denotes the time–stamp of a message *m*, *s* is the number of signatures contained in *m* and *c* is the reading of the processor's synchronised clock.

```
process Order
   var OML:list of message
   process Broadcast
      var m:message
      cycle
         remove m from RMQ
         m.TS := c
         sign m
         diffuse m
         deposit m into OML
      end cycle
   end process
   ||
   process Diffuse
      var m:message
      cycle
         receive m from an internal link
         if m is authentic and c–s(dΔ+ε) ≤ m.TS ≤ c+(sε) and have not received m yet then
            sign m
            diffuse m
            deposit m into OML
         else discard m
         end if
      end cycle
   end process
   ||
   process Deliver
      var m:message
          stable, delivered:list of message
      cycle
         for all m, m in OML and m.TS ≤ c–Δ do add m to stable end for all
         remove spurious messages from stable
         filter duplicate messages from delivered+stable
         delivered := delivered+stable
         order the messages in stable in increasing time–stamp order
         for all m in stable do deposit m into the appropriate DMQi end for all
      end cycle
   end process
end process
```

Figure 4–3: Order process for reference design

The termination bound (Δ) of the atomic broadcast protocol gives the delay suffered by an input message before it is made available to the application process of a particular processor, as measured by the clock of that processor. However, the node can only output the results of the computation of a particular input message when the number of processors that have ordered the input message, and processed it, form at least a majority. Hence, in the context of a failure–masking node, we define the *actual stability delay* (Σ_a) of an order protocol for a particular message to be the *real time* elapsed since a copy of the message is first received by a non–faulty processor of the node until it is ordered and enqueued in the appropriate DMQ_is of all non–faulty processors of the node. We also define Σ_{min} and Σ_{max} to be respectively the lower and the upper bound of the actual stability delay of an order protocol ($\Sigma_{min} \leq \Sigma_a \leq \Sigma_{max}$). Therefore, for the above protocol we have:

$$\Sigma_{min} = \Delta(1-\rho);\ \Sigma_{max} = \Delta(1+\rho)+\varepsilon;\ \text{and}\ \Delta(1-\rho) \leq \Sigma_a \leq \Delta(1+\rho)+\varepsilon.$$

The lower bound Σ_{min} is achieved when the actual difference between the readings of the clocks of any two non–faulty processors at the time the message is being ordered is zero, and the clocks of all non–faulty processors are running at the fastest possible rate, i.e. the interval Δ measured by their clocks corresponds to an interval $\Delta-\Delta\rho$ of real time. On the other hand, the upper bound Σ_{max} is achieved when the clock of at least one non–faulty processor is running at the slowest possible rate (i.e. ρ), and the difference between the reading of the clock of the first non–faulty processor to receive a copy of the message, and the reading of the slowest non–faulty processor at the time the message is being ordered is equal to ε.

4.3. Efficient Order Protocols

We have shown how atomic broadcast protocols can be used to implement an order protocol. The functioning of an atomic broadcast protocol can be analysed in two parts. First it achieves agreement, then it achieves order. The synchronous and deterministic agreement protocols published in the literature require that non–faulty processors have access to synchronised clocks whose readings at any given instance of real time are guaranteed to differ only within a known bound. Since physical clocks of non–faulty processors drift from real time by different rates, meeting this requirement will in turn require each processor to periodically compute the amount

of adjustment to be made to the reading of its local physical clock, and then to make the adjustment so that the synchronous property of the system is maintained. However, if the agreement protocol is executed frequently enough, adjustments can be computed with no message overhead [Babaoglu–Drummond 87]. The need to adjust a read–only clock leads to constructing an *abstraction* of a synchronised clock whose reading is the sum of the physical clock reading and the adjustment that is stored in a memory location (see [Dolev et al. 84] for example).

In a failure–masking node, a given input message can prompt more than one processor to initiate an execution of the agreement protocol. (This is not the case in the transaction commit where only one site – the co–ordinator – initiates one agreement per transaction.) By exploring the execution of these agreement protocols, it is possible to attain some sort of synchronisation among processors, without the need for executing an explicit clock synchronisation protocol. This observation has motivated us to develop atomic broadcast protocols based on agreement protocols that do not require the maintenance of the synchronised–clock abstraction, but relies directly on physical clocks for knowing the current time and for scheduling operations at future times. Many operating systems provide an efficient, low level scheduler that meets our requirement (which is typically in the form of "schedule an execution of process p when clock reads t").

In this section we first present an order protocol whose performance is comparable with other protocols reported in the literature, but, unlike the latter, does not need to maintain explicit synchronised clocks. Then we present an improved version of the protocol for the important case of a Triple Modular Replicated (TMR) node, which performs better than any synchronised based protocol. Finally, we present a derived protocol which assumes that inter–node communication is realised through *first–in–first–out* (*fifo*) channels, and takes advantage of some characteristics of the nodes we are designing to order messages much faster, especially when the node is in a failure–free state. We note that like the order protocol discussed in the previous section, the order protocols discussed in this section require only $\pi < N$, thus, the node requirement that at least a majority of the processors are non–faulty is needed only because of the majority voting mechanism inherent of failure–masking nodes.

The presentation of each protocol is performed in three steps. First an informal description of the protocol is given; then the correctness of the protocol is sketched; and finally the perform-

ance of the protocol is discussed. Later, in Section 4.4, we compare the performance of the protocols presented with the performance of other protocols reported in the literature.

4.3.1. A Protocol without Explicit Clock Synchronisation

4.3.1.1. Protocol Description

Again, the development of the order protocol mainly involves implementing: i) *message diffusion* to ensure that non–faulty processors exchange an identical set of messages between them; and ii) *timeliness checks* to enable a non–faulty processor to assess whether a received message is timely, and therefore should be accepted, or if the message is untimely, in which case it should be discarded. We assume that authentication of input messages diffused by the other processors of the node is performed before the message is made available to the order protocol, thus for the order protocol, the faulty behaviour of a processor is restricted to omitting to send messages or sending them at incorrect times (which includes time–stamping a message with a spurious value).

Message Diffusion

Each processor P_i within a failure–masking node maintains a *message counter*, denoted by MC_i, whose value is an integer that never decreases and is initialised to 1 when the node is first started. When P_i wants to initiate the broadcast of a message received from the network, it composes an internal message $m = <\mu, TS, O, S>$, where $m.\mu$ is the contents of m, i.e. the message received from the network; $m.TS$ is the time–stamp of m; $m.O$ is the originator of m; and $m.S$ is the list of signatures contained in m. A newly formed and unsigned m has empty $m.S$, whilst for any sent or received m, $m.S$ contains a sequence of one or more processor signatures. We use the notation $|m.S|$ to denote the number of signatures in $m.S$.

Before P_i sends the message m it formed, it performs the following actions on m: first, it time–stamps m by setting the message field $m.TS = MC_i$, and increments MC_i by 1 – thus ensuring that any message it later forms gets a time–stamp larger than $m.TS$; then it sets the message originator field $m.O$ to its identifier, i; and finally it generates a signature for m which is put in the field $m.S$. P_i then sends m to all other processors of the node, and *accepts* the sent message by entering a copy of it in a message list called $accepted_i$.

Whenever P_i receives a message m' from another processor of the node, it accepts m' only if m' is timely, and P_i has not accepted m' previously. (We discuss shortly how a non–faulty processor assesses the timeliness of a message.) If m' is accepted, then it is entered in the message list $accepted_i$ and MC_i is set to the maximum of $\{MC_i, m'.TS+1\}$. If the accepted m' has been signed by less than $\pi+1$ processors, P_i generates a signature for m' and add the generated signature to the list of signatures already in $m'.S$; m' is then sent to all processors that have not signed it yet. Thus, any accepted message with less than $\pi+1$ signatures is diffused. As there can be at most π faulty processors, this diffusion ensures that if m enters $accepted_i$ of any non–faulty P_i, then some m' such that $m'.\mu = m.\mu$, $m'.O = m.O$ and $m'.TS = m.TS$, is received by every other non–faulty P_k of the node.

For any given message m, we define an $equiv(m)$ as any m' such that $m'.\mu = m.\mu$, $m'.O = m.O$ and $m'.TS = m.TS$. That is, only the contents of $m.S$ and $[equiv(m)].S$ may differ. Note that an $equiv(m)$ can be m itself.

Lemma 1 (message diffusion): If a non–faulty P_i accepts m at real time $TIME_i$, then every non–faulty P_k receives an $equiv(m)$ at real time $TIME_k$ such that $|TIME_i - TIME_k| < \delta$.

Proof: Let P_j be the first non–faulty processor to accept m at real time $TIME_j$. It follows that $|m.S| < \pi+1$, and if $|m.S| > 0$, then any processor which has signed m must be a faulty processor (since P_j is the first non–faulty processor to accept m). P_j then diffuses m to every other processor which has not signed m. From assumption 4 (see Section 3.3.1, page 54), every non–faulty P_k, $P_k \neq P_j$, P_i inclusive, receives $equiv(m)$ at some real time $TIME_k$, with $TIME_j \leq TIME_k < TIME_j+\delta$, hence the lemma. ❑

To present the protocol in this section, we assume the availability of the primitives, *receive*(m) and *send*(m). These two primitives together implement message diffusion. The *receive*(m) primitive returns a message m that has been received from an internal link and has been authenticated. (This authentication procedure involves checking external signatures appended by processors of the originator node, i.e. checking $m.\mu$, as well as checking internal signatures appended by the processors of the node, i.e. checking $m.S$). Hence, any message m received by a

non–faulty processor by invoking the *receive*(m) primitive is authentic, and further development of the protocol need only be concerned with the timeliness of m. The *send*(m) primitive generates the sending processor's signature for m if m is eligible for diffusion (i.e. $|m.S| < \pi+1$), and appends the generated signature to the signature sequence already in $m.S$. The resulting message is then transmitted to every other processor in the node that has not signed it (see Figure 4–4).

```
send(m)
{
   if |m.S| < (π+1) then
      generate and append the signature for m
      for all P_j, 1 ≤ j ≤ N do
         if P_j has not signed m then transmit m to P_j end if
      end for all
   else discard m
   end if
}
```

Figure 4–4: Send primitive

Timeliness checks

These checks enable a non–faulty processor to accept or discard a received message m such that if m enters $accepted_i$ of a non–faulty P_i then an *equiv*(m) enters $accepted_k$ of every non–faulty P_k within finite time. The protocol employs two timeliness checks based upon time–outs. To illustrate the principles behind the development of these checks, we assume that a processor takes zero time to execute any instruction of the protocol and the *receive*(m) and *send*(m) primitives earlier described. We assume the existence of a clock time interval d, which is of common knowledge to all non–faulty processors, and which will be used to measure a real time interval of at least δ duration. Thus, d must be chosen in such a way that the clock of any non–faulty processor advances from a value t to a value $t+d$ within a real time interval not smaller than δ. Note that since physical clocks drift from real time, the real time interval elapsed when the clock of different non–faulty processors measure a clock time interval d might well be dissimilar.

Suppose that a non–faulty P_i forms and sends m, $m.TS = ts$, $ts \geq 1$, at its clock time t_i. At this time, P_i accepts m, and sets $MC_i = ts+1$. We assume (as an induction hypothesis) that any non–faulty P_k that receives m sent by P_i accepts it (unless, before receiving m, P_k has already received,

and accepted another *equiv*(m)). Thus, since P_k updates its MC_k after accepting *equiv*(m), by lemma 1, MC_k of any P_k becomes larger than ts before a real time interval of δ duration has elapsed, i.e. before P_i's clock time t_i+d. This means that any m' sent by a non–faulty P_k to P_i, with $m'.TS \leq ts$ and $|m'.S| = 1$, must have been sent before P_k had accepted *equiv*(m). Thus, P_i should only accept any m', $m'.TS \leq ts$ and $|m'.S| = 1$ (i.e. a message whose broadcast has been initiated by P_k), if it receives m' before a real time interval of 2δ duration has elapsed since it had sent m, i.e. before its clock time t_i+2d; otherwise m' must be considered late and must be discarded.

Let a non–faulty P_k receive *equiv*(m) from P_i at its clock time t_k and set $MC_k = m.TS+1$ whilst accepting *equiv*(m). Accounting for the possibility of zero transmission time for the message *equiv*(m) received, P_k must assume that MC_j of any non–faulty P_j, $j \neq i \neq k$, becomes at least $m.TS+1$ before P_k's clock reads t_k+d, since by that time every P_j will have received and accepted *equiv*(m). So, P_k must observe the same rule as P_i: it should only accept any m' sent by another processor, with $m'.TS \leq ts$ and $|m'.S| = 1$, if it receives m' before its clock time t_k+2d. Thus, the timeliness check for a single–signed m' is: m', $m'.TS \leq ts$ and $|m'.S| = 1$, is timely only if it is received before local clock time $t+2d$, where t is the smallest local clock time when a message m, $m.TS \geq ts$, was accepted.

We now extend to multiple signatures. Assuming that P_i has formed, and sent m as before, let P_j be a faulty processor that sends m' only to P_k, with $m'.TS \leq ts$ and $|m'.S| = 1$, and let P_k receive m' just before its clock is to read t_k+2d, where t_k was P_k's clock time when P_k had received and accepted *equiv*(m). Non–faulty P_k considers m' as timely, accepts and diffuses it after adding its signature to the diffused m'. As non–faulty P_k has accepted m', non–faulty P_i must also accept the diffused m'. So, any timeliness check carried out by P_i must indicate that the diffused m' is timely. The event of P_k's clock reading t_k happened when P_k accepted *equiv*(m), and therefore happened at latest just before a real time interval of δ duration had elapsed since P_i had sent m, i.e. when P_k's clock reads t_k, P_i's clock reads at most t_i+d. If we neglect, for the time being, the small difference between the drift from real time of P_i's and P_k's clocks when they measure a $2d$ clock time interval, then when P_k's clock reads t_k+2d and m' is diffused to P_i, P_i's clock reads at most t_i+3d. Thus, the double–signed m' diffused by P_k can be received by P_i at latest just before P_i's clock reads t_i+4d, and the timeliness check for a double–signed m' received by P_i must be: m',

$m'.TS \leq ts$ and $|m.S| = 2$, is timely only if it is received before clock time t_i+4d, where t_i is the smallest clock time when a message m, $m.TS \geq ts$, was accepted.

Of course, the difference between the drift of clocks cannot be neglected. Take for instance, the worst case scenario when P_i's clock is running fast, whilst P_k's clock is running slow. In this situation, when P_k's clock reads t_k+2d, P_i's clock may be reading a value greater than t_i+3d, and therefore, it is possible that m' is received by P_i after its clock had already read t_i+4d, which would lead to m' being discarded by P_i.

The $4d$ time–out of the timeliness check executed by P_i can be divided into the following three components: i) the transmission delay of m sent by P_i to the other processors; ii) the $2d$ time–out during which a non–faulty P_k may receive timely m', $m'.TS \leq ts$ and $|m.S| = 1$, after having accepted an *equiv*(m); and iii) the transmission delay of any timely m' received by a non–faulty P_k and diffused to P_i (see Figure 4–5, where T_s and T_d are respectively the real time when m was sent by P_i and the real time when m' was diffused by P_k).

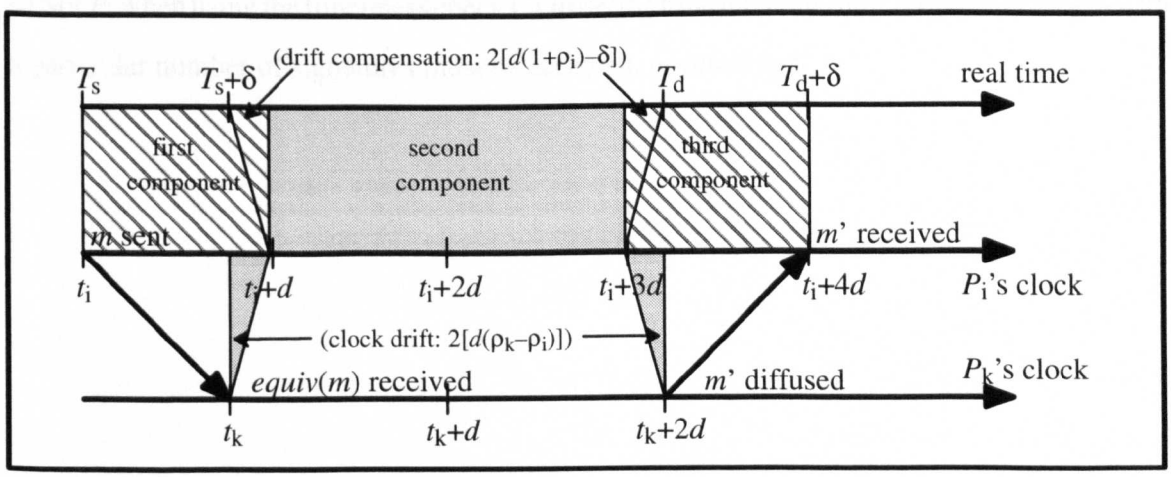

Figure 4–5: Compensating the difference between the drift of clocks in an NMR

Note that the drift problem only applies to the clock time interval of the second component of the time–out. Both the first, and third components of the time–out are due to actual transmission delays of messages, and correspond to real time intervals. Since the maximum drift from real time of any non–faulty clock is known (assumption 5, page 54), it is possible to quantify the maximum difference between the drifts of the clocks of any two non–faulty processors when measuring any particular clock time interval. Thus, by choosing a large enough value for d, it is

possible to compensate the drift difference when measuring the $2d$ clock time interval of the second component of the time–out, with the difference between δ and the actual real time interval elapsed when P_i measures a clock time interval d for the other two components of the time–out. In Figure 4–5 this compensation is illustrated, where ρ_i and ρ_k are the drift rates from real time of the clocks of processors P_i and P_k, respectively.

In Section 4.3.1.2, we show how to precisely calculate the value of d as a function of ρ, and π, so that it is possible to attain the compensation described above. Assuming that d is appropriately chosen, we generalise the timeliness check for messages with any number of signatures s, $1 \leq s \leq \pi+1$, as follows:

> **check C1**: a received m, $m.TS \leq ts$ and $|m.S| = s$, is timely only if it is received before local clock time $t+2sd$, where t is the local clock time when MC first became larger than ts (i.e. the smallest local clock time when a message m, $m.TS \geq ts$ was accepted).

The time diagram presented in Figure 4–6, summarises the reasoning of a non–faulty processor P_i when using the timeliness check C1 to define time intervals during which messages with a particular number of signatures must be accepted, or discarded.

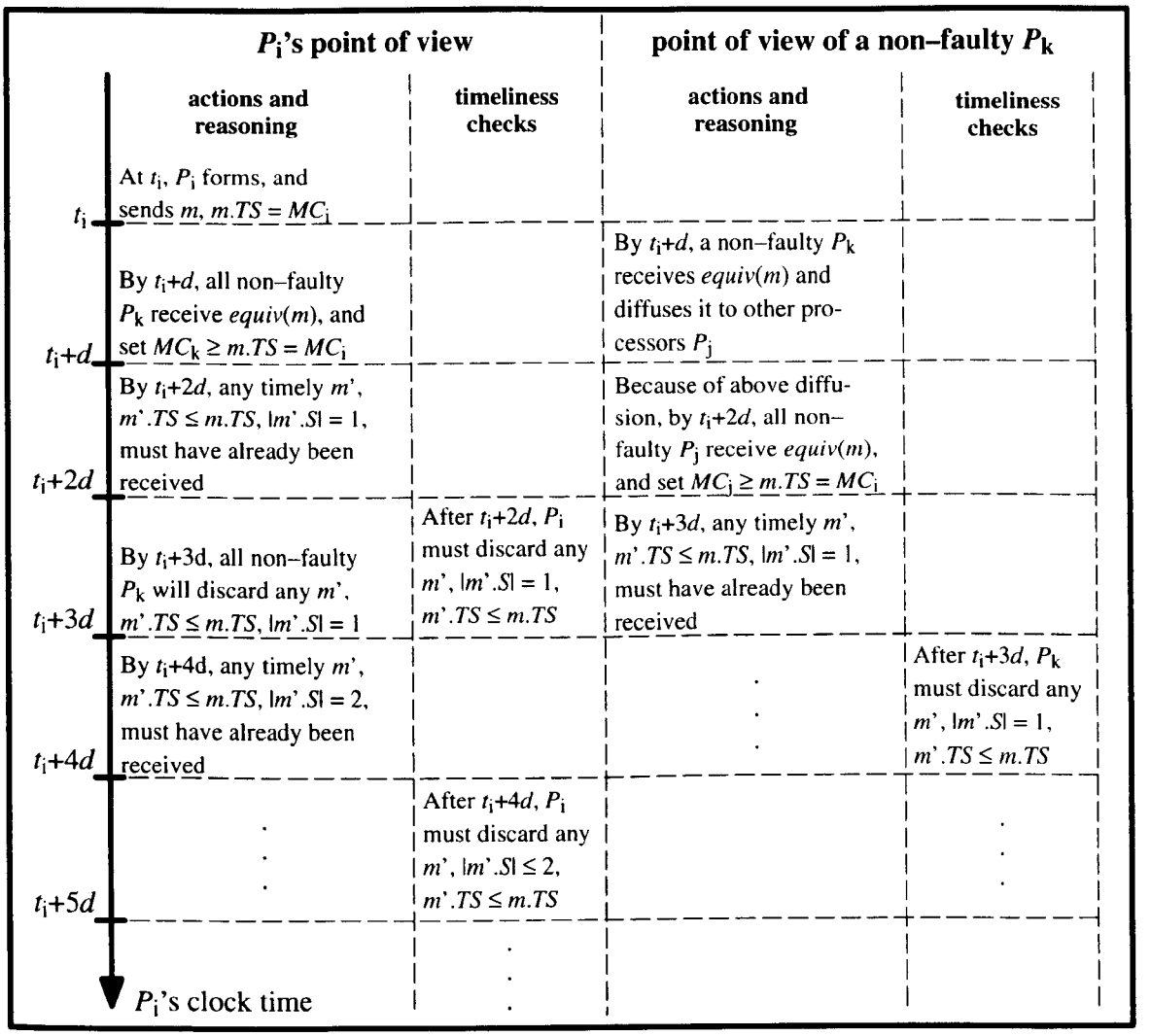

Figure 4–6: Time diagram for timeliness check C1

The check presented above should be superimposed by a second check in some cases. Suppose that non–faulty P_i receives and accepts a message m, $m.TS = ts$ and $m.O = j$, at clock time t_i. In accordance with check C1, P_i should discard any single–signed message m', $m'.TS \leq ts$, received after clock time t_i+2d. However, since a non–faulty P_j must have updated MC_j to ts+1 when sending m, any single–signed m', with $m'.TS \leq ts$, and $m'.O = j$, sent by P_j to P_i must have been sent at a time before P_j had sent m. Thus, accounting for the possibility of m taking zero time to be transmitted from P_j to P_i, P_i should consider any single–signed m', $m'.TS \leq ts$ and $m'.O = j$, timely only if m' is received before clock time t_i+d. (Note that there is no assumption that messages sent from a non–faulty P_j to a non–faulty P_i are received by P_i in the order they have been sent by P_j.)

Again, we extend the reasoning for the case of messages with multiple signatures. Suppose that a non–faulty P_i receives and accepts m, $m.TS = ts$ and $m.O = j$, at clock time t_i. As discussed before, P_i will discard any m', $m'.TS \leq ts$, $|m'.S| = 1$ and $m'.O = j$, received after clock time t_i+d. Message diffusion guarantees that any non–faulty P_k will receive an *equiv*(m) within a real time interval of δ duration. We assume (as an induction hypothesis) that any non–faulty P_k that receives *equiv*(m) accepts it, and thus P_k will also discard any m', $m'.TS \leq ts$, $|m'.S| = 1$ and $m'.O = j$, received after clock time t_k+d, where t_k was its clock time when it had accepted *equiv*(m).

Now suppose P_j is faulty, and sends m', $m'.TS \leq ts$, and $|m'.S| = 1$, only to P_k, and that P_k receives m' just before its clock reads t_k+d. P_k then accepts m', and diffuses m' to the other processors. Since P_k has accepted m', any timeliness check carried out by P_i must indicate that m' is timely. When P_k's clock reads t_k, P_i's clock reads at most t_i+d. Again, we consider a compensation (similar to the one previously discussed) on the value chosen for d to account the difference between the drifts of the clocks of P_i and P_k when measuring a clock interval d. Thus, when P_k's clock reads t_k+d, P_i's clock reads at most t_i+2d; and adding a further transmission delay to allow time for the diffusion to P_i of any timely m' received by P_k, any such timely m' will be received by P_i at latest just before P_i's clock reads t_i+3d. So, the timeliness check for a double–signed m' received by P_i must be: m', $m'.TS \leq ts$, $|m'.S| = 2$ and $m'.O = j$, is timely only if it is received before local clock time t_i+d+2d, where t_i is the local clock time when a message m, $m.TS = ts$ and $m.O = j$, was accepted.

Assuming an appropriate value for d, we can once again generalise the timeliness check for messages with any number of signatures s, $1 \leq s \leq \pi+1$, as follows:

> **check C2**: a received m', $m'.TS \leq ts$, $|m'.S| = s$ and $m'.O = j$, is timely only if it is received before local clock time $t+d+2(s-1)d$, where t is the smallest local clock time when a message m, $m.TS \geq ts$ and $m.O = j$, was accepted.

To implement timeliness checks C1 and C2, each non–faulty P_i maintains *timing counters*, denoted as $TC_i[j, s]$, for every P_j, $j \neq i$, and for every s, $1 \leq s \leq \pi+1$. These timing counters have integer values which are initialised to zero and never decrease. A timing counter $TC_i[j, s]$ is set to

ts, when any received m, $m.TS \leq ts$, $|m.S| = s$ and $m.O = j$, can no longer be considered timely. The two checks derived above indicate that setting $TC_i[j, s]$ and $TC_i[j, s+1]$ to a given ts must be separated by a clock time interval $2d$. A scheduler is used to execute a process, named *Update*, at appropriate times (Figure 4–7 shows P_i's Update process). An execution of the Update process sets a specified timing counter to a specified value and schedules itself to be executed after $2d$ time, if necessary.

```
process Update(j, s, ts)
    var t:clock_time
    if TC_i[j, s] < ts then TC_i[j, s] := ts end if
    if s < (π+1) then
        t := the local physical clock time
        schedule Update(j, s+1, ts) at t+2d
    end if
end process
```

Figure 4–7: Update process for an order protocol based on logical clocks

Define $TC_{i,min}$ to be the minimum of $\{TC_i[j, \pi+1] \mid 1 \leq j \leq N \text{ and } j \neq i\}$. Since for any s, $1 < s \leq \pi+1$, and any $j \neq i$, $TC_i[j, s] \leq TC_i[j, s-1]$ (see Figure 4–7), any m, $m.TS \leq ts$, that is received after $TC_{i,min}$ has become larger than or equal to ts, is considered late and does not enter $accepted_i$, i.e. the message list $accepted_i$ becomes closed for such m after $TC_{i,min} \geq ts$. So, every m', $m'.TS \leq ts$, that is already in $accepted_i$ can be safely ordered after $TC_{i,min} \geq ts$. Such messages are removed from $accepted_i$ and put into another list, called $stable_i$, for ordering. An ordering relation « for distinct m and m' is defined as follows: $m \ll m'$, if $((m.TS < m'.TS)$ **or** $(m.TS = m'.TS$ **and** $m.O < m'.O))$.

As with the order protocol of the reference design, spurious and duplicate messages must be removed from $stable_i$ before message ordering. Message m in $stable_i$ is said to be *spurious* if and only if there is a message m' in $stable_i$ such that $m.O = m'.O$, $m.TS = m'.TS$ and $m.\mu \neq m'.\mu$. Spurious messages must have been formed by a faulty processor that failed by giving the same time–stamp to distinct messages m and m'. On the other hand, a message m' is said to be a *duplicate* of another message m if $m.\mu = m'.\mu$, $m.O \neq m'.O$ and $m \ll m'$.

The Order process to implement the protocol for processor P_i is presented in Figure 4–8. Again, the Order process is composed of three cyclic processes, Broadcast, Diffuse and Deliver, which execute in parallel, and share some data structures.

```
process Order
   var MC_i:message counter
       TC_i,min, TC_i[N, π+1]:timing counter
       accepted_i:list of message
   process Broadcast
      ...
   end process
   ||
   process Diffuse
      ...
   end process
   ||
   process Deliver
      ...
   end process
   ||
   process Update
      ...
   end process
end process
```

Figure 4–8: Order process for an order protocol based on logical clocks

The Broadcast process (Figure 4–9) deals with input messages received from RMQ. Messages received from a link are treated by the Diffuse process (Figure 4–10), whilst message ordering is performed by the Deliver process (Figure 4–11). Note that since the Diffuse process does not enter a received and timely m into $accepted_i$ if $equiv(m)$ is already in the list, and spurious and duplicate messages are removed by the Deliver process, the ordering relation « totally orders the messages in $stable_i$, i.e. if m « m', m is ordered and delivered before m'.

```
process Broadcast
   var M:message
       m:internal_message
       t:clock_time
   cycle
      remove M from RMQ
      form m: m.μ := M; m.TS := MC_i; m.O := i
      MC_i := MC_i+1
      send m
      insert m into accepted_i
      t := the local physical clock time
      for all P_j, 1 ≤ j ≤ N and j ≠ i do schedule Update(j, 1, m.TS) at t+2d end for all
   end cycle
end process
```

Figure 4–9: Broadcast process for an order protocol based on logical clocks

```
process Diffuse
   var m:message
       t:clock_time
   cycle
      receive m
      if m.TS ≤ TC_i[m.O, |m.S|] or an equiv(m) ∈ accepted_i then discard m
      else
         insert m into accepted_i
         MC_i := max{MC_i, m.TS+1}
         t := the local physical clock time
         schedule Update(m.O, 1, m.TS) at t+d
         for all P_j, 1 ≤ j ≤ N and m.O ≠ j ≠ i do
            schedule Update(j, 1, m.TS) at t+2d
         end for all
         send m
      end if
   end cycle
end process
```

Figure 4–10: Diffuse process for an order protocol based on logical clocks

```
process Deliver
  var m:message
      stable_i, delivered_i:list of message
  cycle
    when TC_i,min < min{TC_i[j, π+1] | 1 ≤ j ≤ N and j ≠ i} do
      TC_i,min := min{TC_i[j, π+1] | 1 ≤ j ≤ N and j ≠ i}
      stable_i := {m | m ∈ accepted_i and m.TS ≤ TC_i,min}
      accepted_i := accepted_i–stable_i
      remove spurious messages from stable_i
      filter duplicate messages from delivered_i+stable_i
      delivered_i := delivered_i+stable_i
      order the messages in stable_i according to « relation
      for all m in stable_i do deposit m into the appropriate DMQ_i end for all
    end when
  end cycle
end process
```

Figure 4–11: Deliver process for an order protocol based on logical clocks

4.3.1.2. Protocol Correctness

Notations

We assume the following notations to help in sketching the proof of the correctness of the above protocol (an extended proof can be found in Appendix A). $START_i(j, s, \geq ts)$ denotes the smallest real time instance when $TC_i[j, s]$ becomes larger than or equal to ts. (That is, just before real time $START_i(j, s, \geq ts)$, $TC_i[j, s]$ is less than ts.) $start_i(j, s, \geq ts)$ denotes P_i's clock time at real time $START_i(j, s, \geq ts)$. Also, $END_i(\leq ts)$ denotes the largest real time instance when MC_i is less than or equal to ts. (That is, just after real time $END_i(\leq ts)$, MC_i is larger than ts and P_i will not form and send any m, $m.TS \leq ts$.) Unless stated otherwise, the bounds on ts and s are: $ts \geq 1$ and $1 \leq s \leq \pi+1$. For simplicity, we assume that a non–faulty processor executes the instructions of the protocol in zero real/clock time. Realising this assumption requires an increase in the value of d, which is possible as the proofs impose no upper bound on d.

Lemma 2.1: $start_i(j, s+1, \geq ts) - start_i(j, s, \geq ts) = 2d$, for non–faulty P_i, any P_j, $j \neq i$, and any s, $1 \leq s \leq \pi$.

Lemma 2.2: $|start_i(j, s, \geq ts) - start_i(r, s, \geq ts)| \leq d$, for non–faulty P_i, and any P_j and P_r, $j \neq i \neq r$.

Lemma 2.3: $|START_i(j, 1, \geq ts) - START_k(j, 1, \geq ts)| < \delta + 2d(2\rho)$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$.

Lemma 2.4: $|START_i(j, s, \geq ts) - START_k(j, s, \geq ts)| < \delta + 2d(2\rho s)$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$.

Remark: The above lemmas are true for any randomly chosen, non–negative value of d. (Their proofs do not require any lower or upper bound on d.) The next lemma establishes the relation between the value of d and δ, as a function of ρ and π.

Lemma 2.5: $START_k(j, s+1, \geq ts) - START_i(j, s, \geq ts) > \delta$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$, and any s, $1 \leq s \leq \pi$, provided $d \geq \delta/(1-(2\pi+1)\rho)$.

This lemma ensures that if a non–faulty P_i receives and accepts m, $|m.S| < \pi+1$, then its diffused message is found timely when being received by another non–faulty P_k. If $|m.S| = \pi+1$, then one of the signers must be non–faulty, and the same guarantee applies. So, we state the following corollary.

Corollary 2.1: For any two non–faulty P_i and P_k, and any m, $i \neq m.O \neq k$, if m enters $accepted_i$ at $TIME_i$, then an $equiv(m)$ enters $accepted_k$ at $TIME_k$ such that $|TIME_i - TIME_k| < \delta$, provided $d \geq \delta/(1-(2\pi+1)\rho)$.

Lemma 2.6: $START_i(k, 1, \geq ts) - END_k(\leq ts) > \delta$, for non–faulty P_i and P_k, provided $d > \delta/(1-\rho)$.

This lemma ensures that if a non–faulty P_k forms and sends a message m, every non–faulty P_i accepts m upon reception. So, the corollary 2.1 is strengthened as below.

Corollary 2.2: For any two non–faulty P_i and P_k, and any m, if m enters $accepted_i$ at $TIME_i$, then an $equiv(m)$ enters $accepted_k$ at $TIME_k$ such that $|TIME_i - TIME_k| < \delta$, provided $d \geq \delta/(1-(2\pi+1)\rho)$.

Lemma 2.7: A message m enters $stable_i$ of non–faulty P_i within at most $2d(\pi+1)$ time after having entered $accepted_i$, where time is measured according to P_i's clock.

Lemma 2.8: Any two non–spurious m_1 and m_2 that enter $accepted_i$ of non–faulty P_i are delivered according to the ordering relation «.

Theorem 2.1: The protocol guarantees *agreement* and *order* conditions within a real time interval Σ_a, $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$, if $d \geq \delta/(1-(2\pi+1)\rho)$.

Proof: From corollary 2.2, any m formed and sent by a non–faulty P_i enters $accepted_i$, and will enter $accepted_j$ of every non–faulty P_j within δ time; also, for a given ts, and for every m, $m.TS = ts$, that enters $accepted_i$, there is an $equiv(m)$ that enters $accepted_j$ of non–faulty P_j. Any m that enters $accepted_i$, eventually enters $stable_i$ (Lemma 2.7) together with all m' such that $m'.TS = m.TS$ (see the algorithm of the Deliver process, Figure 4–11, page 82). Therefore, P_i will find m spurious if and only if P_j finds $equiv(m)$ spurious. From lemma 2.8, non–spurious entries of the message list *stable* are ordered according to « relation which is identical for both P_i and P_j since processor ordering is unique and known (assumption 1, page 54). This means that for every given ts, non–faulty processors order an identical set of m, $m.TS = ts$, in an identical manner and after ordering all m', $m'.TS < ts$. Thus the protocol meets atomicity and order requirements.

A message m formed and sent by a non–faulty processor cannot be found spurious in any non–faulty processor (see assumption 2, page 54). From lemma 2.7, we can state that any non–spurious m is ordered by a non–faulty processor within $2d(\pi+1)(1+\rho)$ real time after being accepted. Thus the protocol meets the termination requirement with $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$. Hence the theorem. ❑

Theorem 2.2: The protocol can guarantee ordering agreement in finite time only if $\pi < (1-\rho)/(2\rho)$.

Proof: For Σ_a to be finite and positive, d, $d \geq \delta/(1-(2\pi+1)\rho)$, must be finite and positive. Hence, $1-(2\pi+1)\rho > 0$, i.e. $\pi < (1-\rho)/(2\rho)$. ❑

Remark: For crystal clocks, ρ is typically 10^{-6}, hence π should be less than one half of a million.

4.3.1.3. Protocol Performance

From theorem 2.1, the maximum ordering delay Σ_{max} of the order protocol described in Section 4.3.1.1 is given by $\Sigma_{max} = 2d(\pi+1)(1+\rho)+\delta$. Also, d must be greater or equal to $\delta/(1-(2\pi+1)\rho)$. We choose $d = \delta/(1-(2\pi+1)\rho)$. The maximum stability delay is composed of two parts: the first part – $2d(\pi+1)(1+\rho)$ – is the maximum time interval during which a non–faulty processor will wait for messages m' after having received a new message m, $m' \ll m$; the second part – δ – is the maximum time that the message m received by the first non–faulty processor takes to be received by all other non–faulty processors, after the first processor had initiated m's broadcast. Note that a message normally becomes stable sooner at the processor that has initiated its broadcast rather than at the receivers of this broadcast. Thus, the minimum ordering delay is achieved when the broadcast initiated by the first non–faulty processor to receive the input message takes zero time to be received by all other non–faulty processors. Further, non–faulty processors are running at the fastest rate. In this situation we have $\Sigma_{min} = 2d(\pi+1)(1-\rho)$. Thus, we can express the actual stability delay of the protocol presented as follows:

$$2d(\pi+1)(1-\rho) \leq \Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta.$$

4.3.1.4. Finite Upper Bound on π

Theorem 2.2 indicates that there is a finite upper bound on π, which need not be the case with synchronised clock based protocols. This difference can be intuitively explained as follows: the execution of any synchronous and deterministic agreement protocol (including ours) proceeds in (at most $\pi+1$) rounds of message exchange between processors. Any two non–faulty processors should start and end a given round such that they simultaneously observe that round at least for a real time interval δ, for exchanging messages of that round. The real time difference with which any two non–faulty processors start the first round for agreement over a given message, is bounded by δ in our case (thus we do have implicit clock synchronisation) and by ε – the clock synchronisation bound – in other synchronised based protocols. When clocks are synchronised, the bound on the real time difference with which any two non–faulty processors start any given round of the agreement protocol is constant and independent from the number of the round.

On the other hand, if clocks are not (re)synchronised, the magnitude of this difference can increase with every subsequent round, and therefore is a function of π.

So, when clocks are not synchronised, the round length, say *rl*, must be large enough to ensure that the difference between the real time when an early–starting fast processor completes its $(\pi+1)^{th}$ round and the real time when a late–starting slow processor begins its $(\pi+1)^{th}$ round must be at least as large as δ. That is, $rl(1-\rho)(\pi+1)-(\delta+rl(1+\rho)(\pi)) \geq \delta$. Thus, a finite and positive value for *rl* is possible only if $\pi < (1-\rho)/(2\rho)$.

This is an important observation in our work: when clocks are not explicitly synchronised in a synchronous environment (where δ and ρ are known), reaching agreement becomes an asynchronous problem (where a finite value for *d* cannot be determined), if π exceeds a certain value; when *d* has no finite value, deterministic agreement cannot be guaranteed even in the presence of one benign, let alone Byzantine, failure [Fischer et al. 85].

With current clock technology, this threshold is found to be approximately half a million, which is large enough for protocols that do not rely on explicitly synchronised clocks to be useful in practice.

4.3.2. Reducing the Protocol Stability Delay for TMR nodes

In practice, three–processor failure–masking nodes (TMR nodes) are the most commonly selected replication strategy for masking nodes. In this section we discuss mechanisms to improve the performance of the protocol presented in Section 4.3.1, when applying it to implement TMR nodes. This protocol is essentially the same protocol of Section 4.3.1 for $N = 3$, to which a third timeliness check is added. This extra timeliness check is derived by exploiting the fact that each processor has only two other processors to reach agreement with.

4.3.2.1. Protocol Description

Let a TMR node be $\{P_i, P_j, P_k\}$ and processor P_i be non–faulty. Suppose that P_i receives and accepts a message *m*, $m.O = k$, at clock time t_i. As $\pi = 1$, any message received can either be single–signed or double–signed, and in accordance with check C1, P_i should not accept any *m'*, $m'.TS \leq ts$, $m'.O = j$ and $|m'.S| = 1$ after clock time t_i+2d, and *m'*, $m'.TS \leq ts$, $m'.O = j$ and $|m'.S| = 2$ after clock time t_i+4d. Similarly, in accordance with check C2, P_i should not accept any

m', $m'.TS \leq ts$, $m'.O = k$ and $|m'.S| = 1$ after clock time t_i+d, and m', $m'.TS \leq ts$, $m'.O = k$ and $|m'.S| = 2$ after clock time t_i+3d. However, since P_k is supposed to set MC_k equal to $m.TS+1$ when initiating the broadcast of m, MC_k of P_k must be larger than $m.TS$ when P_i receives m, i.e. at clock time t_i. Hence, a non–faulty P_k (by check C1) should not accept any single–signed m', $m'.TS \leq m.TS$, after t_i+2d. If we ignore the difference between the drift from real time of P_i's and P_k's clocks when they measure a $2d$ clock time interval, any double–signed m', $m'.O = j$ and $m'.TS \leq m.TS$, diffused by P_k must be received by P_i before t_i+3d (instead of t_i+4d, as per check C1).

Again, d has to be chosen in such a way that it compensates the difference between the drift of two non–faulty processors when measuring a $2d$ clock time interval. Figure 4–12 shows a worst case scenario where *equiv*(m) takes zero time to be received by P_i, and P_i is running fast, whilst P_k is running slow. (In Figure 4–12, T_s and T_d are respectively the real time when m was sent by P_k and the real time when m' was diffused by P_k, whilst ρ_i and ρ_k are the drift rates from real time of the clocks of processors P_i and P_k, respectively.)

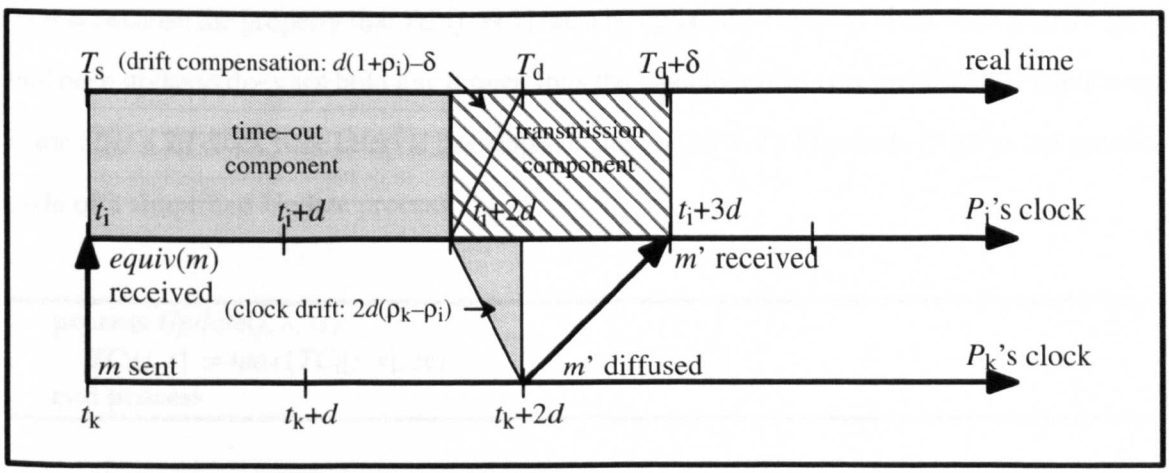

Figure 4–12: Compensating the difference between the drift of clocks in a TMR

In this situation, when P_k's clock reads t_k+2d, P_i's clock reads a value greater than t_i+2d, thus a timely m' received by P_k just before P_k's clock reads t_k+2d and then diffused to P_i may be received by P_i after its clock reads t_i+3d, being then discarded. Thus, d must be such that $3d(1+\rho_i) \geq 2d(1+\rho_k)+\delta$, that is $d \geq \delta/(1+3\rho_i-2\rho_k)$. In the worst case, $\rho_i = -\rho$ and $\rho_k = \rho$, which

leads to $d \geq \delta/(1-5\rho)$. (Note that for a TMR node, the protocol described in Section 4.3.1 required $d \geq \delta/(1-3\rho)$, which is a smaller lower bound for d.)

Thus, assuming that $d \geq \delta/(1-5\rho)$, the third check becomes:

check C3: for any non–faulty P_i, a received m', $m'.TS \leq ts$ and $|m'.S| = 2$, is timely only if it is received before $t+3d$, where t is the smallest local clock time when a message m, $m.O \neq m'.O \neq i$ and $m.TS \geq ts$, was accepted.

As we have shown, the value of $TC_i[k, 2]$ becomes at least ts at P_i's clock time t_i+3d, if a message m, $m.O \neq k$ and $m.TS = ts$, was received and accepted at t_i. In the previous protocol, this was possible only at t_i+4d. So, this optimisation speeds up the progress of $TC_{i,min}$ and reduces Σ_{max} by δ.

The structure of this optimised protocol is very similar to the one of the protocol presented in Section 4.3.1. The changes that are to be made over the previous protocol to incorporate the third timeliness check described above are as follows. First, the Update process must be changed. This is because the property that $TC_i[j, s+1]$ must be updated $2d$ units of clock time after $TC_i[j, s]$ had been updated does not hold any longer, thus the Update process cannot schedule itself to execute after a $2d$ clock time interval has elapsed (see Figure 4–7). Figure 4–13 gives the pseudo–code of a simplified Update process.

```
process Update(j, s, ts)
   TC_i[j, s] := max{TC_i[j, s], ts}
end process
```

Figure 4–13: Update process for TMR nodes

Also, since the Update process no more schedules executions of itself, both the Broadcast and the Diffuse processes must explicitly schedule executions of the Update process for all timing counters $TC_i[j, s]$. Figure 4–14 and Figure 4–15 indicates the changes that are to be made on the Broadcast and Diffuse processes, respectively. On the other hand, the Deliver process of this protocol is the same process presented in the previous protocol, with $N = 3$ (see Figure 4–11).

```
process Broadcast
   var M:message
       m:internal_message
       t:clock_time
   cycle
      remove M from RMQ
      form m: m.μ := M; m.TS := MC_i; m.O := i
      MC_i := MC_i+1
      send m
      insert m into accepted_i
      t := the local physical clock time
      for all P_j, 1 ≤ j ≤ 3 and j ≠ i, do
         schedule Update(j, 1, m.TS) at t+2d
         schedule Update(j, 2, m.TS) at t+4d
      end for all
   end cycle
end process
```

Figure 4–14: Broadcast process for TMR nodes

```
process Diffuse
   var m:message
       t:clock_time
   cycle
      receive m
      if m.TS ≤ TC_i[m.O, |m.S|] or an equiv(m) ∈ accepted_i then discard m
      else
         insert m into accepted_i
         MC_i := max{MC_i, m.TS+1}
         t := the local physical clock time
         schedule Update(m.O, 1, m.TS) at t+d
         schedule Update(m.O, 2, m.TS) at t+3d
         for all P_j, 1 ≤ j ≤ 3 and m.O ≠ j ≠ i do
            schedule Update(j, 1, m.TS) at t+2d
            schedule Update(j, 2, m.TS) at t+3d
         end for all
         send m
      end if
   end cycle
end process
```

Figure 4–15: Diffuse process for TMR nodes

4.3.2.2. Protocol Correctness

We retain the notations $START_i(j, s, \geq ts)$, $start_i(j, s, \geq ts)$ and $END_i(\leq ts)$ of Section 4.3.1.2. Note that for a TMR node s is either 1 or 2, and the changes made to derive the TMR protocol do not modify the way the timing counters for $s = 1$ and the message counter (MC) are being updated. Again, the bound on ts is: $ts \geq 1$. Again, we assume that a non–faulty processor executes the instructions of the protocol in zero real/clock time.

Lemma 3.1: $START_k(j, 2, \geq ts) - START_i(j, 1, \geq ts) > \delta$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$, provided $d \geq \delta/(1-5\rho)$.

This lemma ensures that if a non–faulty P_i receives and accepts a single–signed m, then its diffused double–signed message is found timely when being received by another non–faulty P_k. If a non–faulty P_i receives and accepts a double–signed m, then any other non–faulty processor of the node has already accepted an $equiv(m)$. Lemma 2.6 guarantees that a message m accepted by a non–faulty P_i when P_i formed and sent m is also accepted by all other non–faulty processors. So, we state the following corollary.

Corollary 3.1: For any two non–faulty P_i and P_k, and any m, if m enters $accepted_i$ at $TIME_i$, then an $equiv(m)$ enters $accepted_k$ at $TIME_k$ such that $|TIME_i - TIME_k| < \delta$, provided $d \geq \delta/(1-5\rho)$.

Lemma 3.2: A message m, $m.O = i$, enters $stable_i$ of non–faulty P_i within $4d$ time after having entered $accepted_i$, whilst a message m', $m'.O \neq i$, enters $stable_i$ of non–faulty P_i within $3d$ time after having entered $accepted_i$, where time is measured according to P_i's clock.

Theorem 3.1: The protocol for TMR nodes guarantees *agreement* and *order* conditions within a real time interval Σ_a, $\Sigma_a \leq 4d(1+\rho)$, if $d \geq \delta/(1-5\rho)$.

Proof: From corollary 3.1, any m formed and sent by a non–faulty P_i enters $accepted_i$, and will enter $accepted_j$ of every non–faulty P_j within δ time; also, for a given ts, and for every m, $m.TS = ts$, that enters $accepted_i$, there is an $equiv(m)$ that enters $accepted_j$ of non–faulty P_j. Since the Deliver process for this protocol is the equal to the Deliver process of the previous protocol, any m that enters $accepted_i$, eventually enters $stable_i$

(Lemma 3.2) together with all m' such that $m'.TS = m.TS$ (see the algorithm of the Deliver process, Figure 4–11, page 82). Therefore, P_i will find m spurious if and only if P_j finds $equiv(m)$ spurious. From lemma 2.8, non–spurious entries of the message list *stable* are ordered according to « relation which is identical for both P_i and P_j since processor ordering is unique and known (assumption 1, page 54). This means that for every given ts, non–faulty processors order an identical set of m, $m.TS = ts$, in an identical manner and after ordering all m', $m'.TS < ts$. Thus the protocol meets atomicity and order requirements.

A message m formed and sent by a non–faulty processor cannot be found spurious in any non–faulty processor (see assumption 2, page 54). From lemma 3.3, any non–spurious m, $m.O = i$, in $accepted_i$ enters $stable_i$ within at most $4d(1+\rho)$ (real) time after having entered $accepted_i$; also, any non–spurious m', $m'.O \neq i$, in $accepted_i$ enters $stable_i$ within at most $3d(1+\rho)$ (real) time after having entered $accepted_i$, and may take at most δ time to be received by P_i. As $d \geq \delta/(1-5\rho)$, we have $d(1+\rho) > \delta$, and the protocol meets the termination requirement with $\Sigma_a \leq 4d(1+\rho)$. Hence the theorem. ❑

4.3.2.3. Protocol Performance

From theorem 3.1, the maximum ordering delay Σ_{max} of the protocol described in Section 4.3.2.1 is given by $\Sigma_{max} = 4d(1+\rho)$, with d as small as $\delta/(1-5\rho)$. Note that unlike the case in the previous protocol, in this protocol a message becomes stable later at the processor that have initiated its broadcast rather than at the receivers of this broadcast. Thus, it is easy to see that the actual stability delay of a message for a TMR node using the order protocol discussed in this section varies according with the running rates of the clocks of the non–faulty processors of the node, as indicated in the expression below:

$$4d(1-\rho) \leq \Sigma_a \leq 4d(1+\rho).$$

In other words, the minimum ordering delay is achieved when all non–faulty processors are running at the fastest rate $(1-\rho)$, whilst the maximum stability delay corresponds to the case when

the first non–faulty processor to receive the message from the network is running at the slowest rate $(1+\rho)$.

However, within the architecture of our failure–masking nodes, it is possible to achieve a smaller lower bound for the actual stability delay of a message for this protocol. It is possible that every non–faulty processor of a failure–masking node receives a copy of a particular message from the network. Therefore, copies of the message that have been relayed by other processors of the node may become stable at all non–faulty processors prior to the time when the message they have received from the network is due to stabilise. If we assume that δ_a is the actual time taken by a message received from the network to be relayed to the other non–faulty processors of the node, and that λ_a, $\lambda_a \geq 0$, is the actual message reception skew for a particular message, i.e. the difference between the real times when the first two copies of a particular message are received from the network by any two non–faulty processors of the node, then a relayed message will become stable at any non–faulty processor P_i within a time interval no longer than:

$$\lambda_a+\delta_a+3d(1+\rho_i),$$

where ρ_i is rate with which P_i's clock drift from real time.

Thus, whenever $\lambda_a+\delta_a$ is smaller than $d(1-\rho)$, the lower bound on the actual stability delay of a message is given by $\lambda_a+\delta_a+3d(1-\rho)$. Note that if only one non–faulty processor receives a particular message from the network and the other non–faulty processors do not receive a copy of the same message from the network, then $\lambda_a = \infty$, but the message is still ordered at all non–faulty processors of the node.

4.3.3. An *Early–Order* Protocol for Nodes with *fifo* Internal Channels

The protocols discussed so far make a pessimistic assumption about the faulty behaviour of the processors forming the node. In typical systems, processor failures are rare events, and communication between two non–faulty processors normally takes much less than δ units of time. It is sensible, therefore, to design protocols that can perform better when the system is in a failure–free state. In [Gopal et al. 90], atomic broadcast protocols are presented which can potentially deliver messages earlier when the system is in a failure–free state. As discussed before, (normal) atomic broadcast protocols which rely on synchronising clocks of processors, deliver messages after a

specified interval of time has elapsed; a message received by a processor at clock time t can be delivered by any processor at clock time $t+\Delta$, where Δ is the termination bound of the protocol (*latency* in [Gopal et al. 90] terminology). An early–delivery atomic broadcast protocol, on the other hand, permits some executions where messages are delivered in less time than the protocol's latency [Gopal et al. 90].

Similar to the early–delivery atomic broadcast protocol of [Gopal et al. 90], we develop an *early–order* protocol that *guarantees* that whenever the system is in a failure–free state, messages are ordered in less than the maximum stability delay (Σ_{max}) of previous protocols. Note that in the protocols described before, although some messages can be ordered in less than the protocol's maximum stability delay ($\Sigma_{min} \leq \Sigma_{max}$), those protocols are not early–order protocols because they cannot *guarantee* that messages are ordered in less than Σ_{max}, even when the system is in a failure–free state (e.g. when some processors run at the slowest rate and messages take δ time to be transmitted, it is possible to have $\Sigma_a = \Sigma_{max}$).

It is important to make a distinction between early–delivery/early–order protocols and early stopping protocols [Dolev et al. 90]. Early stopping protocols are protocols whose performance depends on the number of components that have actually failed, i.e. these protocols take advantage from the fact that less than the maximum number of components, whose failures the protocol can tolerate, have failed, and deliver a better performance in such cases. On the other hand, both early–delivery atomic broadcast protocols and early–order protocols only achieve better performance in the case where no components (processors) have failed. (Unlike early–order protocols, early–delivery atomic broadcast protocols do not guarantee early–delivery, even when there are no failures [Gopal et al. 90].)

If failure–masking nodes are constructed in such a way that processors forming the node are fully connected, it is easy to implement an intra–node communication service which guarantees that messages sent over a link connecting a pair of processors within the node are received in the order they are sent. Assuming that intra–node communication in a failure–masking node has this *first–in–first–out* (*fifo*) property, we can modify the protocols previously presented, so that, in the absence of failures, early–order can be achieved, substantially reducing the protocol's actual stability delay.

In the protocols presented earlier the number of processors that have relayed a particular message (i.e. the number of signatures within the message) was a decisive factor on the determination of whether a received message was timely or not. However, in those protocols, receiver processors were not concerned about the actual sequences of processors through which received messages were relayed. To present the early–order protocol in this section we need to distinguish between the various possible sequence of processors through which a received message may have been relayed. We define a *path p* to be an ordered, non–empty sequence of at most $\pi+1$ distinct processors. We also define the path through which a message m has been received, $path(m)$, as the sequence of processors $P_1{:}P_2{:}...{:}P_s$, $1 \leq s \leq \pi+1$, where P_i, $1 \leq i \leq s$, is the processor whose signature is the i^{th} signature contained in m.

In the previous protocols, a non–faulty P_i updates its message counter MC_i whenever it initiates the broadcast of a new message, or when it accepts a message bearing a time–stamp greater or equal to the current value of MC_i. Therefore, if P_i receives a timely message m through a path p, and accepts m, then any non–faulty P_k which has signed m (and therefore is an element in p) must have had $MC_k > m.TS$ at the time it had diffused m. Assume that non–faulty P_k is the last component of p, $p = q{:}P_k$. Then the *fifo* property between any two non–faulty processors guarantees that any timely m', $m'.TS \leq m.TS$, that P_k has received through the path q, and then diffused to P_i, must be received by P_i before m. (This is also the case when $p = P_k$, and P_k has initiated the broadcast of both m and m'.) Thus, the *fifo* assumption has the following impact on checking the timeliness of received messages:

> **check C4**: when a non–faulty P_i receives a message m, through a path p, it can immediately decide that any message m', received later through the same path p, with $m'.TS \leq m.TS$ is untimely and must have been diffused by a faulty processor.

Before introducing the early–order protocol, we discuss how the use of the timeliness check C4 described above by non–faulty processors executing the protocol described in Section 4.3.2 can reduce the actual stability delay of messages for that protocol. Let us assume a failure–free scenario for a TMR node composed of processors $\{P_i, P_j, P_k\}$. For the sake of illustration, assume that P_i, P_j and P_k have each simultaneously received, at real time T, a copy of the same input message, and, as per the protocol of Section 4.3.2, have respectively initiated the broadcast of

messages m_i, m_j, and m_k, with $m_i.\mu = m_j.\mu = m_k.\mu$. At most by $T+\delta$, P_i, for instance, will have received the single–signed messages m_j, and m_k from P_j and P_k respectively; will have applied the timeliness check C4; and will not accept any single–signed messages m_j' and m_k', $m_j'.TS \leq m_j.TS$, $m_k'.TS \leq m_k.TS$, diffused by P_j and P_k, respectively. Similarly, if P_i receives double–signed messages m_k and m_j diffused through paths P_k:P_j and P_j:P_k, respectively, it will not accept any double–signed messages m_j', m_k', $m_j'.TS \leq m_j.TS$, $m_k'.TS \leq m_k.TS$, diffused through paths P_k:P_j and P_j:P_k, respectively. In the absence of failures, double–signed messages m_j, m_k received by P_i will be received by at most $T+2\delta$, thus by this time, if P_i has received these double–signed messages, it is guaranteed that P_i will not accept any message m', either single–signed or double–signed, such that $m'.TS$ is less or equal to the minimum of $\{m_i.TS, m_j.TS, m_k.TS\}$. It is easy to see that the same analysis also follows for processors P_j and P_k. Hence, one of the copies of the input message will become stable at all processors of the node by at most $T+2\delta$, yielding an actual stability delay of at most 2δ, which is much smaller than the minimum stability delay (minimum of $\{\lambda_a+\delta_a+3d(1-\rho), 4d(1-\rho)\}$) of the protocol of Section 4.3.2.

Note however, that a reduced stability delay was achieved only when all processors initiate a broadcast (almost) at the same time. In fact, this is a necessary condition for early–order: early–order of a message m broadcast by P_i can be guaranteed only if all processors other then P_i subsequently broadcast m', $m'.TS \geq m.TS$. Further, every timely *equiv*(m) received, must be diffused so that processors receive a message through every possible path that originates from $m.O$. In the example discussed above, if P_i receives the single–signed message m_k from P_k after it has received the double–signed message m_k through the path P_k:P_j, then P_i must countersign the single–signed m_k received, and send the resulting double–signed m_k to P_j even though a copy of m_k is already in $accepted_i$. Thus, the cost of early–order is twofold: i) every timely message received (not just those that enter *accepted*) must be diffused; and ii) processors must frequently initiate broadcasts.

In the next section we present the modifications necessary to make the protocol of Section 4.3.1 to produce an early–order protocol. It will become clear that the same modifications can also be applied to the protocol presented in Section 4.3.2.

4.3.3.1. Protocol Description

To make the full use of the node's *fifo* property, a non–faulty P_i receives a timely message m, uses it to update its counters and diffuses it if necessary (i.e $|m.S| < \pi+1$), whether or not m is already in its $accepted_i$. We say that P_i has received the *complete broadcast set* of a particular message m initiated by a processor P_j, when P_i has received some $equiv(m)$ through every possible path p beginning from the originator of m (i.e. $m.O = j$). Note that when P_i receives the complete broadcast set of a message m, no more timely messages associated with the broadcast of any m', $m'.TS \leq m.TS$, and $m'.O = m.O$, may be received by P_i. As discussed before, when P_i receives a complete broadcast set from every other processor within the node, and P_i itself has also initiated a broadcast, then it is guaranteed that at least one of these broadcasts is stable. Since the time to receive a complete broadcast set depends on the *actual* transmission delays – δ_a, rather than on the *maximum* transmission delay – δ, ordering is achieved earlier when all processors execute the protocol correctly. To guarantee that processors initiate broadcasts frequently enough, we introduce the notion of a *null–broadcast*, which is initiated by a non–faulty processor whenever a message m associated with a new (non–null) broadcast is received. Messages m associated with a null–broadcast have $m.\mu = \phi$ (we assume that no authentic message received from the network is equal to ϕ).

In order to present the protocol, we define the following notations:

- $path(m)$, which denotes the path through which m was received;
- $|p|$, which denotes the length of the path p; and
- $origin(p)$, which denotes the first processor in the path p.

We also define the following set of paths:

- $paths_i = \{p \mid P_i \text{ not in } p\}$.

In addition to maintaining timing counters $TC_i[j, s]$, for every P_j, $j \neq i$, and for every s, $1 \leq s \leq \pi+1$, each non–faulty P_i also maintains *path counters* $PC_i[p]$ for each path p, through which P_i can receive a message. These path counters are integer values which only increase with the passage of real time, and are initialised to zero when the node is started. P_i maintains the path counters $PC_i[p]$ by either scheduling a revised version of the Update process (see Figure 4–16) to

be executed at appropriated times, or by explicitly updating the value of $PC_i[p]$ whenever a timely message is received through the path p (see the algorithm of the Diffuse process in Figure 4–18).

```
process Update(j, s, ts)
   var t:clock_time
   for all p | p ∈ path_i and |p| = s and origin(p) = j do
      if PC_i[p] < ts then PC_i[p] := ts end if
   end for all
   if TC_i[j, s] < ts then TC_i[j, s] := ts end if
   if s < (π+1) then
      t := the local physical clock time
      schedule Update(j, s+1, ts) at t+2d
   end if
end process
```

Figure 4–16: Update process for early–order protocol

We define $PC_{i,min}$ to be the minimum of $\{PC_i[p] \mid \forall p, p \in paths_i\}$. The pseudo–code for the Broadcast, Diffuse and Deliver processes for this protocol is shown in Figure 4–17, Figure 4–18 and Figure 4–19, respectively.

```
process Broadcast
   var M:message
       m:internal_message
       t:clock_time
   cycle
      remove M from RMQ
      form m: m.μ := M; m.TS := MC_i; m.O := i
      MC_i := MC_i+1
      send m
      insert m into accepted_i
      t := the local physical clock time
      for all P_j, 1 ≤ j ≤ N and j ≠ i do schedule Update(j, 1, m.TS) at t+2d end for all
   end cycle
end process
```

Figure 4–17: Broadcast process for early–order protocol

```
process Diffuse
   var m:message
       t:clock_time
   cycle
      receive m
      if m.TS ≤ PC_i[path(m)] then discard m
      else
         PC_i[path(m)] := m.TS
         send m
         if an equiv(m) ∉ accepted_i and m.μ ≠ φ then
               insert m into accepted_i
               MC_i := max{MC_i, m.TS+1}
               t := the local physical clock time
               schedule Update(m.O, 1, m.TS) at t+d
               for all P_j, 1 ≤ j ≤ N and m.O ≠ j ≠ i do
                  schedule Update(j, 1, m.TS) at t+2d
               end for all
               form m: m.μ ≠ φ; m.TS := MC_i; m.O := i
               MC_i := MC_i+1
               send m
         end if
      end if
   end cycle
end process
```

Figure 4–18: Diffuse process for early–order protocol

```
process Deliver
   var m:message
       stable_i, delivered_i:list of message
   cycle
      when PC_i,min < min{PC_i[p] | ∀p, p ∈ paths_i} do
         PC_i,min := min{PC_i[p] | ∀p, p ∈ paths_i}
         stable_i := {m | m ∈ accepted_i and m.TS ≤ PC_i,min}
         accepted_i := accepted_i–stable_i
         remove spurious messages from stable_i
         filter duplicate messages from delivered_i+stable_i
         delivered_i := delivered_i+stable_i
         order the messages in stable_i according to « relation
         for all m in stable_i do deposit m into the appropriate DMQ_i end for all
      end when
   end cycle
end process
```

Figure 4–19: Deliver process for early–order protocol

Note that despite updating the timing counters $TC_i[j, s]$, the early–order protocol does not use them either for checking the timeliness of received messages, nor for stabilising messages to be delivered. In fact, as will be seen later in Appendix A, we keep the timing counters in the early–order protocol for the sole purpose of simplifying the task of proving the correctness of the early–order protocol.

4.3.3.2. Protocol Correctness

We maintain the notations $START_i(j, s, \geq ts)$ and $END_i(\leq ts)$ as defined previously, and we introduce the following notation: $START_i(p, \geq ts)$, for a non–faulty P_i, all paths $p \in paths_i$ and $ts \geq 1$, denotes the smallest real time instance when $PC_i[p]$ becomes larger than or equal to ts. (That is, just before real time $START_i(p, \geq ts)$, $PC_i[p]$ is less than ts.)

Lemma 4.1: $START_i(p, \geq ts) \leq START_i(origin(p), |p|, \geq ts)$, for any non–faulty processor P_i and any path p, $p \in paths_i$. If $START_i(p, \geq ts) < START_i(origin(p), |p|, \geq ts)$, then P_i must have received a timely message m at some real time $TIME_i$, with $m.TS \geq ts$, $path(m) = p$ and $TIME_i < START_i(p, \geq ts) < START_i(origin(p), |p|, \geq ts)$.

Lemma 4.2: If m_1 and m_2 are two distinct messages sent by a non–faulty P_i, with $path(m_1) = path(m_2)$, then they were sent in the increasing order of their timestamps.

Corollary 4.1: If m_1 and m_2 are distinct messages sent by a non–faulty P_i and received by another non–faulty P_k, then P_k will receive them in the increasing order of their timestamps.

Lemma 4.3: Any message m whose broadcast is initiated by a non–faulty P_i will be found timely by every non–faulty P_k, provided $d > \delta/(1-\rho)$.

Lemma 4.4: Any message m received by a non–faulty P_i from a non–faulty P_k, will be found timely, provided $d \geq \delta/(1-(2\pi+1)\rho)$, and $\pi < (1-\rho)/2\rho$.

Lemma 4.5: If all processors of the node are non–faulty, then the protocol's actual stability delay is given by $0 \leq \Sigma_a < (\pi+2)\delta$.

Lemma 4.6: A message m enters $stable_i$ of non–faulty P_i within at most $2d(\pi+1)$ time after having entered $accepted_i$, where time is measured according to P_i's clock.

Lemma 4.7: Any two non–spurious m_1 and m_2 that enter $accepted_i$ of non–faulty P_i is delivered according to the ordering relation «.

Theorem 4.1: The early–order protocol guarantees *agreement* and *order* conditions, within a real time interval Σ_a, $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$, so long as $d \geq \delta/(1-(2\pi+1)\rho)$, and $\pi < (1-\rho)/2\rho$. In the absence of failures the protocol guarantees early–order of messages in Σ_{eo}, $0 \leq \Sigma_{eo} < (\pi+2)\delta$.

Proof: From lemmas 4.3 and 4.4, any m sent by a non–faulty P_i to a non–faulty P_k will be found timely by P_k; therefore, for a given ts, every m, $m.TS = ts$, that enters $accepted_i$, there is an $equiv(m)$ that enters $accepted_j$ of non–faulty P_j. Any m that enters $accepted_i$, eventually enters $stable_i$ (Lemma 4.6) together with all m' such that $m'.TS = m.TS$ (see the algorithm of the Deliver process, Figure 4–19, page 98). Therefore, P_i will find m spurious if and only if P_j finds $equiv(m)$ spurious. From lemma 4.7, non–spurious entries of the message list *stable* are ordered according to « relation which is identical for both P_i and P_j since processor ordering is unique and known (assumption 1, page 54). This means that for every given ts, non–faulty processors order an identical set of m, $m.TS = ts$, in an identical manner and after ordering all m', $m'.TS < ts$. Thus the protocol meets atomicity and order requirements.

A message m formed and sent by a non–faulty processor cannot be found spurious in any non–faulty processor (see assumption 2, page 54). From lemma 4.6, we can state that any non–spurious m is ordered by a non–faulty processor within $2d(\pi+1)(1+\rho)$ real time after being accepted. Thus the protocol meets the termination requirement with $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$. Early order in Σ_{eo}, $0 \leq \Sigma_{eo} < (\pi+2)\delta$, follows from lemma 4.5. Hence the theorem. ❑

4.3.3.3. Protocol Performance

Since the timeliness check C4 is superimposed on the timeliness checks C1 and C2 (and also to check C3, in the case of TMR nodes), the maximum stability delay of the early–order protocol presented in this section is the same of that for the protocol presented in Section 4.3.1 (and the same as that for the protocol presented in Section 4.3.2 in the case of TMR nodes). On the

other hand, in the absence of failures, the actual stability delay of the protocol is given by Σ_{eo}, $0 \leq \Sigma_{eo} < (\pi+2)\delta$. The minimum stability delay is achieved when all messages associated with the broadcast of a particular message take zero time to be transmitted. Denoting the average time to transmit messages during the broadcast of a particular message by δ_{av}, we have:

$$0 \leq \Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta; \text{ and in the absence of failures } \Sigma_a = (\pi+2)\delta_{av}.$$

For a TMR node we have:

$$0 \leq \Sigma_a \leq 4d(1+\rho); \text{ and in the absence of failures } \Sigma_a = 3\delta_{av}.$$

4.4. Node Overhead Analysis

Running an application on a failure–masking node involves the execution of an order protocol for ordering input messages, and a voter protocol for validating output messages. These operations are not present when the application runs on an unreplicated node, thus, the extra overheads incurred by a failure–masking node are mainly made up by the execution of voter and order protocols. In this chapter we have discussed the design of failure–masking nodes incorporating a standard voter protocol (Section 4.2.1), and different order protocols (Section 4.2.2 and Section 4.3). We now assess the overheads imposed by these protocols in terms of both the extra delay in outputing the response for a request, and the extra intra–node message traffic that is generated when attending a request. We start by concentrating on the overheads imposed by the various order protocols developed. We compare the performance of the efficient order protocols described in Section 4.3 against the performance of the traditional synchronised clock based order protocols, assuming the best possible clock environment for the latter. We then analyse the overheads imposed by the voter protocol.

4.4.1. Ordering Overhead

Order protocols must perform message diffusion, to attain agreement, and must also incorporate mechanisms which guarantee that messages are delivered in the same order at all non–faulty processors of the node. These mechanisms are responsible for the introduction of an stability delay which represents the main time overhead of an order protocol. We first analyse the

stability delay overhead, and later we analyse the extra intra–node traffic generated by each different order protocol presented.

4.4.1.1. Stability Delay

For the sake of comparison, we use the upper bound on the protocols' actual stability delay. From theorem 2.1, the maximum stability delay for the protocol of Section 4.3.1 is given by $\Sigma_{max} = 2d(\pi+1)(1+\rho)+\delta$. We choose $d = \delta/(1-(2\pi+1)\rho)$, which, neglecting the higher order terms of ρ ($O(\rho^2)$), gives a maximum stability delay of $\Sigma_{max} = (3+2\pi+\rho)\delta/(1-(2\pi+1)\rho)$.

Let us denote the maximum stability delay for a synchronised clock based protocol by Σ_{sync}. [Dolev–Strong 82] reports that these protocols require at most $\pi+1$ rounds of message exchange between processors. Thus, if a non–faulty P_i initiates the broadcast of a message m at its synchronised clock time t_i, then every other non–faulty P_k orders m at its respective synchronised clock time $t_i+(\pi+1)(d_\Delta+\varepsilon)$, where $d_\Delta \geq \delta/(1-\rho)$, and ε is the precision of the clock synchronisation protocol (see [Cristian et al. 85] for example). (We ignore protocols such as those in [Babaoglu–Drummond 85] that assume redundant broadcast networks.) Hence, since the clocks of two non–faulty processors can differ by at most ε, in the worst case when P_i's clock is ε units of time ahead of the clock of some non–faulty P_k, $\Sigma_{sync} = (\pi+1)(d_\Delta+\varepsilon)(1+\rho')+\varepsilon$, where $1+\rho'$ is the maximum running rate of a synchronised clock. As clocks are synchronised by periodically adjusting the physical clock reading, $\rho' > \rho$ and the readings of a synchronised clock are not continuous, unless special arrangements are made. To make the comparison simpler, we assume the use of amortization techniques [Schmuck–Cristian 90] to ensure continuity in the readings of synchronised clocks without increasing ρ', and also a mechanism that makes $\rho' = \rho$ (see [Srikanth–Toueg 85]).

The exact value of ε depends on the frequency of clock adjustment and on the algorithm used to compute the adjustment. Adjustments can be computed either by referring to a reliable clock that is external to the node, or by using authenticated clock synchronisation protocols, such as [Halpern et al. 84, Lamport–MelliarSmith 85, Srikanth–Toueg 85], that are suitable for failure–masking nodes. We consider (for simplicity and in favour of synchronised clock based protocols) ε to be the maximum clock difference immediately after adjustments were made; i.e. we

ignore a component of ε that accounts for clock drift until next adjustment. Thus, the value of ε considered for comparison is $\delta/(1+\rho)$, $\delta/(1+\rho)$, and $(1+5\rho)\delta/(1+\rho)$, when respectively, an external clock, [Halpern et al. 84] and [Srikanth–Toueg 85] are used to compute the adjustment. (We omit the authenticated protocol of [Lamport–MelliarSmith 85] as it is admitted to be less efficient than [Halpern et al. 84].) Neglecting the higher order terms of ρ ($O(\rho^2)$), and assuming $d_\Delta = \delta/(1-\rho)$, we have $\Sigma_{sync} = (3+2\pi+2\pi\rho+\rho)\delta$ for $\varepsilon = \delta/(1+\rho)$, and $\Sigma_{sync} = (3+2\pi+7\pi\rho+11\rho)\delta$ for $\varepsilon = (1+5\rho)\delta/(1+\rho)$. When $\varepsilon = \delta/(1+\rho)$, the difference between the maximum stability delay of the protocols, $\Sigma_{max}-\Sigma_{sync}$, is given by $(4\pi^2+6\pi+3)\delta\rho/(1-(2\pi+1)\rho) \approx O(\delta\rho\pi^2)$. For small values of π, the difference is small: approximately $31\delta\rho$ and $13\delta\rho$ when π is 2 and 1, respectively. With [Srikanth–Toueg 85], the difference between the maximum stability delay of the protocols, $\Sigma_{max}-\Sigma_{sync}$, is $(4\pi^2+\pi-7)\delta\rho/(1-(2\pi+1)\rho) \approx O(\delta\rho\pi^2)$, which evaluates to approximately $10\delta\rho$ and $-2\delta\rho$ when π is 2 and 1, respectively.

For TMR nodes however, the optimised version presented in Section 4.3.2 can reach agreement faster than any synchronised clock based protocol, and is better suited for building the more practical TMR nodes. For $\pi = 1$, we have $\Sigma_{max} = 4\delta(1+\rho)/(1-5\rho)$, and for $\varepsilon = \delta/(1+\rho)$, and $\varepsilon = (1+5\rho)\delta/(1+\rho)$, we have $\Sigma_{sync} = (5+3\rho)\delta$ and $\Sigma_{sync} = (5+18\rho)\delta$, respectively. Thus, the difference between the maximum stability delay of the protocols, $\Sigma_{sync}-\Sigma_{max}$, is $(1-26\rho)\delta/(1-5\rho) \approx O(\delta)$, when $\varepsilon = \delta/(1+\rho)$, and $(1-11\rho)\delta/(1-5\rho) \approx O(\delta)$, when $\varepsilon = (1+5\rho)\delta/(1+\rho)$.

We now study the early–order protocol of Section 4.3.3. In the absence of failures, that protocol guarantees early–order in $\Sigma_a = (\pi+2)\delta_{av}$, where δ_{av}, $0 \le \delta_{av} < \delta$, is the average transmission delay experienced by the internal messages associated with the early–order of a particular message. We compare the performance of the protocol of Section 4.3.3 with the performance of an order protocol built upon the early–delivery atomic broadcast protocols presented in [Gopal et al. 90], which assume a synchronised clock abstraction. It is worth noting that since early–delivery is not guaranteed, an order protocol built upon early–delivery atomic broadcast protocols, although able to order some messages in considerably less than Σ_{max}, are not early–order, as they cannot *guarantee* that messages are ordered sooner when the system is in a failure–free state.

A family of early–delivery atomic broadcast protocols is presented in [Gopal et al. 90]. The first protocol presented is an α–protocol which early–delivers messages in $\pi(d+\varepsilon)+\delta_{av}$, as measured by the clock of the processor delivering the message. Thus, an ordering protocol based on the α–protocol may order messages as soon as $\pi(d+\varepsilon)(1-\rho)$. The α–protocol is extended in two directions leading to θ–protocols, and β–protocols. The class of θ–protocols, in its best case θ_0, early–delivers messages in $\Sigma_a = (2\pi+1)\delta_{av}$, whilst the β–protocol early–delivers messages in $\Sigma_a = 2\delta_{av}$. However, both the θ–protocols and the β–protocol are presented within a framework where physical clocks are assumed to be perfectly synchronised (i.e. $\varepsilon = 0$).

4.4.1.2. Intra–Node Message Traffic

In [Cristian et al. 85], the overhead of the atomic broadcast protocol in terms of internal message traffic required when broadcasting a message, is analysed taking in account the average number of internal messages diffused per communication link per broadcast. This reflects the rather general framework within which that work was developed. Within the framework of our failure–masking nodes it is easy to quantify the total number of intra–node messages involved per broadcast. We shall assume that the processors of a node are fully connected. Under this assumption, the extra intra–node message traffic associated with the ordering of input messages, μ_o, for the order protocol in our reference design, based upon the atomic broadcast protocol in [Cristian et al. 85] is given by:

$$\mu_o = (N-1)^2$$

According with the protocol presented in Section 4.3.1, in the absence of failures, when a sender processor initiates the broadcast of a message *m*, it diffuses *N*–1 single–signed copies of *m* to every other receiver processor of the node. In the worst case the *N*–1 receivers receive the single–signed message diffused by the sender as the first message associated with the broadcast of *m*, and therefore, diffuses it to the other *N*–2 processors which have not signed *m* yet. In the best case the *N*–1 receivers receive messages with increasing number of signatures as the first message associated with the broadcast of *m*, and therefore diffuse decreasing number of messages. There-

fore, the number of intra–node messages diffused per broadcast in the order protocol of Section 4.3.1 is given by:

$$\sum_{i=1}^{N-1} i \leq \mu_o \leq (N-1)^2$$

Remark: We note that the atomic broadcast protocol in [Cristian et al. 85] can be easily improved to present an intra–node message traffic overhead comparable with the one expressed above.

For the early–order protocol presented in Section 4.3.3, in the absence of failures, the broadcast of a message m generates as many intra–node messages as the number of different paths starting from the sender processor, and with length less or equal to π+1. Also, since the N–1 receivers start a null–broadcast to acknowledge the broadcast of m, the traffic of internal messages is multiplied by N, and is given by:

$$\mu_o = N \sum_{i=1}^{\pi+1} \prod_{j=1}^{i} (N-j)$$

The protocols in [Gopal et al. 90] also carry a great penalty in message traffic in order to early–deliver messages. In the θ–protocols, for instance, there must be several phases of internal acknowledgement messages exchange, before messages received from the network can be early–delivered. The β–protocol, on the other hand, has an internal traffic comparable with our protocol, however, the former can incur an extra penalty after early–ordering of a message m, whenever a *conflicting* message m' is received after m has been ordered (see [Gopal et al. 90] for details).

The acknowledgement mechanisms inherent to early–order protocols are responsible for the sharp increase on the number of intra–node messages exchanged when ordering messages, making these protocols less attractive, especially when N is large. However, since in the Voltan architecture all processors of the node receive messages from the network, it is possible to reduce the burden of the acknowledgement mechanism of the early–order protocol.

Within the framework of failure–masking nodes, our early–order protocol can be slightly modified, in such a way that early–order of messages can be achieved with a much lower intra–node message traffic overhead. The basic idea is to reduce the number of null–broadcasts necessary for the early–order of messages. Since processors of a failure–masking node continuously receive input messages from the network, it is possible to avoid the necessity of having N–1 null–broadcasts for every message ordered, by using (normal) broadcasts of messages received from the network (instead of null ones) to acknowledge the reception of a new broadcast initiated by another processor of the node (as in the TMR example discussed in the beginning of Section 4.3.1). Each processor keeps track of the time–stamp of the last message it has broadcast, and upon receiving a new broadcast, it only initiates a null–broadcast to acknowledge the received broadcast, if the time–stamp of the new broadcast is greater than the time–stamp of the last broadcast it has initiated. With this small modification, the intra–node traffic per message early–ordered can be reduced by a factor of N, and is given by:

$$\sum_{i=1}^{\pi+1}\prod_{j=1}^{i}(N-j) \le \mu_o \le N\sum_{i=1}^{\pi+1}\prod_{j=1}^{i}(N-j)$$

4.4.2. Voting Overhead

In a failure–masking node, output messages generated by application processes must be voted before being output by the node. Any voter protocol must perform rounds of message exchange between the processors, followed by the execution of a voting upon the messages received. Thus, the main overhead associated with the voter protocol is the unavoidable time to exchange the messages to be voted.

Messages should be exchanged until they have π+1 signatures appended to them, in which case they become valid messages, and can be output. Therefore, there must be π rounds of messages exchange, until valid messages can be generated. The actual stability delay (Σ_a) of an order protocol for a failure–masking node measures the delay since the first copy of a particular message is received by the node, until all non–faulty processors of the node have ordered the message and made it available to the appropriate application process. Thus, if we assume that an application process takes T units of real time to execute a particular request, then after Σ_a+T units of real

time has elapsed since the request was first received, all application processes executing on non–faulty processors will have their output messages available. Hence, the actual validation delay (ς_a) for the voter protocol of Section 4.2.1 is given by:

$$0 \leq \varsigma_a \leq \pi\delta.$$

Also, the extra intra–node message traffic associated with the validation of output messages, μ_v, is given by:

$$\mu_v = N \sum_{i=1}^{\pi} \prod_{j=1}^{i} (N - j)$$

4.5. Concluding Remarks

We have described the design of several failure–masking nodes. The nodes incorporate a standard voter protocol, and differentiate from each other only in the way input messages are ordered. The order protocols described in Section 4.3 make an important contribution towards building efficient failure–masking nodes, and achieve message ordering without requiring the clocks of a node to be maintained in bounded synchronism. The protocol of Section 4.3.1 indicates that when clocks are not explicitly synchronised, reaching agreement in a synchronous environment can become an asynchronous problem, if the number of faults to be tolerated exceeds a certain value. This means that there are situations where deterministic agreement cannot be guaranteed even in the presence of one benign, let alone Byzantine, failure. For the more practical case of TMR nodes, the protocol derived in Section 4.3.2 is guaranteed to give better performance than any synchronised clock based protocol.

The early–order protocol of Section 4.3.3 provides a very reduced stability delay, whenever the system is in a failure–free state. The protocol assumes that the intra–node communication presents an *fifo* property when delivering messages through a communication link between two processors of the node. When N is small it is feasible to have a fully connected intra–node network, over which an *fifo* communication service can be easily implemented. Hence, the protocol of Section 4.3.3, incorporating the modifications discussed in Section 4.3.2 can be used to imple-

ment efficient TMR nodes. We have implemented such nodes, and we discuss their implementation and performance evaluation later in Chapter 6.

Chapter 5
Soft Fail–Silent Nodes

5.1. Introduction

Following the general structure of a soft fail–silent node presented in Chapter 3, constructing a fail–silent node, requires the implementation of Validator and Order processes which execute at each processor of the node. The Validator process for a fail–silent node is a Comparator process. Its function is similar to the Voter process of the failure–masking nodes described in the previous chapter, except that once a mismatch is detected in any message comparison, the Comparator process must bring the node to a halt. Upon the detection of a failure, a non–faulty processor can halt the node by firstly stopping the activities of its Comparator process, so that no new π+1–signed messages are ever output by that processor, and secondly stopping the activities of its Sender process, so that the other processors of the node do not output new π+1–signed messages too (see Figure 3–8, page 56). In [Shrivastava et al. 92] it is suggested how fail–silent nodes can be derived through straightforward modifications of failure–masking nodes. However, in order to achieve an efficient implementation, it is necessary to perform a more thorough investigation on the effects that the reduced redundancy of fail–silent nodes have on the design of such nodes.

Since a fail–silent node must halt as soon as a fault is detected, the Comparator process plays a very important role in the design of the node. Any failure that affects the correct execution of either application or system processes of faulty processors is detected by the Comparator process of non–faulty processors. Thus, unlike failure–masking nodes, in a fail–silent node all sys-

tem processes, except the Comparator, can be designed assuming that they execute in a failure–free environment. Particularly, this property can be used to design a simpler and more efficient Order process. On the other hand, because it is not assumed that a majority of processors are non–faulty, special attention must be taken when designing the Comparator process.

Let an application process running on a correctly functioning unreplicated node take T units of real time to compute the response to an input message. The corresponding correct output from a fail–silent node takes at most $T' = T+T_{delay}$ units of time, where T_{delay}, $T_{delay} > 0$, is the bounded worst–case delay introduced by the redundancy management protocols. If the output from the fail–silent node is produced later than T' then the node is said to have suffered a performance failure [Cristian 91]. A soft fail–silent node can be in one of the following three states (see Figure 5–1):

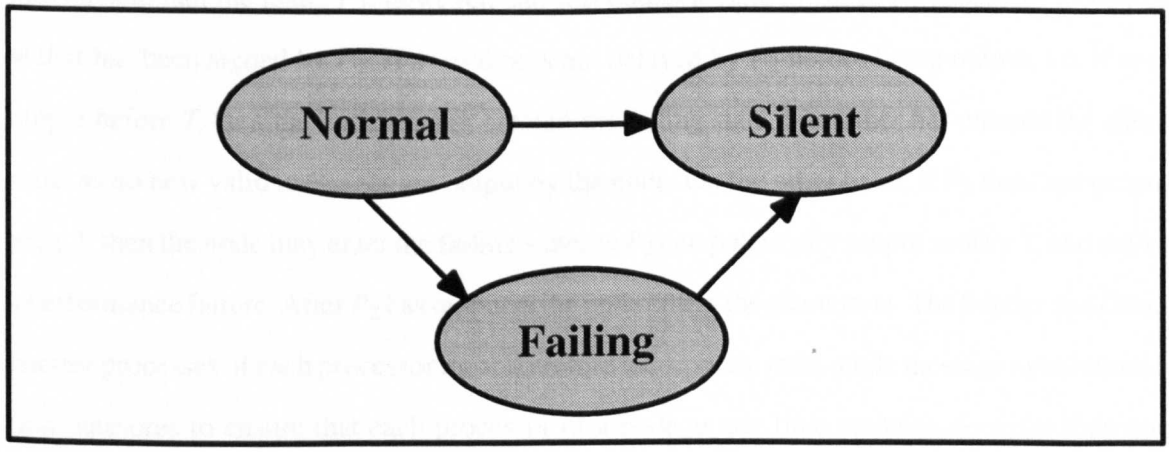

Figure 5–1: State transitions in a fail–silent node

i) *normal state*: in this state, a node produces valid outputs. Detection of an internal failure (by the Comparator) causes the node to irreversibly enter either the *failing* state or the *silent* state;

ii) *failing state*: this is an intermediate state in which the node can suffer at most one performance failure. From this state the node eventually enters the terminal silent state;

iii) *silent state*: when the node is in this state, no new valid messages are produced by the node. Any messages produced by the node can only be in-

valid or copies of previously produced valid messages; any functioning destination node can detect these messages as unwanted.

The reason for the existence of the intermediate failing state is due to the detection latency delay experienced by the processors of the node. Assume, for instance, a two–processor fail–silent node composed of processors P_1 and P_2 – valid messages output by such a node must be double–signed. Suppose that there is a message m that should be output by the node by some real time T, and that P_1 has sent a single–signed copy of m to P_2 before P_1 has received a single–signed copy of m from P_2. If P_2 is faulty and does not send a single–signed copy of m to P_1 before some time–out has expired at P_1 (this time–out is chosen in such a way to ensure that any message sent by P_2 is output within the node worst–case delay, T_{delay}, discussed before), or if P_2 sends a message which fails the comparison at P_1, then P_1 must assume that the node has failed, and must take measures to halt the node. P_2, however, can still output a valid m, since P_2 possesses a copy of m that has been signed by P_1. If a valid m is not delayed by P_2 before being output, i.e. if m is output before T, then the node has not entered the failing state, but rather has entered the silent state, as no new valid messages are output by the node. On the other hand, if P_2 does not output m by T, then the node may enter the failing state, as P_2 can potentially output m after T, and suffer a performance failure. After P_2 has output m the node enters the silent state. The Sender and Comparator processes of each processor must therefore incorporate intra–node message synchronisation measures to ensure that each processor of a node at any time contains no more than one π–signed message for comparison. In this way, the number of performance failures in the failing state can be limited to at most one.

The fact that a fail–silent node can suffer a single performance failure in the intermediate state is not a cause for concern. Consider a system of fail–safe nodes without an intermediate state. A client application with timing constraints and expecting a response from such a node would still be expected to contain timeliness checks for detecting an absent response. The same checks will be adequate for the case of fail–silent nodes for filtering out a late response. If application programs have no timing constraints, then a performance failure suffered by a fail–silent node in the failing state will not cause any inconsistencies. Thus, a soft fail–silent node can be regarded as capable of implementing the abstraction of fail–silence in the following sense:

fail–silent node's semantics: a fail–silent 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.

It is possible to design specialised fault–tolerant network interfaces that could prevent further messages from being output by a node once one of the processors detects a failure. Minimally, we need to provide a network interface with a single switch that can unilaterally and irreversibly be switched off by a control signal sent by any of the processors of the node. (In [Reisinger–Steininger 93] a fail–silent node is presented which incorporates a network interface with a similar functionality.)

Any software solution to the design of a fail–silent node that has no intermediate failing state requires additional redundancy. For example, one could delegate the responsibility of message comparison and output to a separate node that does not fail. A $2\pi+1$–processor failure–masking node (capable of masking π processor failure within a node) could provide the services of message comparison and output to a collection of $\pi+1$–processor fail–silent nodes. Indeed the failure–masking node can provide other services, such as recording the status of fail–silent nodes. This design very much resembles that of a system of fail–stop nodes [Schneider 84] that can switch from the functioning to the halted state, and can provide failure status indication.

The performance of a soft fail–silent node depends on how quickly messages can be ordered and compared. The delay imposed by the comparison protocol is mostly made up of the time spent in message exchanges plus any delay introduced by the intra–node message synchronisation measure necessary to ensure that at any time each processor of a node contains no more than one π–signed message for comparison. Thus, we have sought ways to design more efficient order protocols in order to reduce the overhead associated with ordering, and improve the node's overall performance.

We took the following approach in our quest for a design that minimised both ordering and comparison delays. First we performed a reference design based on a design that was relatively easy to understand. For this reason, in the reference design we have used a well studied clock

synchronised based order protocol and a simple comparison protocol that did not incorporate any synchronisation measure for limiting to just one the number of π–signed messages possessed by any processor (potentially, such a node can suffer more than one performance failure in the failing state). We then investigated a number of ways of reducing message ordering delays. After this we investigated message comparison protocols with synchronisation measures. Our work on order protocols proved highly significant in coming up with a clean and efficient solution for the comparison protocol.

For the sake of simplicity, we assume a two–processor fail–silent node when presenting the reference design of a fail–silent node in the next section and the order protocols of Section 5.3. Note that since the Order process of a fail–silent node can be assumed to execute in a failure–free environment, the order protocols presented for the two–processor node can be easily extended for the more generic case of a π+1–processor fail–silent node. Then, in Section 5.4 we discuss the design of comparison protocols for π+1–processor fail–silent nodes, which incorporate synchronisation measures to limit the number of performance failures to one, when the node is in the failing state. Finally, in Section 5.5 we analyse the overheads in terms of both processing time and extra message traffic incurred by the protocols presented.

5.2. Reference Design

5.2.1. Comparison Protocol

The reference design uses a very a simple comparison protocol. Referring to Figure 3–8 (page 56), and assuming a two–processor node, the Sender process of a processor collects messages from the Processed Message Queue (PMQ), and deposits them in the Internal Candidate Message List (ICL). It also signs a copy of these messages and transmits them to the other processor of the node (its *neighbour*), where they get buffered in the neighbour's External Candidate Message List (ECL). The Comparator process maintains, for each application process $Service_i$, the sequence number of the next message to compare (recall that messages produced by application processes are assign monotonically increasing sequence numbers). Using this criterion, the Comparator matches messages with identical sequence numbers from ECL and ICL and com-

pares them. If the comparison succeeds, the message in ECL is countersigned, and the resulting valid message is deposit in the VMQ for later transmission to its destination. A comparison that detects a disagreement indicates a failure. 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 comparator process stops, and so does the Sender process (Figure 5–2 shows the pseudo–code for this simple Comparator process).

```
process Comparator
   var internal, external:message
       found, time_out:Boolean
   cycle
      do
         internal := a message from ICL
         time_out := has the time–out of internal expired?
         if not time_out then
            found := is there a counterpart of internal in ECL?
            if found then external := the counterpart of internal in ECL end if
         end if
      while not found and not time_out
      if time_out or internal ≠ external then
         kill Sender
         exit
      else
         generate and append the signature for external
         deposit external into VMQ
         discard internal
      end if
   end cycle
end process
```

Figure 5–2: Comparator process

In this simple protocol, the ECL of a processor is permitted to contain more than one valid message from the neighbour, thus potentially, a faulty processor can output more than one late valid message. In Section 5.4 we describe the additional synchronisation measures necessary to prevent this from happening.

5.2.2. Order Protocol with Synchronised Clocks

As with failure–masking nodes, the order protocol of our reference design for fail–silent nodes makes use of the well known approach of using synchronised clocks for message ordering. The clocks of both processors of the node are assumed to be synchronised such that the measurable difference between readings of clocks at any instant is bounded by a known constant, say ε. Because the non–faulty processor stops as soon as a failure is detected, the clock synchronisation protocol need not be fault–tolerant, and can be assumed to execute in a failure–free environment. The Order process of a processor time–stamps a message to be ordered with its local clock reading, and diffuses a copy of the time–stamped message over the link to the Order process of the neighbour processor. If t is the time–stamp of the message received from or diffused to the Order process of the neighbour, then the message becomes stable at local clock time $t+\Delta$, where $\Delta = d_\Delta+\varepsilon$, and $d_\Delta = \delta/(1-\rho)$. Once a message with time–stamp t becomes stable, no timely messages with time–stamp $t' < t$ can be received by an Order process. Stable messages are enqueued in the appropriate Delivered Message Queues (DMQ_i) in increasing time–stamp order, with the action being taken to discard, rather than to enqueue a stable message, if its replica has already been enqueued. (The identifier of the processors is used as a tie–breaker in the case when two different stable messages get the same time–stamp.)

As was the case in the failure–masking nodes discussed in the previous chapter, the Order process is also composed of three cyclic processes: *Broadcast, Diffuse* and *Deliver* (see Figure 4–2, page 67). The Broadcast process picks up messages from the Received Message Queue (RMQ), time–stamps them and sends them to the neighbour. (Note that since we can assume a failure–free environment, there is no need to sign internal messages for ordering.) The Broadcast process also inserts the message into the Ordered Message List (OML). The Diffuse process receives diffused messages from the link, and performs a timeliness check that rejects any message received too early (messages with time–stamp greater than $c+\varepsilon$, where c is the current reading of the processor's clock) or received too late (messages with time–stamp less than $c-\Delta$). Accepted messages are inserted into the OML. The Deliver process takes stable messages (messages with time–stamp less than $c-\Delta$) from the OML, removes duplicates and enqueues the messages on the

appropriate DMQ_is in increasing order of time–stamps. The pseudo–code for this Order process is shown in Figure 5–3.

```
process Order
   var OML:list of message
       c:clock
   process Broadcast
      var m:message
      cycle
         remove m from RMQ
         m.TS := c
         diffuse m to neighbour
         deposit m into OML
      end cycle
   end process
   ||
   process Diffuse
      var m:message
      cycle
         receive m diffused by the neighbour
         if m is authentic and c–Δ ≤ m.TS ≤ c+ε and has not received equiv(m) then
            deposit m into OML
         else discard m
         end if
      end cycle
   end process
   ||
   process Deliver
      var m:message
          stable, delivered:list of message
      cycle
         for all m, m in OML and m.TS < c–Δ do add m to stable end for all
         remove spurious messages from stable
         filter duplicate messages from delivered+stable
         delivered := delivered+stable
         order the messages in stable in increasing time–stamp order
         for all m in stable do deposit m into the appropriate DMQi end for all
      end cycle
   end process
end process
```

Figure 5–3: Order process for synchronised clock based order protocol

In the context of two–processor fail–silent nodes, we define the protocol's actual stability delay (Σ_a) for a particular message to be the real time elapsed since a copy of the message is first

received by one of the processors of the node until it is ordered and enqueued in the appropriate DMQ_i of both processors of the node. We also define Σ_{min} and Σ_{max} to be respectively the lower, and the upper bound of the actual stability delay of an order protocol ($\Sigma_{min} \leq \Sigma_a \leq \Sigma_{max}$). Therefore, for the order protocol just presented, we have:

$$\Sigma_{min} = \Delta(1-\rho);\ \Sigma_{max} = \Delta(1+\rho)+\varepsilon;\ \text{and}\ \Delta(1-\rho) \leq \Sigma_a \leq \Delta(1+\rho)+\varepsilon.$$

The lower bound Σ_{min} is achieved when the clocks of both processors are running at the fastest possible rate, i.e. the clock time interval Δ measured by their clocks corresponds to a real time interval $\Delta-\Delta\rho$. Also, either the difference between the readings of their clocks at the time a copy of the message is first received from the network is zero, or they both receive a copy of the message from the network at the same real time. On the other hand, the upper bound Σ_{max} is achieved when the reading of the clock of the processor that first receives a copy of the message from the network is ε ahead of the reading of the clock of its neighbour, and the clock of the neighbour processor is running at the slowest possible rate (i.e. ρ). Also, if the neighbour receives a copy of the message from the network, this message is received at least after a real time interval of ε has elapsed since the the first copy of the message had been received. The fixed stability delay of at least $\Delta(1-\rho)$, implicit in this order protocol, has motivated us to seek enhancements.

5.3. Efficient Order Protocols

For the order protocol of the reference implementation previously discussed, there are three parameters that will affect the protocol's actual stability delay for a particular message: i) the actual drift from real time of the clocks of each processor; ii) the actual clock synchronisation error as perceived by the first processor to receive a copy of the message from the network, i.e. the difference between its neighbour's current clock time and its clock time, at the time the message is received from the network; and iii) the message reception skew, i.e. the actual difference between the real times when each processor receives its copy of the message from the network. Let ε_a, $-\varepsilon \leq \varepsilon_a \leq \varepsilon$, be the actual clock synchronisation error as defined above, and λ_a, $\lambda_a \geq 0$, be the message reception skew for any particular message. (Note that if only one processor receives the message from the network and its neighbour does not, then $\lambda_a = \infty$, but the message is still ordered at both processors.)

Assume a two–processor fail–silent node comprised of processors P_1, and P_2, with ρ_1, c_1, and ρ_2, c_2 being respectively the drift from real time, and the reading of their clocks when the first copy of a particular message is received by one of the processors. If we assume that P_1 is the processor that receives the first copy of a message at real time T, then $\varepsilon_a = c_2 - c_1$, and the message will become stable at P_1 at real time $T+\Delta(1+\rho_1)$. If λ_a is finite, i.e. if P_2 eventually receives a copy of the message from the network, then the message diffused by P_2 to P_1 will become stable at P_1 at real time $T+\lambda_a(1+\rho_1)/(1+\rho_2)+(\Delta+\varepsilon_a)(1+\rho_1) \approx T+\lambda_a+(\Delta+\varepsilon_a)(1+\rho_1)$. On the other hand, the copy of the message that P_2 has received from the network will become stable at P_2 at real time $T+\lambda_a+\Delta(1+\rho_2)$, whilst the message diffused by P_1 to P_2 will become stable at P_2 at real time $T+(\Delta-\varepsilon_a)(1+\rho_2)$. The local stability delay for a particular processor is given by the minimum delay necessary to stabilise either the message received from the network (if any), or the message diffused by the neighbour (if any), i.e. minimum of $\{\Delta(1+\rho_1), \lambda_a+(\Delta+\varepsilon_a)(1+\rho_1)\}$ for P_1, and minimum of $\{(\Delta-\varepsilon_a)(1+\rho_2), \lambda_a+\Delta(1+\rho_2)\}$ for P_2. The protocol's actual stability delay is then given by the maximum delay between the local stability delays for P_1, and P_2.

Therefore, as observed previously, the actual stability delay for ordering any message with the order protocol presented in Section 5.2.2 is at least $\Delta(1-\rho)$. We begin our search for ways of reducing the lower bound on the actual stability delay of order protocols for two–processor fail–silent nodes by describing a method which reduces the lower bound of the clock synchronised based protocol of Section 5.2.2. We then describe new protocols with similar lower bounds, but which do not require the clocks of a node to be kept synchronised.

5.3.1. Improving the Synchronised Clock Protocol

The arrival of a diffused message can be used to reduce the constant stability delay $\Delta(1-\rho)$ imposed by the lower bound of the order protocol presented in Section 5.2.2. We shall assume that messages sent over the internal communication link of the node are received in the sent order (such communication service can be easily implemented within a two–processor node where processors are connected to each other through an internal link). Given this *fifo* assumption, a processor can use the time–stamp of a message diffused by its neighbour processor to define a new lower bound on its local stability delay. Figure 5–4 is used to illustrate the idea.

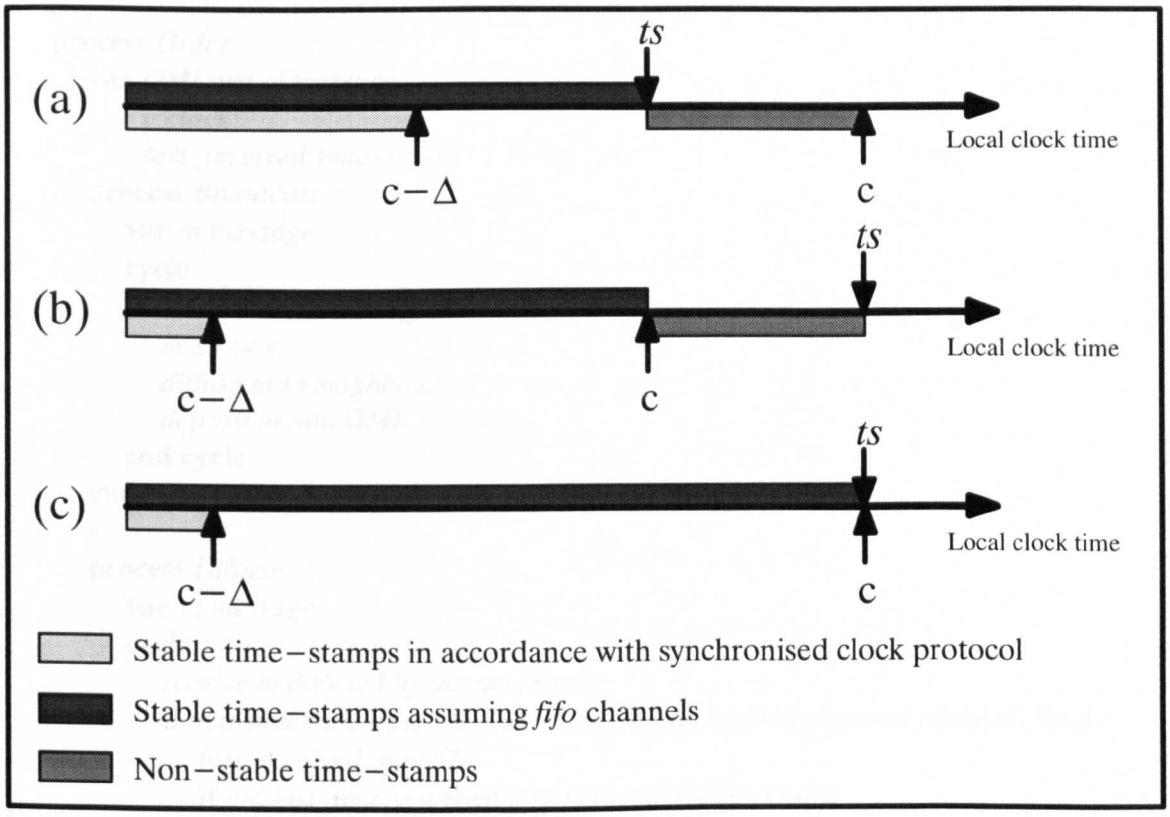

Figure 5–4: Stability intervals

In case (a) of Figure 5–4, a diffused message with time–stamp *ts* is received when the local clock reading, *c*, is greater than *ts*. As no more messages are received for ordering from the neighbour bearing a time–stamp smaller than or equal to *ts* (*fifo* assumption), and any new local message for ordering gets a time–stamp greater than or equal to *c* (clock values increase monotonically), all messages for ordering (messages deposit into OML, Figure 4–2, page 67) with time–stamps smaller than or equal to *ts* are stable.

Figure 5–4(b) shows the case where a message with time–stamp *ts* is received for ordering from the neighbour when the local clock reading *c* is less than *ts*. In this case all messages for ordering with time–stamp smaller than *c* are stable. Note that in this case it is guaranteed that the neighbour's clock is ahead of the processor's clock, thus it is possible, without causing harm to the clock synchronisation protocol properties, to advance the local clock to read *ts*+1. With this update, a diffused message with time–stamp *ts* received by a processor defines a new stabilisation interval such that all the messages with time–stamp smaller than or equal to *ts* are stable (case (c) of Figure 5–4). The pseudo–code for this modified protocol is shown in Figure 5–5.

```
process Order
  var OML:list of message
      c:clock
      last_received:time-stamp
  process Broadcast
    var m:message
    cycle
      remove m from RMQ
      m.TS := c
      diffuse m to neighbour
      deposit m into OML
    end cycle
  end process
  ||
  process Diffuse
    var m:message
    cycle
      receive m diffused by the neighbour
      if m is authentic and c-Δ ≤ m.TS ≤ c+ε and has not received equiv(m) then
        last_received := m.TS
        if c < last_received then c := last_received+1 end if
        deposit m into OML
      else discard m
      end if
    end cycle
  end process
  ||
  process Deliver
    var m:message
        stable, delivered:list of message
        stability_time:time-stamp
    cycle
      stability_time := maximum of {last_received, c-Δ}
      for all m, m in OML and m.TS ≤ stability_time do add m to stable end for all
      remove spurious messages from stable
      filter duplicate messages from delivered+stable
      delivered := delivered+stable
      order the messages in stable in increasing time-stamp order
      for all m in stable do deposit m into the appropriate DMQ_i end for all
    end cycle
  end process
end process
```

Figure 5–5: Order process for synchronised clock based order protocol with *fifo* channels

From the stability intervals of Figure 5–4(a) and Figure 5–4(c), it is easy to see that messages diffused from one processor to the other become stable at the receiver processor as soon as they are received. If we take again a two–processor fail–silent node comprised of processors P_1 and P_2, with clocks drifting from real time by ρ_1 and ρ_2, respectively, and assume that P_1 receives the first copy of a particular message from the network at real time T, then this message will become stable at P_1 at $T+\Delta(1+\rho_1)$, whilst the copy of this message received from P_2 will become stable at P_1 at $T+\lambda_a+\delta_{21}$, where δ_{21} is the actual transmission delay of the message diffused by P_2 to P_1. On the other hand, at P_2, the same message will become stable at $T+\lambda_a+\Delta(1+\rho_2)$ – for the message received from the network, and at $T+\delta_{12}$ – for the message received from P_1, where δ_{12} is the actual transmission delay of the message diffused by P_1 to P_2. Therefore, the actual stability delay for a particular message ordered by this modified protocol is given by:

$$\Sigma_a = \text{maximum of } \{\text{minimum of } \{\Delta(1+\rho_1), \lambda_a+\delta_{21}\}, \delta_{12}\}. \text{ Also,}$$

$$\Sigma_{min} = 0;\ \Sigma_{max} = \Delta(1+\rho);\ \text{and } 0 \leq \Sigma_a \leq \Delta(1+\rho).$$

5.3.2. Order Protocol with Logical Clocks

We can take the idea discussed before a step further and eliminate the requirement of having the physical clocks of the processors forming a node to be kept synchronised, and instead use logical clocks for generating time–stamps [Lamport 78].

In this order protocol each processor of a node maintains two logical clocks (counters), namely the local logical clock (*LLC*) and the remote logical clock (*RLC*), which are initialised to 1 and 0, respectively. *LLC* is used to time–stamp messages diffused to the neighbour for ordering, whilst *RLC* is used to store an estimation of the neighbours *LLC*. These clocks are updated in the following way: whenever a processor diffuses a message to its neighbour, it time–stamps the message with the current value of *LLC*, and increments *LLC* by one; whenever a message with time–stamp *ts* is received from the neighbour, *RLC* is set to *ts* and *LLC* is set to the maximum of its current value and *ts*+1. These updates ensure the following properties:

i) messages are diffused to the neighbour bearing increasing time–stamps; and

ii) the value of *RLC* of a processor is smaller than that of its *LLC* as well as that of its neighbour's *LLC*.

Property (ii) guarantees that all messages for ordering with time–stamps smaller than or equal to *RLC* are stable. So, as before, a diffused message becomes stable at the receiver processor as soon as it is received.

The protocol as presented above has one shortcoming. Messages at a processor can become stable only after the arrival of a diffused message from the neighbour (because *RLC* is updated only when a message diffused from the neighbour is received). However, a processor can only diffuse a message if it receives it from the network, so if only one of the processors receives a message from the network ($\lambda_a = \infty$), this processor will be prevented from stabilising that message, since it will not receive a copy of the message from the neighbour.

To solve this problem we introduce a scheme based on time–outs that allows a processor to update *RLC* even if its neighbour does not diffuse a message. When a processor (say P_1) diffuses a message (say m_1) with time–stamp *ts* to its neighbour (say P_2), it schedules an update of *RLC* to value *ts* to occur at time $t+2d$, where t is the value read on its physical local clock when m_1 was diffused, and d is the common clock time interval to measure a real time interval of at least δ duration, i.e. d is known to all non–faulty processors, and must be chosen such that $d \geq \delta/(1-\rho)$. At time $t+2d$, *RLC* is updated to *ts* only if its value is less than *ts*. The $2d$ time–out interval follows from the fact that after receiving m_1 with time–stamp *ts*, *LLC* of P_2 has the value of at least $ts+1$; therefore any message with time–stamp smaller than or equal to *ts* diffused from P_2 to P_1 (say m_2) will have been diffused before P_2 had received m_1. In the worst case, this would have been done just before the reception of m_1, with m_1 and m_2 each taking at most δ units of real time to be transmitted. Thus P_1 must wait for at least $2d$ units of clock time before advancing its *RLC*.

The Order process of this protocol is also composed of the three cyclic processes which work in a fashion similar to those discussed in the previous protocols (see Figure 5–6).

```
process Order
  var OML:list of message
      LLC, RLC:logical–clock
  process Broadcast
    var m:message
        t:clock–time
    cycle
      remove m from RMQ
      m.TS = LLC
      diffuse m to neighbour
      deposit m into OML
      t := the local physical clock time
      at t+2d set RLC to minimum of {RLC, LLC}
      LLC := LLC+1
    end cycle
  end process
  ||
  process Diffuse
    var m:message
    cycle
      receive m diffused by the neighbour
      if m is authentic and m.TS ≥ RLC and has not received equiv(m) then
        deposit m into OML
        RLC := m.TS
        if LLC < RLC then LLC := RLC+1 end if
      else discard m
      end if
    end cycle
  end process
  ||
  process Deliver
    var m:message
        stable, delivered:list of message
    cycle
      for all m, m in OML and m.TS < RLC do add m to stable end for all
      remove spurious messages from stable
      filter duplicate messages from delivered+stable
      delivered := delivered+stable
      order the messages in stable in increasing time–stamp order
      for all m in stable do deposit m into the appropriate DMQ_i end for all
    end cycle
  end process
end process
```

Figure 5–6: Order process for logical clock based order protocol

The Broadcast process picks up a message on its RMQ, time–stamps it with the value *ts* read on *LLC*, and places the message into its OML. Then, a copy of the time–stamped message is sent over the link to the neighbour processor. Finally, the processor's *LLC* is incremented by one, and an update of *RLC* to *ts* is scheduled to be executed in $2d$ units of time. The Diffuse process receives a diffused message with time–stamp *ts* from the link, performs a timeliness check on the message (a message is considered timely if its time–stamp is greater than the current value of *RLC*), and if the message is considered to be timely, the Diffuse process places it in the OML. *LLC* and *RLC* are then updated if necessary as discussed before. Messages in OML with time–stamps less or equal to *RLC* are stable.

For a two–processor fail–silent node comprised of processors P_1, and P_2, and assuming that P_1 is the first processor to receive a copy of a particular message from the network, we have the actual stability delay for this protocol given by:

$$\Sigma_a = \text{maximum of } \{\text{minimum of } \{2d(1+\rho_1), \lambda_a+\delta_{21}\}, \delta_{12}\};$$

where, δ_{21} is the actual transmission delay of the message diffused by P_2 to P_1, and δ_{12} is the actual transmission delay of the message diffused by P_1 to P_2.

The minimum and maximum actual stability delay for a particular message are given by:

$$\Sigma_{min} = 0;\ \Sigma_{max} = 2d(1+\rho);\ \text{and}\ 0 \le \Sigma_a \le 2d(1+\rho).$$

5.3.3. Asymmetric Order Protocol

The last two order protocols presented, have an actual stability delay that is affected by the message reception skew λ_a. When λ_a is large, the actual stability delay of those protocols deteriorates to their worst case Σ_{max} ($\Delta(1+\rho)$ for the protocol of Section 5.3.1 and $2d(1+\rho)$ for the protocol of Section 5.3.2). Thus, for systems where the message reception skew is large, the two protocols presented do not solve the problem of reducing the actual stability delay when ordering messages. We now present an order protocol whose actual stability delay is not a function of λ_a, but rather a function of δ_a, i.e. the actual transmission delay of messages between the two processors of the node. Thus, since our node model makes no assumptions on the upper bound of λ_a (see Section 3.3.1), this protocol is more suitable for implementing the Order process of the nodes we are constructing.

In this asymmetric protocol we assign different roles to each of the two processors forming a node. We term one processor the *leader* and its neighbour the *follower*. Figure 5–7 is a modified version of the node model pictured in Figure 3–8 (page 56), presenting the inter–communication of processes at both leader and follower processors.

It is the responsibility of the leader to determine the order of messages received from the network. Having selected a message for processing, the leader sends a copy of the message to the follower, which then processes messages in the order dictate by the leader. (The inspiration for this way of building a fail–silent node comes from the leader–follower replication protocol for application level processes used in the Delta–4 system [Powell 92, Barrett et al. 90].) Due to the simplicity of this ordering mechanism, there is no need for a special Order process within a processor. Instead we have enhanced the functionality of the Receiver processes executing in the leader and in the follower (see Figure 5–7).

The node works as follows. A valid double–signed message received by the Receiver of the leader is deposited in the appropriate DMQ_i and a copy of the message is also sent to the follower across the link. Double–signed messages from the leader reach the follower where they also get deposited in the appropriate DMQ_is. The *fifo* property of the communication channel guarantees the ordering of messages. (Note that a simple message sequencing mechanism can be used to implement a logical *fifo* channel over any kind of physical communication channel available.)

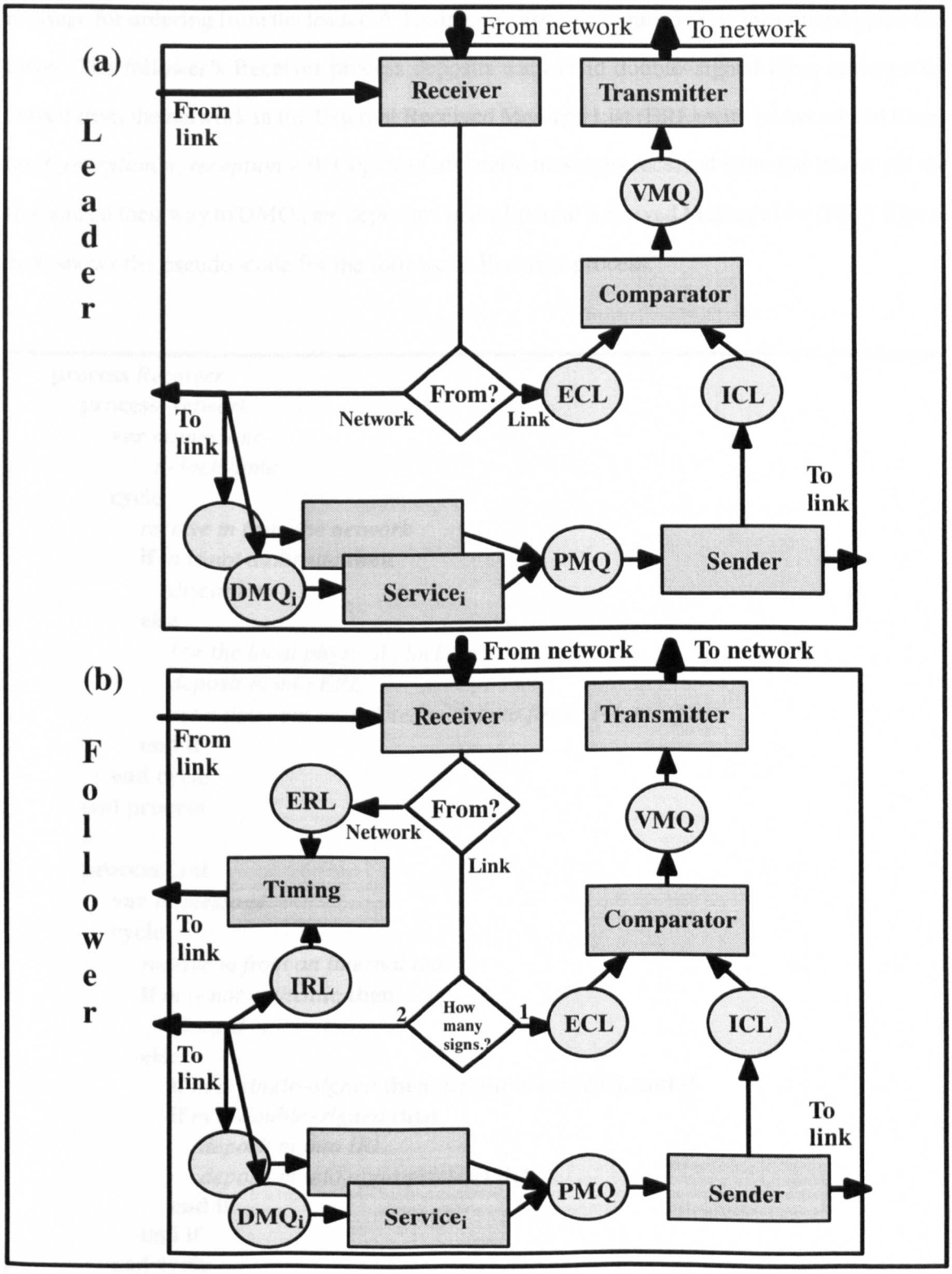

Figure 5–7: Asymmetric fail–silent node

The asymmetry introduced by assigning different roles to the two processors of a node requires us to introduce an extra mechanism in the follower for detecting late or non arrival of a

message for ordering from the leader. A Timing process (see Figure 5–7) is introduced in the follower. The follower's Receiver process deposits each valid double–signed input message received from the network in the External Received Message List (ERL) with an associated time–out *t_reception*, *t_reception* ≥ 0. Copies of authentic messages received from the leader via the link and on their way to DMQ_i, are deposited in the Internal Received Message List (IRL). Figure 5–8, shows the pseudo–code for the follower's Receiver process.

```
process Receiver
   process Network
      var m:message
          t:clock–time
      cycle
         receive m from the network
         if m is not authentic then
            discard m
         else
            t := the local physical clock time
            deposit m into ERL
            set a time–out associated with m to fire at t+t_reception
         end if
      end cycle
   end process
   ||
   process Link
      var m:message
      cycle
         receive m from an internal link
         if m is not authentic then
            discard m
         else
            if m is single–signed then deposit m into ECL end if
            if m is double–signed then
               deposit m into IRL
               deposit m into appropriate DMQi
            end if
         end if
      end cycle
   end process
end process
```

Figure 5–8: Follower's Receiver process

The Timing process picks up each messages in the ERL and tries to find its counterpart in the IRL. If a counterpart is found, then it resets the time–out associated with the message in ERL and discards both messages. If a time–out expires, the follower assumes that the leader has failed to send a message for ordering. This can happen either because the leader has failed, or because the leader has not received a copy of the message from the network. The follower processor can try to prevent a premature shut down (when the leader has not failed) by feeding the leader with the missing input message. This 'feedback' mechanism works as follows: after the time–out *t_reception* has expired, the follower sends a copy of the missing input message to the leader in order to have it properly ordered. A second time–out *t_feedback*, $t\_feedback \geq 2d$, is associated with the message. If this time–out also expires, then the follower may assume that the leader has failed, and can bring the node to a halt by stopping its Comparator and Sender processes. The pseudo–code for the Timing process, and for the implementation of the feedback mechanism by the leader's Receiver process are shown in Figure 5–9 and Figure 5–10, respectively.

```
process Timing
   var internal, external:message
       t:clock–time
   cycle
      external := a message from ERL
      t := the local physical clock time
      if time–out t_reception has expired for external then
         diffuse external to leader
         set a time–out associated with external to fire at t+t_feedback
      end if
      if time–out t_feedback has expired for external then
         kill Sender
         kill Comparator
         exit
      end if
      if there is a counterpart of external in IRL
         internal := counterpart of external in IRL
         reset any time–out associated with external
         discard internal
         discard external
      end if
   end cycle
end process
```

Figure 5–9: Timing process

```
process Receiver
   process Network
      var m:message
      cycle
         receive m from the network
         if m is not authentic then
            discard m
         else
            diffuse m to follower
            deposit m into appropriate DMQ_i
         end if
      end cycle
   end process
   ||
   process Link
      var m:message
      cycle
         receive m from an internal link
         if m is not authentic then
            discard m
         else
            if m is single-signed then deposit m into ECL end if
            if m is double-signed and has not diffused m yet then
               diffuse m to follower
               deposit m into appropriate DMQ_i
            end if
         end if
      end cycle
   end process
end process
```

Figure 5–10: Leader's Receiver process

The comparison protocol of this protocol is the same used in the reference design implementation. Thus, output messages from an application process, Service_i, follow the same path as discussed before, and message buffers ECL, ICL, VMQ and the Comparator process have the same role as before.

Unlike the previous protocols, in order to calculate the actual stability delay of the asymmetric order protocol it is relevant to identify the processor that first receives a copy of a particular input message. We define λ_{LF} as the message reception skew as perceived by the follower processor, i.e. the difference between the real time that the leader receives a copy of a particular mess-

age from the network and the real time that the follower receives a copy of the same message. The actual stability delay for this protocol is then given by:

$$\Sigma_a = \Sigma_F = \Sigma_L + \delta_{LF}; \text{ and}$$

$$\Sigma_L = \begin{cases} 0, \text{ if } \lambda_{LF} < 0 \\ \text{minimum of } \{\lambda_{LF},\ t\_reception(1 + \rho_{follower}) + \delta_{FL}\}, \text{ otherwise;} \end{cases}$$

where Σ_F is the local stability delay for the follower, Σ_L is the local stability delay for the leader, δ_{LF} is the actual transmission delay of the message diffused from the leader to the follower, $\rho_{follower}$ is the rate of the drift of the follower's clock and δ_{FL} is the actual transmission delay of the message diffused from the follower to the leader.

A sensible strategy is for the follower to set *t_reception* = 0 (thus, as soon as the follower receives a message from the network it checks for the presence of the corresponding diffused message from the leader). Hence, we have:

$$\Sigma_a = \Sigma_F = \Sigma_L + \delta_{LF}, \text{ and } \Sigma_L = \begin{cases} 0, \text{ if } \lambda_{LF} < 0 \\ \text{minimum of } \{\lambda_{LF},\ \delta_{FL}\} \text{ otherwise.} \end{cases}$$

Thus, for this protocol, the minimum and maximum actual stability delay for a particular message are given by:

$$\Sigma_{min} = 0;\ \Sigma_{max} = 2\delta; \text{ and } 0 \leq \Sigma_a \leq 2\delta.$$

5.4. Comparison Protocols

The comparison protocol discussed in Section 5.2.1 permitted a node in the failing state to commit more than one performance failure. The only fool–proof way of preventing this from happening is to use a comparison protocol that guarantees that a processor sends the next message for comparison to the other processors of the node *only after* the former has successfully compared and output a π+1–signed copy of the previous one.

A simple way to modify the comparison protocol presented in our reference design to achieve this functionality, is to introduce a synchronisation mechanism between the Comparator and the Sender processes of a non–faulty processor, such that the Sender process only sends a new

message across the links to the other processors of the node, after the Comparator process has deposit a valid message into the Validated Message Queue (VMQ), thus enabling the Transmitter process to output this valid message (see Figure 3–8, page 56).

In this modified comparison protocol, the Sender and Comparator processes of a non–faulty processor work as follows. Whenever allowed by the Comparator process, the Sender process removes a message from the Processed Message Queue (PMQ), deposits it into the Internal Candidate List (ICL), countersigns a copy of this message and diffuses it through the internal links to the other processors of the node. The Comparator process removes the message from the ICL (the way the Comparator process grants access to ICL to the Sender process guarantees that at any time there is no more than one message in the ICL of a non–faulty processor), and continuously scans the External Candidate List (ECL) searching for matching messages that have been sent by the other processors of the node for comparison. Matching messages are compared, and successfully compared messages with less than π signatures are countersigned and diffused to the other processors which have not signed these messages yet. A successfully compared π–signed message is countersigned and the resulting valid message is deposit into the VMQ for later transmission to its destination node. The Comparator process then signals the Sender process, allowing the latter to remove a new message from the PMQ. If no successfully compared π–signed message is found before a node specific time–out interval has expired, then the node is shut down. Note that there is no problem if a faulty processor does not follow the synchronisation policy, and send several messages for comparison. There is at least one processor that is non–faulty, and that only countersigns messages that match the message present in its local ICL.

However, since there is no assumptions on how application processes are scheduled within a non–faulty processor of the node, messages are deposit into the PMQ in an arbitrary order[1]. As a result, if the Sender process of two non–faulty processors select different messages from their PMQ for comparison, no π–signed messages matching the message in the ICL of any processor of the node are ever generated, and the node is shut down, even tough all processors of the node might be non–faulty.

In order to prevent this from happening, it is also necessary that the processors agree on the next message to compare. In our architecture, a logical way of achieving this agreement would be to insert an Order process between the $Service_i$ processes and the Sender process of each processor. We need to add an extra queue, called Output Message Queue (OMQ), to allow the communication between the new Order process inserted and the Sender process. Figure 5–11 is an extract of the node model presented in Figure 3–8 (page 56), which shows the relevant processes and queues after the introduction of the output Order process.

Note that this new Order protocol introduced can also be implemented under the assumption that the node is in a failure–free situation. Any failure that leads to a wrong order of messages in the OMQ will be detected by the time–out mechanism of the Comparator process. Thus, we can use any of the order protocols discussed in Section 5.3 to implement the output Order process.

---

1. Two distinct output messages produced by the same application process of a non–faulty processor are deposit in PMQ in the order they are generated, thus the Comparator process of a fail–silent node must also provide means to guarantee that if these messages are ever output by the node, they will be output in the same order that they have been generated, i.e. the Comparator process must be able to detect the failure of the Sender process of a faulty processor which has failed by sending an 'out of sequence' message for comparison. The Voter process of a failure–masking node is able to mask this sort of failure without any additional mechanism, hence this is yet another difference between that process and the Comparator process of a fail–silent node.

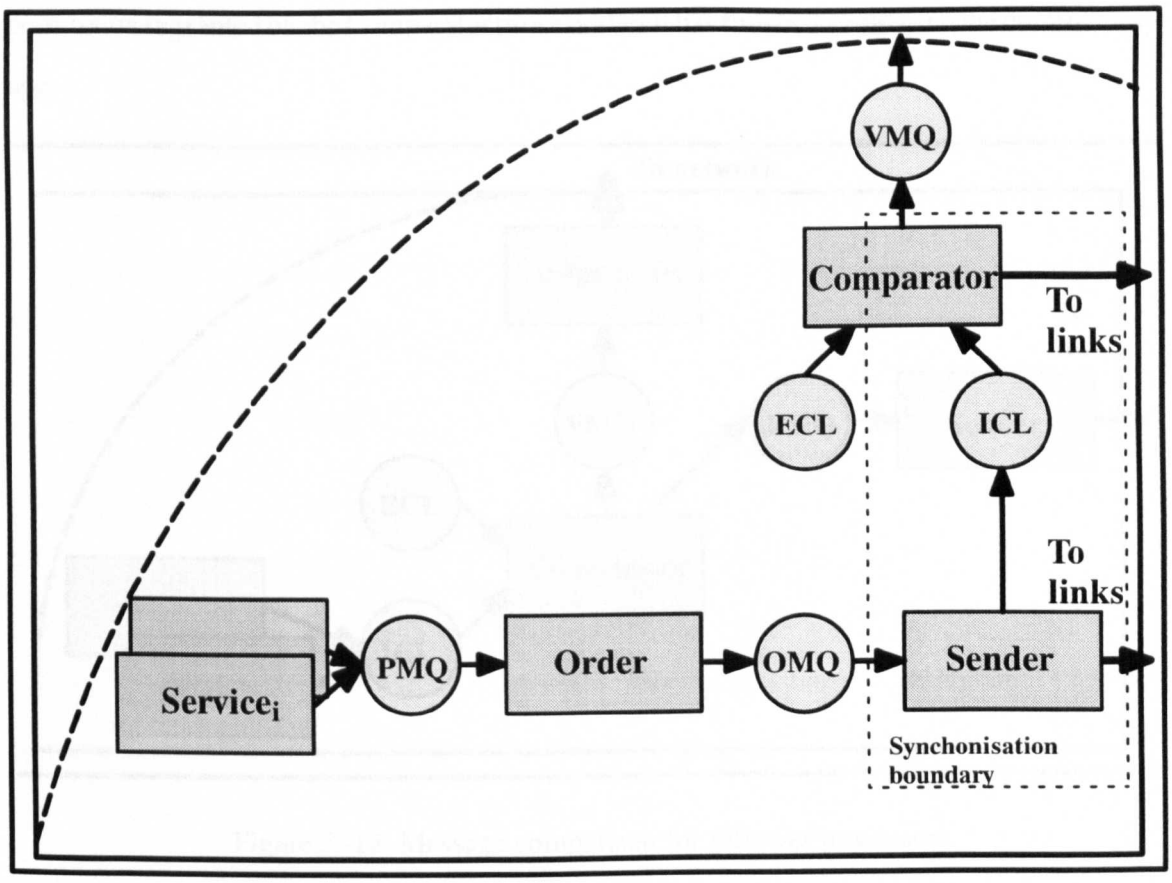

Figure 5–11: Extended Voltan fail–silent node

The asymmetric ordering approach discussed in Section 5.3.3 provides a very convenient way of integrating ordering with comparison. Accordingly, we present a comparison protocol based around the leader–follower technique. It is worth noting that this comparison protocol can be used within a node that uses any of the order protocols previously described (synchronised clock, logical clock or the asymmetric), since ordering for input messages is independent from ordering for output messages. For the sake of simplicity, we describe this protocol in the context of a two–processor fail–silent node. The description concentrates on the message synchronisation aspects of the protocol, as the other aspects remain as they were presented in Section 5.2.1.

For the purpose of message comparison then, one processor is assigned the role of the leader, and the other, is the follower. In the leader, messages in the PMQ follow the same path as before (see Figure 3–8, page 56). However, the following synchronisation mechanism between the Sender and the Comparator processes is introduced: the Sender process is allowed to send a new message over the link for comparison only if permitted by the Comparator process, and this

permission is granted by the Comparator process after it has finished comparing the current message.

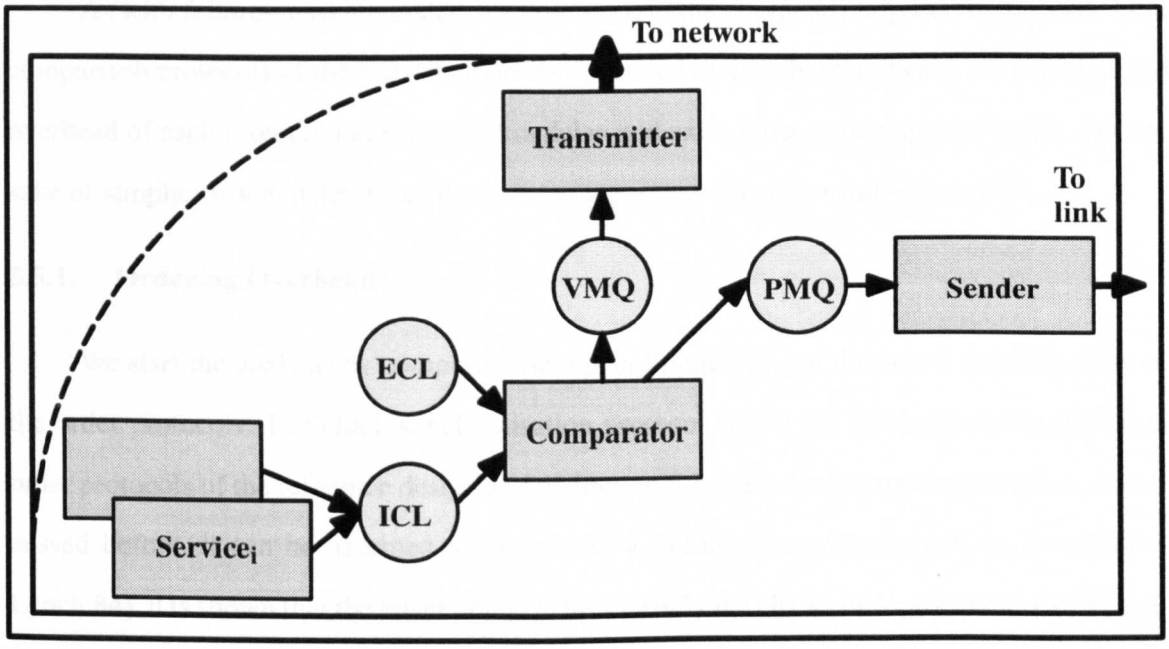

Figure 5–12: Message comparison for follower processors

On the follower side, messages produced by application processes follow a slightly different path, as shown in Figure 5–12. The Comparator process of the follower compares the message in the ECL (sent by the leader) with the correspondent locally produced one in the ICL; if the comparison succeeds, the message in ECL is countersigned, and the resulting valid message is deposited in the VMQ for network delivery. The locally produced message (the message in ICL) is also signed, and deposited into the PMQ for delivery over the link to the leader. This message arrives in the ECL of the leader, gets compared and, if the comparison succeeds, the leader's Comparator process then permits the next message from the leader to be transmitted by the leader's Sender process for comparison by the follower's Comparator process. This mechanism guarantees that the fail–silent node will suffer at most one performance failure if it enters the failing state before entering the silent state (see Figure 5–1).

5.5. Node Overhead Analysis

As with failure–masking nodes, we now analyse the overheads imposed by the order and comparison protocols of the fail–silent nodes discussed in this chapter. Again, we calculate the overhead of each protocol in terms of extra delay and extra intra–node message traffic. For the sake of simplicity, we make the analysis assuming a two–processor fail–silent node.

5.5.1. Ordering Overhead

We start the analysis by comparing the upper bound Σ_{max} of the actual stability delay of the order protocols. The clock synchronisation protocol which the clock synchronised based order protocols of the reference design and of Section 5.3.1 require is very simple, since, as discussed before, it can be assumed to execute in a failure–free environment. In [Lundelius–Lynch 84], it is shown that the lower bound at how closely the clocks of N processors can be synchronised is given by $\eta(1-1/N)$, where η is the uncertainty in message delivery. Thus, we can take for a two–processor node $\varepsilon = \delta/2$.

For the clock synchronised based order protocol of our reference design, $\Sigma_{max} = \Delta(1+\rho)+\varepsilon$, where $\Delta = (d_\Delta+\varepsilon)$ and $d_\Delta = \delta/(1-\rho) \approx \delta(1+\rho)$. After some algebra and neglecting the higher order terms of ρ ($O(\rho^2)$), we have Σ_{max} given by: $\Sigma_{max} = 2\delta+5\delta\rho/2$; similarly, Σ_{max} for the clock synchronised based order protocol of Section 5.3.1 is given by: $\Sigma_{max} = \Delta(1+\rho) = 3\delta/2+5\delta\rho/2$; for the logical clock based order protocol of Section 5.3.2, we have: $\Sigma_{max} = 2d(1+\rho)$, where $d = \delta/(1-\rho) \approx \delta(1+\rho)$, leading to $\Sigma_{max} = 2\delta+4\delta\rho$; finally, for the asymmetric order protocol of Section 5.3.3, we have $\Sigma_{max} = 2\delta$. Thus, the protocol of Section 5.3.1 presents the smallest value for Σ_{max}, which is approximately $\delta/2$ smaller than the value of the other protocols.

We note however, that since ε is proportional to N, unlike the upper bound of both the logical clock based and the asymmetric order protocols, the upper bound of the order protocols based on synchronised clocks increases with the number N of processors in the node.

It is also important to understand what are the conditions that force the actual stability delay of a particular message to approach the upper bound value of a particular order protocol. Take for instance the symmetric protocols of Section 5.3. The actual stability delay of those protocols

is given by their respective upper bound whenever the message reception skew (λ_a) is greater than $3\delta/2+5\delta\rho/2-\delta_a$, for the protocol of Section 5.3.1, and greater than $2\delta+4\delta\rho-\delta_a$, for the protocol of Section 5.3.2, where δ_a is the actual transmission delay of messages diffused from one processor to the other.

On the other hand, for the asymmetric order protocol of Section 5.3.3, the actual stability delay of messages does not depend on λ_a. For a particular message, the actual stability delay of the asymmetric protocol approaches its upper bound only when δ_{av} approaches the worst case transmission delay δ. Also, the upper bound 2δ is attained only in those cases when the first copy of the message received from the network is received by the follower processor. When the leader processor is the first processor to receive a copy of a particular message from the network, then the actual stability delay is bounded by δ. Thus, if we assume that both leader and follower processors of a two–processor fail–silent node have the same probability of receiving the first copy of a particular message from the network, then, in average the maximum actual stability delay of the asymmetric protocol is at most $3\delta/2$, which is smaller than the maximum actual stability delay of any other protocol presented.

In respect to the extra intra–node message traffic generated, all order protocols presented in this chapter need only one extra message to be transmitted through the internal link connecting the two processor in the node, for each input message received from the network.

5.5.2. Comparison Overhead

The actual validation delay (ς_a) of a comparison protocol is mainly made up by the time necessary to exchange the messages to be compared. However, the synchronisation mechanisms used to prevent the occurrence of more than one performance failure (in case the node enters the failing state – see Figure 5–1), may also incorporate a considerable overhead to the actual validation delay of a message. This makes the analysis of ς_a for the comparison protocol not as straightforward as it was the case for the ς_a of the voter protocol in the previous chapter.

In the asymmetric comparison protocol presented in Section 5.4, the follower processor always outputs messages before the leader processor; also, an output message deposit in the PMQ of the leader processor is only sent for comparison at the follower after the leader has received

and successfully compared a copy of the previous output message sent to the follower. Thus, the synchronisation delay encompasses the interval of time since an output message is deposit in the PMQ of the leader until the previous output message is deposit in the VMQ of the leader; and the synchronisation delay for the $(n+1)^{th}$ message output by the node, σ_{n+1}, is given by:

$$0 \leq \sigma_{n+1} \leq \sigma_n + 2\delta,$$

where σ_n is the synchronisation delay experienced by the n^{th} message output by the node, $n > 1$, and $\sigma_1 = 0$.

The reasoning for the maximum synchronisation delay of an output message presented above is as follows. After having sent the n^{th} output message from its PMQ to the follower, the leader will have to wait at most 2δ units of time until the correspondent message sent by the follower is received. It is only after receiving a copy of the n^{th} output message from the follower, that the leader can initiate the output of the $(n+1)^{th}$ message. Also, if at the time the $(n+1)^{th}$ output message is deposit in the PMQ of the leader the n^{th} output message is still there, then the synchronisation delay of the $(n+1)^{th}$ output message will also incorporate some of the synchronisation delay associated with the n^{th} output message; in the worst case when output messages n and $n+1$ are deposit in PMQ almost at the same time, the whole of σ_n must be incorporated in σ_{n+1}.

Thus, for any particular message, the actual validation delay for the comparison protocol of Section 5.4 is given by:

$$0 \leq \varsigma_a \leq \delta + \sigma,$$

where σ, $\sigma \geq 0$, is the synchronisation delay for the particular message.

The discussion above showed that the synchronisation delay is a function of both the actual transmission delay of the (output) messages (δ_a), and the rate with which messages are deposit in the PMQ of the leader (output rate). Further, the output rate represents the rate with which messages are generated by the application processes, which, among other things, depends on the rate with which messages are received from the network. The smaller the output rate, the smaller the synchronisation delay; in fact, if the output rate is greater than 1 message per each 2δ time interval, then it is guaranteed that the synchronisation delay is zero.

Regarding extra intra–node message traffic, the comparison protocol requires the exchange of only two messages per message output. It is worth noting that the output message sent by the follower for comparison at the leader is used also to implement the necessary synchronisation at the leader side which guarantees that a fail–silent node at the failing state will suffer no more than one performance failure.

5.6. Concluding Remarks

In this chapter we have described how efficient fail–silent nodes can be design. We have first performed a reference design that made use of a simple comparison and order protocols. We have then investigated how the performance of the order protocol can be improved. This led to a much simpler protocol based purely on logical clocks, obviating any need for keeping intra–node clocks explicitly synchronised. We have also discussed the design of an asymmetric order protocol. Further, we have described how the asymmetric ordering approach can be exploited for the construction of an efficient message comparison protocol.

Extensive experiments have been performed to evaluate the performance of two–processor fail–silent nodes under the order protocols presented in this chapter. The implementation and the performance results obtained are discussed in the next chapter.

Chapter 6
Implementation and Performance Evaluation of Soft Replicated Nodes

6.1. Introduction

In this chapter we discuss the implementation of both failure–masking and fail–silent nodes. We then analyse the performance figures obtained after running a variety of experiments in different implementations of such nodes. Our main objective has been to assess the degradation in performance suffered by a replicated node as it is called upon to execute the redundancy management software not present in an ordinary processor (unreplicated node).

Efficient implementations of the protocols described in the two previous chapters require that the processors forming a node be capable of exchanging messages quickly. *Transputers* are processing units with interfaces to fast point–to–point communication links, providing just the kind of functionality we require. For this reason, we have chosen to implement TMR failure–masking and two–processor fail–silent nodes using a network of T800 Inmos transputers [INMOS 88]. Each transputer has four internal links which connects it with four other transputers, providing a fast communication link. Figure 6–1 shows how six transputers can be coupled together to implement a network of replicated nodes. Figure 6–1(a) shows the inter–connection of TMR nodes, whilst Figure 6–1(b) shows the inter–connection of two–processor fail–silent nodes.

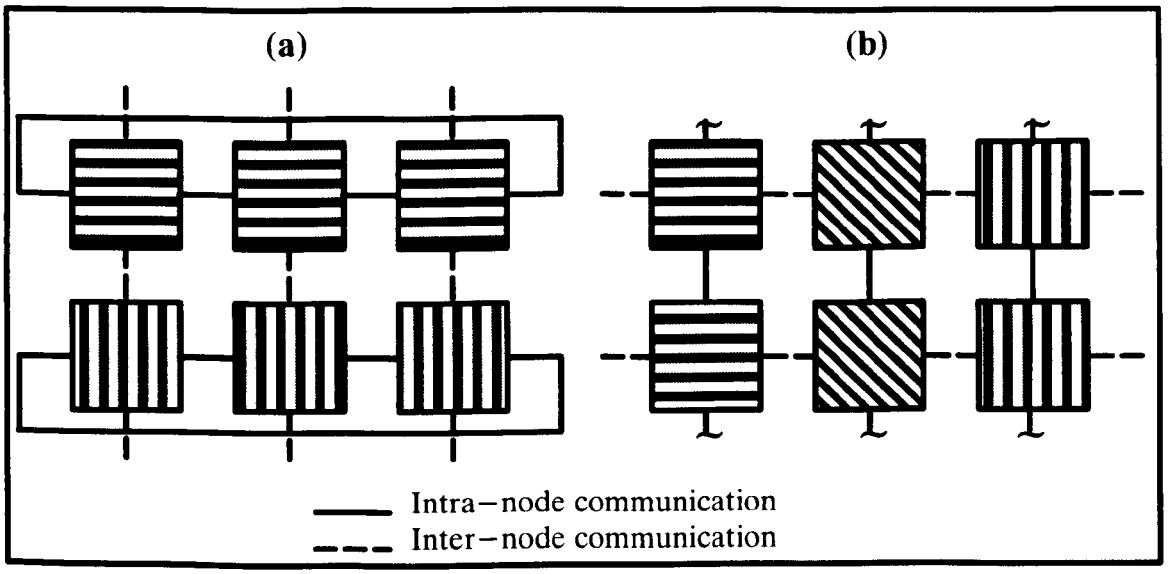

Figure 6–1: Structuring replicated nodes on a six–transputer network

6.2. Implementation Details

We have chosen to implement the protocols in an object oriented language, C++ [Stroustrup 92], and have used the facilities of the Helios operating system [Perihelion 91], a Unix–like operating system which runs on transputers and supports the client–server model for structuring programs. These choices are not central to our design and implementation, and have been taken mainly because we have extensive Unix/C++ based system programming experience. Some familiarity with C++ is assumed in this section.

Our implementation makes extensive use of classes, inheritance and virtual operations to implement the software architectures shown in Figure 3–8 (page 56), Figure 5–7 (page 126) and Figure 5–12 (page 134). Base classes exist for implementing processes, messages, queues and lists. The functionality of the system is then implemented in classes derived from these base classes.

Our implementation has been performed in a layered fashion. The lowest layer uses Helios services for providing basic system services for constructing programs composed of active objects communicating via message queues and message lists. These services are then used by the next layer, the communication layer, which provides intra–node communication facilities. Then comes the replication layer that implements the capability for replicated processing.

6.2.1. System Services

In our system, low and high priority processes are spawned by a call to the *Fork*() and *HighFork*() functions, respectively. In our implementation we encapsulate process spawning into a common *Active_Object* base class from which individual process classes may then be derived. In this way, the base class provides the thread of activity, whilst the derived class provides the algorithm by specifying the behaviour of the virtual operation *main*(). Further, process priority is defined in the way that derived classes are instantiated (see Figure 6–2).

```
class Active_Object
{
    Semaphore       started;
    public:
                    Active_Object(priority object_priority, word stack_size);
                    ~Active_Object();
      void          StartMain();
      virtual void  main();
};

void inter_main(Active_Object *object_pointer)
{
    Wait(object_pointer->started);
    object_pointer->main();
}

Active_Object::Active_Object(priority object_priority = LOW, word stack_size = 2000)
{
    InitSemaphore(&started, 0);
    if object_priority == LOW then
       Fork(stack_size, (VoidFnPtr) inter_main, sizeof(Active_Object *), this);
    else
       HighFork(stack_size, (VoidFnPtr) inter_main, sizeof(Active_Object *), this);
}

Active_Object::~Active_Object() { };

void Active_Object::StartMain()
{
    Signal(&started);
}
```

Figure 6–2: The Active_Object class

In a class hierarchy, C++ constructors are executed in a bottom–up fashion, i.e. the Active_Object constructor is executed before that of the derived class. It is conceivable therefore that main() will be called before the derived class has had time to initiate the data structures used by main(). A semaphore is used to prevent this situation from occurring. Thus the derived class controls when main() becomes active by calling the function *StartMain*() at the appropriate time. The code for the Active_Object class is shown in Figure 6–2. A parameter supplied to the derived class constructor determines the priority of the thread. The two priorities available, high and low, correspond to the priorities handled by the transputer scheduler.

```
class Producer : public Active_Object
{
   Queue   *output_queue;
   public:
           Producer(Queue *queue);
           ~Producer();
     void  main();
};

Producer::Producer(Queue *queue) :
   Active_Object(), output_queue(queue)
{
   StartMain();
}

Producer::~Producer() {};

void Producer::main()
{
   String  *message;
   while(true)
   {
      message = new String("Hello World\n");
      output_queue->enqueue(message);
   }
}
```

Figure 6–3: Active object derived class (Producer)

Two active objects can communicate with each other by accessing a common passive object via a pointer parameter passed to both constructors. The passive object can be an instance of a *Queue* class. For example, an active object *Producer* (Figure 6–3) may communicate with an ac-

tive object *Consumer* (Figure 6–4) if the pointer parameter passed to both constructors refers to the same queue, as shown in Figure 6–5.

```
class Consumer : public Active_Object
{
   Queue   *input_queue;
   public:
           Consumer(Queue *queue);
           ~Consumer();
     void  main();
};

Consumer::Consumer(Queue *queue) :
   Active_Object(), input_queue(queue)
{
   StartMain();
}

Consumer::~Consumer() {};

void Consumer::main()
{
   String  *message;
   while(true)
   {
      input_queue->dequeue((void **) &message);
      cout >> message->string() >> "\n";
      delete(message);
   }
}
```

Figure 6–4: Active object derived class (Consumer)

```
main()
{
   Queue       *queue;
   Consumer    consumer(&queue);
   Producer    producer(&queue);
   while(true);
}
```

Figure 6–5: Connecting active objects

Note that any implementation of a passive object class must consider issues of concurrency control. The Queue class allows concurrent active objects to *enqueue*() and *dequeue*() without interference. We also provide a *List* class which can be used to allow the communication of two active processes, and which provides the more elaborated *find*() and *insert*() functions to access data. The interface to the Queue and List classes are shown in Figure 6–6.

```
class Queue
{
   void          data_pointer;
   Semaphore     access;
   Semaphore     new_item;
   public:
                 Queue();
                 ~Queue();
      bool       enqueue(void *item);
      bool       dequeue(void **item);
};

class List
{
   void          data_pointer;
   Semaphore     access;
   Semaphore     new_item;
   virtual bool  match(void *information, void *item);
   public:
                 List();
                 ~List();
      void       *find(void *information);
      void       *insert(void *information, void *item);
};
```

Figure 6–6: Interface of Queue and List classes

The processes implementing the replicated nodes communicate with each other via messages. A replicated node message is implemented as a passive *Message_Block* object and is defined as a class which represents the structure of a message accepted by queues and lists, and transmitted across transputer links. A Message_Block object stores message data in the form of a sequence of bytes. Message_Block objects also contain a control component which handles all the system information relating to a message (for example, signatures, sequences numbers, timestamps, etc.). The interface for the class Message_Block is shown in Figure 6–7.

```
class Message_Block
{
   Message        data;
   Control_Block  control;
   public:
                  Message_Block();
                  Message_Block(Message_Block& message_block);
                  ~Message_Block();
      void        sign();
      bool        authenticate();
};
```

Figure 6–7: Interface of Message_Block class

Hence the structure of our implementation consists of several processes (active objects) communicating asynchronously with each other via queues and lists (passive objects) using messages (passive objects) to transfer information.

The Queue and List class objects presented earlier, manage a collection of void pointers which may refer to arbitrary data structure elements, so all users of a queue must implicitly know the type of the elements. In order to implement additional type checking it has been found desirable to derive classes *Message_Block_Queue* (*MBQ*) and *Message_Block_List* (*MBL*) from the Queue and List classes, respectively. The construction of MBL, for instance, is simplified by containing list complexity and concurrency control entirely within the List base class. The derived class is also the mechanism through which particular list operations may be performed. The List class provides two functions *find*() and *insert*(), and a virtual function *match*(). The derived class calls find() and insert() and provides a real function match(). All the message lists and queues required by our replicated nodes are declared as instances of the MBL and the MBQ classes, which are derived from the base classes List and Queue, respectively.

6.2.2. Communication Layer

Helios provides two different communication mechanisms: primitives for client–server processes and direct point–to–point communication over 'raw' links. Intra–node communication in our replicated nodes uses this raw link–level service, whereas inter–node communication at present uses the client–server facilities provided by Helios. Our basic approach is to distinguish

between intra–node and inter–node communication. Intra–node communication is used for agreement, order and validation of messages, so for efficiency reasons, it takes place over the fast directly connected links between the processors of a node.

Using the link level primitives, together with the system services described in the previous subsection, we have built a point–to–point communication service for communicating messages between any two processors of a node. Each processor contains active objects *Tx* and *Rx* for respectively transmitting and receiving intra–node messages. Tx dequeues messages from a queue specified in its constructor and sends them over a given link also specified in the constructor. The Rx object awaits messages on a link specified in its constructor. It receives messages and decides on which queue the messages should be placed, depending on the number of signatures present in the message. The possible destination queues are specified by passing the Rx constructor a pointer to an array of possible destination queues.

Each processor runs as many Tx and Rx objects as necessary to allow communication with all other processors in the node. Thus, in the case of the TMR nodes, there are two instances of Tx and Rx objects allowing communication with the other two processors in the node, whilst in the case of a two–processor fail–silent node there is only one instance of each object. Hence, to send a message to any other processor in the node, a process simply specifies the destination queue in the message control field and enqueues it on the queue associated with the appropriate Tx object. (On our two–processor fail–silent nodes this queue is named the *Neighbour Message Queue – NMQ*, whilst on TMR nodes, the queues are named *Lower Neighbour Message Queue – LMQ*, and *Higher Neighbour Message Queue – HMQ*.) The intra–node communication service will then ensure delivery of the message to the correct queue in the destination processor. Both Tx and Rx run at high priority so as to minimize communication delays within a node.

6.2.3. Replication Layer

The replication layer implements the functionality of the objects pictured in Figure 3–8 (page 56), Figure 5–7 (page 126) and Figure 5–12 (page 134). Inter–node communication is provided by a *Receiver* and a *Transmitter* active objects, using the client–server facilities provided by Helios. The remaining active objects in the replication layer are responsible for the ordering

of input messages and the validation of output messages. Our object–oriented approach has shown to be very flexible and allowed rapid implementation of replicated nodes with different protocols. As all nodes have the same basic structure irrespective of the protocols they incorporate, it is possible to change the internal functioning of a particular node by simply making the necessary changes on the parameters and the code of the main() function of the active objects which should be changed to attain the required functionality of the new replicated node.

```
class Sender: public Active_Object
{
   MBQ     *PMQ, *NMQ;
   MBL     *ICL;
   int     processor_id;
   public:
           Sender(MBQ *pmq, MBL *icl, MBQ *nmq, int p_id);
           ~Sender();
      void main();
};

Sender::Sender(MBQ *pmq, MBQ *vmq, MBQ *nmq, int p_id) :
   Active_Object(), PMQ(pmq), ICL(icl), NMQ(nmq), processor_id(p_id)
{
   StartMain();
}

Sender::~Sender() {};

void Sender::main()
{
   Message_Block *internal_message, *external_message;
   while(true)
   {
      PMQ->dequeue(&internal_message);
      external_message = new Message_Block(*internal_message);
      external_message->sign(processor_id);
      ICL->insert(internal_message);
      NMQ->enqueue(external_message);
   }
}
```

Figure 6–8: Sender process for a two–processor fail–silent node

Given the infra–structure services described above, system processes are relatively simple to implement. To illustrate this, we show the implementation of the Sender process for both fail–

silent (Figure 6–8) and TMR nodes (Figure 6–9).

```
class Sender: public Active_Object
{
   MBQ      *PMQ, *LMQ, *HMQ;
   MBL      *ICL;
   int      processor_id;
   public:
            Sender(MBQ *pmq, MBL *icl, MBQ *lmq, MBQ *hmq, int p_id);
            ~Sender();
      void  main();
};

Sender::Sender(MBQ *pmq, MBL *icl, MBQ *lmq, MBQ *hmq, int p_id) :
   Active_Object(), PMQ(pmq), ICL(icl), LMQ(lmq), HMQ(hmq), processor_id(p_id)
{
   StartMain();
}

Sender::~Sender() {};

void Sender::main()
{
   Message_Block *internal_message, *lower_message, *higher_message;
   while(true)
   {
      PMQ->dequeue(&internal_message);
      lower_message = new Message_Block(*internal_message);
      lower_message->sign(processor_id);
      higher_message = new Message_Block(*internal_message);
      higher_message->sign(processor_id);
      ICL->insert(internal_message);
      LMQ->enqueue(lower_message);
      HMQ->enqueue(higher_message);
   }
}
```

Figure 6–9: Sender process for a TMR node

The Sender process works as follows: messages are dequeued from the PMQ, then copies of the messages are made so that it can be diffused to the neighbour(s). These messages are signed with the processor's unique identifier, and enqueued in the appropriate queues to be transmitted by a Tx process. (Single–signed messages transmitted by a Tx process are received at the other end by an Rx process which will insert them in the local ECL.) A copy of the messages is also

inserted in the ICL, where the Validator process will access them in order to validate them against the messages inserted in the ECL by the Rx process. Note that, as we have pointed out before, it is very simple to modify the Sender process of a fail–silent node in order to incorporate it to a TMR node. Likewise, NMR nodes with $N > 3$ should also be simple to implement by modifying the active objects used to implement TMR nodes.

Remark: for the nodes whose order protocols require clocks to be synchronised, the clock synchronization algorithm implemented is that proposed by Halpern et al.[Halpern et al. 84]. Its implementation consists of two active objects, which respectively monitor the time (*Time_Monitor* process) and await incoming clock synchronization messages (*Message_Manager* process), together with a passive object (*Clock*) which maintains the time, and implements the abstraction of a synchronised clock. Consistency, with respect to concurrent accesses to the clock, is maintained by a semaphore. Clock synchronization objects communicate via the internal links and run at high priority so that the synchronization will be as tight as possible and will be relatively independent of the load on each processor.

6.3. Performance Evaluation

We have implemented several versions of failure–masking and fail–silent nodes, each of them incorporating a different order protocol. A set of experiments have been executed on the various nodes implemented, and the performance of each node has been measured. In this section we describe the versions of failure–masking and fail–silent nodes implemented, and the experiments that have been realised. We then analyse the performance of these nodes and confront the figures obtained for the time overhead incurred by the nodes against the expected values discussed in the previous chapters.

6.3.1. Nodes Description

We have implemented three different versions of three–processor failure–masking nodes (TMR nodes). All TMR nodes implemented incorporate the voter protocol of the reference design presented in Section 4.2, and they are differentiated from each other only in the way input

messages are ordered. We have used the following order protocols to implement each of the three versions of TMR nodes:

i) *synchronised clocks*: this implementation corresponds to the reference design discussed in Section 4.2, and incorporates an order protocol based on having the physical clocks of the processors forming the node explicitly synchronised;

ii) *logical clocks*: this version incorporates the order protocol presented in Section 4.3.2, which is an improvement for TMR nodes on the order protocol presented in Section 4.3.1; and

iii) *early–order*: in this node, the early–order protocol of Section 4.3.3 is used to order input messages. (Communication between processors forming the node is assumed to have an *fifo* property, which is very easy to achieve within the transputer based architecture used.)

We have also implemented three versions of two–processor fail–silent nodes. All nodes incorporate the asymmetric comparison protocol discussed in Section 5.4, and again are differentiated from each other only in the way input messages are ordered. We have evaluated their performance for the following three order protocols:

i) *synchronised clocks*: the order protocol of the reference implementation presented in Section 5.2.2, which needs the implementation of a clock synchronisation algorithm;

ii) *logical clocks*: the protocol based on logical clocks, discussed in Section 5.3.2; and

iii) *asymmetric*: the leader–follower asymmetric protocol studied in Section 4.3.3. (For this node, the processor acting as the leader for ordering is also the leader for comparison.)

We have also implemented an unreplicated node model which executes on single transputers. Apart from the application processes, the unreplicated node incorporates Receiver and Transmitter system processes. (The Receiver process for the unreplicated node is slightly simpler

than the one for replicated nodes, since input messages do not need to be authenticated.) Figure 6–10 shows the unreplicated node structure in terms of the processes and queues of the replicated model presented in Section 3.3.2 (see Figure 3–8, page 56).

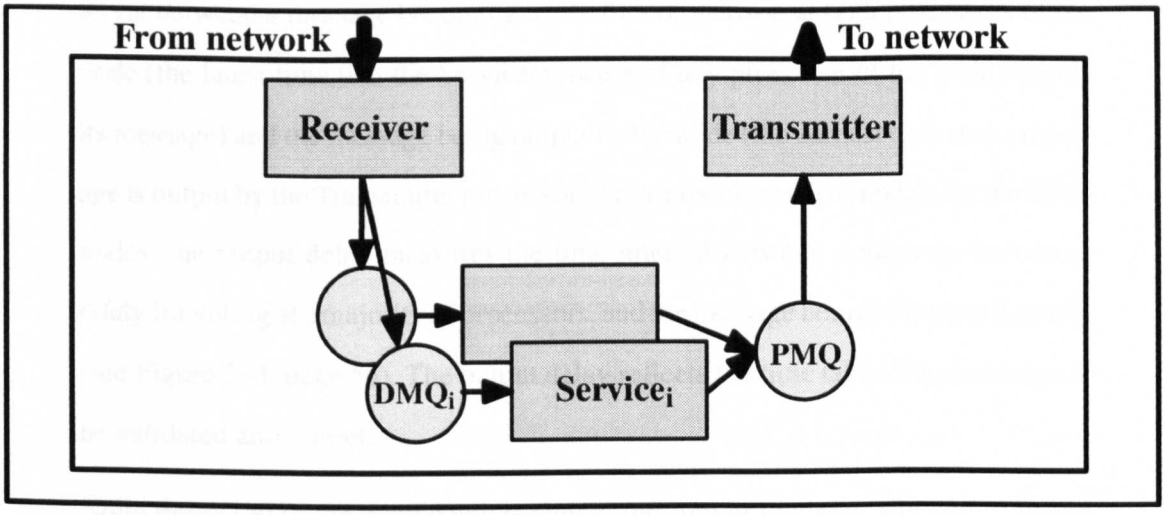

Figure 6–10: Unreplicated node model

6.3.2. Experiments Description and Evaluation

We have executed a number of experiments using both TMR and fail–silent nodes, with the objective of assessing the overhead incurred by the redundancy management protocols of these nodes. The general structure of the experiments is that of a client application process executing on an unreplicated node, and which makes requests to a server application process executing on a replicated node. We are particularly interested in the overheads associated with ordering of input messages and validation of output messages. Thus, we have measured the following time intervals for the replicated nodes:

Input delay (ID): for the fail–silent nodes, the input delay measures the time interval between a message entering the node (the earliest time that a particular message is received by the Receiver process of any processor of the node) and the message being removed from the DMQ_i of both processors of the node; for the TMR nodes, the input delay measures the time interval between a message entering the node and the message being removed from the DMQ_i of a majority of the processors of the node (see Figure 3–8, page 56). Thus, the input delay is made up of the actual stability delay

for a message (Σ_a) plus the time taken up by authentication and queue manipulation within the node; it reflects the overhead involved in ordering messages at a node.

Output delay (OD): for the fail–silent nodes, the output delay measures the time interval between a message becoming ready for comparison at both processors of the node (the latest time that the $Service_i$ process of any processor of the node outputs its message) and the message being output by the node (the earliest time that a message is output by the Transmitter processor of any processor of the node); for the TMR nodes, the output delay measures the time interval between a message becoming ready for voting at a majority of processors, and the message being output by the node (see Figure 3–8, page 56). The output delay reflects the time taken for a message to be validated and output.

Node delay (ND): the node delay is simply the sum of the input and output delays (ID+OD). It reflects the earliest response from a node to a given input message, i.e. the overhead associated with replication.

For the unreplicated node we have measured the following time interval:

Response latency (RL): the response latency is the time that the client will have to wait for the response for a particular request to arrive. It is made up by the small processing overhead associated with the output of a request and the reception of the correspondent response at the unreplicated node, the inter–node transmission delay for both the request and the response messages, and the processing overhead at the replicated node where the server executes.

Overhead for a simple client–server application

In the first experiment a single client application process executing on an unreplicated node requests a simple service from the server application process which executes on a replicated node. The client process issues a request to the server process by broadcasting the request to the server process replicas executing on each of the processors forming the replicated node. It then waits for a response from any of the server replicas. Each server process replica executing on the replicated node receives a request from the client, services it (the actual computation performed is

minimal) and sends the response back to the client (this simulates the communication involved in a simple RPC operation [Birrell–Nelson 84]). The client issues a new request upon reception of the first response message received from the server.

We have collected data for ten runs of the experiment, each run involving the client node sending 100 request messages of 64 bytes, and receiving response messages of the same size. For each run we have measured the input, output and node delays for the replicated node, as well as the response latency for the unreplicated node. For each one of these time intervals we have averaged the values measured for each of the requests processed. We have also measured the average message reception skew (λ_{av}) and the average link transmission delay of internal messages, making a distinction between the transmission delay of internal messages associated with the ordering of input messages (δ_{input}) and the the transmission delay of internal messages associated with the validation of output messages (δ_{output}).

We have also executed the experiment for a server process executing on a single processor, i.e. on an unreplicated node. As we would anticipate, for the case of ordinary processors, the overheads are small. They exist because it is still necessary to enqueue and dequeue messages in the system. The measured node delay for the unreplicated node amounted to about 1.49 ms, of which about 0.75 ms was due to input overheads, whilst about 0.74 ms was due to output overheads. The average response latency measured was 5.74 ms.

We first concentrate the analysis on the overheads for the replicated nodes, starting with the overheads for the fail–silent node implementations. Table 6–1 summarises the average delays obtained for each fail–silent node implementation exercised.

Model	Delay (milliseconds)					
	ID	OD	ND	δ_{input}	δ_{output}	λ_{av}
fail–silent synchronised clocks	21.59	6.31	27.90	2.70	1.46	0.41
fail–silent logical clocks	8.45	6.67	15.12	2.77	1.40	0.40
fail–silent asymmetric	4.79	3.32	8.11	3.92	1.30	0.48

Table 6–1: Performance overhead for a client–server application on fail–silent nodes

For the case of the fail–silent node with synchronised clock based order protocol, experiments under worst case circumstances determined the smallest safe value for δ to be 12 ms. This

reference implementation of a node uses a simplified version of the clock synchronisation algorithm presented in [Halpern et al. 84]. As discussed in the previous chapter, we can assume a failure–free environment for the execution of the clock synchronisation protocol, thus allowing ε to be set to $\delta/2$. Hence we have fixed $\varepsilon = 6$ ms which gives the stability delay, Δ, of 18 ms (since $\Delta = \delta+\varepsilon$). Measurements indicated that throughout the relatively small duration of the experiment, the actual synchronisation error (ε_a) was very small. Thus, on average, the stability delay (Σ_a) is almost the same as Δ, and the values shown in Table 6–1 for ID indicate that for this implementation the overheads due to message authentication and queue manipulation take up to 3.59 ms.

From the discussion in Section 5.3.2, on average, the stability delay of the fail–silent node using the logical clock based order protocol would be equal to the minimum value between 2δ, and $\delta_{input}+\lambda_{av}$, plus any extra overheads. In our experiment $\delta_{input}+\lambda_{av}$ is smaller than 2δ, thus, from the figures given in Table 6–1 the overheads due to message authentication and queue manipulation for this implementation take up to 5.28 ms. The increased overhead of this implementation suggests that in terms of execution time, the maintenance of logical clocks is more costly than the maintenance of synchronised physical clocks.

For the fail–silent node using the asymmetric order protocol, it is necessary to examine separately the performance of leader and follower processors since they are executing different protocols. From the analysis presented in the Section 5.3.3, ID corresponds to the follower's stability delay ($\Sigma_a = \Sigma_F = \Sigma_L+\delta_{input}$), plus any overheads due to message authentication and queue manipulation. In our experiment, the leader processor nearly always is the first processor of the node to receive a copy of a particular input message. Thus, most of the time we will have $\lambda_{LF} < 0$, and consequently $\Sigma_a = \delta_{input}$. The values shown in Table 6–1 indicate that the input overheads (0.87 ms) for the asymmetric implementation of a fail–silent node are close to those experienced by the unreplicated node. This is because the functions of the order protocol are incorporated into the Receiver process, consequently reducing the overheads associated with context switching and queue manipulation. The overheads are slightly larger because, in the replicated node, messages must be authenticated. (Currently, simple checksums are being used as signatures to provide the authentication facility that our node model assumes, and so, have a relatively small impact upon

system performance. The performance overhead of more complex signature mechanisms has not yet been assessed.)

Indeed, for small messages, most of the extra input overheads incurred by the different implementations of fail–silent nodes are due to context switching of processes and queue manipulation. This also contributes to the smaller overheads of the synchronised clock based implementation when compared to those for the logical clock based implementation. In the former, the tighter synchronisation guaranteed by the time triggered characteristics of the synchronised clock based order protocol allows more efficient scheduling of the actions taken at each processor forming the node. We believe that a better control of the scheduling policy of the processors forming a fail–silent node can lead to a certain degree of improvement on the implementations of such nodes, particularly for the case of the logical clock based implementation.

The positive effects of a more efficient scheduling strategy can also be noticed at the validation of output messages. Despite the fact that all fail–silent nodes implemented make use of the same comparison protocol, figures in Table 6–1 show that a node implemented with the asymmetric order protocol for input messages suffers less output delay than a node with a symmetric one. The reason for this is that the asymmetry introduced for input ordering and for comparison helps the follower at comparison time: by the time a message becomes available in the follower's ICL (see Figure 5–7, page 126), the leader's message will usually be already available in the followers ECL. From the figures presented in Table 6–1 we can deduce that the overheads for validation (OD–δ_{output}) of the two first implementations take up respectively 4.85 ms, and 5.27 ms, whilst the overheads for the asymmetric implementation take up only 2.02 ms.

We now analyse the overheads for the TMR nodes. Table 6–2 gives the average delays obtained when we executed the experiments on the different TMR node implementations.

Again, we have exposed the implementations to a worst case situation, and have measured the smallest safe value for δ, which turned out to be $\delta = 50$ ms. This much larger value for δ when compared with the value we have chosen for the fail–silent implementations reflects the fact that there is a considerable increase in the number of internal messages exchanged by the processors

forming a TMR node in comparison with the number of internal messages exchanged by the processors forming a fail–silent node.

Model	Delay (milliseconds)					
	ID	OD	ND	δ_{input}	δ_{output}	λ_{av}
TMR synchronised clocks	202.78	10.76	213.54	5.60	4.25	1.25
TMR logical clocks	163.50	11.09	174.59	8.77	4.41	0.96
TMR early–order	35.55	10.71	46.26	9.60	4.32	1.40

Table 6–2: Performance overhead for a client–server application on TMR nodes

For the synchronised clock based implementation we have used the fault–tolerant clock synchronisation algorithm presented in [Halpern et al. 84], for which we can choose the maximum error on the reading of the clocks of any two non–faulty processors to be $\varepsilon = \delta/(1+\rho) \approx 50$ ms, which leads to $\Delta \approx 200$ ms. Following the same reasoning that we have used when discussing the fail–silent implementations, we can deduce from the value of ID presented in Table 6–2 that the input overheads of the synchronised clock based TMR node take up approximately 2.78 ms.

As studied in Section 4.3.2, on average, the stability delay of a TMR node which uses a logical clock based order protocol is given by the minimum between 4δ and $\lambda_{av}+\delta_{input}+3\delta$. Since for our experiment $\lambda_{av}+\delta_{input}$ is much smaller than δ, from the figures shown in Table 6–2 we can assess the extra input overheads of this node to be 3.77 ms.

For the case of the early–order implementation, the average value expected for its stability delay is given by $3\delta_{input}$. Thus, from the value of ID presented in Table 6–2 we can conclude that the input overheads for this implementation take up to 6.75 ms. Among the reasons for the much larger overhead of this implementation are: i) the overheads associated with the maintenance of several logical clocks; ii) the overheads associated with the management of null acknowledgement broadcasts; and iii) the event triggered nature of the early–order protocol which is responsible for extra overheads in the scheduling of processes.

The overheads associated with the validation phase are pretty much the same for all TMR implementations (6.51 ms, 6.68 ms, and 6.39 ms for the synchronised clock based, logical clock based and early–order implementations, respectively). Furthermore, the validation overheads are

only slightly larger than those presented by the symmetric implementations of fail–silent nodes (see Table 6–1), reflecting the increased complexity of the voting function of TMR nodes, when compared with the complexity of the comparator function of fail–silent nodes.

Now we analyse how the overheads at the node where the server process is executed are perceived by the client process executing on the unreplicated node. We have measured the average response latency (RL) experienced by the client process executing on the unreplicated node when the client issues requests to server processes executing on the different replicated nodes implemented. We have also measured the response latency for the case where the server process executes on an unreplicated node ($RL_{unreplicated}$). We define the Relative Replication Performance Overhead (RRPO) of a node as: $RRPO = 1 - RL_{unreplicated}/RL$. Table 6–3 below gives the values obtained for RL and RRPO.

Model	RL (milliseconds)	RRPO (%)
unreplicated node	5.74	0.00
fail–silent synchronised clocks	35.67	83.91
fail–silent logical clocks	22.38	74.35
fail–silent asymmetric	14.46	60.30
TMR synchronised clocks	221.73	97.41
TMR logical clocks	182.49	96.85
TMR early–order	54.65	89.50

Table 6–3: Response latency and RRPO for a client–server application

In our experiment there is no processing at the server apart from that associated with queue manipulation. Thus, we can consider the figures presented in Table 6–3 as worst case performances for a lightly loaded node. It should be appreciated that the price in performance becomes significant in only these distributed applications where processes interact frequently. If on the other hand, application processes are involved in computations requiring less frequent interactions then the performance impact of replication protocols can be quite small. The next experiment illustrates this point.

Relative replication performance overhead for increasing server processing time

In this experiment we have extended the experiment previously described one step further. We have modified the server process in such a way that we can control its processing time. We

then executed the experiment for different server processing times. Figure 6–11 shows the figures obtained for RRPO when the server processing time was increased from (almost) zero up to 100 ms.

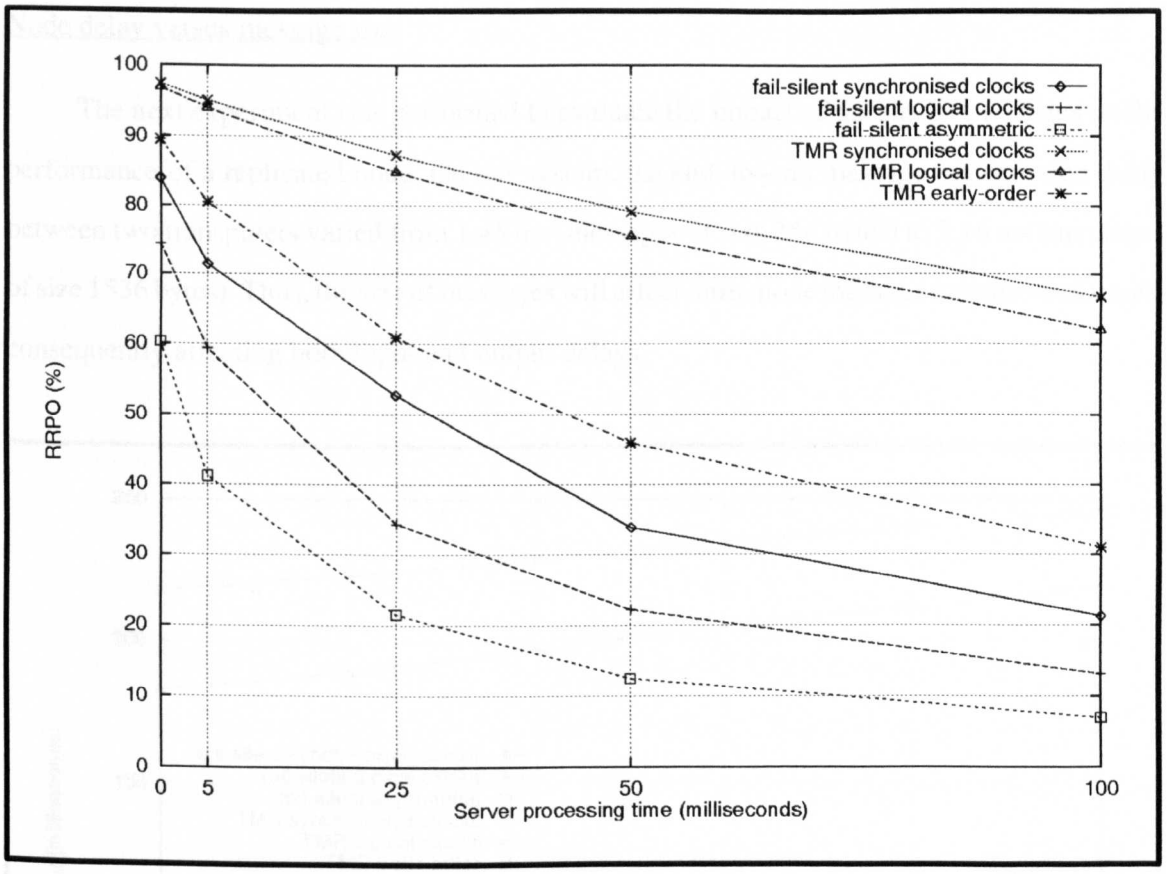

Figure 6–11: RRPO versus server processing time

As expected, when the server processing time increases there is a reduction on the relative replication performance overhead of replicated nodes. The values shown in Figure 6–11 indicate that for our particular environment, if application processes do not communicate during intervals of time of at least 100 ms duration, then the burden associated with replication in the fail–silent nodes can be decreased from more than 80% to less than 25%. Also, for server processing times as little as 25 ms, the asymmetric fail–silent node can reach about 75% of the performance of an unreplicated node.

For the synchronised clock based and logical clock based TMR nodes, and server processing times of up to 100 ms, the decrease of the relative replication performance overhead is not as distinctive as was the case for the fail–silent nodes. However, the relative replication perform-

ance overhead of the early–order TMR node is reduced from more than 80% to nearly 30%. Given their more robust semantics, TMR nodes should naturally be expected to be more expensive to build.

Node delay versus message size

The next experiment was performed to evaluate the impact of the size of messages on the performance of a replicated node. On our system, the end–to–end message transmission delay between two transputers varied from 1.45 ms (messages of size 256 bytes) to 2.16 ms (messages of size 1536 bytes). Thus, the size of messages will affect intra–node message transmission times, consequently affecting both input and output delays.

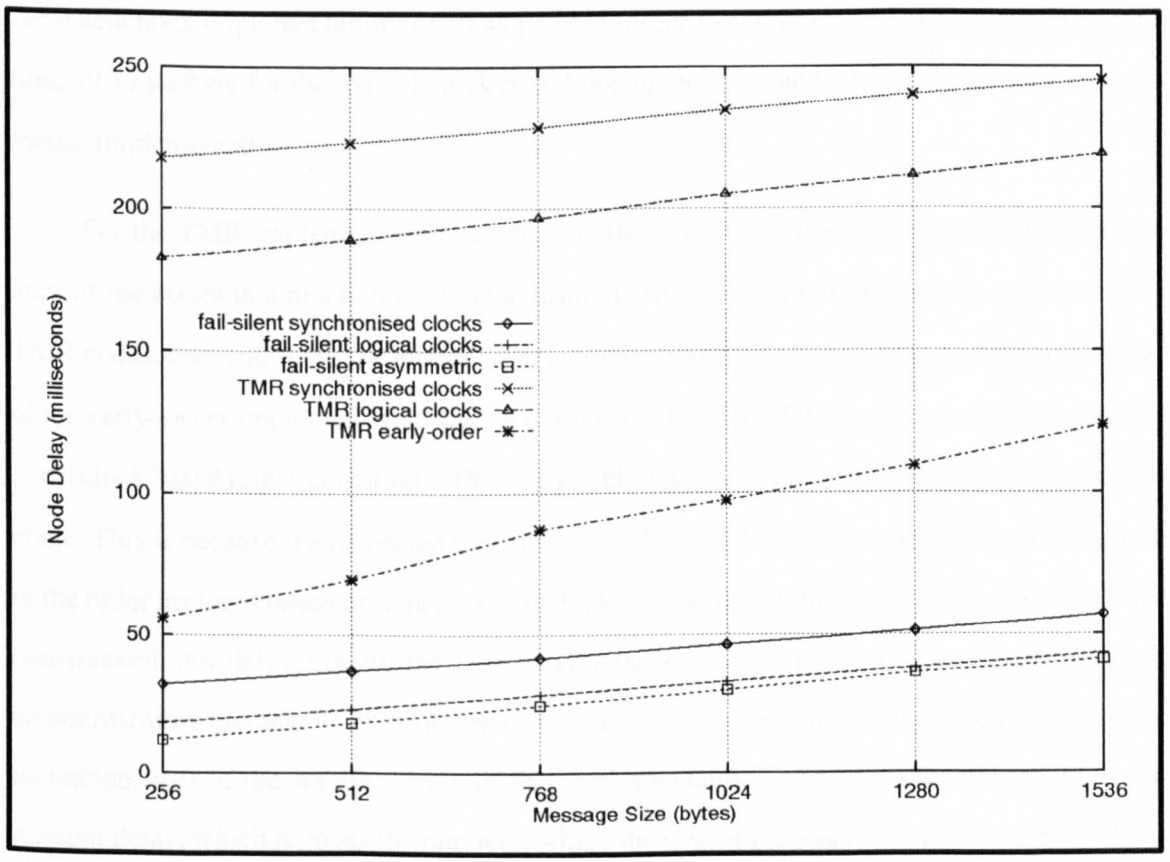

Figure 6–12: Node delay versus message size

Using the original client–server application previously described (i.e. with minimal processing at the server), we have measured the node delay for the various fail–silent and TMR nodes implemented, as the message size was increased from 256 to 1536 bytes. Figure 6–12 presents

the average node delays obtained when the experiment was executed on the different replicated nodes implemented.

The impact of the increase of message size on the performance of the various replicated nodes implemented is not uniform. For instance, the performance of the fail–silent nodes suffers only a relatively small decrease when compared with the reduction of performance experienced by the TMR nodes. The fail–silent node implementations require only a small number of intra–node messages to be exchanged between the processors forming the node. For these nodes, most of the increase on the node delay will be due to the extra time needed to authenticate, sign, and copy messages within the node, rather than the increased transmission delay. This is confirmed by the values shown in Figure 6–12 which indicate an average increase of the node delay for the fail–silent node implementations of 19.41 μs/byte for the synchronised clock based implementation, 19.77 μs/byte for the logical clock based implementation and 23.09 μs/byte for the asymmetric implementation.

For the TMR implementations, increasing the size of the messages impacts the performance of the nodes in a much more diverse fashion. The much larger internal message traffic of TMR nodes, particularly in the input phase, is responsible for a sharp increase of the node delay of the early–order implementation (52.69 μs/byte on average). On the other hand, the synchronised clock based implementation suffers only a relatively small increase of 21.21 μs/byte on average. This is because the increased transmission delay will have no impact on the performance of the order protocol based on synchronised clocks, since its stability delay is based on the worst case transmission delay. Finally, the node delay of the logical clock based implementation suffers an intermediary increase of 28.40 μs/byte on average. Like the synchronised clock based implementation, part of the stability delay of this implementation is based on the worst case transmission delay, which reduces the rate with which the size of messages increases the node delay of this implementation.

Response latency versus number of clients

In the experiments so far the server process executing on the replicated nodes attends a single request at a time. Thus, the figures obtained correspond to the case where the server process

is subjected to a light load. We now analyse the behaviour of our system when the processing load is increased. We increase the load by increasing the number of clients that simultaneously issue requests to the server. The experiment is structured in such a way that a client process only issues another request to the server when all client processes have received the response of their previous request. Also, there is no processing at the server. Figure 6–13 shows the average response latency attained when the number of clients was increased from 2 up to 20.

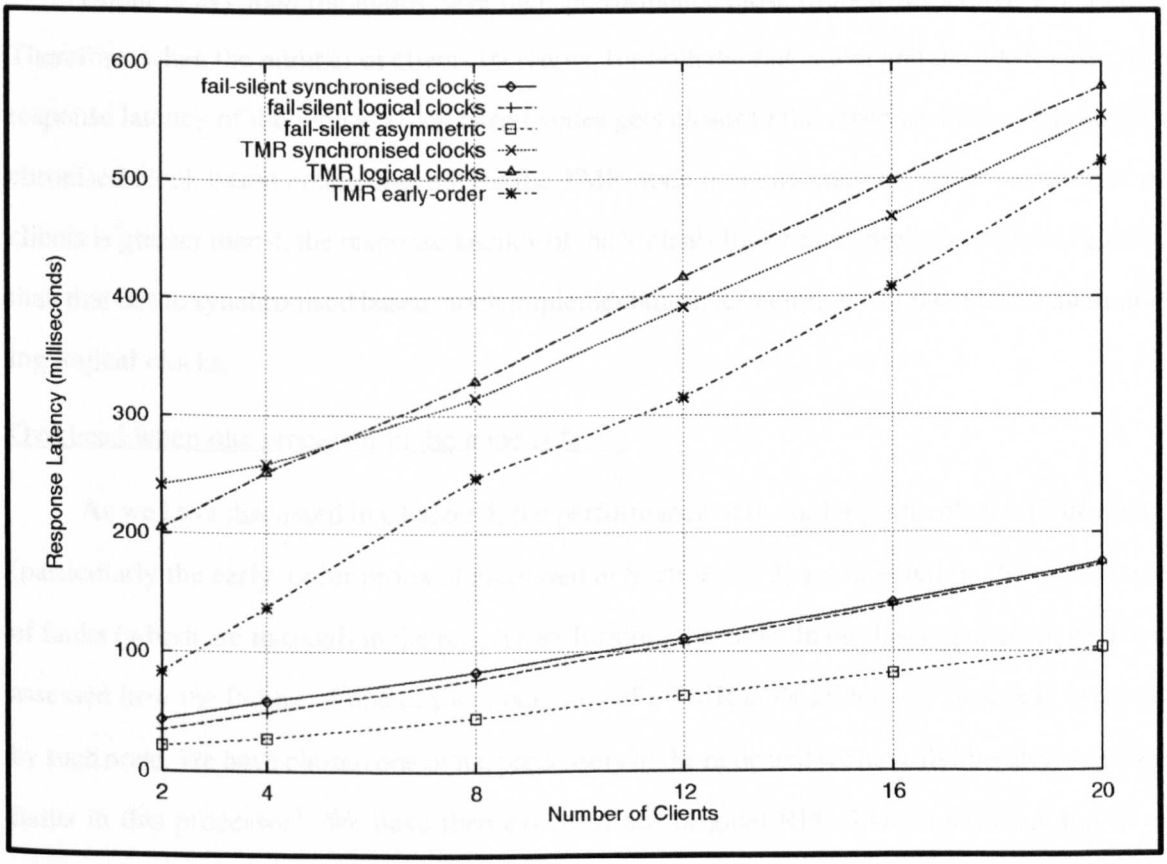

Figure 6–13: Node delay versus number of clients

For the TMR implementations, the early–order node is the one that presents the largest increase in the response latency (24.03 ms/client on average). Both the synchronised clock based and the logical clock based TMR implementations present a much lower increase on their response latency as the number of clients increases (17.38 and 20.82 ms/client on average, respectively). The response latency for these two implementations is less affected because the processing time that would be normally wasted whilst the processor is idle waiting for an input message to become stable can now be used to attend requests from other clients. This behaviour can also be

observed for the fail–silent implementations, although in this case the impact is not as noticeable as it was the case for the TMR implementations. The increase in the response latency of the synchronised clock based implementation is on average 7.35 ms/client, whilst the logical clock based implementation has its response latency increased by 7.70 ms/client on average. The asymmetric fail–silent implementation still out–performs the other two implementations by a considerable margin, and has an increase on its response latency of 4.57 ms/client on average.

Under heavy load the nodes have their performance close to their respective worst case. Therefore, when the number of clients increases, for both the fail–silent and the TMR cases, the response latency of the logical clock based nodes gets closer to the response latency of the synchronised clock based nodes. Indeed, for the TMR node implementations, when the number of clients is greater than 4, the response latency of the logical clock based implementation is greater than that of the synchronised based clock implementation, reflecting the higher cost in maintaining logical clocks.

Overhead when one processor in the node is faulty

As we have discussed in Chapter 4, the performance of the order protocols for TMR nodes (particularly the early–order protocol discussed in Section 4.3.3) are affected by the occurrence of faults (which are masked) in the processors forming the node. In our last experiment we have assessed how the failure of one of the processors of a TMR node affects the overhead incurred by such node. We have chosen one of the processors of the node and we have deliberately injected faults in this processor[1]. We have then executed our original RPC–like experiment and have measured the various node delays.

Table 6–4 summarises the figures obtained when we simulate a crash failure of one of the processors forming the node. The crash scenario is achieved by injecting omission faults on both internal links of the processor (see [Tao et al. 94] for more details on this fault injection technique).

1. We have used the fault injection techniques developed in [Tao et al. 94].

Model	Delay (milliseconds)		
	ID	OD	ND
TMR synchronised clocks	202.75	8.28	211.03
TMR logical clocks	163.88	8.73	172.61
TMR early–order	162.57	8.01	170.58

Table 6–4: Performance overhead for TMR nodes containing a crashed processor

As expected, the performance of the early–order node implementation is the most affected by the failure of one of the processors of the TMR node. The sharp increase of the node delay of the early–order implementation is due to the fact that its order protocol only achieves optimum performance when the node is in a failure–free state. As indicated by the values shown in Table 6–4, when there is a faulty processor within the node the early–order protocol has its performance comparable to the performance of the logical clock based order protocol.

On the other hand, for the other two implementations, the failure of one processor does not cause an increase on their respective node delays. In fact, for crash failures such as the one we have simulated, the node delay of the synchronised clock based and logical clock based implementations are slightly reduced. This is due to the fact that since one processor has crashed, the internal traffic of messages is reduced, which in turn decreases the internal message transmission delays. Comparing the values shown in Table 6–4 with those presented in Table 6–2, we can see that the output delay (OD) of all three implementations (including the early–order) is reduced by about 2.60 ms when there is a faulty processor within the node.

We note that since fail–silent nodes are expected to halt after the detection of a failure, it was pointless to perform this fault injection experiment for fail–silent nodes. Nevertheless, in a parallel research this fault injection technique has been used to validate the design and the implementation of replicated nodes (see [Tao et al. 94]).

6.4. Concluding Remarks

We have implemented both three–processor failure–masking (TMR) and two–processor fail–silent nodes on a network of T800 Inmos transputers. Extensive experiments were performed to evaluate the performance of the nodes under the various order protocols presented in Chapter 4 (failure–masking nodes) and Chapter 5 (fail–silent nodes).

Most of the performance overheads of the TMR node implementations are associated with the ordering of input messages. In a lightly loaded, error–free system, the early–order implementation presents a much reduced performance overhead when compared with the performance of both the synchronised clock and the logical clock based TMR implementations. For our particular system, the early–order implementation out–performs the synchronised clock based implementation by a factor of 4.6, and the logical clock based implementation by a factor of 3.8. On the other hand, when there is a faulty processor within the node the performance of the early–order implementation is comparable to that of the logical clock based implementation, still out–performing the synchronised clock based implementation. As will be seen in the next chapter, reconfiguration mechanisms can be introduced to replicated nodes, so that they are expect to be in an error–free state for most of their lifetime. Thus, taking this in account, we believe that a TMR node comprising the early–order protocol can be a feasible solution for a number of applications.

Concerning the implementation of fail–silent nodes, the results obtained indicate that adopting the asymmetric mechanism for ordering of input messages within a fail–silent node represents the best design choice. Our performance figures have been obtained after quite a careful engineering of the message passing software. It is unlikely therefore that significantly better performance can be produced from soft fail–silent nodes. So the asymmetric node described here probably indicates the limits of what can be achieved using standard 'off–the–shelf' processors.

In our particular implementation, the performance impact of using fail–silent nodes is to produce a delay in response of about 8 ms per message in a lightly loaded system. Further, when processes possess a communication intensive characteristic, a fail–silent node can achieve about 40% of the performance of its unreplicated counterpart. On the other hand, in those cases where application processes are involved in computations requiring less frequent interaction, then the performance impact of adding software–implemented fail–silence can be quite small. For our particular environment, the burden in the performance of an asymmetric fail–silent node can be reduced to less than 10% when the interval between two intra–process communication is larger than 100 ms.

Thus, bearing in mind the discussion on the advantages of soft replicated nodes over hard replicated nodes given in Section 3.2.3, we can anticipate a range of applications for which our software implemented replicated nodes offer an attractive alternative to their hardware implemented counterparts.

Chapter 7
Reconfigurable Replicated Nodes

7.1. Introduction

Regarding the four constituent phases of fault tolerance (see Section 2.2, pages 13–16), the replicated nodes we have discussed in the past three chapters only incorporate fault tolerance mechanisms which are associated with the first two error treatment phases, namely error detection, and damage confinement. (Error recovery is not necessary, since failure–masking nodes incorporate enough redundancy to mask failures without the need of recovering from errors, and fail–silent nodes are expected to halt soon after a failure is detected.) In both failure–masking and fail–silent nodes, error detection is attained by comparing the output messages generated by replica processes executing at distinct processors which fail independently. A message authentication mechanism based on digital signatures is used to confine damage within the node, and allow non–faulty receivers to recognise, and therefore ignore, spurious messages transmitted by faulty transmitters.

Hence, although the failure–masking and fail–silent nodes previously presented are able to treat errors and avoid the undesirable effects of a failure to be seen by the software executing at an upper level, they do not incorporate mechanisms to treat faults and thereby renew the fault tolerance capabilities of the node, once failures have occurred. Fault treatment involves the detection and diagnosis of faulty components, followed by either the isolation of such components or

the replacement of faulty components by non–faulty ones. We term a replicated node capable of treating faults, a *reconfigurable replicated node*.

In this chapter we discuss how reconfiguration mechanisms have been used to provide fault tolerance renewal to fault–tolerant systems described in the literature. We then discuss the possibilities of incorporating similar mechanisms to the failure–masking replicated nodes we have designed, so to construct replicated nodes that are able not only to treat errors, but to treat faults as well.

7.2. Reconfigurable Fault–Tolerant Systems

Many of the systems we have discussed in Chapter 2 use some sort of reconfiguration procedure after failures are detected, in order to maintain the continuity of the service they provide. Systems based on error detection mechanisms normally rely on spare nodes that can take over the computation being executed on the nodes that have failed. Other systems cannot afford the potential unavailability of the system during the interval of time on which reconfiguration is being carried out. Thus, these systems are normally based on some kind of failure–masking mechanism.

Safety–critical multiprocessor systems like SIFT and FTMP use triple modular redundancy to mask the failure of one of the processors in a triad. However, the extremely high reliability requirements of these systems cannot be sustained over their mission lifetime (normally a ten hours flight) by using simply triple modular redundancy, without some sort of reconfiguration procedure being performed to re–establish system's reliability once a failure has been detected. In these systems it is necessary not only to mask failures, but also detect faults, identify faulty components and replace them by non–faulty ones.

Reconfiguration in SIFT

The basic element of computation in SIFT is a task. Tasks can be either unreplicated or replicated. Each unreplicated task has an instance executing at every processor of the system, and each instance of a particular unreplicated task behaves as an independent entity. On the other hand, replicated tasks behave as a single entity which is replicated in different processors (nor-

mally three) for the purpose of fault tolerance. Replicated tasks can be either applications or system tasks, whilst unreplicated tasks are always system tasks.

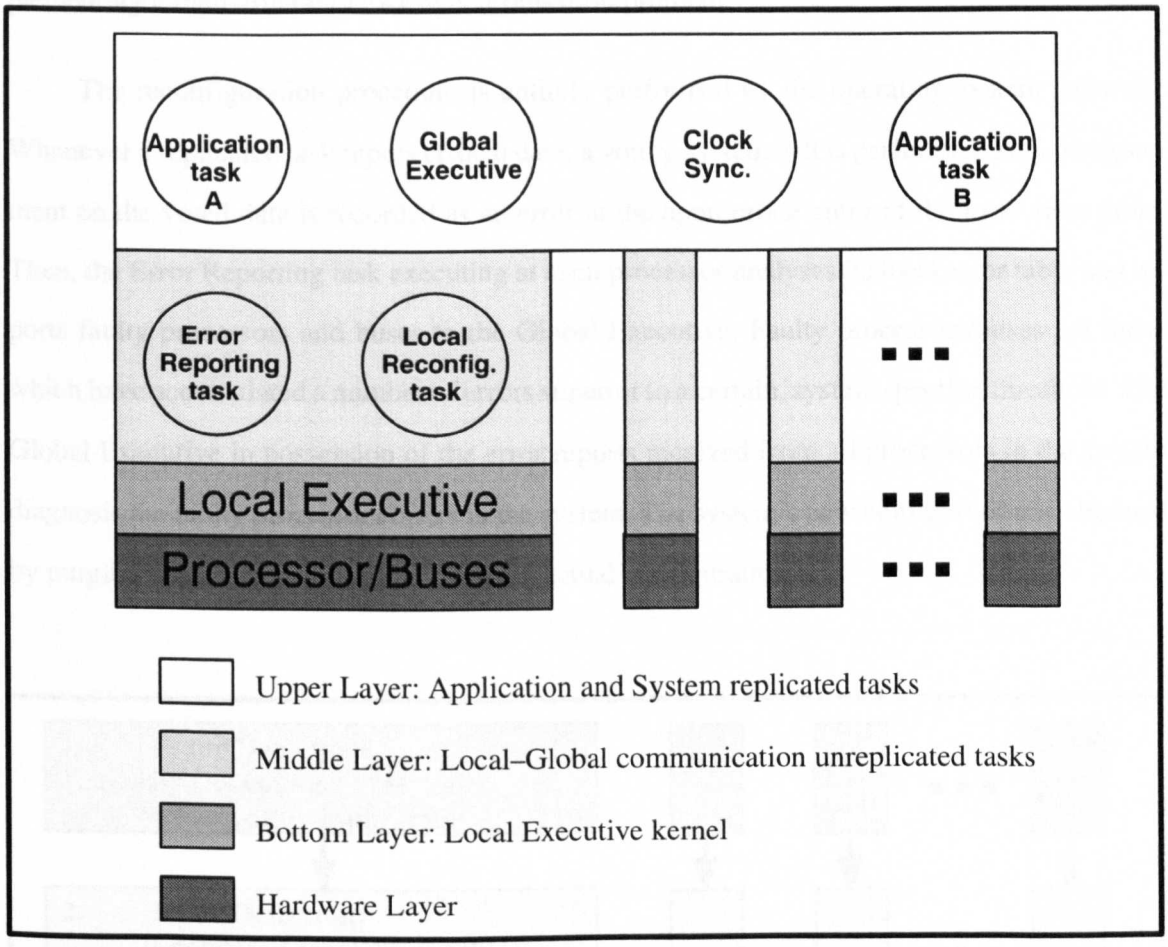

Figure 7–1: Operating system structure of the SIFT system

SIFT's operating system is divided into three layers (see Figure 7–1). At the bottom layer, interfacing with SIFT's hardware, there is a small kernel called the *Local Executive* which is present in all processors in the system. The Local Executive maintains local buffers and tables, and offers system calls to perform error handling, task scheduling, inter–task communication and voting of critical data. In the middle layer, on top of the Local Executive, there are two unreplicated *Local–Global Communication* tasks. The *Error Reporting* task is responsible for analysing the contents of a local error table and determining the local processor's view on which processors and buses in the system are faulty. The *Local Reconfiguration* task is responsible for deciding the role of the local processor in the current configuration of the system. The replicated tasks execute at the upper layer. Apart from the replicated application tasks, there are two replicated sys-

tem tasks. They are the *Global Executive*, which diagnosis faults and provides information for updating the scheduling tables; and the *Clock Synchronisation* task, which is responsible for implementing a fault–tolerant clock synchronisation protocol.

The reconfiguration procedure is entirely performed by the operating system software. Whenever a replicated task inputs critical data, a voting system call is performed. Any disagreement on the voted data is recorded as an error at the appropriate entry of the local error table. Then, the Error Reporting task executing at each processor analyses its local error table and reports faulty processors and buses to the Global Executive. Faulty processors/buses are those which have accumulated a number of errors superior to a certain, system specific, threshold. The Global Executive in possession of the error reports received from all processors in the system diagnosis the faulty processors/buses in the system. The system's new configuration is obtained by purging the faulty components from the actual configuration.

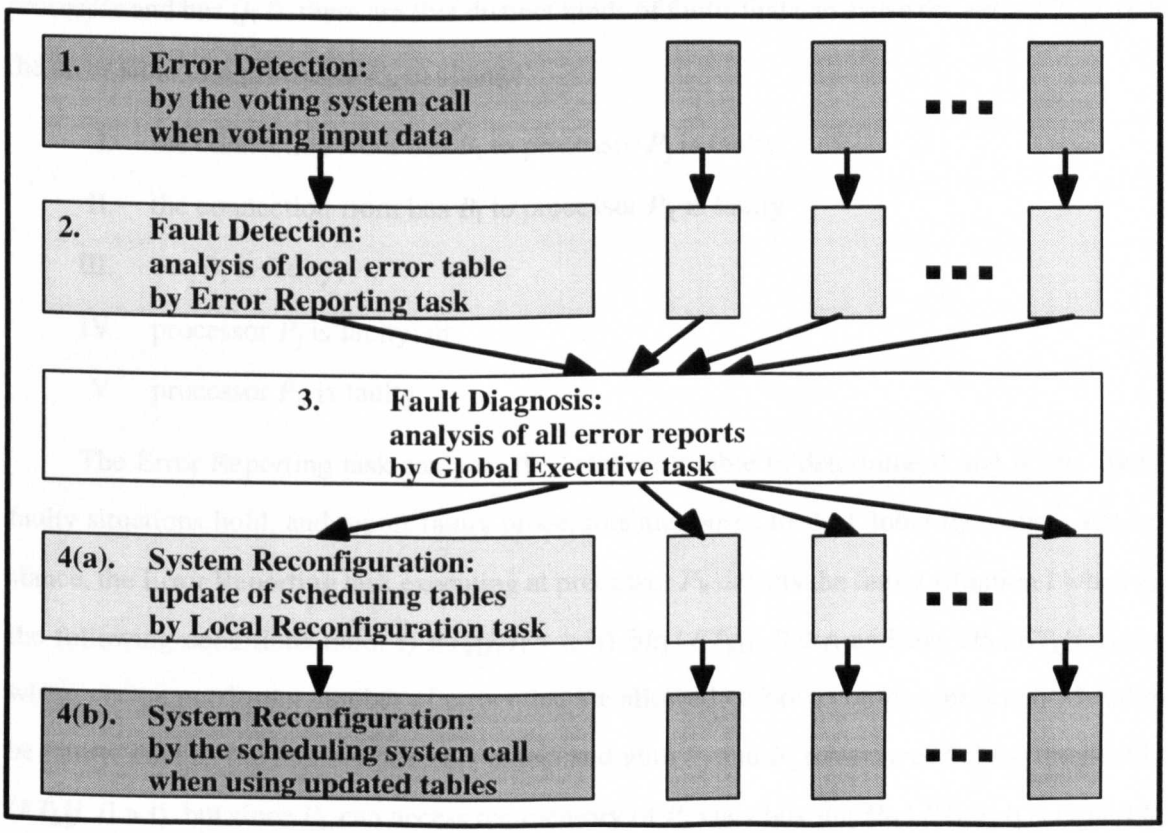

Figure 7–2: Reconfiguration phases

Tasks are scheduled following a predefined scheduling policy. There is a policy for every possible system configuration. Therefore, after diagnosing faulty processors, the Global Executive reports these processors to the Local Reconfiguration tasks executing at each processor which can choose the appropriate scheduling policy for the surviving system. Faulty buses are also reported, so that they are no longer used for accessing input data for voting. Figure 7–2 gives a schematic view of the several phases associated with the reconfiguration procedure.

The local error table is a p by b matrix ET, where p is the number of processors and b is the number of buses in the system. Since the system is assumed to start operation from a failure–free situation, initially, $ET[j, i] = 0$, for every pair (j, i) formed by a processor P_j and a bus B_i. As errors are detected by the voting of input data, the values in the error table ET are updated. Voting is performed at each processor by reading the memory of the three processors that comprises a particular triad, via three different buses. Thus, whenever an error is detected for the data read at processor's P_j memory via bus B_i, $ET[j, i]$ is incremented by one. For a particular pair of processor and bus (j, i), there are five distinct kinds of faults that can cause the entry $ET_k[j, i]$ of the error table of a processor P_k to change:

I. the connection from bus B_i to processor P_j is faulty;

II. the connection from bus B_i to processor P_k is faulty

III. bus B_i is faulty;

IV. processor P_j is faulty; or

V. processor P_k is faulty.

The Error Reporting task analyses the local error table to determine if any of the above faulty situations hold, and report faulty processors and buses to the Global Executive. For instance, the Error Reporting task executing at processor P_k detects the faulty situation I when all the following conditions hold: i) $ET_k[j, i] > t$; ii) $\exists B_l \mid ET_k[j, l] \leq t$; and iii) $\exists P_l \mid ET_k[l, i] \leq t$; where t is the maximum number of errors that are allowed before a component is considered to be faulty. That is, the number of errors associated with P_j and B_i have exceeded the threshold t ($ET_k[j, i] > t$), but since P_k can access the memory of P_j via a bus B_l ($\exists B_l \mid ET_k[j, l] \leq t$), and B_i can be used by P_k to access the memory of a processor P_l ($\exists P_l \mid ET_k[l, i] \leq t$), then the only possible faulty situation is the one where the connection from B_i to P_j is faulty. After detecting this

faulty situation, the Error Reporting task takes the action of stopping using bus B_i to read processor's P_j memory. Table 7–1 gives a summary of the conditions to detect each of the above faulty situations, together with the actions to be taken by the Error Reporting task in each case.

Faulty Situation	Condition	Action
I	$ET_k[j, i] > t$; and $\exists B_i \mid ET_k[j, i] \leq t$; and $\exists B_i \mid ET_k[j, i] \leq t$	Stop using bus B_i to access processor's P_j memory
II or III	$\forall P_j, ET_k[j, i] > t$	Stop using bus B_i to access processor's P_j memory, and report bus B_i as faulty
IV	$\forall B_i, ET_k[j, i] > t$	Report processor P_j as faulty
V	$\forall B_i, ET_k[k, i] > t$	The Error Reporter cannot be depended upon

Table 7–1: Local fault diagnosis

The Global Executive receives error reports from all processors in the system and makes the final decision about which components are faulty. From the information in Table 7–1 we note that the Error Reporting task cannot distinguish between faulty situations II and III, since either faulty situations can hold when $\forall P_j, ET_k[j, i] > t$. Thus, these two cases can only be distinguished by the Global Executive, with the help of the error reports from all processors. Faulty situation III holds if all processors in the system report bus B_i as faulty, otherwise faulty situation II holds. In case faulty situation III holds, the Global Executive reports the faulty bus to every processor in the system. Further, any processor that have been reported as faulty by at least two other processors in the system, are considered to be faulty. Obviously, the error report of a processor that detects itself as being faulty is not considered in the analysis, since although that processor must be indeed faulty, for this very reason, its error report cannot be trusted.

Reconfiguration in FTMP

FTMP uses a similar approach to achieve fault tolerance renewal. However, in FTMP most of the reconfiguration procedure is confined to the hardwired bus interfaces and voters circuits. In FTMP, any three modules (e.g. processors, memories) can be combined to form a triad. Access to modules is achieved via a triad of redundant buses. Every module is able to receive data from all incident buses, and contains a voting component to mask an erroneous data received via one of the buses forming the active triad. At any time there is only one triad of active buses for acces-

sing a particular module. Each module possesses a *bus guardian* which is responsible, among other things, for selecting which triad of buses should be used to access the module. The hard-wired voters provide FTMP with its failure–masking capability. The voters are also responsible for detecting errors. They are equipped with latches that are set in case of a disagreement on the data being voted, indicating which bus was in disagreement. A replicated fault detection program executes periodically reading the error latches. The error latches are cleared after a read operation, and each error indication present in an error latch is credit as a demerit to the correspondent component. If a component accumulates a number of demerits superior to a system dependent threshold, then the component is considered to be faulty. Reconfiguration is executed by issuing the necessary commands to the appropriate bus guardians.

The main goal of a multiprocessor system is to achieve high throughput by executing parallel computations on the replicated processors of the system. Fault–tolerant multiprocessor systems like SIFT and FTMP, take advantage of the inherent redundancy of components to achieve dependability as well. In both SIFT and FTMP, tolerance renewal is achieved by diagnosing faulty processors, and distributing their load among the remaining non–faulty processors, with a consequent trade–off between performance and dependability. Another approach is the one taken in the construction of the quadrupled version of FTP (QuadFTP[1]) reported in [Lala 86]. In that system both system throughput and system dependability are maintained, irrespective of the occurrence of faults, by the use of spare components that can take over faulty ones, with little system disruption.

Reconfiguration in QuadFTP

The QuadFTP architecture is an extension of the triple modular redundant FTP reported in [Smith 84]. It was designed to attain very high safety requirements, which ruled out the utilisation of the limited failure–masking capabilities of the original FTP architecture. QuadFTP operates in a ^{1}fail–operational/^{2}fail–operational/fail–safe failure semantics [Somani–Sarnaik 89], i.e. the system uses a reconfiguration mechanism which enables it to mask up to two failures,

1. This terminology is ours.

maintaining the system operational after the first and second failures are detected, and then failing safely after the detection of a third failure.

The system is composed of four processors executing in lock–step, and having their outputs validated by hardwired voter circuits. Information is exchanged among the processors using inter–stage circuits which provide a communication service resilient to Byzantine faults. The circuit formed by the voters, the inter–stages and the associated error latches is referred as a *communicator* (see Figure 3–4, page 42). Apart from masking failures, the voters also detect errors, and set their error latches to indicate the processors which have provided erroneous data. Reconfiguration is performed by a system program which periodically reads the voters' error latches and diagnosis faulty processors. Faulty processors are turned off by using a processor inter–lock mechanism which allows a majority of processors to turn off a failed processor and its output.

QuadFTP's real time operating system is timer interrupt driven. It schedules foreground tasks that cannot be interrupted and that run every 100 milliseconds. Further, it schedules background tasks (which can be interrupted) to execute in any remaining time left of a particular 100 milliseconds cycle. The software architecture of the system is depicted in Figure 7–3.

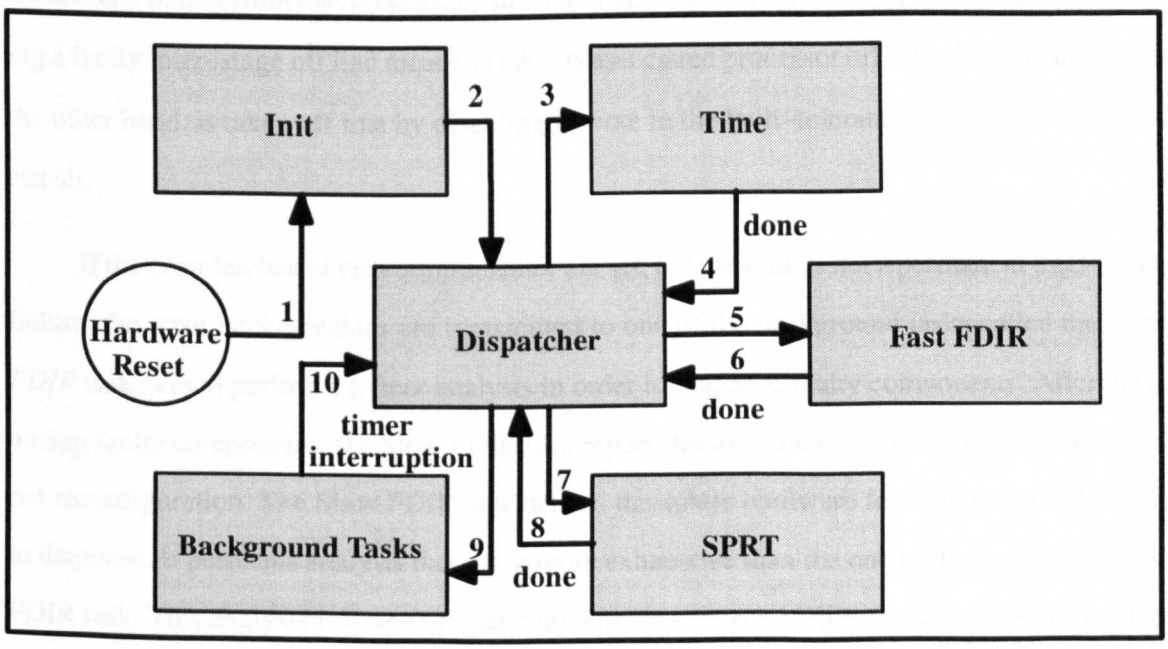

Figure 7–3: Software architecture of the QuadFTP architecture

The *Init* task is executed following system start up synchronisation, after the system is powered on. The Init task sets up data structures and hardware devices, and then switches control to the *Dispatcher* task. Since the Init task is executed only once, the timer interrupt handler is responsible for switching the control to the Dispatcher task after attending a timer interruption every 100 milliseconds. At each cycle, the Dispatcher task schedules the foreground tasks to execute in a predefined order, starting from the *Time* task which is responsible for maintaining the system time, continuing with the *Fast Fault Detection, Isolation and Reconfiguration* task (Fast FDIR) which performs the reconfiguration procedure, and finishing with the *Sequential Probability Ratio Test* task (SPRT), which is the main application task. The background tasks are then scheduled to execute in any remaining time available.

The Fast FDIR task is responsible for carrying out the reconfiguration of the system. At each cycle the Fast FDIR task is activated to try to identify the most common cases of faulty components. It performs a set of tests which are able to detect processors that are out of synchronism, clock faults and permanent inter–stage failures. It then, invokes the reconfiguration procedure to take any faulty component off line. Taking a processor off line means that it will be voted out in any data exchange through the communicator, and that the processor inter–lock monitor will disable any output from that processor. An inter–stage is present in each processor, therefore taking a faulty inter–stage off line means to take its associated processor off line. A faulty clock, on the other hand, is taken off line by disabling its vote in the fault–tolerant clock synchronisation circuit.

If the error latches of the communicator are set, but the fault is not a permanent inter–stage failure, the error latches values are transmitted to one of the background tasks called the *Slow FDIR* task, which performs a finer analysis in order to diagnose faulty components. After diagnosing faulty components, the Slow FDIR task reports them to the Fast FDIR task which carries out reconfiguration. The Slow FDIR task detects the subtle hardware faults which are difficult to diagnose. It performs analysis that is far more exhaustive than the one performed by the Fast FDIR task. This analysis is based on matching known patterns of failures with the error latches values received. Another function of the Slow FDIR task is that of bringing processors that are believed to have suffered a transient fault back into operation. The Slow FDIR task maintains a

'health' indication for each processor (similar to the demerit assignment in FTMP), which is used to decide whether the processor should rejoin the system or whether it has failed permanently.

Latent faults and and other reconfiguration problems

In the simplistic view we have taken when presenting the reconfigurable systems above, the reconfiguration procedure works either by substituting a faulty component for a non–faulty one, or by purging faulty components from the system, the first time the component is detected as being faulty. This view has the shortcoming of not taking into account the latency of faults, i.e. the first fault may not be detected until a later time where a second fault has occurred, which may well preclude recovery. To reduce this possibility, it is imperative to expose latent faults by systematic exercising the error detection mechanisms of the system. For the systems discussed, this means that each voter or communicator must be tested routinely to ensure that its error correction/detection capability is undiminished. This can be achieved by having a system program that tests the voting mechanisms outputs with appropriate input data to generate all possible kinds of errors, and determining if the expected behaviour is attained.

In multiprocessor systems, it is possible to have a test task executing periodically and flexing the system, whilst other triads carry on normal functions. FTMP, for instance, uses this approach to detect latent faults. On the other hand, in a system like QuadFTP, where redundancy is present only for the purpose of fault tolerance, the introduction of periodical test tasks is more problematic. The solution adopted by QuadFTP is to use background tasks (see Figure 7–3), which are executed only when the system would otherwise be idle. We have already discussed the functioning of the Slow FDIR background task, when describing the reconfiguration procedure. Figure 7–4 shows a list with all the background tasks executing on a QuadFTP system, and the order on which they are executed.

The other background tasks shown in Figure 7–4 are the *Alarm Display* task which produces a system status display and outputs information on logged errors; the *Pick–up Test* task, and the *Lost Soul Synchronisation* task, which are also part of the reconfiguration procedure (these tasks are responsible for bringing processors that have experienced transient faults back

into the system); and the *Self Test* task, which is responsible for the periodical flexing of the system and the detection of latent faults.

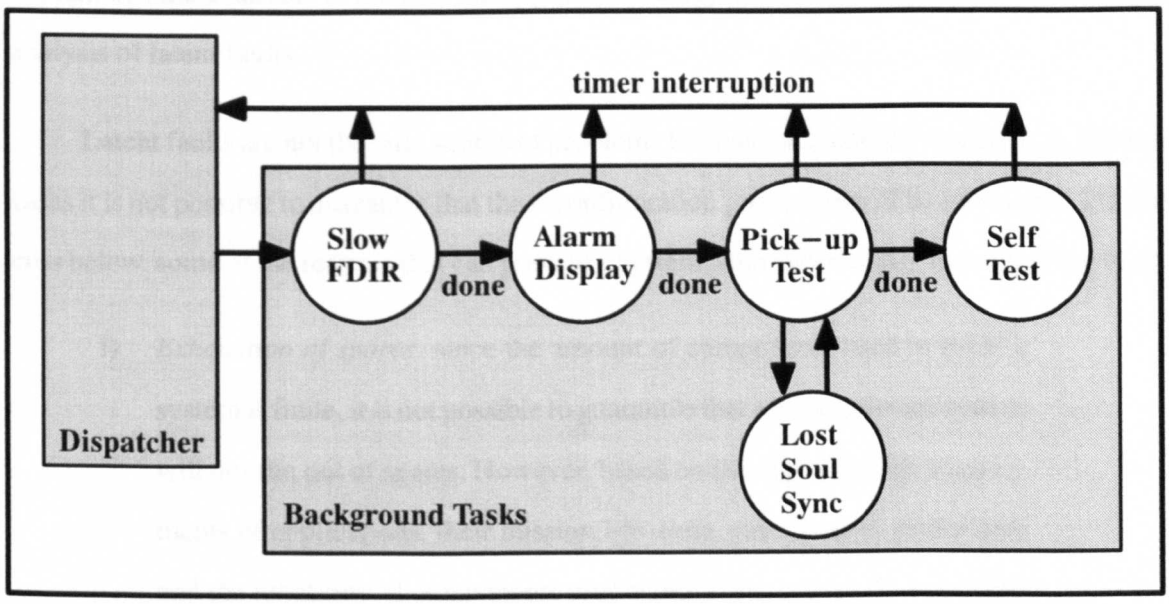

Figure 7–4: Background tasks of the QuadFTP architecture

The Self Test task periodically tests all hardware components of the system. Voters are subjected to spurious data to check if their masking and error detection capabilities have been reduced. Checksums are used to test non–volatile memory, whilst volatile memory of redundant processors are compared on a word by word basis for a quick detection of bit failures. Also, pattern sensitive failures in volatile memory are tested by suitable pattern writing tests.

Background tasks are executed in the remaining time available of every 100 milliseconds cycle. When the last foreground task (the SPRT task) returns the control to the Dispatcher task, the latter switches the control to the first background task (the Slow FDIR task). After completion, the Slow FDIR task switches control to the next background task (the Alarm Display task), and so on. When a timer interruption occurs, signaling the end of the cycle, the state of the background task which is currently executing is saved, thus allowing that task to continue from that point the next time it gets control of the processor. The Dispatcher task regains control of the processor and schedules the first foreground task, initiating a new scheduling cycle (see Figure 7–3). Note that since a background task in the head of the list executes more often than one in the tail, the order on which the background tasks are executed defines a priority hierarchy among these

tasks. Thus, by choosing an appropriate cycle duration, and priority relation among background tasks, the designers of QuadFTP could find the correct balance in the trade–off between the amount of time available to execute the application tasks, and the time dedicated to exhaustive analysis of latent faults.

Latent faults are not the only source of problems for a reconfiguration mechanism. In some cases it is not possible to guarantee that the reconfiguration procedure will be successful. We discuss below, some of the reasons that can prevent a system from successfully reconfiguring itself.

i) *Exhaustion of spares*: since the amount of components used to build a system is finite, it is not possible to guarantee that a fault–tolerant system will not run out of spares. However, based on the dependability requirements of applications, their mission life–time, maintenance procedures and the reliability of components used to build the system, it is possible to calculate the number of spares necessary to guarantee the desired level of dependability. In QuadFTP, for instance, it was sufficient to have just a single spare processor to provide an acceptable degree of dependability for the particular applications executing on that system.

ii) *Use of defective spares*: to avoid the use of defective spares, it is necessary to test spare components for latent faults, so that defective spares will not be introduced into the system. FTMP reduces the probability of using defective spares by continuously reconfiguring the system, in such a way, that all modules, including those part of the spare contingent, are constantly used and therefore tested.

iii) *Malfunction of the reconfiguration system*: clearly, the reconfiguration procedure of a fault–tolerant system must also be fault–tolerant. In the systems we have discussed, reconfiguration is performed by replicated software executing in processor triads. Furthermore, it is assumed that faults occur sequentially, i.e. failure detection, fault diagnosis and system reconfiguration are executed fast enough, so that the possibility of

another fault occurring during this procedure is negligible [Somani 90]. Hence, it is highly improbable that the reconfiguration procedure will fail, even when the triad where it is executing is injured. Note that in the systems we have discussed, the reduction of the failure detection latency by periodically flexing the system is an important factor to reduce the probability of a second fault occurring on an injured triad before it has been successfully reconfigured.

iv) *Failure to detect the need for reconfiguration*: in a system where processors can act maliciously, it is always possible that a faulty processor behave in such a way that only part of the processors in the system will detected it as faulty. This in turn can prevent fault diagnosis, and consequently, system reconfiguration. In FTMP this possibility is reduced by frequently reconfiguring and flexing the system. On the other hand, both SIFT and QuadFTP use the knowledge about their particular hardware organisation and the characteristics of the applications they execute, in order to design accurate testing procedures which can detect faults with very high probability. In QuadFTP, for instance, this is performed by the Slow FDIR background task. This task matches known failure patterns against the error latches it receives from all processors in the system. The known patterns used were determined by an analysis of failure modes of the communicator hardware and examining the patterns they would produce for different faulty scenarios.

7.3. Constructing Reconfigurable Replicated Nodes

So far, we have studied replicated nodes with two kinds of failure semantics, namely failure–masking and fail–safe nodes. The fail–silent nodes discussed in Chapter 5 present a safe failure semantics. They follow the conservative approach of halting as soon as a failure within the node is detected. As seen in the previous chapter, these nodes can be very cheap and simple to implement (a two–processor fail–silent node is the cheapest replicated node that can be con-

structed). Further, for a great number of applications, the overhead incurred by the replication protocols necessary to implement fail–silent nodes generate only a relatively small burden on the throughput of the node.

The failure–masking nodes discussed in Chapter 4 are more expensive to implement, and impose a heavier overhead on the node's throughput. On the other hand, they are able to mask a bounded number of internal failures, which make their survivability greater than than that of fail–safe nodes. Masking however, has two distinct sides that must be pondered. On one side, it can cope successfully with failures and yet sustain the node's dependability, whilst on the other side it may obscure the fact that a fault has occurred, and thereby has reduced the node's tolerance to future faults. Thus, even though masking mechanisms can cope with the immediate danger of a failure, they can also hide the fact that the masking capability of the node has degraded, up to a point where a further failure will defeat the fault tolerance mechanisms of the node, with potentially catastrophic results.

An obvious strategy to provide fault tolerance renewal to failure–masking replicated nodes is the introduction of spare processors. Using this strategy, a failure–masking node can be built by coupling N active processors, $N = 2\pi+1$, which execute the replicated applications, together with S spare processors, which can take over the computation of active processors that have failed. A fault diagnosis procedure is periodically executed to identify processors that have failed. Faulty processors are reconfigured out of the node and substituted by processors from the spare pool. Such node is able to tolerate the failure of up to $\pi+S$ processors, provided that no more than π processors fail between two executions of the reconfiguration mechanism.

This solution however does not avoid the possibility of a catastrophic behaviour, since the introduction of spares only reduces the probability of the node having its masking capabilities entirely degraded. The possibility of exceptional behaviour after $S+\pi$ faults have occurred is still present. Thus, what is needed is a replicated node which could simultaneously incorporate the survivability property of failure–masking nodes, together with the safety property of fail–safe nodes.

In [Shrivastava et al. 91] the notion of a *failure–masking before stopping node* (FMS node) is defined. A π–FMS node is a replicated node composed of N processors, $N = 2\pi+1$, which is able to mask up to π failures. However, unlike the failure–masking nodes previously discussed, once the failure of π processors have been detected, the node must fail safely, rather than continue processing. This failure semantics avoids the unpredictable behaviour of the node in case another fault occur after the masking capabilities of the node had been entirely degraded. A π–FMS node periodically executes a fault diagnosis procedure to identify processors that have failed within the period between two executions of the fault diagnosis procedure. Whenever the total number of faulty processors diagnosed reaches π, then the node must halt. Fault diagnosis, is therefore a crucial part on the design of a π–FMS node. Assuming that a π–FMS node initially contains only non–faulty processors, every execution of the fault diagnosis procedure must guarantee the following two conditions, in the presence of up to π failures:

agreement: all non–faulty processors must reach agreement on the processors that should be diagnosed faulty; and

completeness: all faulty processors, whose failure have been detected by non–faulty processors, and only these, are diagnosed faulty.

The failure of a faulty processor can only be detected through the messages it sends or fails to send. Thus, there is always a possibility that a faulty processor behave in a two–facing manner on which it sends timely messages with correct contents to some processors in the node, whilst at the same time it sends messages either untimely, or with incorrect contents to other processors in the node. A faulty processor behaving in this way is detected as faulty by some processors, whilst some others are not able to detect it as being faulty. It is easy to see that there are cases where it is not possible to reach a consensus on which processors are faulty. For instance, in a three–processor node formed by processors P_1, P_2 and P_3, where P_2 is faulty, and is detected faulty by P_1, but not by P_3, it is possible that P_3 receives a report from P_1 stating that P_2 is faulty, and also a report from the faulty P_2 stating that P_1 is faulty. P_3 cannot trust any of the reports, and is not able to identify which processor is faulty. This means that the completeness property is not achieved, and therefore, the method described above cannot be used to construct an FMS

node from its equivalent failure–masking node. To the best of our knowledge, there is no fault diagnosis protocol which can guarantee the completeness condition above, even when $N > 2\pi+1$.

Based on the previous discussion, the authors of [Shrivastava et al. 91] conjecture that it is not possible to build FMS nodes from processors which can fail in an arbitrary way (even if they have access to a message authentication service). In the next section we prove the [Shrivastava et al. 91] conjecture to be false, by presenting an FMS node constructed by the replication of authentication–detectable fail–arbitrary processors.

7.3.1. Constructing Reconfigurable Nodes from Fail–Safe Components

As discussed before, if processors can fail arbitrary, it is possible to have faulty processors behaving in a two–faced way such that the remaining non–faulty processors are not able to achieve consensus on which processors are faulty. Since fault diagnosis is a crucial feature of an FMS node, an alternative approach for the construction of such nodes is the utilisation of processors with more restrictive failure modes, which do not allow the two–facing behaviour which defeats diagnosis. Thus, we start our discussion by considering the construction of π–FMS nodes composed of fail–safe processors.

A π–FMS node has a ^{1}fail–operational/.../$^{\pi-1}$fail–operational/fail–safe failure semantics when $\pi > 1$, and a fail–safe semantics when $\pi = 1$. In other words, when $\pi > 1$ the node is able to mask up to π failures, maintaining the node operational after π–1 failures are detected, and then failing safely after the detection of the π^{st} failure; whilst when $\pi = 1$ the node fails safely as soon as a failure is detected.

Hence, the design of a π–FMS node composed of π fail–safe processors is very simple. In such a node, application processors are replicated and executed in each fail–safe processor. In order to guarantee that replicas executing at different fail–safe processors do not diverge, input messages are subjected to an order protocol which provides each replica with an identical input message stream. Note that since applications execute in fail–safe hardware, their outputs need not be subject to a voting mechanism. Also, the order protocol can be made very simple and efficient. Figure 7–5 shows how two fail–safe processors can be coupled together to construct a 2–FMS node.

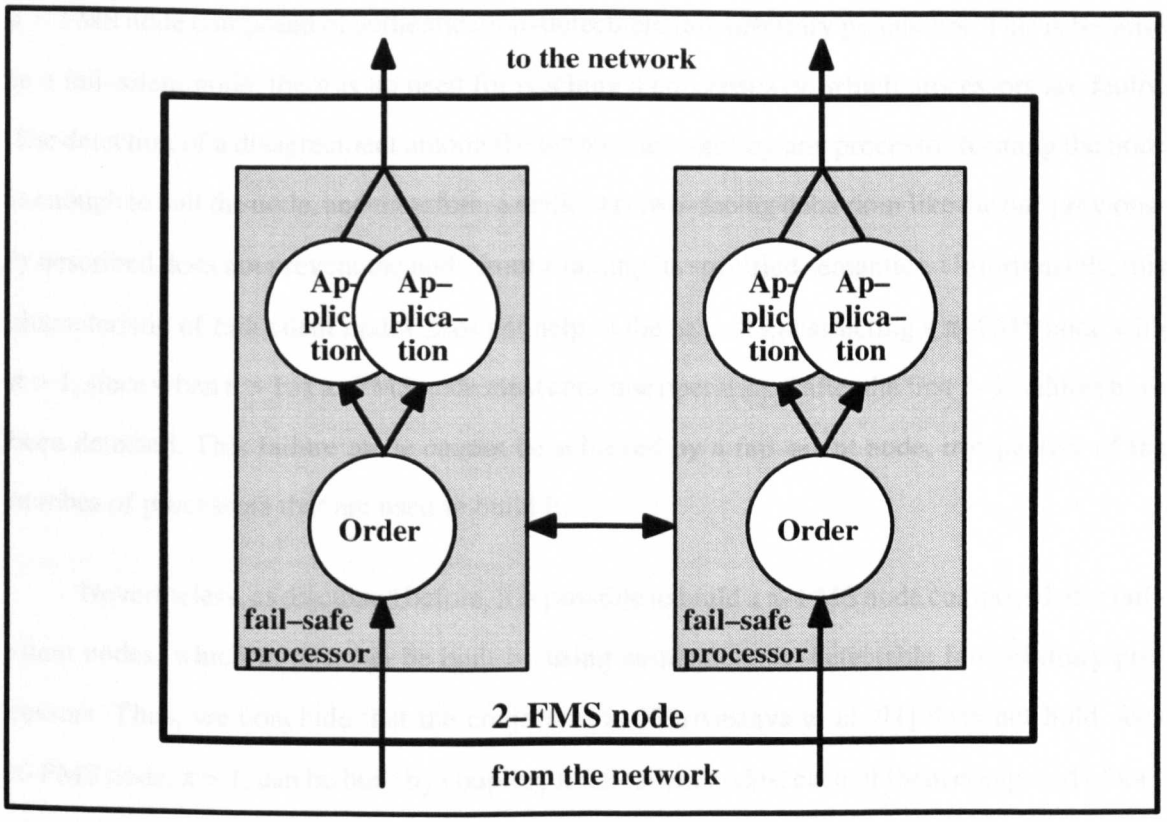

Figure 7–5: Using a pair of fail–safe processors to construct a 2–FMS node

With this structure, the failure of up to π–1 fail–safe processors is automatically masked, and after the failure of the π^{st} processor the node fails safely. The safe failure semantics of the processors which comprise the node guarantees that they do not interfere with the node output once they have failed, thus there is no need for the non–faulty processors of the node to engage themselves in fault diagnosis procedures to assess the number of processors that have failed.

From the structure of a π–FMS node presented above, it is easy to see that when $\pi = 1$, and hence applications are not replicated, there is no need for ordering input messages. In fact, a closer investigation shows that the failure semantics of a 1–FMS node is the same as that of a fail–safe processor, i.e. both must stop after an internal failure is detected. This observation has motivated us to contest the conjecture that it is not possible to build FMS nodes from processors that can fail arbitrary.

In Chapter 5 we have shown how fail–silent nodes, a kind of fail–safe node, can be built from authentication–detectable fail–arbitrary processors. Thus, since a fail–safe processing site can be built from authentication–detectable fail–arbitrary processors, it is also possible to build

a 1–FMS node composed of authentication–detectable fail–arbitrary processors. This is because in a fail–silent node, there is no need for reaching a consensus on which processors are faulty. The detection of a disagreement among the output messages by any processor forming the node is enough to halt the node, and therefore, a malicious two–facing behaviour like the one previously described does not prevent the node from attaining its specified semantics. Unfortunately, this characteristic of fail–silent nodes does not help in the case of constructing a π–FMS node with $\pi > 1$, since when $\pi > 1$, a π–FMS node must continue operational after the first π–1 failures have been detected. This failure mode cannot be achieved by a fail–silent node, irrespective of the number of processors that are used to build it.

Nevertheless, as discussed before, it is possible to build a π–FMS node composed of π fail–silent nodes, which in turn can be built by using authentication–detectable fail–arbitrary processors. Thus, we conclude that the conjecture in [Shrivastava et al. 91] does not hold, as a π–FMS node, $\pi > 1$, can be built by coupling π fail–silent nodes, each of them composed of π+1 authentication–detectable fail–arbitrary processors. Figure 7–6 shows the structure of a 2–FMS node constructed from authentication–detectable fail–arbitrary processors.

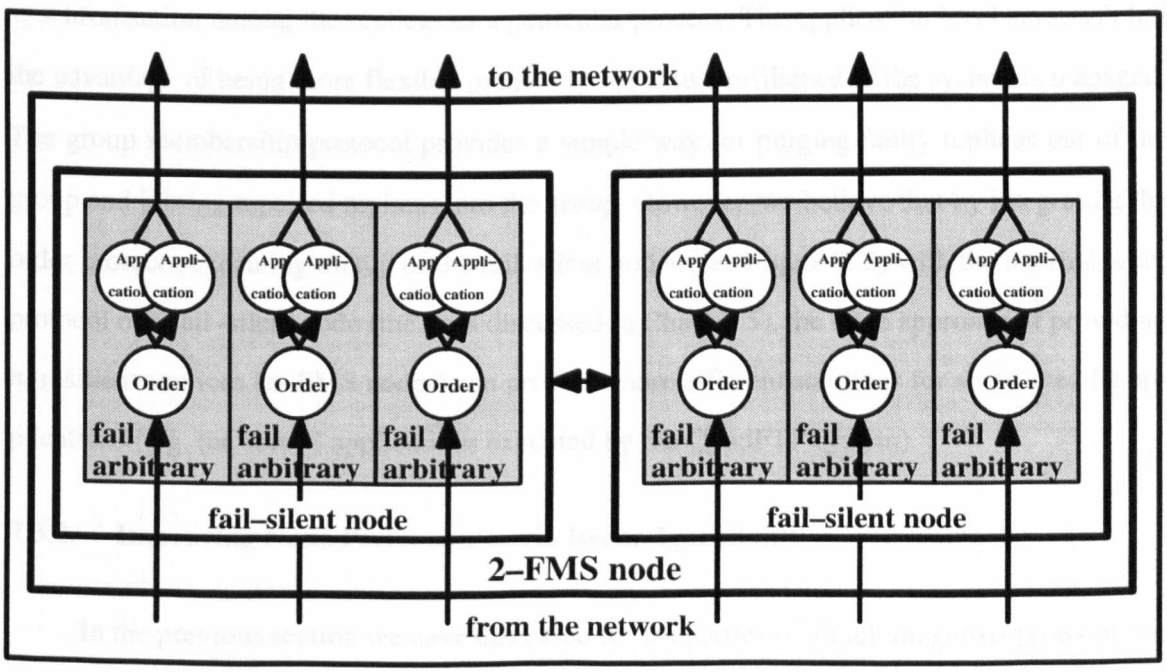

Figure 7–6: Using fail–arbitrary processors to construct FMS nodes

A π–FMS node, $\pi > 1$, constructed in this way has a ^{1}fail–operational/.../$^{\pi-1}$fail–operational/fail–safe failure semantics, provided that at each fail–silent node no more than π processors are faulty. Note that since up to π faults can occur simultaneously, each fail–silent node must be able to tolerate up to π failures, and therefore must be composed of at least π+1 authentication–detectable fail–arbitrary processors. Thus, the number of authentication–detectable fail–arbitrary processors necessary to build a π–FMS node following this approach is given by $\pi(\pi+1)$, which, for $\pi > 2$, is much larger than the 2π+1 processors necessary to implement a failure–masking node with equivalent survivability. However, if faults are assumed to occur sequentially, then, more economical two–processor fail–silent nodes can be used, and the total number of processors needed to build a π–FMS node is reduced to only 2π.

Achieving π–resilience in this way, is of course a well known technique and has been used to implement highly available services at the application level on a variety of distributed systems. (See the Delta–4 XPA architecture [Barrett et al. 90] and the protocols in [Ezhilchelvan–Shrivastava 91].) These systems assume that the underlying nodes are fail–safe, and develop protocols to allow the communication of replicated applications. They are normally based around group membership and group communication protocols [Birman et al. 91] which provide the necessary synchronisation among the replicas of a particular process. The application level approach has the advantage of being more flexible in the way the actual resilience of the system is managed. The group membership protocol provides a simple way for purging faulty replicas out of the group and joining repaired replicas into the group. However, we believe that by integrating the order protocol executing on top of the fail–silent nodes (see Figure 7–5) with the internal order protocol of a fail–silent node (the ones discussed in Chapter 5), the node approach of providing π–resilient services (π–FMS nodes) can produce more efficient solutions for some specific applications (e.g. the sort of applications executed by the QuadFTP system).

7.3.2. Improving Node Performance via Reconfiguration

In the previous section we have discussed the properties of a fault diagnosis protocol, for diagnosing faulty processors within a replicated node composed of processors which can fail arbitrary. Such fault diagnosis protocols must guarantee agreement and completeness properties.

If processors are assumed to fail arbitrary, there is no known protocol which can guarantee the completeness property. However, a *partial fault diagnosis protocol* which only presents the agreement property can be implemented with the assumption that processors fail arbitrary [Shin–Ramanathan 87]. In this section we discuss how a partial fault diagnosis protocol can be used to improve the performance of replicated nodes.

In Section 4.3.3 we have presented an early–order protocol for replicated nodes composed of authentication–detectable fail–arbitrary processors possessing *fifo* internal channels for intra–node communication. A node using that protocol can order messages within a very short time provided that the node is in a failure–free state. However, when at least one processor is faulty, the performance of the protocol can be dramatically reduced (see the fourth experiment described in Section 6.3.2). However, as will be shown shortly, by purging faulty processors from the node, and having the node reconfigured to a failure–free state, it is possible to restore the performance of the ordering protocol to its optimum value.

In the early–order protocol of Section 4.3.3, a non–faulty processor needs to receive messages that have been broadcast by every other processor in the node before it can order a message. Since faulty processors may fail to broadcast a message, or may broadcast a message too late, a non–faulty processor must also use time–out mechanisms to detect the absence, or the untimeliness of such broadcasts. These time–outs are based on the communication maximum delay to transmit a message within the node, which is normally much larger than the actual communication delay. Thus, a faulty processor can slow down the ordering of messages by delaying or failing to broadcast a message.

The performance of a failure–masking node which uses the early–order protocol of Section 4.3.3 can be improved by introducting into the node design a partial fault diagnosis protocol. If all non–faulty processors agree that a particular processor is faulty, then they can simultaneously stop considering messages that such a faulty processor may broadcast, resulting in not having to wait for broadcasts that are never received, or that are delayed. Note that provided non–faulty processors agree on which processors have failed, failure in diagnosing *all* the processors that have failed may only prevent the reduction on the delay imposed by the order protocol, but does not compromise the correct functioning of the order protocol. Also, it is noteworthy that crashed

processors, i.e. processors that have failed by stopping to send messages out, are detected as faulty by all non–faulty processors of the node, and therefore are always successfully diagnosed as faulty. Since crash is the most common fault behaviour experienced by 'off–the–shelf' processors, it is reasonable to consider that in most cases a failure–masking node which incorporates the mechanism described in this section is able to reconfigure itself. Thus, most of the time the node will be in a failure–free state, which allows early ordering of input messages.

7.4. Concluding Remarks

We have studied the reconfiguration strategy of a number of fault–tolerant distributed systems reported in the literature. Reconfiguration aims to increase the survivability of systems by purging faulty components and replacing them by non–faulty ones. We have shown that in some situations a faulty processor can behave in such a malicious way, that it can prevent reconfiguration. We have studied the mechanisms that different systems use to reduce the probability of reconfiguration failure. We then tried to apply similar mechanisms in order to build a reconfigurable replicated node.

We have presented the design of a π–FMS node constructed from 'off–the–shelf' processors, which can fail–arbitrary. This contradicts a conjecture in [Shrivastava et al. 91] which argued the impossibility of constructing such nodes. π–FMS nodes built in the way described in this chapter are very expensive, since the number of processors needed is a function of π^2. For values of π greater than 2, it is possible to build cheaper failure–masking nodes with better survivability (however subjected to a potential uncontrolled behaviour if the number of faulty processors exceed the upper bound of maskable faults). Nevertheless, if faults are assumed to occur sequentially, a much cheaper solution requiring only 2π processors can be attained.

Finally, we have also discussed how a partial fault diagnosis protocol can be introduced into the design of a failure–masking node to implement a node reconfiguration mechanism. This mechanism can then be used to purge faulty processors out of the node, allowing the node to remain failure–free most of the time. As discussed in Section 4.3.3, input messages can be ordered extremely fast when there are no faulty processors within the node.

Chapter 8
Conclusions

8.1. Discussion

In this thesis we have presented our research on the design and implementation of fault–tolerant fail–controlled nodes which can be used as a processing platform for the development of dependable distributed computer systems. Designing and implementing dependable distributed systems is a difficult task, especially when the design must cope with processors and communication failures. In Chapter 2 of this thesis we have shown different fault tolerance mechanisms that have been used to develop some of the dependable computer system reported in the literature. We have put emphasis in differentiating the design of centralised specific purpose systems from the design of general purpose distributed systems. We have shown that a common approach, which attempts to simplify the design of dependable distributed systems, is to assume that the underlying processing hardware and communication system present a well defined failure mode (fail–controlled semantics) [Bartlett 81, Kopetz–Merker 85, Shrivastava 89, Powell 92, Birman et al. 91]. However, since all hardware components will eventually fail, possibly in an unpredictable way, for applications with high dependability requirements, conventional hardware cannot be assumed to have a fail–controlled semantics. Thus, in order to have their dependability requirements guaranteed, such applications must execute on specially designed hardware which can provide, with sufficiently high probability, the fail–controlled behaviour assumed by the upper level software protocols.

Replicated processing on distinct processors which fail independently is a practical means of constructing fault–tolerant fail–controlled nodes. In these nodes, redundant outputs produced

by the processors forming the node are subjected to a validation mechanism (e.g. comparator, voter), which prevents the outputs of faulty processors from appearing at the application level. A number of dependable computer systems reported in the literature have incorporated fail–controlled replicated nodes following this design; some of them have been discussed in Chapter 3.

The design of fail–controlled replicated nodes may follow either a hardware based approach (hard nodes) or a software based approach (soft nodes). In hard replicated nodes, redundant processors are coupled together in tight synchronism and execute identical software in lock–step. Thus, if all processors are functioning, their output are the same at each clock cycle, and by applying an appropriate hardware validation circuit to the outputs of processors, it is possible to attain different fail–controlled behaviours. FTMP [Hopkins et al. 78] and FTP [Smith 84], for instance, are two systems whose fault tolerance mechanisms rely on their underlying hard failure–masking nodes. MARS [Reisinger–Steininger 93], Sequoia [Bernstein 88] and Stratus [Webber–Beirne 91], on the other hand, are examples of systems which have the implementation of their fault tolerance mechanisms simplified by the use of underlying hard fail–silent nodes.

Opposing the tight micro–instruction level synchronisation strategy of hard replicated nodes, soft replicated nodes have a much looser synchronisation strategy, which synchronises replicas at the task level. In systems using soft replicated nodes, application programs are implemented as a collection of processes or tasks, which communicate with each other through well defined primitives. Replicas are normally assumed to have a deterministic behaviour, thus provided that all functioning replicas receive identical input, they will produce identical outputs, which can be validated by a software validation mechanism. Soft failure–masking replicated nodes have been used in the implementation of the SIFT [Wensley et al. 78] and VOTRICS [Theuretzbacher 86] systems.

The performance overhead incurred by hard replicated nodes is normally very small, and indeed is the main advantage of this approach. Also, the tight synchronism, characteristic of the hardware approach, allows software that have been developed to be executed on unreplicated platforms to be ported and executed on replicated platforms with virtually no modifications. However, as discussed in previous chapters of this thesis, there are also some problems associated with this approach, the most grave of them being the inflexibility of their design. This in turn is

a direct consequence of having the synchronisation and validation mechanisms intimately associated with the design of the hardware components of the system. Hard nodes are normally ad hoc solutions tailored to attend the requirements of a particular system. Therefore, node upgrades due to technology advances or even changes in the degree of replication of the node [Lala 86] cannot be attained, unless substantial redesign is performed, which in turn may increase costs dramatically.

Soft replicated nodes are much more flexible than their hardware counterpart. The principles behind the redundant management protocols of soft replicated nodes are not linked to the hardware design of the node, therefore, technology upgrades involve only porting the software protocols from one platform to another. Also, since the synchronisation and validation mechanisms are not hardwired, the hardware design can be made much simpler and more prone to scale. Further, by employing different types of processors within a node, and diverse implementation of both applications and system software [Randell 75, Avizienis 85], there is a possibility that a measure of tolerance against design faults in both hardware and software can be obtained, without recourse to any specialised hardware assistance.

The main drawbacks of soft replicated nodes are the substantial performance overhead incurred by their redundancy management protocols, and the necessity of structuring application programs in accordance with guide-lines dictated by the replica synchronisation strategy. In SIFT, for instance, application programs must be structured as a collection of cyclic tasks which execute at an a priori known iteration rate. Further, the overhead associated with redundancy management in that system can consume as much as 80% of the processor throughput [Palumbo–Butler 85].

Soft replicated nodes based on the state machine approach [Schneider 90, Shrivastava et al. 91, Shrivastava et al. 92] require application programs to be structured as a collection of processes that communicate with each other exclusively via messages. Since most distributed applications follow this structuring paradigm, this requirement is not a real limitation when these nodes are used as the underlying processing hardware of dependable distributed systems. Further, application processes can be allowed to incorporate some degree of non–determinism [Tully–

Shrivastava 90]. Nevertheless, the performance overhead incurred by the redundancy management protocols is still a concern.

The Voltan family of replicated nodes [Shrivastava et al. 91, Shrivastava et al. 92] follows the state machine approach, to present the design of a general node structure which can be used to implement a variety of fail–controlled nodes, ranging from failure–masking to fail–silent nodes. The general structure of a Voltan replicated node have been studied in detail in Section 3.3 of this thesis. Apart from the processes corresponding to the application programs, each processor of a Voltan node executes five system processes that are responsible for the management of the node's redundancy, and which will ultimately provide the node with a particular fail–controlled semantics (see Figure 3–8, page 56). The five system processes are: Transmitter and Receiver processes, which are responsible for inter–node communication; Order process, which guarantees that replica processes receive identical input messages, in identical order; Sender process which is responsible for diffusing information to the other processors of the node for validation; and Validator process, which implements the validation mechanism of the node.

An earlier implementation of failure–masking Voltan nodes has been reported in [Speirs et al. 93]. Apart from the trivial Transmitter, Receiver, and Sender processes, this implementation incorporates an Order process implemented by a clock synchronised based order protocol, and a Validator process, which in the case of a failure–masking node is a Voter process, implemented by a simple voter protocol (see the reference design of a failure–masking node reported in Section 4.2). In [Speirs et al. 93] it is also suggested that the implementation of a fail–silent node can be easily derived from the implementation of a failure–masking node by simply substituting the Voter process by a Comparator process. Further, the Comparator process can be implemented by a comparison protocol adapted from the voter protocol which implements the Voter process of the failure–masking node (see the reference design of a fail–silent node reported in Section 5.2).

In our research we have thoroughly investigated the design of soft fail–silent nodes. Our research revealed some fundamental differences between the design of soft fail–silent nodes and that of soft failure–masking nodes. The results of this investigation were twofold: firstly, we have found that a simple Comparator process adapted from the Voter process of a failure–masking

node cannot provide the fail–controlled semantics of a fail–silent node; secondly, since a fail–silent node is expected to halt after a failure is detected, all systems processes, except the Comparator process, can be designed under the assumption that they will execute on a failure–free environment. Based on these facts, we have developed new order protocols which are much more efficient than the traditional clock synchronised based order protocol adapted from earlier implementations of failure–masking nodes. We have also developed an efficient comparison protocol which, different from the previously known protocols, can guarantee the correct functioning of soft fail–silent nodes (Chapter 5). This has allowed us to produce what is, to the best of our knowledge, the first implementation of a soft fail–silent node [Brasileiro et al. 92] (Chapter 6). Further, we have developed efficient order protocols to implement the Order process of failure–masking Voltan nodes, which substantially reduced the performance overhead of such nodes (Chapter 4).

Fail–silent nodes are very cheap and simple to implement (a two–processor soft fail–silent node is the cheapest replicated node that can be constructed). Further for a great number of applications, the overhead incurred by the replication protocols necessary to implement fail–silent nodes generate only a relatively small burden on the performance of the node. In our particular implementation, the performance impact of using fail–silent nodes is to produce a delay in response of about 8 ms per message in a lightly loaded system. Further, in a worst case scenario where application processes are frequently exchanging messages, a fail–silent node can achieve about 40% of the performance of its unreplicated counterpart. It should be appreciated that this price in performance becomes significant in only those distributed applications which are communication intensive. As discussed in Section 6.3.2, in client–server applications where the server processing time is at least 100 ms, then the performance impact of adding software–implemented fail–silence can be reduced to less than 10%.

Failure–masking nodes, on the other hand, are more expensive to implement, and impose a heavier overhead on the node's performance. Nevertheless, they are able to mask a bounded number of internal failures, which make their survivability greater than that of fail–silent nodes. However, masking also has the undesirable property of obscuring the fact that a fault has occurred, and thereby has reduced the node's tolerance to future faults. Thus, even though masking mechanisms can cope with the immediate danger of a failure, they can also hide the fact that the

masking capability of the node has degraded, up to a point where a further failure will defeat the fault tolerance mechanisms of the node, with potentially catastrophic results.

A π–FMS node [Shrivastava et al. 91] is a replicated node which attempts to encompass the desirable properties of both fail–silent and failure–masking nodes. A π–FMS node is able to mask up to π failures, and will fail silently after the detection of the π^{th} failure, thus combining the safety property of fail–silent nodes and the survivability property of failure–masking nodes. In [Shrivastava et al. 91] the authors show that deriving the implementation of a π–FMS node from the implementation of a failure–masking node composed of N authentication–detectable fail–arbitrary processors, $N = 2\pi+1$, involves the introduction of a fault diagnosis protocol into the node, which is executed periodically, and must be able to diagnosis all processors that have failed since the last execution of the fault diagnosis protocol. However, since there is no known faulty diagnosis protocol that can guarantee the diagnosis of all faulty processors in the node (even when $N > 2\pi+1$), the authors conjectured that it is impossible to build π–FMS nodes from authentication–detectable fail–arbitrary processors.

In contradiction with this conjecture, in Chapter 7 of this thesis we have presented the design of a π–FMS node constructed from authentication–detectable fail–arbitrary processors. The design of such node can be divided into two logical levels, namely a lower, fail–arbitrary level; and an upper fail–silent level. The node is composed of $\pi(\pi+1)$ authentication–detectable fail–arbitrary processors that are logically seen as π fail–silent nodes, each of these composed of $\pi+1$ authentication–detectable fail–arbitrary processors. Each fail–silent node must execute an extra system process which is responsible for ordering input messages at the fail–silent level, and therefore prevent fail–silent replicas from diverge. π–FMS nodes implemented in this way are very expensive, since the number of processors required is a function of π^2. However, for small values of π ($\pi \leq 2$), or when failures can be assumed to occur sequentially, implementations of π–FMS nodes can be obtained with a cost comparable with that of a corresponding failure–masking node.

8.2. Directions for Further Research

As presented in Chapter 6, our current implementation of failure–masking and fail–silent nodes was developed on a network of T800 Inmos transputers [INMOS 88], using the facilities

provided by the Helios operating system [Perihelion 91]. Recently, the protocols have been ported, with relative ease, to another platform consisting of Hewlett–Packard work–stations, and using the facilities provided by their real–time operating system [Morgan 93]. Nevertheless, in both implementations, the redundancy management protocols of the replicated nodes execute as normal application processes. We believe that an interesting exercise would be to assess the overheads associated with the redundancy management protocols when they are incorporated at different levels of a distributed system. Apart from the application level which was attempted by our implementation, the redundancy management protocols of a soft replicated node could be introduced at the micro–kernel level of a true distributed operating systems like Amoeba [Mullender et al. 90] or Mach [Accetta et al. 86], where the possibility of a better control on the policy of scheduling system processes could lead to a reduction of the overheads associated with the management of the node redundancy. It should also be of interest to investigate the use of the software replication concept to provide high dependability to centralised multiprocessor time-sharing systems.

Finally, we believe that the implementation of a practical system using soft replicated nodes as their underlying processing hardware is a very interesting exercise, which can give a full understanding of the use of soft replicated nodes as the underlying processing platform of distributed systems. For instance, applications developed using the Arjuna system [Shrivastava 89] make the assumption that the underlying nodes are fail–silent. Currently, most of the known applications developed using Arjuna execute on conventional processors that might not provide the fail–silent semantics required. A challenging exercise is therefore to develop Arjuna applications which execute on top of soft fail–silent nodes, like the ones we have implemented.

8.3. Concluding Remarks

We have investigated alternative ways of constructing efficient fail–controlled replicated nodes suitable for the development of a reliable processing platform on top of which dependable distributed systems can be more easily implemented. The fail–controlled nodes presented in this thesis are based solely on the utilisation of 'off–the–shelf' processors (which can fail in an arbitrary way), and software protocols to control system redundancy, without recourse to any specia-

lised hardware. The experiments carried out with implementations of the fail–controlled nodes presented, showed that the overheads associated with the execution of the new redundancy management protocols presented in this thesis are much smaller than the overheads of any previously known protocols. Thus, bearing in mind the discussion on the advantages of software–implemented fail–controlled nodes over hardware–implemented nodes, we can anticipate a range of applications for which these nodes offer an attractive alternative to their hardware implemented counterparts. We summarise the main contributions of this thesis as follows:

- the presentation of a precise definition of the semantics of a soft fail–silent node (Chapter 5);
- the design of efficient protocols for the construction of both soft failure–masking (Chapter 4) and soft fail–silent nodes (Chapter 5);
- the implementation and performance evaluation of software based replicated fail–controlled nodes, which indicates the feasibility of the utilisation of such nodes in a wide range of applications (Chapter 6); and
- the design of soft π–FMS nodes (Chapter 7).

Appendix A
Correctness Proof of Order Protocols

A.1. System Assumptions, Definitions and Notations

In this appendix we prove the correctness of the order protocols presented in Chapter 4. We adopt the style of writing real time values in Greek or italicised upper case Roman letters, and clock time values in italicised lower case Roman letters; the term 'clock' is used to refer to a processor's physical clock.

We start by summarising the main assumptions we have made.

Assumption 1: In a failure–masking node, at least π+1 out of N processors are non–faulty and never fail, where $N = 2\pi+1$. Processors are assigned a unique numbering which is known to all non–faulty processors.

Assumption 2: A non–faulty processor's signature for a given message is unique and cannot be generated by any other processor. Furthermore, any attempt to alter the contents of a non–faulty processor's signed message is detected by any other non–faulty processor.

Assumption 3: Processors within a node are connected to processors of another node through point–to–point connections.

Assumption 4: When a non–faulty processor sends a message to a subset of processors of the node at real time *SENT*, every non–faulty destination receives the mess-

age at real time *RECEIVED*, $SENT \leq RECEIVED < (SENT+\delta)$, where δ, $\delta > 0$, is known.

Assumption 5: A non–faulty processor's clock measures an interval of time x in a real time interval $x(1+\rho_a)$, where $|\rho_a| \leq \rho$ and ρ is a known positive constant.

Each processor P_i within a failure–masking node maintains a *message counter*, denoted by MC_i, whose value is an integer that never decreases and is initialised to 1 when the node is first started. When P_i wants to initiate the broadcast of a message received from the network, it composes an internal message $m = <\mu, TS, O, S>$, where $m.\mu$ is the contents of m, i.e. the message received from the network; $m.TS$ is the time–stamp of m; $m.O$ is the originator of m; and $m.S$ is the list of signatures contained in m. A newly formed and unsigned m has empty $m.S$, whilst for any sent or received m, $m.S$ contains a sequence of one or more processor signatures. We use the notation $|m.S|$ to denote the number of signatures in $m.S$.

The development of the order protocols involves the implementation of: i) *message diffusion* to ensure that non–faulty processors exchange an identical set of messages between them; and ii) *timeliness checks* to enable a non–faulty processor to assess whether a received message is timely, and therefore should be accepted, or if the message is untimely, in which case it should be discarded.

P_i *accepts* a message by entering a copy of it in a message list, called $accepted_i$. For any given message m, we define an $equiv(m)$ as any m' such that $m'.\mu = m.\mu$, $m'.O = m.O$ and $m'.TS = m.TS$. That is, only the contents of $m.S$ and $[equiv(m)].S$ may differ. Note that an $equiv(m)$ can be m itself. In all order protocols presented in Section 4.3, an accepted message m is always diffused to the other processors that have not signed m. Thus, the following lemma is valid for all order protocols of Section 4.3.

Lemma 1 (message diffusion): If a non–faulty P_i accepts m at real time $TIME_i$, then every non–faulty P_k receives an $equiv(m)$ at real time $TIME_k$ such that $|TIME_i - TIME_k| < \delta$.

Proof: Let P_j be the first non–faulty processor to accept m at real time $TIME_j$. It follows that $|m.S| < \pi+1$, and if $|m.S| > 0$, then any processor which has signed m must

be a faulty processor (since P_j is the first non–faulty processor to accept m). P_j then diffuses m to every other processor which has not signed m. From assumption 4, every non–faulty P_k (including P_i), $P_k \neq P_j$, receives $equiv(m)$ at some real time $TIME_k$, with $TIME_j \leq TIME_k < TIME_j+\delta$, hence the lemma. ❑

To implement timeliness checks, each P_i maintains *timing counters*, denoted as $TC_i[j, s]$, for every $P_j, j \neq i$, and for every s, $1 \leq s \leq \pi+1$. These timing counters have integer values which are initialised to zero and never decrease.

We assume the following notations: $START_i(j, s, \geq ts)$ denotes the smallest real time instance when $TC_i[j, s]$ becomes larger than or equal to ts. (That is, just before real time $START_i(j, s, \geq ts)$, $TC_i[j, s]$ is less than ts.) $start_i(j, s, \geq ts)$ denotes P_i's clock time at real time $START_i(j, s, \geq ts)$. Also, $END_i(\leq ts)$ denotes the largest real time instance when MC_i is less than or equal to ts. (That is, just after real time $END_i(\leq ts)$, MC_i is larger than ts and P_i will not form and send any $m, m.TS \leq ts$.) We also assume the notation ρ_i to denote the rate with which the clock of a processor P_i drifts from real time.

Unless stated otherwise, the bounds on ts and s are: $ts \geq 1$ and $1 \leq s \leq \pi+1$. For simplicity, we assume that a non–faulty processor executes the instructions of the protocols in zero real/clock time. Realising this assumption requires an increase in the value of d, which is possible as the proofs impose no upper bound on d.

A.2. Proof of Correctness of the Protocol of Section 4.3.1

Lemma 2.1: $start_i(j, s+1, \geq ts) - start_i(j, s, \geq ts) = 2d$, for non–faulty P_i, any $P_j, j \neq i$, and any s, $1 \leq s \leq \pi$.

Proof: Follows directly from the algorithm of the Update process (Figure 4–7, page 79). ❑

Lemma 2.2: $|start_i(j, s, \geq ts) - start_i(r, s, \geq ts)| \leq d$, for non–faulty P_i, and any P_j and $P_r, j \neq i \neq r$.

Proof: Let us first consider $s = 1$, and abbreviate $start_i(j, 1, \geq ts)$ and $start_i(r, 1, \geq ts)$ as $start(j)$ and $start(r)$, respectively. Let the executions of the Update process at clock

times $start(j)$ and $start(r)$ be scheduled due to P_i accepting messages m_j and m_r, respectively. Note that $m_j.TS \geq ts$ and $m_r.TS \geq ts$.

Suppose that P_i accepts m_j at clock time $time_j$. Accepting m_j would lead to the Update process being scheduled to update the timing counter $TC_i[k = m_j.O, 1]$ at $time_j+d$ and each timing counter $TC_i[k \neq m_j.O, 1]$ at $time_j+2d$. Note that scheduling a timing counter update to a value ts will not be effective only if that timing counter has been updated to or above ts, prior to the scheduled time. So, $start(r) \leq time_j+2d$. As the acceptance of m_j did set $TC_i[j, 1] \geq ts$, either $start(j) = time_j+d$ or $start(j) = time_j+2d$ depending on whether $j = m_j.O$ or not. This means that $start(r) \leq start(j)+d$. Similar arguments can be made with m_r: if $time_r$ is the clock time when P_i accepted m_r, then $start(j) \leq time_r+2d$ and $start(r)$ is either $time_r+d$ or $time_r+2d$; so, $start(j) \leq start(r)+d$. Hence the lemma is true for $s = 1$. From lemma 2.1, this lemma is true for any s, $1 < s \leq \pi+1$. ❑

Lemma 2.3: $|START_i(j, 1, \geq ts)-START_k(j, 1, \geq ts)| < \delta+2d(2\rho)$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$.

Proof: Let us abbreviate $START_i(j, 1, \geq ts)$ and $START_k(j, 1, \geq ts)$ as $START_i$ and $START_k$ respectively. Without loss of generality, we assume that $START_i < START_k$ and prove that $START_k–START_i < \delta+2d(2\rho)$. We prove it by showing that $START_k–START_i \geq \delta+2d(2\rho)$ cannot be true.

Let m_i be the message whose acceptance by P_i at real time, say $TIME_i$, caused P_i to set $TC_i[j, 1] \geq ts$ at $START_i$. From lemma 1, P_k receives an $equiv(m_i)$ before $TIME_i+\delta$. Let m_k denote the first $equiv(m_i)$ that P_k receives. Note that m_k is accepted by P_k, only if it is timely, i.e. only if $TIME_i+\delta \leq START_k(m_k.O, |m_k.S|, \geq m_k.TS)$. We will prove the lemma by showing that if $START_k \geq START_i+\delta+2d(2\rho)$ then P_k must find m_k timely; however, if P_k finds m_k timely then, as we will show, $START_k$ is guaranteed to be less than $START_i+\delta+2d(2\rho)$, which is a contradiction.

From the algorithms of the Broadcast and Diffuse processes (Figure 4–9, page 81; and Figure 4–10, page 81, respectively), the execution of the Update process at $START_i$ is scheduled to be executed at least d units of clock time after P_i has received m_i, thus, $START_i \geq TIME_i + d(1+\rho_i)$. By Lemma 2.2, $START_k(m_k.O, 1, \geq m_k.TS) + d(1+\rho_k) \geq START_k$. If $START_k \geq START_i + \delta + 2d(2\rho)$, then $START_k(m_k.O, 1, \geq m_k.TS) + d(1+\rho_k) \geq START_i + \delta + 2d(2\rho)$. So, we have $START_k(m_k.O, 1, \geq m_k.TS) + d(1+\rho_k) \geq TIME_i + d(1+\rho_i) + \delta + 2d(2\rho)$ and, after some simple algebra, $START_k(m_k.O, 1, \geq m_k.TS) \geq TIME_i + \delta + d(4\rho+\rho_i-\rho_k)$. Since $|\rho_i|$, and $|\rho_k|$ are bounded by ρ (assumption 5), $(4\rho+\rho_i-\rho_k)$ is non–negative and $START_k(m_k.O, 1, \geq m_k.TS) \geq TIME_i + \delta$. Note that since $|m_k.S| \geq 1$, $START_k(m_k.O, |m_k.S|, \geq m_k.TS) \geq START_k(m_k.O, 1, \geq ts) \geq TIME_i + \delta$. Thus, if $START_k \geq START_i + \delta + 2d(2\rho)$, P_k cannot find m_k untimely and must accept m_k.

If P_k accepts m_k then from the algorithm of the Diffuse process (Figure 4–10, page 81), $START_k < TIME_i + \delta + xd(1+\rho_k)$, with $x = 1$ if $m_k.O = j$ or $x = 2$ if $m_k.O \neq j$. Also, $START_i = TIME_i + xd(1+\rho_i)$, with $x = 1$ if $m_i.O = j$ or $x = 2$ if $m_i.O \neq j$. As $m_k.O = m_i.O$, the value of x will be identical in the expressions for both $START_k$ and $START_i$. So, $START_k < START_i - xd(1+\rho_i) + \delta + xd(1+\rho_k)$, which leads to $START_k < START_i + \delta + 2d(2\rho)$. Hence the lemma. ❑

Lemma 2.4: $|START_i(j, s, \geq ts) - START_k(j, s, \geq ts)| < \delta + 2d(2\rho s)$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$.

Proof: Follows from lemmas 2.3 and 2.1, and from assumption 5. ❑

Remark: The above lemmas are true for any randomly chosen, non–negative value of d. (Their proofs do not require any lower or upper bound on d.) The next lemma establishes the relation between the value of d and δ, as a function of ρ and π.

Lemma 2.5: $START_i(j, s+1, \geq ts) - START_k(j, s, \geq ts) > \delta$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$, and any s, $1 \leq s \leq \pi$, provided $d \geq \delta/(1-(2\pi+1)\rho)$.

Proof: Let us abbreviate $START_i(j, s+1, \geq ts)$, $START_i(j, s, \geq ts)$ and $START_k(j, s, \geq ts)$ as $START_i(s+1)$, $START_i(s)$ and $START_k(s)$, respectively. From lemma 2.1, we have

$START_i(s) = START_i(s+1) - 2d(1+\rho_i)$. From lemma 2.4, we have $START_i(s) > START_k(s) - (\delta + 2d(s)(2\rho))$. Combining the two expressions we have $START_i(s+1) > START_k(s) - (\delta + 2d(s)(2\rho)) + 2d(1+\rho_i)$. The lemma will be true, if $2d(1+\rho_i) - (\delta + 2d(s)(2\rho)) \geq \delta$, i.e. $d \geq \delta/(1+\rho_i - 2s\rho)$. As $|\rho_i| \leq \rho$ (assumption 5), and letting $s = \pi$, this implies $d \geq \delta/(1-(2\pi+1)\rho)$, hence the lemma. ❑

This lemma ensures that if a non–faulty P_i receives and accepts m, $|m.S| < \pi+1$, then its diffused message is found timely when being received by another non–faulty P_k. If $|m.S| = \pi+1$, then one of the signers must be non–faulty, and the same guarantee applies. So, we state the following corollary.

Corollary 2.1: For any two non–faulty P_i and P_k, and any m, $i \neq m.O \neq k$, if m enters $accepted_i$ at $TIME_i$, then an $equiv(m)$ enters $accepted_k$ at $TIME_k$ such that $|TIME_i - TIME_k| < \delta$, provided $d \geq \delta/(1-(2\pi+1)\rho)$.

Lemma 2.6: $START_i(k, 1, \geq ts) - END_k(\leq ts) > \delta$, for non–faulty P_i and P_k, provided $d > \delta/(1-\rho)$.

Proof: Let the execution of the Update process at $START_i(k, 1, \geq ts)$ be due to P_i accepting m_i, $m_i.TS \geq ts$, at real time $TIME_i$. Regarding $m_i.O$ we consider two cases:

(i) $m_i.O = k$; from the algorithm of the Broadcast process (Figure 4–9, page 81), just before sending m_i, $m_i.TS \geq ts$, P_k sets MC_k to $m_i.TS+1$. Therefore, $END_k(\leq ts) < TIME_i$. Also, $START_i(k, 1, \geq ts) = TIME_i + d(1+\rho_i)$. Thus, $START_i(k, 1, \geq ts) > END_k(\leq ts) + d(1+\rho_i)$; since $d > \delta/(1-\rho)$ implies $d(1+\rho_i) > \delta$, we have $START_i(k, 1, \geq ts) - END_k(\leq ts) > \delta$.

(ii) $m_i.O \neq k$; from lemma 1, P_k receives an $equiv(m_i)$, say m_k, at real time $TIME_k < TIME_i + \delta$. If $TIME_k \geq START_k(m_k.O, |m_k.S|, \geq m_k.TS)$ then from the algorithms of the Broadcast and Diffuse processes (Figure 4–9, page 81; and Figure 4–10, page 81, respectively), it is easy to see that $TIME_k > END_k(\leq ts)$; if, on the other hand, $TIME_k < START_k(m_k.O, |m_k.S|, \geq ts)$ then either m_k is accepted at $TIME_k$ or another $equiv(m_i)$ has been accepted before $TIME_k$. Thus, $TIME_k \geq END_k(\leq ts)$. In both cases

we have $END_k(\leq ts) < TIME_i+\delta$. When $m_i.O \neq k$, from the algorithm Diffuse process (Figure 4–10, page 81) we have $START_i(k, 1, \geq ts) = TIME_i+2d(1+\rho_i)$. Therefore, $START_i(k, 1, \geq ts) > END_k(\leq ts)-\delta+2d(1+\rho_i)$, and since $d > \delta/(1-\rho)$ implies $2d(1+\rho_i) > 2\delta$, $START_i(k, 1, \geq ts)-END_k(\leq ts) > \delta$. Hence the lemma. ❑

This lemma ensures that if a non–faulty P_k forms and sends a message m, every non–faulty P_i accepts m upon reception. So, the corollary 2.1 is strengthened as below.

Corollary 2.2: For any two non–faulty P_i and P_k, and any m, if m enters $accepted_i$ at $TIME_i$, then an $equiv(m)$ enters $accepted_k$ at $TIME_k$ such that $|TIME_i-TIME_k| < \delta$, provided $d \geq \delta/(1-(2\pi+1)\rho)$.

Lemma 2.7: A message m enters $stable_i$ of non–faulty P_i within at most $2d(\pi+1)$ time after having entered $accepted_i$, where time is measured according to P_i's clock.

Proof: At any given instance of real time, $TC_{i,min} \leq TC_i[j, s]$ for any $j, j \neq i$, and any s, $1 \leq s \leq \pi+1$. Therefore, no m can enter $accepted_i$ after $m.TS$ has become larger than or equal to $TC_{i,min}$. From the algorithms of the Broadcast and Diffuse processes (Figure 4–9, page 81; and Figure 4–10, page 81, respectively), it can be seen that all timing counters of P_i become larger than or equal to $m.TS$, within $2d(\pi+1)$ time after m is accepted, where time is measured according to P_i's clock. Hence the lemma follows from the algorithm of the Deliver process (Figure 4–11, page 82) which removes every m from $accepted_i$ into $stable_i$ immediately after $m.TS$ has become larger than or equal to $TC_{i,min}$. ❑

Lemma 2.8: Any two non–spurious m_1 and m_2 that enter $accepted_i$ of non–faulty P_i are delivered according to the ordering relation «.

Proof: Say $m_1 \ll m_2$, thus $m_1.TS \leq m_2.TS$. Note that $TC_{i,min}$ never decreases with the passage of real time, and also that no message m enters $stable_i$ after $TC_{i,min}$ becomes larger than or equal to $m.TS$ (see the algorithm of the Deliver process, Figure 4–11, page 82). Therefore, it is impossible for m_1 to enter $stable_i$ after m_2 has entered. Hence the Lemma. ❑

Theorem 2.1: The protocol guarantees *agreement* and *order* conditions within a real time interval Σ_a, $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$, if $d \geq \delta/(1-(2\pi+1)\rho)$.

Proof: From corollary 2.2, any m formed and sent by a non–faulty P_i enters $accepted_i$, and will enter $accepted_j$ of every non–faulty P_j within δ time; also, for a given ts, and for every m, $m.TS = ts$, that enters $accepted_i$, there is an $equiv(m)$ that enters $accepted_j$ of non–faulty P_j. Any m that enters $accepted_i$, eventually enters $stable_i$ (Lemma 2.7) together with all m' such that $m'.TS = m.TS$ (see the algorithm of the Deliver process, Figure 4–11, page 82). Therefore, P_i will find m spurious if and only if P_j finds $equiv(m)$ spurious. From lemma 2.8, non–spurious entries of the message list *stable* are ordered according to « relation which is identical for both P_i and P_j since processor ordering is unique and known (assumption 1). This means that for every given ts, non–faulty processors order an identical set of m, $m.TS = ts$, in an identical manner and after ordering all m', $m'.TS < ts$. Thus the protocol meets atomicity and order requirements.

A message m formed and sent by a non–faulty processor cannot be found spurious in any non–faulty processor (see assumption 2). From lemma 2.7, we can state that any non–spurious m is ordered by a non–faulty processor within $2d(\pi+1)(1+\rho)$ real time after being accepted. Thus the protocol meets the termination requirement with $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$. Hence the theorem. ❑

Theorem 2.2: The protocol can guarantee ordering agreement in finite time only if $\pi < (1-\rho)/(2\rho)$.

Proof: For Σ_a to be finite and positive, d, $d \geq \delta/(1-(2\pi+1)\rho)$, must be finite and positive. Hence, $1-(2\pi+1)\rho > 0$, i.e. $\pi < (1-\rho)/(2\rho)$. ❑

A.3. Proof of Correctness of the Protocol of Section 4.3.2

Note that for a TMR node s is either 1 or 2, and the changes made to derive the TMR protocol do not modify the way the timing counters for $s = 1$ and the message counter (MC) are being

updated. Thus, lemma 2.6–2.8 also hold for this protocol. (Note that the proofs of lemma 2.6–2.8 do not require lemmas 2.1–2.5.)

Lemma 3.1: $START_k(j, 2, \geq ts)$–$START_i(j, 1, \geq ts) > \delta$, for non–faulty P_i and P_k, and any P_j, $i \neq j \neq k$, provided $d \geq \delta/(1-5\rho)$.

Proof: Let us abbreviate $START_k(j, 2, \geq ts)$, and $START_i(j, 1, \geq ts)$ as $START_k(2)$, and $START_i(1)$, respectively. Let m be the message whose acceptance by P_k scheduled the execution of the Update process at $START_k(2)$. Regarding $m.O$, we consider each of the three processors in the TMR node:

(i) $m.O = k$; From the algorithm of the Broadcast process (Figure 4–14, page 89) P_k must have formed and sent m just before real time $START_k(2)$–$4d(1+\rho_k)$. From lemma 1, P_i must have received $equiv(m)$ before $START_k(2)$–$4d(1+\rho_k)+\delta$. From lemma 2.6, P_i accepts $equiv(m)$, and from the algorithm of the Diffuse process (Figure 4–15, page 89) we have $START_i(1) < START_k(2)$–$4d(1+\rho_k)+\delta+2d(1+\rho_i)$, leading to $START_k(2)$–$START_i(1) > \delta$ whenever $d \geq \delta/(1+2\rho_k-\rho_i)$, which is always true since $d \geq \delta/(1-5\rho) > \delta/(1+2\rho_k-\rho_i)$.

(ii) $m.O = i$; Let P_i form and send m at real time $TIME_i$. From lemma 2.6, and the algorithm of the Diffuse process (Figure 4–15, page 89), P_k receives and accepts $equiv(m)$ at some real time $TIME_k$, $TIME_i \leq TIME_k < START_k(2)$–$3d(1+\rho_k)$. For P_i, $START_i(1) \leq TIME_i+2d(1+\rho_i)$, thus $START_k(2)$–$3d(1+\rho_k) > START_i(1)$–$2d(1+\rho_i)$, and $START_k(2)$–$START_i(1) > \delta$ whenever $d \geq \delta/(1+3\rho_k-2\rho_i)$, which is always true since $d \geq \delta/(1-5\rho) \geq \delta/(1+3\rho_k-2\rho_i)$.

(iii) $m.O = j$, $k \neq j \neq i$; as per the algorithm of the Diffuse process (Figure 4–15, page 89), P_k must have received m just before real time $START_k(2)$–$3d(1+\rho_k)$. Let P_i receive $equiv(m)$ at real time $TIME_i$; from lemma 1, $TIME_i < START_k(2)$–$3d(1+\rho_k)+\delta$. If $TIME_i \geq START_i(1)$, then for any non–negative value of d we have $START_k(2)$–$START_i(1) > \delta$. Now consider $TIME_i < START_i(1)$, thus $equiv(m)$ is accepted by P_i. From the algorithm of the Diffuse process (Figure 4–15, page 89), $START_i(1) \leq TIME_i+d(1+\rho_i)$. Thus, $START_k(2)$–$START_i(1) > 3d(1+\rho_k)$–$\delta$–$d(1+\rho_i)$,

and $START_k(2)$–$START_i(1) \geq \delta$ whenever $d \geq \delta/(1+3\rho_k/2-\rho_i/2)$, which is always true since $d \geq \delta/(1-5\rho) > \delta/(1+3\rho_k/2-\rho_i/2)$. Hence the lemma. ❑

This lemma ensures that if a non–faulty P_i receives and accepts a single–signed m, then its diffused double–signed message is found timely when being received by another non–faulty P_k. If a non–faulty P_i receives and accepts a double–signed m, then any other non–faulty processor of the node has already accepted an $equiv(m)$. Lemma 2.6 guarantees that a message m accepted by a non–faulty P_i when P_i formed and sent m is also accepted by all other non–faulty processors. So, we state the following corollary.

Corollary 3.1: For any two non–faulty P_i and P_k, and any m, if m enters $accepted_i$ at $TIME_i$, then an $equiv(m)$ enters $accepted_k$ at $TIME_k$ such that $|TIME_i–TIME_k| < \delta$, provided $d \geq \delta/(1-5\rho)$.

Lemma 3.2: A message m, $m.O = i$, enters $stable_i$ of non–faulty P_i within $4d$ time after having entered $accepted_i$, whilst a message m', $m'.O \neq i$, enters $stable_i$ of non–faulty P_i within $3d$ time after having entered $accepted_i$, where time is measured according to P_i's clock.

Proof: At any given instance of real time, $TC_{i,min} \leq TC_i[j, s]$ for any $j, j \neq i$, and any s, $1 \leq s \leq 2$. Therefore, no m can enter $accepted_i$ after $m.TS$ has become larger than or equal to $TC_{i,min}$. For m, $m.O = i$, from the algorithm of the Broadcast process (Figure 4–14, page 89), it can be seen that all timing counters of P_i become larger than or equal to $m.TS$ within $4d$ time after m is accepted, where time is measured according to P_i's clock. Similarly, for m, $m.O = i$, from the algorithm of the Diffuse process (Figure 4–15, page 89), it can be seen that all timing counters of P_i become larger than or equal to $m.TS$ within $3d$ time after m is accepted. Hence the lemma follows form the algorithm of the Deliver process (Figure 4–11, page 82) which removes every m from $accepted_i$ into $stable_i$ immediately after $m.TS$ has become larger than or equal to $TC_{i,min}$. ❑

Theorem 3.1: The protocol for TMR nodes guarantees *agreement* and *order* conditions within a real time interval Σ_a, $\Sigma_a \leq 4d(1+\rho)$, if $d \geq \delta/(1-5\rho)$.

Proof: From corollary 3.1, any m formed and sent by a non–faulty P_i enters $accepted_i$, and will enter $accepted_j$ of every non–faulty P_j within δ time; also, for a given ts, and for every m, $m.TS = ts$, that enters $accepted_i$, there is an $equiv(m)$ that enters $accepted_j$ of non–faulty P_j. Since the Deliver process for this protocol is the equal to the Deliver process of the previous protocol, any m that enters $accepted_i$, eventually enters $stable_i$ (Lemma 3.2) together with all m' such that $m'.TS = m.TS$ (see the algorithm of the Deliver process, Figure 4–11, page 82). Therefore, P_i will find m spurious if and only if P_j finds $equiv(m)$ spurious. From lemma 2.8, non–spurious entries of the message list *stable* are ordered according to « relation which is identical for both P_i and P_j since processor ordering is unique and known (assumption 1). This means that for every given ts, non–faulty processors order an identical set of m, $m.TS = ts$, in an identical manner and after ordering all m', $m'.TS < ts$. Thus the protocol meets atomicity and order requirements.

A message m formed and sent by a non–faulty processor cannot be found spurious in any non–faulty processor (see assumption 2). From lemma 3.3, any non–spurious m, $m.O = i$, in $accepted_i$ enters $stable_i$ within at most $4d(1+\rho)$ (real) time after having entered $accepted_i$; also, any non–spurious m', $m'.O \neq i$, in $accepted_i$ enters $stable_i$ within at most $3d(1+\rho)$ (real) time after having entered $accepted_i$, and may take at most δ time to be received by P_i. As $d \geq \delta/(1-5\rho)$, we have $d(1+\rho) > \delta$, and the protocol meets the termination requirement with $\Sigma_a \leq 4d(1+\rho)$. Hence the theorem. ❑

A.4. Proof of Correctness of the protocol of Section 4.3.3

We define a *path* p to be an ordered, non–empty sequence of at most $\pi+1$ distinct processors. We also define the following notations:

- $path(m)$, which denotes the path through which a message m was received;
- $|p|$, which denotes the length of the path p (i.e. the number of processors in p); and

- *origin*(p), which denotes the first processor in the path p.

Further, we define the following set of paths:

- $paths_i = \{p \mid P_i \text{ not in } p\}$.

In addition to maintaining timing counters $TC_i[j, s]$, for every P_j, $j \neq i$, and for every s, $1 \leq s \leq \pi+1$, each non–faulty P_i also maintains *path counters* $PC_i[p]$ for each path p through which a message can be received (i.e. $\forall p \mid p \in paths_i$). These path counters are integer values which only increase with the passage of real time, and are initialised to zero when the node is started.

Comparing the algorithms of the Broadcast and Diffuse processes of the protocol of Section 4.3.1 (Figure 4–9, page 81; and Figure 4–10, page 81, respectively) and the algorithms of the Broadcast and Diffuse processes of the early–order protocol (Figure 4–17, page 97; and Figure 4–18, page 98, respectively) it is easy to see that for each execution of *Update*(j, 1, $m.TS$) at clock time t in the protocol of Section 4.3.1, there is a corresponding execution of *Update*(j, 1, $m.TS$) at clock time t in the early–order protocol. From the algorithms of the Update process of the protocol of Section 4.3.1 (Figure 4–7, page 79) and the Update process of the early–order protocol (Figure 4–16, page 97), both versions of Update will schedule executions of *Update*(j, s+1, $m.TS$) $2d$ units of clock time after having executed *Update*(j, s, $m.TS$), $1 \leq s \leq \pi$. Thus, the timing counters $TC_i[j, s]$, for every P_j, $j \neq i$, and for every s, $1 \leq s \leq \pi+1$, will be update in the early–order protocol in the same way as they are updated in the protocol of Section 4.3.1. Further, the message counter MC_i is also updated in an equivalent way in both protocols. Hence lemmas 2.1–2.6 also hold for the early–order protocol.

We maintain the notations $START_i(j, s, \geq ts)$ and $END_i(\leq ts)$ as defined previously, and we introduce the following notation: $START_i(p, \geq ts)$, for a non–faulty P_i, and all paths $p \in paths_i$, and $ts \geq 1$, denotes the smallest real time instance when $PC_i[p]$ becomes larger than or equal to ts. (That is, just before real time $START_i(p, \geq ts)$, $PC_i[p]$ is less than ts.)

Lemma 4.1: $START_i(p, \geq ts) \leq START_i(origin(p), |p|, \geq ts)$, for any non–faulty processor P_i and any path p, $p \in paths_i$. If $START_i(p, \geq ts) < START_i(origin(p), |p|, \geq ts)$,

then P_i must have received a timely message m at some real time $TIME_i$, with $m.TS \geq ts$, $path(m) = p$ and $TIME_i < START_i(p, \geq ts) < START_i(origin(p), |p|, \geq ts)$.

Proof: $START_i(p, \geq ts) \leq START_i(origin(p), |p|, \geq ts)$ follows directly from the algorithm of the early–order Update process (Figure 4–16, page 97).

If $START_i(p, \geq ts) < START_i(origin(p), |p|, \geq ts)$, then the path counter $C_i[p]$ must have been explicitly updated to a value greater or equal to ts by the Diffuse process. From the algorithm of the early–order Diffuse process (Figure 4–18, page 98) P_i must receive m at $TIME_i$, with $m.TS \geq ts$, $path(m) = p$ and $TIME_i < START_i(p, \geq ts)$. Hence the lemma. ❑

Lemma 4.2: If m_1 and m_2 are two distinct messages sent by a non–faulty P_i, with $path(m_1) = path(m_2)$, then they were sent in the increasing order of their timestamps.

Proof: Assume that $|path(m_1)| = |path(m_2)| = 1$. A non–faulty processor forms and sends messages in the increased order of timestamps. Hence the lemma is true when $|path(m_1)| = |path(m_2)| = 1$.

Say $|path(m_1)| = |path(m_2)| > 1$. Let m_1' and m_2' denote the messages which P_i signed and sent as m_1 and m_2, respectively. So, $path(m_1'):P_i = path(m_1)$ and $path(m_2'):P_i = path(m_2)$. We will assume that m_1 was sent before m_2 and prove that $m_1.TS < m_2.TS$.

Since P_i signed and sent m_1', it must have found m_1' timely. So, upon receiving m_1', it will set $PC_i[path(m_1')] \geq m_1'.TS$. If $m_2'.TS \leq m_1'.TS$, P_i will not find m_2' timely and will not send m_2. In other words, if m_2 was sent, then $m_2.TS > m_1.TS$. ❑

Corollary 4.1: If m_1 and m_2 are distinct messages sent by a non–faulty P_i and received by another non–faulty P_k, then P_k will receive them in the increasing order of their timestamps.

Proof: Follows from Lemma 4.1 and the *fifo* assumption. ❑

Lemma 4.3: Any message m whose broadcast is initiated by a non–faulty P_i will be found timely by every non–faulty P_k, provided $d > \delta/(1-\rho)$.

Proof: If $START_k(P_i, \geq m.TS) = START_k(i, 1, \geq m.TS)$, then the lemma follows from lemma 2.6.

If $START_k(P_i, \geq m.TS) < START_k(i, 1, \geq m.TS)$, then from lemma 4.1 P_k must have received a timely message m' at some time $TIME_k'$, with $m'.TS \geq m.TS$, $path(m') = P_i$ and $TIME_k' < START_k(P_i, \geq m.TS)$. From corollary 4.1, m must have been received at some time $TIME_k$, $TIME_k \leq TIME_k' < START_k(P_i, \geq m.TS)$. Thus, P_k finds m timely, hence the lemma. ❑

Lemma 4.4: Any message m received by a non–faulty P_i from a non–faulty P_k, will be found timely, provided $d \geq \delta/(1-(2\pi+1)\rho)$, and $\pi < (1-\rho)/2\rho$.

Proof: If $START_k(path(m), \geq m.TS) = START_k(m.O, |m.S|, \geq m.TS)$, then the lemma follows from lemma 2.5.

If $START_k(path(m), \geq m.TS) < START_k(m.O, |m.S|, \geq m.TS)$, then from lemma 4.1 P_k must have received a timely message m' at some time $TIME_k'$, with $m'.TS \geq m.TS$, $path(m') = path(m)$ and $TIME_k' < START_k(path(m), \geq m.TS)$. From corollary 4.1, m must have been received at some time $TIME_k$, $TIME_k \leq TIME_k'$. Since $TIME_k < START_k(path(m), \geq m.TS)$, P_k finds m timely. Hence the lemma. ❑

Lemma 4.5: If all processors of the node are non–faulty, then the protocol's actual stability delay is given by $0 \leq \Sigma_a < (\pi+2)\delta$.

Proof: Suppose P_i initiates the broadcast of a message m at real time T. Since all processors are non–faulty, every other P_j receives a copy of m at latest by real time $T+\delta$, and initiates a null–broadcast m_j. At latest by real time $T+\delta+(\pi+1)\delta$ all messages associated with the broadcast of m initiated by P_i, and with the broadcast of each null–broadcast m_j initiated by every other P_j will have been received by every processor within the node. Thus, at latest by this time P_i will have updated $PC_{i,min}$ to $m.TS$, and every other P_j will also have updated $PC_{j,min}$ to $m.TS$. Therefore, at latest by

$\Sigma_a < (\pi+2)\delta$, all processors will have ordered m. The lower bound is attained when messages are exchanged in zero time, hence the lemma. ❑

Lemma 4.6: A message m enters $stable_i$ of non–faulty P_i within at most $2d(\pi+1)$ time after having entered $accepted_i$, where time is measured according to P_i's clock.

Proof: At any given instance of real time, $PC_{i,min} \leq PC_i[p]$ for any p, $p \in paths_i$. Therefore, no m can enter $accepted_i$ after $m.TS$ has become larger than or equal to $PC_{i,min}$. From the algorithms of the Broadcast and Diffuse processes (Figure 4–17, page 97, and Figure 4–18, page 98, respectively), it can be seen that all path counters of P_i become larger than or equal to $m.TS$, within $2d(\pi+1)$ time after m is accepted, where time is measured according to P_i's clock. Hence the lemma follows from the algorithm of the Deliver process (Figure 4–19, page 98) which removes every m from $accepted_i$ into $stable_i$ immediately after $m.TS$ has become larger than or equal to $PC_{i,min}$. ❑

Lemma 4.7: Any two non–spurious m_1 and m_2 that enter $accepted_i$ of non–faulty P_i is delivered according to the ordering relation «.

Proof: Say $m_1 \ll m_2$, thus $m_1.TS \leq m_2.TS$. Note that $PC_{i,min}$ never decreases with the passage of real time, and also that no message m enters $stable_i$ after $PC_{i,min}$ becomes larger than or equal to $m.TS$ (see the algorithm of the Deliver process, Figure 4–19, page 98). Therefore, it is impossible for m_1 to enter $stable_i$ after m_2 has entered. Hence the Lemma. ❑

Theorem 4.1: The early–order protocol guarantees *agreement* and *order* conditions, within a real time interval Σ_a, $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$, so long as $d \geq \delta/(1-(2\pi+1)\rho)$, and $\pi < (1-\rho)/2\rho$. In the absence of failures the protocol guarantees early–order of messages in Σ_{eo}, $0 \leq \Sigma_{eo} < (\pi+2)\delta$.

Proof: From lemmas 4.3 and 4.4, any m sent by a non–faulty P_i to a non–faulty P_k will be found timely by P_k; therefore, for a given ts, every m, $m.TS = ts$, that enters $accepted_i$, there is an $equiv(m)$ that enters $accepted_j$ of non–faulty P_j. Any m that enters

$accepted_i$, eventually enters $stable_i$ (Lemma 4.6) together with all m' such that $m'.TS = m.TS$ (see the algorithm of the Deliver process, Figure 4–19, page 98). Therefore, P_i will find m spurious if and only if P_j finds $equiv(m)$ spurious. From lemma 4.7, non–spurious entries of the message list *stable* are ordered according to « relation which is identical for both P_i and P_j since processor ordering is unique and known (assumption 1). This means that for every given ts, non–faulty processors order an identical set of m, $m.TS = ts$, in an identical manner and after ordering all m', $m'.TS < ts$. Thus the protocol meets atomicity and order requirements.

A message m formed and sent by a non–faulty processor cannot be found spurious in any non–faulty processor (see assumption 2). From lemma 4.6, we can state that any non–spurious m is ordered by a non–faulty processor within $2d(\pi+1)(1+\rho)$ real time after being accepted. Thus the protocol meets the termination requirement with $\Sigma_a \leq 2d(\pi+1)(1+\rho)+\delta$. Early order in Σ_{eo}, $0 \leq \Sigma_{eo} < (\pi+2)\delta$, follows from lemma 4.5. Hence the theorem. ❑

References

[Accetta et al. 86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," Proceedings of the Summer 1986 USENIX Conference, Atlanta, USA, pp. 93–112, July 1986.

[Avizienis 78] A. Avizienis, "Fault–Tolerance: The Survival Attribute of Digital Systems," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1109–1125, October 1978.

[Avizienis 85] A. Avizienis, "The N–Version Approach to Fault–Tolerant Software," IEEE Transactions on Software Engineering, Vol. SE–11, No. 12, pp. 1491–1501, December 1985.

[Babaoglu–Drummond 85] O. Babaoglu, and R. Drummond, "Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts," IEEE Transactions on Software Engineering, Vol. 11, No. 6, pp. 546–554, June 1985.

[Babaoglu–Drummond 87] O. Babaoglu, and R. Drummond, "(Almost) No cost Clock Synchronisation," Digest of Papers, FTCS–17, Pittsburgh, USA, pp.42–47, July 1987.

[Barborak–Malek 93] M. Barborak, and M. Malek, "The Consensus Problem in Fault–Tolerant Computing," ACM Computing Surveys, Vol. 15, No. 2, pp. 171–220, June 1993.

[Barrett et al. 90] P.A. Barrett, A.M. Hilborne, P. Veríssimo, L. Rodrigues, P.G. Bond, D.T. Seaton, and N.A. Speirs, "The Delta–4 Extra Performance Architecture (XPA)," Digest of Papers, FTCS–20, Newcastle upon Tyne, UK, pp. 481–488, June 1990.

[Bartlett 81] J.F. Bartlett, "A NonStop Kernel," ACM 8th Symposium on Operating Systems Principles, Pacific Grove, USA, Vol. 15, No. 5, pp. 22–29, December 1981.

[Bernstein 88] P.A. Bernstein, "Sequoia: A Fault–Tolerant Tightly Coupled Multiprocessor for Transaction Processing," IEEE Computer, Vol. 21, No. 2, pp. 37–45, February 1988.

[Birman et al. 91] K.P. Birman, A. Shiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 272–314, August 1991.

[Birrell–Nelson 84] A.D. Birrell, and B.J. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, Vol. 2, pp. 39–59, February 1984.

[Brasileiro et al. 92] F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs, and S. Tao, "Efficient Protocols for Fail–Silent Nodes in Distributed Systems," Technical Report No. 413, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, December 1992.

[Brière–Traverse 93] D. Brière, and P. Traverse, "AIRBUS A320/A330/A340 Electrical Flight Controls – A Family of Fault–Tolerant Systems," Digest of Papers, FTCS–23, Toulouse, France, pp. 616–623, June 1993.

[Chérèque et al. 92] M. Chérèque, D. Powell, P. Reynier, J–L. Richier, and J. Voiron, "Active Replication in Delta–4," Digest of Papers, FTCS–22, Boston, USA, pp. 28–37, June 1992.

[Cristian et al. 85] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement," Digest of Papers, FTCS–15, Ann Arbor, USA, pp. 200–206, June 1985.

[Cristian 91] F. Cristian, "Understanding Fault–Tolerant Distributed Systems," Communications of the ACM, Vol. 34, No. 2, pp. 57–78, February 1991.

[Dolev 82] D. Dolev, "The Byzantine Generals Strike Again," Journal of Algorithms, Vol. 3, No. 1, pp. 14–30, January 1982.

[Dolev–Strong 82] D. Dolev, and H.R. Strong, "Requirements for Agreement in a Distributed System," Proceedings of the 2nd Symposium on Distributed Databases, West Berlin, Germany, pp. 115–129, September 1982.

[Dolev–Strong 83] D. Dolev, and H.R. Strong, "Authenticated Algorithms for Byzantine Agreement," SIAM Journal of Computing, Vol. 12, No. 4, pp. 656–666, November 1983.

[Dolev et al. 84] D. Dolev, J. Halpern, and H.R. Strong, "On the Possibility and Impossibility of Achieving Clock Synchronization," Proceedings of the 16th ACM STOC, Washington, D.C., USA, pp. 504–511, May 1984.

[Dolev et al. 90] D. Dolev, R. Reischuk, and H.R. Strong, "Early Stopping in Byzantine Agreement," Journal of the ACM, Vol. 37, No. 4, pp. 720–741, October 1990.

[Ezhilchelvan–Shrivastava 91] P.D. Ezhilchelvan, and S.K. Shrivastava, "A Distributed Systems Architecture Supporting High Availability and Reliability," Proceedings of the 2nd IFIP Conference on Dependable Computing for Critical Applications, Arizona, USA, pp. 36–48, February 1991.

[Fischer–Lynch 82] M.J. Fischer, and N.A. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency," Information Processing Letters, Vol. 14, No. 4, pp. 183–186, June 1982.

[Fischer et al. 85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," Journal of the ACM, Vol. 32, No. 2, pp. 374–382, April 1985.

[FTCS 71–94] Digest of the Annual International Symposium on Fault–Tolerant Computing, FTCS 1–24, 1971–1994, IEEE Computer Society.

[Gopal et al. 90] A. Gopal, R. Strong, S. Toueg, and F. Cristian, "Early–Delivery Atomic Broadcast," Proceedings of the 9th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing, Québec City, Canada, pp. 297–309, August 1990.

[Halpern et al. 84] J.Y. Halpern, B. Simons, H.R. Strong, and D. Dolev, "Fault Tolerant Clock Synchronization," Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, pp. 89–102, August 1984.

[Harper et al. 88] R.E. Harper, J.H. Lala, and J.J. Deyst, "Fault Tolerant Processor Architecture Overview," Digest of Papers, FTCS–18, Tokyo, Japan, pp. 252–257, June 1988.

[Hopkins et al. 78] A.L. Hopkins, T.B. Smith, and J.H. Lala, "FTMP – A Highly Reliable Fault–Tolerant Multiprocessor for Aircraft," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1221–1239, October 1978.

[Ihara et al. 78] K. Ihara et al., "Fault–Tolerant Computer System with Three Symmetric Computers," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1160–1177, October 1978.

[INMOS 88] INMOS Limited, Transputer Instruction Set, Prentice Hall International (UK) Ltd, 1988, ISBN 0–13–929100–8.

[Katsuki et al. 78] D. Katsuki et al., "Pluribus – An Operational Fault–Tolerant Multiprocessor," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1146–1159, October 1978.

[Kieckhafer et al. 88] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance," IEEE Transactions on Computers, Vol. 37, No. 4, pp. 398–405, April 1988.

[Kopetz–Merker 85] H. Kopetz, and W. Merker., "The Architecture of MARS," Digest of Papers, FTCS–15, Ann Arbor, USA, pp. 274–279, June 1985.

[Kopetz et al. 90] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating Transient Faults in MARS," Digest of Papers, FTCS–20, Newcastle upon Tyne, UK, pp. 466–473, June 1990.

[Krol–van Gils 85] Th. Krol, and W.J. van Gils, "The Input/Output Architecture of the (4,2) Concept Fault–Tolerant Computer", Digest of Papers, FTCS–15, Ann Arbor, USA, pp. 254–259, June 1985.

[Lala 86] J.H. Lala, "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications," Digest of papers, FTCS–16, Vienna, Austria, pp. 338–343, July 1986.

[Lala–Alger 88] J.H. Lala, and L.S. Alger, "Hardware and Software Fault Tolerance: A Unified Architectural Approach," Digest of Papers, FTCS–18, Tokyo, Japan, pp. 240–245, June 1988.

[Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21, No. 7, pp. 558–565, July 1978.

[Lamport et al. 82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine General Problem," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 382–401, July 1982.

[Lamport–MelliarSmith 85] L. Lamport, and P.M. Melliar–Smith, "Synchronising Clocks in the Presence of Faults," Journal of the ACM, Vol. 32, No. 1, pp. 52–78, January 1985.

[Lampson 81] B.W. Lampson, "Atomic Transactions," Chapter 11 in Distributed Systems – Architecture and Implementation, B.W. Lampson, M. Paul, and H.J. Siegert (Eds.), Spring–Verlag, pp. 246–264, 1981, ISBN 3–540–10571–9.

[Laprie 89] J–C. Laprie, "Dependability: a Unifying Concept for Reliable Computing and Fault Tolerance," in Dependability of Resilient Computers, Anderson, T. (Ed.), BSP Professional Books, 1989, ISBN 0–632–02054–7.

[Larman 83] B.T. Larman, "The Project Galileo Fault Protection System," Digest of Papers, FTCS–13, Milan, Italy, pp. 460–466, June 1983.

[Lee–Anderson 90] P.A. Lee, and T. Anderson, Fault–Tolerance Principles and Practice, Spring–Verlag, 1990, ISBN 3–211–82077–9.

[Lundelius–Lynch 84] J. Lundelius, and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," Information and Control, Vol. 62, No. 2/3, pp. 190–204, August/September 1984.

[Morgan 93] K.D. Morgan, "The HP–RT Real–Time Operating System," Hewlett–Packard Journal, Vol. 44, No. 4, pp. 23–30, August 1993.

[Mullender et al. 90] S.J. Mullender, G. van Rossum, A.S. Tanembaum, R. van Renesse, and H. van Staveren, "Amoeba: A Distributed Operating System for the 1990's," IEEE Computer, Vol. 23, No. 5, pp. 44–53, May 1990.

[Okamoto 88] T. Okamoto, "A Digital Multisignature Scheme Using Bijective Public–Key Cryptosystems," ACM Transactions on Computer Systems, Vol. 6, No. 8, pp. 432–441, November 1988.

[Palumbo–Butler 85] D.L. Palumbo, and R.W. Butler, "Measurements of SIFT Operating System Overhead," NASA Tech. Memo. 86322, 1985.

[Pease et al. 80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," Journal of the ACM, Vol. 27, No. 2, pp. 228–234, April 1980.

[Perihelion 91] Perihelion Software Limited, The helios Parallel Operating System, Prentice Hall, 1991, ISBN 0–13–381237–5.

[Powell 92] D. Powell (Ed.), Delta–4 – A Generic Architecture for Dependable Distributed Computing, Spring–Verlag, 1992, ISBN 3–540–54985–4.

[Powell et al. 88] D. Powell, P. Veríssimo, G. Bonn, F. Waeselynck, D. Seaton, "The Delta–4 Approach to Dependability in Open Distributed Computing Systems," Digest of Papers, FTCS–18, Tokyo, Japan, pp. 246–251, June 1988.

[Randell 75] B. Randell, "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, Vol. SE–1, No. 2, pp. 220–232, June 1975.

[Reisinger–Steininger 93] J. Reisinger, and A. Steininger, "The Design of a Fail–Silent Processing Node for the Predictable Hard Real–Time System MARS," IEE Distributed System Engineering, Vol. 1, No. 2, pp. 104–111, December 1993.

[Rennels 78] D.A. Rennels, "Architectures for Fault–Tolerant Spacecraft Computers," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1255–1268, October 1978.

[Rivest et al. 78] R. Rivest, A. Shamir and L. Adleman, "A Method of Obtaining Digital Signatures and Public–key Cryptosystems," Communications of the ACM, Vol. 21, No. 2, pp.120–126, February 1978.

[Schlichting–Schneider 83] R.D. Schlichting, and F.B. Schneider, "Fail–Stop Processors: An Approach to Designing Fault–Tolerant Computing Systems," ACM Transactions on Computer Systems, Vol. 1, No. 3, pp. 222–238, August 1984.

[Schmuck–Cristian 90] F. Schmuck, and F. Cristian, "Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronisation Algorithm," Proceedings of the 9th ACM symposium on Principles of Distributed Computing, Québec City, Canada, pp. 133–141, August 1990.

[Schneider 84] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail–Stop Processors," ACM Transactions on Computer Systems, Vol. 2, No. 2, pp. 145–154, May 1984.

[Schneider 90] F.B. Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial," ACM Computing Surveys, Vol. 22, No. 4, pp. 299–319, December 1990.

[Shin–Ramanathan 87] K.G. Shin, and P. Ramanathan, "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System," Digest of Papers, FTCS–17, Pittsburgh, USA, pp. 55–60, June 1987.

[Shrivastava 89] S.K. Shrivastava, "The Design and Implementation of ARJUNA," Technical Report No. 205, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, March 1989.

[Shrivastava et al. 91] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, and D.T. Seaton, "Fail–Controlled Computer Architectures for Distributed Systems," Technical Report No. 333, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, July 1991.

[Shrivastava et al. 92] 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, Vol. 41, No. 5, pp. 452–549, May 1992.

[Siewiorek et al. 78a] D. Siewiorek et al., "A Case Study of C.mmp, Cm*, and C.vmp: Part I – Experiences with Fault Tolerance in Multiprocessor Systems," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1178–1199, October 1978.

[Siewiorek et al. 78b] D. Siewiorek et al., "A Case Study of C.mmp, Cm*, and C.vmp: Part II – Predicting and Calibrating Reliability of Multiprocessor Systems," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1200–1220, October 1978.

[Siewiorek–Swarz 92] D.P. Siewiorek, and R.S. Swarz, Reliable Computer Systems: Design and Evaluation (second edition), Digital Press, USA, 1992, ISBN 0–13–772021–1.

[Smith 84] T.B. Smith, "Fault Tolerant Processor Concepts and Operation," Digest of Papers, FTCS–14, Kissimmee, USA, pp. 158–163, June 1984.

[Somani 90] A.K. Somani, "Sequential Fault Occurrence and Reconfiguration in System Level Diagnosis," IEEE Transactions on Computers, Vol. 39, No. 12, pp. 1472–1475, December 1990.

[Somani–Sarnaik 89] A.K. Somani, and T.R. Sarnaik, "Reliability Analysis and Comparison of Two Fail–op/Fail–op/Fail–safe Architectures," Digest of Papers, FTCS–19, Chicago, USA, pp. 566–573, June 1989.

[Speirs–Barrett 89] N.A. Speirs, and P.A. Barrett, "Using Passive Replicates in Delta–4 to Provide Dependable Distributed Computing," Digest of Papers, FTCS–19, Chicago, USA, pp. 184–190, June 1989.

[Speirs et al. 93] 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, Vol. 1, No. 2, pp. 93–109, November 1993.

[Srikanth–Toueg 85] T.K. Srikanth, and S. Toueg, "Optimal Clock Synchronisation," Proceedings of the 4th ACM symposium on Principles of Distributed Computing, Minaki, Canada, pp. 71–86, August 1985.

[Strong et al. 90] R. Strong, D. Dolev, and F. Cristian, "New Latency Bounds for Atomic Broadcast (Extended Abstract)," Proceedings of the 11th Real Time System Symposium, Lake Buena Vista, USA, pp. 156–165, December 1990.

[Stroustrup 92] B. Stroustrup, The C++ Programming Language (2nd edition), Addison–Wesley Publishing Company, 1992, ISBN 0–201–53992–6.

[Tanembaum 92] A.S Tanembaum, Modern Operating Systems, Prentice–Hall, Inc., 1992, ISBN 0–13–595752–4.

[Tao et al. 94] S. Tao, P.D. Ezhilchelvan, S.K. Shrivastava, and N.A. Speirs, "Using Focused Fault Injection Testing for Validating Software Implemented Fault Tolerance Mechanisms," Technical Report, University of Newcastle upon Tyne, 1994.

[Theuretzbacher 86] N. Theuretzbacher, "'VOTRICS': Voting Triple Modular Computing System," Digest of Papers, FTCS–16, Vienna, Austria, pp. 144–150, July 1986.

[Toy 78] W.N. Toy, "Fault–Tolerant Design of Local ESS Processors," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1126–1145, October 1978.

[Toy–Gallaher 83] W.N. Toy, and L.E. Gallaher, "Overview and Architecture of 3B20D Processor," Bell Systems Technical Journal, Vol. 62, No. 1 (pt. 2), pp. 181–190, January 1983.

[Tully–Shrivastava 90] A. Tully, and S.K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs," Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, USA, pp. 104–113, October 1990.

[Webber–Beirne 91] S. Webber, and J. Beirne, "The Stratus Architecture," Digest of Papers, FTCS–21, Montréal, Canada, pp. 79–85, June 1991.

[Wensley et al. 78] J.H. Wensley et al., "SIFT: Design and Analysis of a Fault–Tolerant Computer for Aircraft Control," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1240–1255, October 1978.