DESIGN AND DEVELOPMENT OF ALGORITHMS FOR FAULT TOLERANT DISTRIBUTED SYSTEMS

Paul D. Ezhilchelvan

Ph.D. Thesis

NEWCASTLE UNIVERSITY LIBRARY

089 05273 X

THESIS L3517

University of Newcastle upon Tyne

Computing Laboratory

September 1989

ABSTRACT

This thesis describes the design and development of algorithms for fault tolerant distributed systems. The development of such algorithms requires making assumptions about the types of component faults for which tolerance is to be provided. Such assumptions must be specified accurately. To this end, this thesis develops a classification of faults in systems. This fault classification identifies a range of fault types from the most restricted to the least restricted. For each fault type, an algorithm for reaching distributed agreement in the presence of a bounded number of faulty processors is developed, and thus a family of agreement algorithms is presented. The influence of the various fault types on the complexities of these algorithms is discussed. Early stopping algorithms are also developed for selected fault types and the influence of fault types on the early stopping conditions of the respective algorithms is analysed. The problem of evaluating the performance of distributed replicated systems which will require agreement algorithms is considered next. As a first step in the direction of meeting this challenging task, a pipeline triple modular redundant system is considered and analytical methods are derived to evaluate the performance of such a system. Finally, the accuracy of these methods is examined using computer simulations.

ACKNOWLEDGEMENTS

A special debt of gratitude is due to Professor Santosh Shrivastava for his continuous and constructive advice in supervising this thesis. In particular, his scrupulous reading and criticisms of preliminary drafts have provided me with invaluable guidance. I am grateful to him for his comments and criticisms. I am pleased to acknowledge that Professor Shrivastava and I collaborated for the work on fault classification.

I am also grateful to my colleague Dr. Isi Mitrani with whom I established a very fruitful collaboration for the work on performance evaluation. My thanks are also due to Professor Brian Randell for his technical advice during my research work and for his careful reading of earlier versions of some of the chapters in this thesis.

I would like to thank all my colleagues at the Computing Laboratory who discussed with me the work described in this thesis. In particular, I wish to thank Dr. Neil Speirs for his careful reading of some of the crucial parts of my thesis, that revealed some inaccuracies in the text.

I also acknowledge my wife who typed some parts of my thesis and gave constant encouragement. The UK Science and Engineering Research Council (SERC), and the DELTA-4 consortium of ESPIRIT research programme are also acknowledged for their financial support.

Finally, my *namaskhaars* are offered to the Almighty.

CONTENTS

Chapter 1. Introduction 1

Chapter 2. A Classification of Faults in Systems 12

2.1. Introduction 12

2.2. Components and Their Behaviour 13

2.2.1. Fault/Failure Classification 15

2.2.2. Fault/Failure Ordering 19

2.2.3. Selfchecking Components 22

2.2.4. Selecting Fault Models for Components 24

2.3. Replicated Responses 25

2.3.1. Fault/Failure Classification 27

2.3.2. Fault/Failure Ordering 31

2.4. Composite Components 33

2.4.1. Processor with a Clock 33

2.4.1.1. Types of Clock Faults 34

2.4.1.2. Types of Processor Faults 36

2.4.2. Processor Interconnections 37

2.4.2.1. Processors with Unreplicated Responses 37

2.4.2.2. Composite Component with Replicated Responses 38

2.5. Concluding Remarks 41

Chapter 3. A Family Of Agreement Algorithms 42

3.1. Introduction 42

3.2. Assumptions 46

3.2.1. Clock Synchronism 46

3.2.2. Message Signatures and Authentication 46

3.2.3. Bounded Communication Delay 48

3.2.4. Message Timestamps 50

3.2.5. Unanimity and Validity Conditions 50

3.3. Consistent Omission Fault 52

3.3.1. Algorithm ACO - Algorithm (for)

Consistent Omission (Faults) . 52

3.4. Consistent Value Fault 53

3.5. Consistent Timing Fault 54

3.5.1. Algorithm ACT-1 for Consistent Timing Faults 55

3.5.2. Algorithm ACT-2 for Overloaded Processors 61

3.6. Consistent Emission Fault 67

3.6.1. Algorithm ACE-1 for Consistent Emission Faults .. 69

3.6.2. Algorithm ACE-2 for Overloaded Processors 75

3.7. Omission Fault 80

3.7.1. Algorithm AO for Omission Faults 81

3.8. Value Fault 83

3.8.1. Algorithm AV for Value Faults 84

3.9. Timing Fault and Emission Fault 87

3.10. Byzantine Fault 87

3.10.1. Algorithm AB for Byzantine Faults 88

3.11. The Generic Algorithm 89

3.11.1. Discussions 94

3.12. Concluding Remarks 97

Chapter 4. Early Stopping Agreement Algorithms under Omission and Timing Fault Types . 99

4.1. Introduction 99

4.2. Assumptions and Notations 103

4.3. Permanent Omission Fault Tolerant Algorithm 105

4.3.1. Correctness of the Algorithm 109

4.4. Omission Fault Tolerant Algorithm 117

4.4.1. Correctness of the Algorithm 118

4.5. Timing Fault Tolerant Algorithm 120

4.5.1. Correctness of the Algorithm 123

4.6. Consistently Late Timing Fault Tolerant Algorithm 126

4.6.1. Assumptions 126

4.6.2. Development of the Algorithm 127

4.6.3. The Algorithm 130

4.6.4. Correctness of the Algorithm 132

4.7. Concluding Remarks 136

Chapter 5. Performance Evaluation 138

5.1. Introduction 138

5.2. Distributed Replicated Processing Systems 140

5.3. Model Definition 145

5.4. Analytical Approximations 149

5.4.1. Operative State Distribution for Model 0 154

5.4.2. Operative State Distribution for Model 1 157

5.5. Experimental Results 159

5.5.1. Results for Model 0 161

5.5.1.1. Group 1 161

5.5.1.2. Group 2 166

5.5.2. Results for Model 1 171

5.5.3. Computer Times for Simulations and Analytical Estmations 175

5.6. Concluding Remarks 176

Chapter 6. Conclusions 178

6.1. Directions for Further Research 183

References 186

CHAPTER 1

INTRODUCTION

A system can be considered to be made up of a set of components which interact under the control of a design. A component of a system is a system by itself and can be considered, where appropriate, to be atomic with the implication that any further decomposition of a component is of no interest and can be ignored. According to the terminology presented in [Ander81, Lapri85], *faults* in a system are the (potential or actual) causes of the failures of the system. A violation from the specified behaviour of a system will be termed a *failure*. Given the complexity of modern computing systems, one approach for making them reliable is to accept that despite whatever efforts that have been made to avoid or remove faults, systems can nevertheless remain potentially faulty and to incorporate provisions to enable the system to cope with the faults that remain or develop. This approach is termed the fault tolerance approach and is generally considered to be necessary for building systems that can be assured of providing a high degree of operational reliability to the user(s) of the system.

A given component can usually fail in many different ways. That is, a faulty component can have many failure modes. Thus, in order to be able to provide any kind of guarantee of service, the designer of a fault tolerant system should specify not only the maximum number of components that he presumes might be faulty, but also the types of failures a faulty component is presumed to have. In other words, the fault tolerance specification should state explicitly the type and number of component failures a system is

supposed to tolerate. Then, provided the actual component failures that occur do not violate these assumptions and the fault tolerance strategies are correctly designed, the overall system will not fail. Putting this another way, it is meaningless to claim that a system is fault tolerant, without indicating the assumptions that have been made regarding the number and types of component failures that could occur.

1.1. Fault and failure classification

A distributed system will be defined as a collection of autonomous processors which can communicate with each other, and each of which can provide one or more services and can cooperate with other processors on a common goal or task [Enslo78]. By considering processors as components in a distributed system, different types of processor failures have been considered in the literature. One of the most restricted failure types is an omission failure [Mohan83] whereby a faulty processor fails by producing no output for a given input that requires an output to be produced by the processor. A fault that causes such a failure will be called an omission fault. When a processor's omission failure persists for all such inputs, the processor will be said to have failed in a permanent omission manner. With subtle differences, a permanent omission failure has been termed in the literature as processor crash, halting failure [Birma87], fail-silent failure [Powel88], and fail-stop failure [Schli83]. Failures of these types are relatively easy to tolerate when compared to Byzantine failures. A Byzantine failure is caused by a Byzantine fault and is *any* violation of the specified behaviour. A Byzantine faulty processor is customarily considered to be capable of being malicious in trying to "sabotage" any fault tolerance provisions in the system. Faults that can appear to be of malicious nature were first discussed in [Daly73, Davie78] and have been considered, for example, in the

design of SIFT system [Wensl78] and in [Lampo82] where the name "Byzantine" was coined.

A significant advantage to be gained by assuming that processors may have Byzantine failures is that the analysis required for justifying the fault assumptions made about the processors of systems used in life critical systems is greatly simplified. In order to consider anything less than Byzantine failure behaviour on the part of a processor in the design of a fault tolerant system, one must provide a convincing argument (based on knowledge of the processor's design, its components, and any provisions that the design considers for faults in these components) of why it can fail only in some restricted manner. Since, in the Byzantine fault model, no assumption needs to be made about failure modes of a processor, system analysis is simplified. However, attempts to build systems which can tolerate Byzantine failures of processors involve a significant cost in terms of the number of redundant processors required in the system, and of message and time complexity in providing the system services. For example, when processors are considered to suffer only omission failures, only one (redundant) processor is required in addition to the number of processors that are assumed to fail in providing a service; when Byzantine failures are considered, the non-faulty processors in the system should form a majority if they are to produce identical outputs for a given input (as in systems with majority voting [Lyons62]), and they should out-number the faulty processors by more than three to one [Pease80] if their outputs are unlikely to be identical (such as outputting the reading of a local clock [Lampo85]).

In the design of a reliable, but not life critical systems, provisions for tolerance to Byzantine failures may be sacrificed in the interest of economic considerations. In such a situation, if omission failures are considered to be

too restrictive, then the design of a fault tolerant system requires a realistic means of identifying failure types that are more restricted than in the Byzantine model and less restricted than in the omission model. In this thesis we present, in chapter 2, a fault and failure classification using "expected-value" and "timing" as the two properties of a component's response. The resulting fault and failure classes are ordered according to their relative restrictiveness. Examples are drawn from distributed systems. We further extend this classification to apply to a particularly important type of components that are required to provide replicated responses. Our fault and failure classification is an improvement over [Mohan83, Crist85], and our earlier classification in [Ezhil86]. An interesting observation from our classification of faults and failures is that for a given problem in distributed computing one can design a family of algorithms - from relatively simple ones tolerating failures of restricted types to increasingly complex ones tolerating failures of less restricted (and unrestricted) types. One such fundamental problem considered here will be the agreement problem.

1.2. The agreement problem and algorithms

Processors in a distributed system cooperate on a common goal or a task. Fundamental to such cooperation is the problem of agreeing on a piece of data upon which a computation depends. For example, the data managers in a distributed database system need to agree on whether to commit or abort a given transaction [Gray79]. In a replicated database system, the processors need to agree on an identical sequence of incoming transactions [Garci86], and might need to agree on where a particular piece of data (a file, for example) is supposed to reside [Giffo79, Popek81]. In a flight control system for an airplane [Wensl78], the engine control module and the flight surface control module need to agree on whether to continue or abort

a landing in progress. The key point is not *what* the processors are agreeing on but that they must all come to the *same* conclusion.

An obvious approach to achieving agreement is for the processors to vote and agree on the majority value. In the absence of faults, this works fine, but in a close election, the vote of one faulty processor can swing the outcome. Suppose distinct non-faulty processors receive conflicting votes from a faulty processor, then they might reach conflicting conclusions and hence fail to reach agreement. Thus specific algorithms need to be developed to guarantee that non-faulty processors reach agreement by arriving at the same conclusion. This problem is called the agreement problem [Lampo82] or the interactive consistency problem [Pease80] in the literature and can be briefly described as follows: A processor, called the sender, in a distributed system of at least three processors wants to disseminate a value to all other processors. The non-faulty processors in the system, which are at least two in number, will be said to have reached agreement on the sender's value if they all decide on the same value, and on the sent value if the sender is non-faulty. Extending a solution to the agreement problem mentioned above into a general context where every processor can act as a sender is straightforward.

The agreement problem has been studied under a variety of assumptions mainly concerning the synchrony of processors and message communication and the types of faults processors and communication medium are subjected to. (A brief survey is presented in [Fisch83a].) This thesis presents, in chapters 3 and 4, deterministic algorithms developed for a distributed system where relative processing speeds of, and message communication delays between, processors are assumed to be known and bounded. An upper bound on the number of processors that can possibly fail is also

assumed. Execution of these algorithms will guarantee agreement in a known and bounded time interval. Algorithms are developed in two contexts: the sender processor's broadcast time is not known, and is known, to other processors a priori. Algorithms designed in the context of unknown broadcast time can be developed into broadcast protocols as in [Crist85] which can provide agreement and ordering abstractions [Schne86] which are essential for constructing systems with replicated processing. In this context, in chapter 3, we develop agreement algorithms, and show them to be correct, for faults of each type defined in our classification and thus present a family of agreement algorithms illustrating the relative complexity of these algorithms.

Solutions to the agreement problem when the broadcast time of the sender is known can be useful in systems such as a distributed database system where data managers have a prior knowledge of the time the agreement on commit or abort decision for a given transaction should commence. When processors in a distributed system know the sender's broadcast time a priori, it may be desirable to have them reach agreement early, when the actual number of failed processors is less than the expected. Agreement algorithms that guarantee an early agreement in the presence of less-than-expected number of processor failures are called early stopping algorithms [Dolev82a]. In an execution of an early stopping algorithm, non-faulty processors may reach agreement at different timing instants; some can be earlier than the others. So, these algorithms are useful in applications where processors, following an agreement, carry out actions that need not be time-coordinated. In distributed transaction commit, for example, non-faulty processors need not commit (or abort) a transaction at a coordinated time, so long as they all decide to do, and eventually do, the same thing; therefore, a

non-faulty processor can commit a transaction and continue processing as soon as it has reached a commit decision and thereby knows that all other non-faulty processors will eventually commit the transaction. For such applications, early stopping algorithms can be used to make the processors reach agreement as early as possible. The authors of [Dolev82a] considered Byzantine failures for developing an early stopping algorithm. In chapter 4 of this thesis, we consider a few restricted types of failures and provide early stopping algorithms that are faster than the ones reported in the literature for these failure types.

1.3. Performance evaluation

The agreement algorithms presented in chapters 3 and 4 are useful in constructing systems with replicated processing. Replicated processing with majority voting - N modular redundant processing - provides a powerful means of constructing fault tolerant systems [Mathu70, Carte79]. In N modular redundant processing, NMR processing for short, a given computational task is carried out in N, $N \geq 3$, processing modules. These modules must not have any common mode of failures so that they can fail independently of each other. The results produced by these modules will be subject to a majority vote to obtain the final result. A majority vote is possible, and the final result will be correct, if (i) at least majority of the processing modules are non-faulty and (ii) non-faulty processing modules are to produce identical results. Thus, in NMR processing, tolerance can be provided to failures of at most less than half the number of modules and the failures may even be of Byzantine type.

Fundamental to the fault tolerance capabilities in NMR processing is that non-faulty modules produce identical results for a given computational task. Modules may maintain some state information which can affect

processing of a computational task and hence the results produced. In the case of a deterministic processing model, when non-faulty modules with identical state information process a given computational task, they undergo identical state transitions and produce identical results. Given that processing is deterministic and non-faulty modules have identical initial state information, it is necessary to guarantee that non-faulty modules process the computational tasks in an identical order. When modules can receive task messages from multiple sources or from a single source via different communication paths, they cannot be expected to receive task messages in an identical order. This means that non-faulty modules should agree on the processing order for every given task message to be processed. If it cannot be guaranteed that a source will provide different modules with task messages of identical contents, then non-faulty modules should not only agree on the processing order for a task message but also on the contents of the task message. The agreement algorithms presented in chapter 3 can be used to meet these requirements in systems with NMR processing.

The common form of NMR processing in practical systems is triple modular redundant processing, TMR processing for short, where three processing modules are used to process the computational tasks concurrently. FTMP (Fault tolerant Multiple processor) [Hopki78] achieves fault tolerance through TMR processing and is one of the early practical systems developed for flight control applications. The cost of fault tolerance in TMR processing (NMR processing in general) is mainly the redundant processing modules, the majority voters, and the time taken to agree on, and order, the input messages and to perform majority voting. This time overhead, among other factors, influences the response time for a given computational task - the overall time taken for the final (voted) result of a computational task to be obtained.

Consider a computational task being carried out by a TMR node - a triad of processors grouped for TMR processing. The three processors of the TMR node need not produce their results exactly at the same time, since their processing speeds may be different and they cannot be guaranteed to start processing the task exactly at the same time. Consequently, they may be producing their results to a voter at different timing instants. A majority vote for the final result cannot be carried out until at least two of the three processors have produced their results. If any one of these two processors has failed by producing incorrect results, then the results from the third processor has to be waited for, before performing a majority vote. Thus, the response time of a TMR node not only gets influenced by the time taken to carry out majority voting, agreement and ordering (on input messages) but also varies depending on whether all three or just two processors in a TMR node are non-faulty. This means that an evaluation of response times needs to consider the operational status of processors in the node. Thus, performance evaluation of a system with replicated processing should take into account of a number of factors such as voting times, processor failure modes and failure probabilities, processing and message transmission times, etc. This is a challenging task.

In chapter 5 of this thesis, we study the performance of a distributed replicated system that is made up of a collection of TMR nodes connected in tandem. We present analytical methods to evaluate the performance of such a TMR pipeline system. The derivation of these methods involve analytical approximations, the accuracy of which is examined by computer simulations. Despite their simplicity and roughness due to approximations, the analytical methods presented here can be observed to be quite accurate in estimating system performance measures. These methods can serve as alternatives to simulation methods, when the latter are considered to be

expensive to carry out. We also examine the influence of majority voting times and processor failure rates on system performance. It should be mentioned here that performance evaluation of distributed replicated systems have not (yet) been reported in the literature and our work presented in this thesis, to the best of our knowledge, is the first of its kind.

To summarise the ideas presented in this thesis: A classification of component faults and failures is presented. Agreement algorithms are developed in the two contexts defined by whether the sender's broadcast time is or is not known to other processors in the system a priori. In the first context early stopping algorithms are developed for failures of selected types. In the second context, algorithms are developed for failures of each type defined in our classification - thus a family of algorithms is presented. Next we consider the problem of evaluating the performance of distributed systems which require agreement protocols. Analytical methods for evaluating the performance of a pipeline TMR system are derived based on some approximations, and the accuracy of these approximations is examined using computer simulations. The approximations turn out to estimate performance fairly accurately.

1.4. Thesis organisation

In the next chapter, we present a classification of component faults and failures using "expected-value" and "timeliness" as the two properties of a component's response. We extend this classification to apply to components required to produce replicated responses. The different fault types defined are ordered in the form of a lattice according to their relative restrictiveness and the unrestricted type is shown to be Byzantine. Based on the fault and failure classifications for components with replicated and unreplicated responses, a fault analysis of composite components made up of potentially faulty components is performed. Composite components subject to such a

fault analysis are a processor considered to be made up of computational unit and digital clock, and a distributed system made up of processors and communication subsystem.

In chapter 3, deterministic agreement algorithms tolerant to processor faults are developed in the context of the sender's broadcast time not being known a priori. The types of processor faults considered will be the ones that are defined in chapter 2. A generic algorithm is presented to collectively represent algorithms tolerant to different types of processor faults. Based on the generic algorithm, the influence of processor fault types on algorithm complexity is discussed. For processors in a distributed system having a prior knowledge of sender's broadcast time, early stopping agreement algorithms are presented in chapter 4. Only selected types of processor faults are considered. The early stopping capabilities and message requirement of algorithms presented in this chapter will reveal the fact that each algorithm has been developed making complete use of the distinct features that characterise the respective types of faults tolerated. Chapter 5 presents analytical methods for estimating the performance of a pipeline TMR system. Two system models are studied: in the first model faulty processors stay faulty until the observation period, and in the second faulty processors are repaired after a finite and random delay. The accuracy of analytical approximations involved in the derivation of the analytical methods is examined by computer simulations. Performance estimates obtained by analytical methods are observed to be reasonably close to simulation estimates. The influence of majority voting times and processor failure rates on system performance is also observed. Chapter 6 concludes the thesis and suggests directions for further research.

CHAPTER 2

A CLASSIFICATION OF FAULTS IN SYSTEMS

2.1. Introduction

A fault tolerant computing system must be capable of providing specified services in the presence of a finite number of component failures. In order to be able to provide any kind of guarantee of services, the system designer must specify what kinds of, and how many, component failures the system is intended to tolerate. Suppose a system is constructed out of n components, then its fault tolerance specification could be along the lines that if there are no more than f component failures (where $f < n$) and if each failure is of an assumed type, then the system will continue to function as specified. That is, the type of component failures a system is supposed to tolerate has to be stated explicitly. A given component can usually have many failure modes (that is, a failed component can behave in one of many different ways) some of which can be relatively easier to tolerate than others; at the same time, certain failure modes of a component are likely to occur with greater probability than others. Given that the failure data of system components, such as, failure modes and their probability of occurrences are available, the design of a reliable system will often involve making engineering judgements regarding the classes of component failures for which tolerance is to be provided. For example, if a particular type of component failure is hard to tolerate and if the probability of occurrence of

such failures is extremely small, then, in applications that are not life critical, provisions for tolerating that type of failure may well be sacrificed in the interest of economic and performance considerations. An interesting observation is that for a given system function (e.g. maintaining consistency of replicated data in a distributed system), one can design a *family of algorithms* - from relatively simple ones tolerating restricted types of failures to increasingly complex ones tolerating less restricted (and unrestricted) types of failures.

In this chapter, we present a classification of component failures which we believe provides a convenient and realistic means for specifying faulty behaviour of components and for designing corresponding fault tolerant algorithms. Section 2.2 presents this classification. In section 2.3, we extend our classification to apply to a particularly important class of components that are required to provide *replicated responses.* In the following section we study the behaviour of systems composed of possibly faulty components. Section 2.5 concludes this chapter.

2.2. Components and Their Behaviour

A system can be considered to be made up of a set of components which interact under the control of a design. A component of a system is a system by itself and can be considered to be atomic with the implication that any further decomposition of a component is of no interest and can be ignored. A component's behaviour in response to an input from the environment will be defined by the component's specification prescribing state transitions and a real time interval within which the transitions should occur in response to a given input. Following the terminology developed elsewhere [Ander81, Lapri85], a component *fails,* when its behaviour deviates from that specified.

The term *fault* will be used to refer to the cause of the failure. Consider an input that requires the component to produce an output. A *non-faulty* component, by definition, will produce (i.e. respond with) an output that is in accordance with the specification. The response of a faulty component, on the other hand, need not be as specified. Following [Kopet85], we will consider the response of a component for a given input to be correct, if not only the output value is as expected, but also the output is produced on time. Formally, a component's correct response will be defined as follows:

Definition: Correct Response

Let a component receive at time t_i an input requiring an output from the component and as a result respond by producing an output with value v at time t_j, $t_j > t_i$. For that input, the response is correct iff:

(i) the value is as expected: $v = w$, where w is the expected value consistent with the specification; and,

(ii) it is produced on time: $t_{min} \leq t_j - t_i \leq t_{max}$, where $[t_i + t_{min}, t_i + t_{max}]$ is the interval during which the specified output is expected to be produced; and t_{min} ($t_{min} > 0$) and t_{max} ($t_{max} > t_{min}$) are constants denoting respectively the minimum delay time and maximum delay time for the output. A component's correct response for an input requiring an output can be expressed concisely as:

CR: $v = w$ **and** $t_{min} \leq t_j - t_i \leq t_{max}$.

For notational convenience, CR will also be denoted as:

CR: *expected−value* **and** *ontime*.

The minimum delay time, t_{min}, indicates that the output of a component cannot be produced instantaneously but must experience a finite minimum

delay of non-zero amount. The maximum delay time, t_{max}, indicates the upper bound on the output delay.

The above definition is based on the expected input-output behaviour of the component and does not refer to any internal state transitions caused by inputs. There can, however, be inputs that may require the component to behave by making appropriate changes in its internal state and by producing no output. The value w, and the quantities t_{max}, t_{min} in the above definition are meaningful only when output values are expected to be produced by the component in response to inputs. This also implies that the definition is directly applicable to "demand driven" components: components that produce outputs in response to having received an input. However, there are also *autonomous components*, such as clocks, which continuously produce outputs. A treatment on the behaviour of such components will be presented in section 2.4.

Definition: Incorrect Response

A response will be said to be incorrect, if either the output value or the output timing or both are incorrect. The output value will be termed incorrect, if the value of the output produced is not the expected value consistent with the specification, i. e. $v \neq w$; similarly, the output timing will be said to be incorrect, if the output is produced outside the expected interval, i.e. either $t_j - t_i < t_{min}$ (early) or $t_j - t_i > t_{max}$ (late).

2.2.1. Fault/Failure Classification

In the following, five classes of faults are presented. This classification has been developed by considering failures in the value domain, in the time domain, and then in both the domains. Our classification of faults is based

on, and an improvement over, earlier work reported in [Mohan83, Crist85, Ezhil86].

Omission Fault/Failure

A fault that causes a component not to respond to an input and, thereby, fail by not producing the expected output will be termed an *omission fault* and the corresponding failure an *omission failure*.

A component with an omission fault behaves in a very simple fashion: either a correct response is produced or no response is produced. A processor that (perhaps momentarily) stops functioning, a sensor that occasionally fails to produce output signals, and a communication link which loses messages are examples of components with omission faults. In the literature, many fault tolerant algorithms can be found to have been designed under this fault assumption.

Value Fault/Failure

A fault that causes a component to respond, for a given input, within the correct time interval but with an incorrect value will be termed a *value fault*. The corresponding failure will be called a *value failure* which, using our notation, is defined as:

VF: $v \neq w$ **and** $t_{min} \leq t_j - t_i \leq t_{max}$.

$= \textbf{not } expected-value \textbf{ and } ontime.$

A processor producing erroneously computed values on time, a timely delivery of a corrupted message by a communication link are examples of value failures.

Timing Fault/Failure

A *timing fault* causes a component to respond to a given input with the correct value but outside the correct interval (either early or late). The corresponding failure will be called a *timing failure:*

TF: $v = w$ **and** $(t_j - t_i < t_{min}$ **or** $t_j - t_i > t_{max})$.

$=$ *expected − value* **and not** *ontime*.

For example, an overloaded processor producing correct values with excessive delay and a fast timer which sends an early timeout signal, will be said to have suffered timing failures. A timing failure in which the response is produced late (early) will be called a late timing failure (an early timing failure). A late timing failure is also referred to as a performance failure [Crist86].

Emission Fault/Failure

A component with an *emission fault* fails by producing an incorrect response to a given input. The corresponding failure is called an *emission failure.* Using our notation, an emission failure is:

EF $=$ **not** CR

$=$ $v \neq w$ **or** $(t_j - t_i < t_{min}$ **or** $t_j - t_i > t_{max})$.

$=$ **not** *expected − value* **or not** *ontime*.

An emission fault can cause a component to emit a response which can be incorrect in the value as well as in the time domains. Emission failures are a combination of value and timing failures. An overloaded processor that responds too late to a given input with erroneously computed values can be said to suffer an emission failure.

Note that from the definitions of *VF*, *TF*, and *EF*, we have:

VF **implies** *EF* and *TF* **implies** *EF*.

Thus, value and timing failures (faults) are special cases of emission failures (faults).

Byzantine (or General) Fault/Failure

In the above four failure classes, a component's failure modes have been defined by analysing the properties of an output for a given input. It is also possible to consider a faulty component to fail in an "arbitrary" manner, i.e., in a manner that cannot be perceived within the framework of the above failure classes. For example, a failed component may produce an output without a valid input. All such failure modes that cannot be considered to be in the above four failure classes will be included in the last *general* fault class:

A *Byzantine fault* causes a component to violate the specified input-output behaviour in *any* manner and a *Byzantine failure* will be *any* violation of a component's specified input-output behaviour.

By definition,

EF **implies** *BF*, where *BF* is a Byzantine failure.

Note that it is not generally feasible to enumerate all possible failure modes of a faulty component. A Byzantine faulty component is customarily considered in the literature to be capable of being "malicious" in its responses to its environment. The following examples of a faulty processor's behaviour in a distributed system can be considered to be malicious: a processor P_i masquerading as processor $P_j, j \neq i$, and P_i altering source/destination of a message it is relaying. Faults of malicious nature

were first discussed in [Daly73, Davie78] and have been considered in system designs such as SIFT [Wensl78], and in algorithms for reaching agreement in a distributed system [Lampo82].

2.2.2. Fault/Failure Ordering

A Byzantine fault causes any violation of a component's specified input-output behaviour; as such, no restrictions are applicable in the resulting faulty behaviour. All other fault types preclude certain types of faulty behaviour, the omission fault type being the most restrictive. Thus the omission and Byzantine faults represent two ends of the fault classification spectrum, with the other fault types placed in between. The relationship between the five types of faults can be further developed as follows.

If an omission failure can be interpreted as equivalent to 'producing a null value at some finite time', then it can be defined as follows:

OF: $v = NULL$ **and** $t_i < t_j < \infty$.

Since $v = NULL$ **implies** $v \neq w$, an omission failure as defined above can be seen as a special case of an emission failure: *OF* **implies** *EF*.

OF can also be shown to be a special case of *VF*, by reasoning as follows:

In a value failure, the incorrect output value can be a null value. Define a proper subset of value failures in which the output values are *NULL* as:

VF_{null}: $v = NULL$ **and** $t_{\min} \leq t_j - t_i \leq t_{\max}$.

By definition, VF_{null} **implies** *VF* and also **implies** *OF*. A null value produced on time is the same, for all practical reasons, as a null value produced at any time. Therefore, VF_{null} in which a null value is produced on time can be treated to be the same as ($v = NULL$ **and** $t_i < t_j < \infty$), and therefore to

represent *OF* itself; so *OF* **implies** *VF* in practice.

If an omission failure can be interpreted as equivalent to 'producing some value at the time of infinity', then, it can be defined also as:

OF: ($v \neq NULL$ **and** $t_j = \infty$). Since $t_j = \infty$ **implies** $t_j - t_i > t_{max}$, an omission failure as defined above can be seen as a special case of an emission failure: *OF* **implies** *EF*.

It can also be shown that *OF* **implies** *TF*, by using arguments similar to those employed to show that *OF* **implies** *VF*:

Define a proper subset of timing failures in which $t_j = \infty$ as:

TF_{null}: { $v = w$ **and** $t_j = \infty$}.

By definition, TF_{null} **implies** *TF* and also **implies** *OF*. For all practical purposes, producing the correct value at time ∞ has the same meaning as producing any value at time ∞. Therefore TF_{null} in which the expected value is produced at time ∞ can be considered to be equivalent to {$v \neq NULL$ and $t_j = \infty$} and to represent *OF* itself; thus, *OF* **implies** *TF* in practice.

Thus, omission faults (failures) can be treated as a special case of, and hence a proper subset of, emission, value, and timing faults (failures). From their definitions, value and timing faults (failures) can be seen to form a proper subset of emission faults (failures).

The relationship among these five types of faults (failures) can be expressed by the ordering diagram shown in figure 2.1, where an arrow from A to B, A $\rightarrow$ B, indicates that fault (failure) of type A is a special case of, or a proper subset of, fault (failure) of type B and therefore fault assumptions of type B are less restrictive than those of type A. In the figure, the two circles in omission type represent VF_{null} and TF_{null} which, by definition, form proper subsets of omission type and, in practice, become omission type

itself. Note that the ordering relation $\rightarrow$ is transitive. From the ordering diagram, it can be stated that an algorithm designed to tolerate f, $f > 0$, value (or timing) failures can also tolerate f omission failures; similarly an algorithm designed to tolerate f emission failures can also tolerate f failures of either omission, value, or timing failures, and finally an algorithm designed to tolerate f Byzantine failures can tolerate f failures of any type.

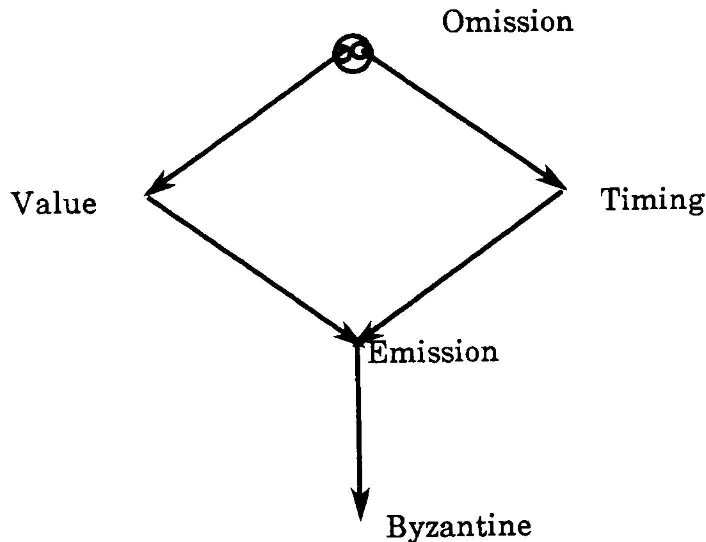


Figure 2.1. Fault/Failure Ordering Diagram.

Remarks

1. Output Sequences

The above classification is based on the behaviour of a component with respect to a single input. When a sequence of inputs over a given time interval is considered, the type of fault suffered by a faulty component will be the most restrictive (or the least serious) one (as per the ordering

diagram in figure 2.1) of which all types of failures occurred during the interval can be considered to be special cases. For example, if, over a given time interval, value and timing failures have occurred, then the component will be said to have suffered emission failures during that interval; if value and omission are the types of failures, then the component will have suffered value failures. If a given type of faulty behaviour persists for a "sufficiently lengthy" sequence of inputs, then the failures can be classified as a permanent failure of that type. A permanent omission fault causes the component to halt functioning for ever. In the literature, it is called a *crash* failure or a *fail-silent* failure [Powel88].

2. Outputs with no timing requirements

There may be situations where a component's responses do not need to follow the rigid timing requirements specified here. Under these circumstances, a timing failure cannot be defined and the component can only suffer three types of failures, namely, omission, value, and Byzantine.

2.2.3. Selfchecking Components

The fault/failure classification can be applied to understand and specify the behaviour of components with builtin redundancy (for selfchecking) where the redundancy is employed to minimise the likelihood of the occurrences of failures of certain types. For example, consider a component with value checks; for valid inputs, such a component is designed to produce either "normal values" or "value exceptions" - the expected values when failures within the component occur for which a degree of tolerance has been provided. A component with dual processors and a comparator that compares the output values of both the processors and outputs a value exception whenever a disagreement is detected can be such a component. It

can suffer only timing failures, provided the comparator is non-faulty and no more than one processor fails; given these fault assumptions for the comparator and processors, the overall behaviour of this selfchecking component is *CR* **or** *TF*:

(*expected − value* **and** *ontime*) **or** (*expected − value* **and not** *ontime*), where the *expected − value* is given by:

normal − value **or** *value − exception*.

That is, such a component produces the normal value or a value exception either on time or not on time. Another example of a selfchecking component will be a processor with a 'watch-dog' timer that is used to prevent timing failures by signalling a 'timing-exception' whenever the processor is deemed not capable of producing its output on time. Thus the expected behaviour of such a processor in the value domain includes generating a response indicating a timing exception. The watch-dog timer cannot, however, detect the processor's value and Byzantine failures. Given that the processor does not suffer Byzantine failures and the watch-dog timer is non-faulty, the overall behaviour of such a processor will be *CR* **or** *VF*:

(*expected − value* **and** *ontime*) **or** (**not** *expected − value* **and** *ontime*), where the *expected − value* is given by:

normal − value **or** *timing − exception*.

That is, such a selfchecking processor produces timely responses which could be wrong in the value domain. If a processor has been constructed with both value and timing checks, then this means that its expected behaviour in the value domain is extended to include the production of value and timing exceptions (or simply failure exceptions).

A fail-stop processor [Schli83] is an example of a component proposed with value and timing checks and to raise a failure exception, in case the possibility for producing an incorrect response is detected. It is also designed to stop responding for ever to input requests after having raised a failure exception.

2.2.4. Selecting Fault Models for Components

In the fault classification presented here, a Byzantine fault has been defined to be a fault which can cause the component to fail in any manner. Choosing the Byzantine fault model for components will mean that no restrictive assumption need be made regarding the components' failure modes. In practice, the type of failures that a component may be assumed to suffer should be decided by considering engineering factors such as the failure data of the component (i.e. failure modes and probability of their occurrences), and application specific details such as the task load the component is designed for, safety factors, and the consequences of the component failing in a manner other than what was assumed. If the failure data are not available or if it is judged that it is not safe to predict the failure modes given the criticality of the application at hand, it will be appropriate to expect the component to fail in Byzantine manner; otherwise, faults of appropriate non-Byzantine class can be chosen to model the component's faulty behaviour. A choice of value, timing, or emission faults can be refined, if necessary, with a set of additional assumptions to precisely model the faults of a component. For example, it is common, in a value fault model, to restrict the failure modes by assuming that corruption of a message by a communication link are limited such that mechanisms such as checksums can be utilised for error detection. In [Veris89], value failures of

processors and communication links which result in detectable message corruption are classified as *syntactic* failures.

2.3. Replicated Responses

In this section the fault classification is extended to a particularly important type of systems where components are required to produce replicated responses for a given input. For example, in triple modular redundant systems, a processor is required to send its output to three other processors; similarly, when processors (considered as components in a distributed system) are taking part in some agreement protocol, every processor is required to send its output to every other processor in the system.

Consider a component that is required to produce a replicated response containing r individual outputs, where r, $r \geq 1$, is the specified replication level, as a result of receiving an input at time t_i. We will use the following vector notation to specify the replicated response:

$V = \{v_1, v_2, \cdots, v_r\}$, where v_k is the value of the kth, $1 \leq k \leq r$, individual output.

$T_j = \{t_j1, t_j2, \cdots, t_jr\}$, where t_jk is the time at which the kth individual output appeared.

Definition: Correct Replicated Response

Let a component receive at time t_i an input requiring a replicated response. For that input, the replicated response with value V at time T_j is correct iff:

(i) the output value is correct: $V = W$, where W is the vector of expected output values; and

(ii) the output timing is correct: $t_i + t_{min} \leq t_jk \leq t_i + t_{max}$, for all t_jk, $1 \leq k \leq r$, in T_j, where t_{min}, t_{max} are as defined in the definition for correct unreplicated responses in section 2.2.

Note that, by definition, W has the property that $w_p = w_k = w$, for all p,k, $1 \leq p,k \leq r$.

Remark: The skew interval

In a correct replicated output, all individual outputs are produced with the correct, hence identical, values and on time but not necessarily at 'the same time'; thus, for any two of the r individual outputs:

$0 \leq |t_jp - t_jk| \leq t_{max} - t_{min}$, for all p,k, $1 \leq p,k \leq r$.

The interval 0 .. S, where S = $t_{max} - t_{min}$, will be called the *skew interval* within which all individual outputs are expected to be produced.

Definition: Incorrect Replicated Response

An incorrect replicated response is defined first by defining failures in value and time domains:

The output value V will be termed incorrect, if $V \neq W$, i.e. there exists some p, $1 \leq p \leq r$, such that $v_p \neq w$.

The response time T_j will be termed incorrect, if there exists some p, $1 \leq p \leq r$, such that:

$t_jp < t_i + t_{min}$ (response too early), or

$t_jp > t_i + t_{max}$ (response too late).

A replicated response will be said to be incorrect, if either V or T_j or both are incorrect.

Definition: Consistently Incorrect Replicated Responses

For replicated responses, it is possible to consider a restricted violation of the specification by considering the notion of *consistent incorrectness* among individual responses of an incorrect replicated response:

In a replicated response, the output value V is said to be *consistently incorrect,* if,

(i) V is incorrect, and

(ii) for all p,k, $1 \leq p,k \leq r$, $v_p = v_k$

That is, all individual output values are *identically* incorrect. Similarly, the response time T_j of a replicated response is said to be *consistently incorrect,* if

(i) T_j is incorrect, and

(ii) for all p,k, $1 \leq p,k \leq r$, $|t_{jp} - t_{jk}| \leq$ S.

That is, while the response is not produced on time, all of the individual responses are produced within the skew interval. A replicated response is said to be *consistently incorrect,* if:

(i) V is consistently incorrect and T_j is correct, or

(ii) V is correct and T_j is consistently incorrect, or

(iii) both V and T_j are consistently incorrect.

2.3.1. Fault/Failure Classification

A given replicated response being consistently incorrect in the value domain or in the time domain or in both the domains is a special case of it being incorrect respectively in the value domain or in the time domain or in

both the domains. As our fault classification is based on the input-output behaviour of a component, the definition of consistently incorrect responses will give rise to a set of fault types that were not defined for components with unreplicated responses. With the definitions of correct, consistently incorrect, and incorrect responses, the following nine classes of faults will be identified for components with replicated responses.

Consistent Omission Fault/Failure

Faults of this type cause a component to fail by not responding to a given input and, consequently, by not producing a response when a replicated response is expected. The corresponding failure will be termed a *consistent omission* failure. A processor that has stopped functioning, a processor that occasionally fails to broadcast a message are examples of components with consistent omission faults.

Consistent Value Fault/Failure

A *consistent value* fault causes a component to respond to a given input by producing a replicated output on time but with identically incorrect values. That is, in a consistent value failure, V is consistently incorrect and T_j is correct. A processor broadcasting an incorrectly computed value is an example of a consistent value failure.

Consistent Timing Fault/Failure

A *consistent timing* fault causes a consistent timing failure in which V is correct and T_j is consistently incorrect. This fault type causes a component to produce correct values either too early or too late, but within the

specified skew interval. A processor with too many computational tasks can suffer a consistent timing failure, when it produces a replicated output with correct values during the interval $[t+\delta, t+\delta+S]$ instead of $[t, t+S]$, where δ is the excess delay due to processor overloading.

A consistent timing failure in which outputs are produced late (early) will be called a consistently late timing failure (a consistently early timing failure).

Consistent Emission Fault/Failure

A *consistent emission* fault causes a component to produce a consistently incorrect response for a given input. In a consistent emission failure, V and/or T_j will be consistently incorrect. An overloaded processor that broadcasts erroneously computed values suffers a consistent emission failure.

Omission Fault/Failure

An omission fault causes an omission failure in which none or some of the individual outputs of a replicated response are not produced. A processor that occasionally stops functioning while outputting the individual outputs of a replicated response is an example of a component suffering an omission failure. An omission fault can cause a consistent omission failure and hence is more general than a consistent omission fault.

Value Fault/Failure

A value fault causes a component to respond with incorrect values on time. In a value failure, V is incorrect and T_j is correct and but any two individual outputs of a replicated output need not be identical. For an example of value failure, consider a processor broadcasting a message to a group of processors to which it is connected by point to point links. A broadcast will then consist of sequentially transmitting a copy of the message held in a buffer to each member of the group. Such a processor can suffer a value fault if the buffer gets corrupted during a broadcast. Value failures (faults) subsume consistent value failures (faults).

Timing Fault/Failure

A timing fault causes a component to respond with correct output value at incorrect time. In a timing failure, V is correct and T_j is incorrect, and any two individual outputs need not appear within the skew interval.

In the previous example of a value failure, instead of buffer corruption, if the processor slows down (due to overloading), then the individual outputs may not be produced within the skew interval. Timing failures (faults) subsume consistent timing failures (faults).

Emission Fault/Failure

An emission fault causes a component to produce an incorrect response for a given input. Both value and timing failures are special cases of an emission failure.

Byzantine (or General) Fault/Failure

A Byzantine fault, as in the case of unreplicated responses, is defined to be the most general fault that can cause the component to deviate from the specified input-output behaviour in any manner; the corresponding failure is defined to be a Byzantine failure which will include the component producing arbitrary responses when no input was supplied and producing responses with 'malicious' intentions. The behaviour of a "traitorous general" in the Byzantine generals problem of [Lampo82] is a classic example of how a processor with Byzantine faults can be malicious in its responses to other (faulty or non-faulty) processors in a distributed system.

2.3.2. Fault/Failure Ordering

When a consistent omission failure is interpreted as a failure of producing null output values (identically incorrect values) at any time after the input was supplied, or as a failure of producing identically correct or incorrect output values at time ∞, it can be seen to form a special case of failures of every other consistent type. By their definitions, consistent omission, consistent value, consistent timing, and consistent emission types of faults/failures are respectively special cases of omission, value, timing, and emission types of faults/failures. The ordering diagram shown in figure 2.2 indicates the 'special case' relationship between various classes of faults and failures. As in figure 2.1, an arrow from A to B, $A \rightarrow B$, in figure 2.2 indicates that faults/failures of type A are a special case of faults/failures of type B, and the ordering relation $\rightarrow$ is transitive.

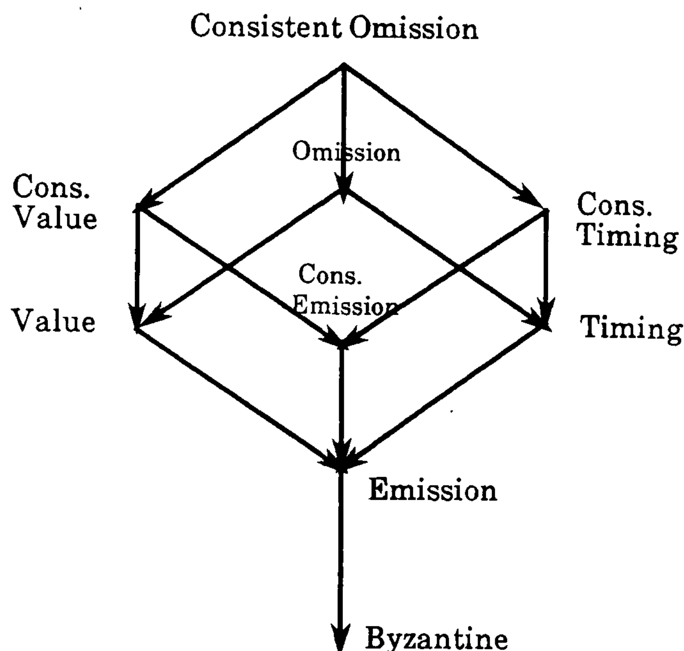


Figure 2.2. Fault/Failure Ordering Diagram For Replicated Responses.

Remark: Output Sequences

As in the case of components with unreplicated responses, the above classification is based on the behaviour of a component with respect to a single input; and, when a sequence of replicated responses over a given time interval is considered, the type of failure suffered by a faulty component will be the most restrictive one (as per the ordering diagram in figure 2.2) of which all types of failures occurred during the interval can be considered to be special cases. Note that an unreplicated response becomes a special case of a replicated response, when r, the degree of replication, is taken to be 1. When r becomes 1, a component's omission, value, timing, and emission failures can be respectively regarded as: consistent omission, consistent

value, consistent timing, and consistent emission failures. Thus, a sequence of replicated and unreplicated responses over a given time interval can also be treated as a sequence of replicated responses and, thereby, the type of fault suffered by the component during that interval can be determined from the ordering diagram in figure 2.2. For example, if, over a given time interval, value and timing failures have occurred for unreplicated responses and consistent timing failures for replicated responses, the component will be said to have a consistent emission fault during that interval; if value and (consistent) omission are the types of failures respectively for unreplicated and replicated responses, then the component will have a (consistent) value fault; if Byzantine and consistent omission failures occur while producing respectively unreplicated and replicated responses, the component becomes Byzantine faulty.

2.4. Composite Components

Following the classification of faults and failures of a single individual component, the behaviour of composite components made up of potentially faulty components is investigated. To start with, the behaviour of a processor is studied by considering a digital clock as one of its components. The study is then extended to a distributed system which is considered to be made up of processors and communication links.

2.4.1. Processor with a Clock

A processor, P, will be considered to be made up of two components: (i) computational and communication unit, CCU, that processes computational tasks and handles communication with the environment, and (ii) a digital clock, CL, that measures the passage of real time and provides the current

time. CCU can use CL to read the current time or to measure time intervals. For some specified ρ, $0 < \rho << 1$, a correct clock measures the passage of one time unit when a real time period between $(1 - \rho)$ and $(1 + \rho)$ has elapsed [Ellin73]. It is natural to model a digital clock as an *autonomous* component - a component that produces outputs (display of current time) not by receiving input requests but simply in response to the passage of real time. One such model is developed here. It is nevertheless possible to model a digital clock as a demand driven "time server" device that outputs current time only for an input request.

2.4.1.1. Types of Clock Faults

A clock's display of current time will be its response produced at every given timing instant. Such a sequence of responses will start from T_0 reflecting the time t_0 when the clock started functioning. Thus, a digital clock is an autonomous component which, once turned on, is expected to produce an infinitely long sequence of responses of monotonically non-decreasing values such that the output value of every response (i.e. the value displayed) at any given timing instant will indicate the clock's measurement of the passage of real time from t_0 to that timing instant. If T is the value displayed at real time t, $t \geq t_0$, then the response will be said to be correct iff:

*CR*1 (measurement of time):

$T_0 + (t - t_0)/(1 + \rho) \leq T \leq T_0 + (t - t_0)/(1 - \rho)$ and,

*CR*2 (monotonic display):

$T \geq T-$, where $T-$ is the display value at $t-$, $t- = t - \Delta t$, as $\Delta t \rightarrow 0$.

While condition $CR1$ states the correctness requirement for the measurement of the passage of real time, condition $CR2$ ensures that values displayed are monotonically non-decreasing. For example, a clock that showed T_1 and T_2 respectively at $t-$ and t such that

$T_0 + (t - t_0)/(1 + \rho) \leq T_2 < T_1 \leq T_0 + (t - t_0)/(1 - \rho)$ can satisfy $CR1$, due to non-zero ρ, but will not satisfy $CR2$.

It can be seen from $CR1$ that the correctness of an output value (T) and the instant of time the output value is produced (t) are interdependent and, hence, value and timing failures cannot occur independently of each other; in other words, an occurrence of a value failure will imply that of a timing failure and *vice versa*. Thus, according to our fault/failure classification, CL can have the following three types of faults/failures: omission, emission, and Byzantine.

A clock that occasionally fails to display the current time will be said to have an omission fault and a clock that fails to display the current time for a "sufficiently long time" will be said to have a permanent omission failure. An emission failure is the clock producing an incorrect response, i.e., **not** ($CR1$ **and** $CR2$). A Byzantine faulty clock can fail in any manner and no assumption can be considered on its failure modes. For a clock, there is little difference between an emission fault and a Byzantine fault. A proper subset of emission failures can be identified by considering emission failures in which only $CR1$ is violated, i.e. (**not** $CR1$ **and** $CR2$). Such failures will be referred to as *monotonic emission failures*. For example, a fast or slow clock or a clock that stops by displaying the same value (running infinitely slow) are examples of clocks with monotonic emission faults. In the following analysis of P's behaviour, CL is considered to have omission, monotonic emission, and Byzantine types of faults/failures.

2.4.1.2. Types of Processor Faults

The faulty behaviour of a processor (P) with a clock, can be studied in terms of the types of faults in its components CCU and CL. Suppose that CCU is non-faulty and CL is faulty. An omission faulty CL will make P to suffer at most omission failures (P's computational results that do not involve the use of CL will be correct and be produced on time). Suppose that CL has a monotonic emission fault. When CCU uses the CL's display as a value in its computation (e.g. to generate a sequence number), P can suffer a value failure. If CL is used by CCU to set timeouts, then a fast or slow clock can result in P's timing failure. Thus a monotonic emission faulty CL can make P emission faulty. A Byzantine failure in CL may result in a Byzantine failure in P. When CCU is faulty and CL is non-faulty, P will suffer the types of faults in CCU. An interesting observation is that when CL becomes unduly faster or slower, P can suffer an emission failure - quite a serious type of failure - even when CCU is non-faulty.

When both CCU and CL are faulty, it is possible for the failures of one component to mask or nullify that of the other component. For example, a permanent omission failure in CCU will make P also to suffer that type of failure, irrespective of the types of faults in CL; similarly, fast computation by CCU and a slow CL may nullify failures of each component resulting in P producing a timely response. However, the possibilities of components' failures nullifying each other cannot be relied upon to happen all the time and, therefore, P should be considered to be faulty for all practical purposes. Thus, except in the case of the CCU having a permanent omission fault, the types of faults in P will be the least serious one which, according to the ordering diagram in figure 2.1, subsumes the types of faults in CCU and CL, given that every fault type is considered to subsume itself and that

monotonic emission fault of CL is taken to be equivalent to, and represented as, emission fault in figure 2.1. For example, when both CCU and CL have omission faults, P will suffer failures of omission type; if CCU has timing faults and CL monotonic emission faults, failures of P will be of emission type.

2.4.2. Processor Interconnections

2.4.2.1. Processors with unreplicated responses

Consider first the behaviour of a component C composed of a processor P1 and a link L that connects P1 to a second processor P2 (see figure 2.3). The function of L is merely to transmit the outputs of P1 to P2.

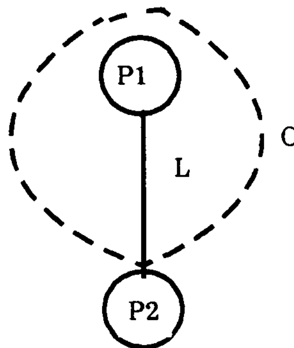


Figure 2.3. A Three Component System.

When one component in C is faulty the other one is non-faulty, C will suffer the type of faults suffered by its faulty component. Suppose that both P1 and L are faulty. A permanent omission failure in L will cause C also to have a permanent omission failure, irrespective of the type of fault in P1. If it can be assumed that faulty L cannot generate messages on its own accord,

then a permanent omission failure in P1 will also mean that C as a whole has a fault of that type. Excepting the above two cases, the fault type of C will be the least serious type which, according to the ordering diagram in figure 2.1, will subsume the types of faults suffered by P1 and by L.

2.4.2.2. Composite component with replicated responses

To analyse the faulty behaviour of a processor-link component producing replicated responses, consider a distributed system made up of n processors, $P_0, P_1, \cdots, P_{n-1}$, capable of exchanging messages using a communication medium L. Suppose that a composite component C_0 is made up of processor P_0 and L. Let L, as shown in figure 2.4, be a bus capable of providing a broadcast service.

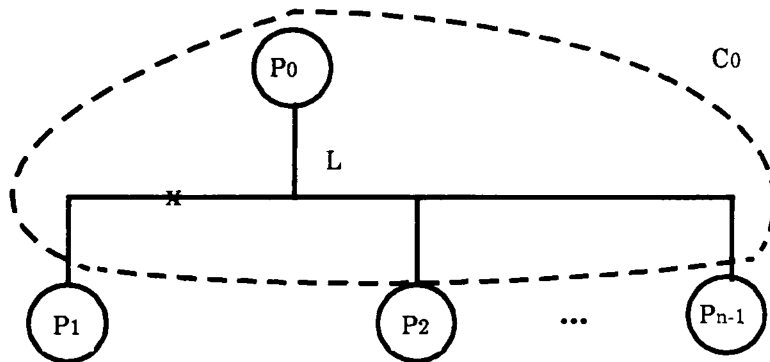


Figure 2.4. Components With Replicated Responses.

Suppose that P_0 is faulty and L is non-faulty; then C_0 will suffer the same fault as in P_0. Similarly, if P_0 is non-faulty and L is faulty, then the type of fault in L will be the fault type of C_0. Since P_0 is not responsible for replicating its outputs, its fault types can only be consistent or Byzantine;

but those of L can vary from consistent omission to Byzantine. A fault in L can cause the failures of L not to be consistent. If, for example, there is a break at the place (x) shown in the figure, L may fail by delivering P_0's output to all processors other than P_1. When both P_0 and L are faulty, the fault type of C_0 can be determined as discussed in the previous subsection.

It is interesting to observe that if L is assumed to be reliable and if P_0 is faulty, then C_0 can fail only in consistent or Byzantine manner. Suppose the functionality of L is further enhanced such that processors receiving a message can authenticate the identity of the sender of the message, then faulty P_0 cannot masquerade as any other processor. The DELTA-4 distributed system[Powel88] is such a system where the communication subsystem has been designed to be reliable and with the authentication facility thereby considerably reducing the probability of masquerading Byzantine failures occurring in the system. Instead of making L reliable, if P_0 is made reliable, then the overall faulty behaviour of C_0 can still encompass all possible failure modes.

Suppose that processors in a distributed system are connected by links such that there exists a communication path between any two processors in the system. The communication paths will be made up of links and, if necessary, processors which will be expected to relay messages according to some routing algorithm.

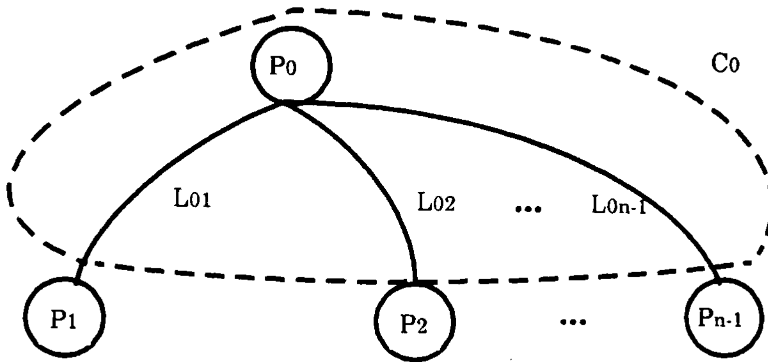


Figure 2.5. Components With Replicated Responses.

In figure 2.5, L_{0i} represents the logical communication path between P_0 and P_i, $i \le i \le n-1$. If any L_{0i} is faulty, then the fault type will be determined by the type of faults in processors and in links that make up L_{0i} and the fault analysis of L_{0i} will be similar to that discussed for the unreplicated case. Suppose that the composite component C_0 is made up of P_0 and communication paths, $L_{01}, L_{02}, ..., L_{0(n-1)}$, incident on P_0, and P_0 produces replicated output by sending a copy of the message through each communication path. Then the fault types of P_0 can vary from consistent omission to Byzantine. As before (discounting permanent omission failures in all L_{0i}'s), the fault type of C_0 will be the least serious one which, according to the ordering diagram in figure 2.2, will subsume the type of faults in all faulty components of C_0. For example, if P_0 has consistent value faults and some L_{0i} is faulty with omission (emission) faults, then C_0 will be considered to have value (emission) faults.

A similar fault analysis can be performed for composite components $C_1, C_2, ..., C_{(n-1)}$ constructed respectively with $P_1, P_2, ..., P_{(n-1)}$ and

communication paths incident on them. Such a fault analysis of composite components in terms of constituent components can provide useful insights on the faulty behaviour of the composite components. Given the nature of processor interconnections and the type of faults that might be suffered by processors and communication links or bus, the behaviour of one processor with respect to other processors can be analysed.

This section has illustrated how the fault classification scheme presented here can be applied to develop accurate fault models of composite components. Precise specifications of component faults can be exploited to develop efficient fault tolerant algorithms as illustrated in the next two chapters.

2.5. Concluding Remarks

Using 'timeliness' and 'expected value' as the specified properties of a component's response, we have presented five types of faults for components with unreplicated responses and extended the classification to nine types of faults for components with replicated responses. These classifications together with the fault lattices presented represent one of the main contributions of this chapter. We have also discussed how faulty behaviour of a component can be determined, given the fault types of its constituent components. This was illustrated by constructing fault models of clocks and processors in a distributed system. Our fault classification provides a very convenient means not only for specifying the faulty behaviour of components but also for the construction of increasingly more sophisticated fault tolerant algorithms tolerating faults of increasingly more complex types. This will be demonstrated by the fault tolerant agreement algorithms presented in the following two chapters.

CHAPTER 3

FAMILY OF AGREEMENT ALGORITHMS

3.1. Introduction

Reaching agreement in the presence of faults is a fundamental problem in fault tolerant distributed computing. The agreement problem originally formulated as the *interactive consistency* problem [Pease80] and later as the *Byzantine Agreement* problem [Lampo82] can be described as follows: A distributed system is made up of n, $n > 2$, processors capable of communicating with each other only by message passing. Among these n processors in the system, one processor is designated as the *sender* and the other processors as *receiver* processors. The sender wants to disseminate some value to all receiver processors. The difficulty is that some processors, possibly including the sender, may be faulty and that a non-faulty processor cannot ascertain which other processors are faulty. When the sender is faulty, it cannot be guaranteed that all non-faulty receiver processors receive the same value directly from the sender. Thus it is necessary to develop an agreement algorithm that can be executed by receiver processors to guarantee that the following conditions will be met in the presence of at most f, $f < n-1$, faulty processors:

C1 all non-faulty receiver processors decide on the *same* value, and

C2 when the sender is non-faulty, all non-faulty receiver processors decide

on the value sent.

By C1, it is ensured that all non-faulty receiver processors reach a *unanimous* decision on a value; by C2, it is guaranteed that if the sender is non-faulty, then every non-faulty receiver processor reaches a *valid* decision by deciding on the value sent by the sender. When C1 and C2 are met, the agreement will be said to have been reached (on the sender's value) by non-faulty receiver processors. When the sender is non-faulty, C2 implies C1.

The agreement problem has been studied under a variety of assumptions concerning the synchrony of processors, the types of failures to which processors are subject, the properties of communication network, and deterministic versus probabilistic nature of a solution (see [Fisch83a] for a brief survey). We solve the agreement problem for a synchronous distributed system in which the relative computational speeds of, and message communication delays between, non-faulty processors are assumed to be bounded. In a synchronous system, the agreement problem can be solved by considering that the receiver processors know, or do not know, *a priori* the time at which the sender is to send its value. In this chapter, receiver processors are considered not to have prior knowledge of the sender's send time. Deterministic agreement algorithms are developed assuming that the communication medium is fault free and that faults occur only in processors. The types of faults that are considered to occur in processors will be the ones that are defined in the previous chapter for components with replicated responses. The range of fault types considered here thus starts from consistent omission and ends with emission and Byzantine. For each fault type, an agreement algorithm is presented - thus presenting a *family of agreement algorithms*. An execution of any of these algorithms presented here will guarantee that the conditions C1 and C2 are met within some known and bounded

time interval denoted by Δ. Since these algorithms are designed in the context of synchronous and deterministic processors with no prior knowledge of sender's send time, they can provide agreement and ordering abstractions [Schne86] which are essential for constructing systems with replicated processing (e.g. [Ezhil89]).

In the literature, agreement algorithms have been developed in the above-mentioned context under omission, timing and Byzantine types of faults. In [Pease80, Dolev83], faulty processors are considered to fail in a Byzantine manner. These algorithms, of necessity [Dolev82b], require at most $(f+1)$ rounds of message exchange to be carried out between processors. Faster algorithms for Byzantine faults have been developed in [Babao85] using redundancy in the communication medium. Omission faults, timing faults, and Byzantine faults are the three types of faults considered by Cristian et. al. in solving the agreement problem [Crist85].

Agreement algorithms for faults other than omission, timing, and Byzantine faults will be developed in this chapter. The complete presentation of the family of agreement algorithms helps the reader to compare the complexities of algorithms for different types of faults and thus illustrates the advantages, or lack of them, in assuming a particular type of fault for processors. Under consistent timing and consistent emission faults, a specific type of timing faults is also considered: processor overloading is considered to be the only fault that can cause a processor to suffer (late) timing failures. Assumptions are made to restrict the failure modes of overloaded processors and they lead to the development of faster agreement algorithms. So, for applications where these restrictive assumptions for overloaded processors can be considered to be realistic, these algorithms can be used to achieve distributed agreement in a fast and cost-effective manner.

It should be emphasised that these algorithms are developed with the assumption that the communication medium is fault free and that this assumption can be relaxed when processor faults are not considered to be consistent. One way to relax this assumption is to identify communication failures as processor failures. This may result in an overly pessimistic view of the reliability of the system. Alternatively, the communication network can be made redundant such that faulty links and faulty processors do not disconnect non-faulty processors.

The rest of this chapter will be organised as follows: In the next section, the assumptions made for designing the family of agreement algorithms are stated and explained. In sections 3.3 to 3.10, algorithms for different types of processor faults - ranging from consistent omission to Byzantine faults - are respectively presented and proved to be correct, where necessary. The algorithms for omission, timing, and Byzantine faults are variants of protocols in [Crist85]. In each of these sections, important observations about the respective algorithms have been made. In section 3.11, a generic algorithm is developed which collectively represents the entire family of agreement algorithms. Based on the generic algorithm, the complexities of the agreement algorithms presented in previous sections are compared and the relative influence of processor fault types on the complexities of agreement algorithms is presented. Section 3.12 concludes the chapter.

3.2. Assumptions

3.2.1. Clock Synchronism

The hardware clocks of processors have small differences in their running rates and their readings tend to drift apart with the passage of real time [Ellin73]. Design of agreement algorithms with bounded and known execution time will require the processors to observe time within some bounded and known difference. In order to meet the requirement, the processors have to adjust the readings of their clocks to counter the difference that has so far developed. The readings of clocks can be adjusted periodically either through the execution of a fault tolerant clock synchronisation algorithm, e.g. [Halpe84, Kopet87, Crist86], or with reference to some external and reliable time service. The following is assumed about processor's clocks.

Assumption A1:

At any given instant of real time, the observable difference between clock readings of any two non-faulty processors will be at most ϵ.

Remark:

The term "non-faulty" is not necessary in the above assumption when faulty processors are considered to suffer only (consistent) omission or (consistent) value failures, since processors with faults of omission and value types do not fail by producing an untimely output. Thus processors with faults of above types will be taken to satisfy A1.

3.2.2. Message Signature and Authentication

The messages, on being relayed by processors, may get corrupted accidentally or deliberately by faulty processors and may, thereby, have their contents altered. A corrupted message, on being received, can deliver

a value other than what the source processor intended to deliver, if it is not detected to have been corrupted. In an attempt to avoid being "misinformed" by corrupted messages, each processor is assumed to have facilities to "sign" every message it sends, and to "authenticate" the signature of every message it receives in order to detect any apparent attempt to corrupt a message.

In [Rives78], a scheme has been proposed by which processors can generate message signatures such that the signatures are signer-dependent and contents-dependent, and can authenticate signed messages so that any attempt to alter the contents will be detected with high probability; it is also possible for processors to over sign an already signed message by considering the signature(s) in the message as yet another piece of data, and to authenticate a multiply signed message by recursively authenticating every individual signature starting from the one that was last added on, and ending with the first one. Another such scheme is presented in [Okamo88]. These schemes, when implemented in processors of a distributed system, will guarantee the following:

(i) a non-faulty processor's signature for a given message is *highly likely* to be unique for it to be generated by any other processor, and

(ii) any attempt to alter the contents of a non-faulty processor's signed message is *highly likely* to be detected.

Thus, when a non-faulty processor signs and sends its messages, it is highly unlikely that contents of its messages can be undetectably altered; similarly, when a non-faulty processor authenticates signed messages of another non-faulty processor, it is highly likely that an authentic message will contain what the sender sent. Assumption A2 is made on processors' signature and authentication capabilities.

Assumption A2:

A non-faulty processor's signature for a given message cannot be forged by any other processor and any attempt to alter the contents of a message signed (or over signed) by a non-faulty processor can be detected.

Remarks:

From what is guaranteed by schemes for generating message signatures and for message authentication, it can be seen that there exists a non-zero probability of (i) a faulty processor being able to generate the same signature that a non-faulty processor would generate for a given message, and (ii) the contents of a message signed by a non-faulty processor being altered, accidentally or deliberately, in such a way that the corrupted message cannot be detected as unauthentic. In A2, this probability is assumed to be zero. Such an assumption has often been made in the literature [Dolev83, Crist85, Lampo82] for designing (what are called signed message or authenticated) agreement algorithms. So, by A2, a non-faulty processor's signature for a given message cannot be undetectably generated by a faulty processor. However, a faulty processor can undetectably forge another faulty processor's signature for a given message. This means that a faulty processor can undetectably alter the contents of a message that is signed or oversigned only by faulty processors. Thus, by A2, only the messages that are signed by at least one non-faulty processor are protected against undetectable corruption by a faulty processor.

3.2.3. Bounded Communication Delay

The processors in the distributed system communicate only by message passing via a fully connected communication medium that is assumed to be fault free. The next assumption bounds the message transmission delay.

Assumption A3:

If, at time T, an event occurs in a non-faulty processor p and causes a message to be formed and broadcast, then any other non-faulty processor q can receive the message at time T_r, $T \leq T_r < T + d$ - where time is measured according to any non-faulty processor's clock and $d > 0$.

Remark 1:

d is fixed by considering the processing time taken when the occurrence of an event requires a processor to decide on sending messages, and message routing and transmission in the communication medium.

Remark 2:

Based on A1 and A3, the following can be stated:

If an event occurs in a non-faulty processor p at time T according to p's clock that causes a message to be formed and broadcast, then any other non-faulty processor q will receive the message at time T_r, $T - e \leq T_r < T + d+e$, according to q's clock.

Remark 3:

The term non-faulty is not necessary in A3 and in remark 2, if processors are considered to have consistent omission or consistent value faults; when omission or value faults are considered, A3 and remark 2 will be true without the term non-faulty only for those q's that receive p's message.

Remark 4:

When processor faults are not of consistent type, the full connectivity and the reliability requirements on the communication medium can be relaxed: the number of faults in the communication medium should be such that processors remain connected in the system that survives after removing the faulty links, faulty processors, and the links incident on faulty processors. The types of faults in the communication medium should be no more

serious than the types of processor faults considered.

3.2.4. Message Timestamps

Finally, the sender processor will be required to indicate its time of transmission by appending a timestamp (local clock reading at the time of transmission) to the message so that messages transmitted at different timing instants can be distinguished. For the sake of simplicity in handling of sender's messages by receiver processors, the following assumption is made on the sender processor's timestamps.

Assumption A4:

A non-faulty sender processor will not carry out more than one broadcast with the same timestamp.

Remarks:

If, at the same clock time, two or more distinct values are decided to be delivered, then A4 will require the sender processor to transmit all these values in a single message with one timestamp. When processors are considered to suffer only omission or value faults, the term non-faulty is not necessary in A4, since faulty processors fail only by sending incorrect values and, by A1, have properly synchronised clocks.

3.2.5. Unanimity and Validity Conditions

When the sender broadcasts a value with a timestamp, the decision made by a receiver processor will be associated with that timestamp. In this context, conditions, C1 and C2, for agreement will be modified as *unanimity* and *validity* conditions respectively:

When the sender broadcasts a message with timestamp T_s,

Unanimity:

all non-faulty receiver processors either reach the same decision for the message with timestamp T_s by their clock time $T_s + \Delta$, where Δ ($\Delta > 0$) is known and bounded, or do not ever take any decision for T_s, and

Validity:

when the sender is non-faulty, all non-faulty receiver processors decide for T_s by their clock time $T_s + \Delta$ on the value sent by the sender.

For a given broadcast by the sender, condition C1 requires a non-faulty receiver processor to decide on a value; the unanimity condition however permits a non-faulty receiver not to make any decision. This modification of C1 is necessary because the agreement problem is being solved in a bounded and known interval and in the context of the sender's broadcast time not being known to receiver processors a priori. When the sender's broadcast time is not known a priori, it may not be always possible for a receiver processor to reach a decision in a bounded interval for every broadcast carried out by a faulty sender.

In the following, presented are the algorithms tolerant of at most, $f, f \leq n-2$, distinct processors in the system suffering from faults of a given type. For each agreement algorithm designed, Δ will be expressed as a function of d and e. In fixing the size of Δ, it is assumed that a receiver processor will take no time for executing the instructions of the algorithm (this will require an increase on the value of d to accommodate execution time overheads).

3.3. Consistent Omission Fault

The agreement problem becomes a non-problem under consistent omission faults, since a sender with consistent omission faults can fail only by not sending its message to any receiver processor. So, either all or none of the receiver processors receive a given message from the sender. So no agreement protocol is necessary.

3.3.1. Algorithm ACO - Algorithm (for) Consistent Omission (Faults)

The messages exchanged between processors are taken to be of the following *record* structure:

```
type M = record
    v:value; Ts:Time; id: string of char
    end;
```

A message of type M contains a value in v, a timestamp in T_s, and the identifier of the sending processor in id.

The sender executes the following algorithm:

```
sender:
  const own-id=....;
  var msg:M; local-value:value;
begin
  msg.v := local-value; msg.Ts := clock.get;
  msg.id := own-id; send(msg)
end.
```

The sender processor has a unique identifier that is assumed to be known to every receiver processor. The constant 'own-id' contains that identifier. We assume that the object 'clock' at each processor is responsible for maintaining the local clock synchronised with the other clocks in the system; its function 'get' returns the current clock reading. The current clock reading is given to the message as timestamp and the timestamped message is sent to all other receiver processors by executing the "send(msg)" primitive. The algorithm executed by a receiver processor will be as follows:

```
receiver:
 var msg:M;
begin
 cycle
  receive(msg)
  decide(msg.v, msg.T_s)
 endcycle
end.
```

The "receive(msg)" primitive returns a message, if there is one or more messages to be received; it blocks itself, otherwise. Upon receiving a message with sender's identifier, a receiver processor decides on msg.v for msg.T_s by executing "decide(msg.v, msg.T_s)".

When a non-faulty receiver processor receives a message with timestamp, say, T_s, its clock will read less than $T_s + (d+e)$ due to assumptions A1 and A3 (see remark 2 under assumption A3). So a non-faulty receiver processor decides for T_s by its clock time less than $T_s + (d+e)$ and, therefore, the size of Δ necessary to guarantee agreement is $(d+e)$.

3.4. Consistent Value Fault

Under consistent value fault assumptions, a faulty sender can fail only by sending messages with identically incorrect value to all receiver processors. By A1, a faulty sender's messages will have correct timestamp on them. Therefore, agreement will be guaranteed when receiver processors make their decision with every message they receive from the sender. Thus, the algorithm for consistent value faults, ACV, becomes the same as ACO. Since a faulty sender does not fail by producing untimely responses, all non-faulty receiver processors reach agreement by their clock time $T_s + \Delta$, $\Delta = d+e$ (by A1 and A3), on the sender's value sent with timestamp T_s.

3.5. Consistent Timing Fault

A processor with a consistent timing fault fails by producing some or all of message replicates either early or late. All the message replicates will nevertheless be produced within the specified skew interval that is included in message delays bounded by d (A3). Consider a faulty sender suffering such a failure in sending its messages with timestamp, say, T_s. Let one non-faulty receiver processor, say, p, receive the sender's message at its clock time T_p. Since the sender's failure is consistent, no other non-faulty receiver processor can receive the sender's message earlier than $T_p - d$ and later than $T_p + d$, according to p's clock. If q, q $\neq$ p, is any other non-faulty receiver processor and T_q is the time q received the sender's message according to its clock, then, by A1, $| T_p - T_q | < (d+e)$.

Let the sender's failure be such that T_p is between T_s and $T_s + (d+e)$ and T_q is greater than $T_s + (d+e)$. While p can regard the sender's message timely (by assumptions A1 and A3) and decide on the value contained therein, q cannot do so, because unless late messages are ignored by receiver processors, agreement cannot be guaranteed to be reached within a bounded amount of time. Therefore, reaching agreement in a bounded amount of time and in the presence of consistent timing faults will require messages to be exchanged between receiver processors. A receiver processor which received a timely message directly from the sender should relay the message to every other receiver processor - thus initiating the second round of message exchange following the sender's broadcast which is counted as the first round. Since there can be at most f faulty processors, an execution of agreement algorithm should allow for at most $f+1$ rounds of message exchange between processors.

The messages exchanged between processors during an execution of the agreement algorithm, are taken to be of the following *record* structure:

```
type M = record
    v: value; Ts: Time; ids: sequence of identifiers;
    end;
```

Every processor has a unique identifier that is assumed to be known to every other processor. The message variable, ids, can have a sequence of processors' identifiers.

Two algorithms will be developed for consistent timing faults. The first algorithm is general in nature. A special version of the algorithm will be developed for overloaded processors with the following two assumptions: overloaded processors have their clocks synchronised with those of non-faulty processors within the bounded difference of e (A1); secondly, whenever an overloaded processor receives and subsequently relays the message, the ratio of the delay involved in sending the message to the communication medium (as a part of relay operation), to the the delay involved in receiving the message from the communication medium is assumed to be bounded by a known quantity θ. The value of θ is assumed to be known and to bound the ratio of the two delays irrespective of the variations in processing loads during the period the message is being handled. The second algorithm turns out to be faster than the first one, if $(f-1) > \theta$.

3.5.1. Algorithm ACT-1 for Consistent Timing Faults

The sender sends its message to all receiver processors as before. A receiver processor, on receiving a message from the sender, will accept the message, if the message has been received at its clock time T_r such that $T_s - e \leq T_r < T_s + (d+e)$, where T_s is the timestamp in the message. If the message is accepted, the receiver processor takes its decision for T_s on the value contained in the message, appends its identifier to the message, and sends the message to every other receiver processor.

When a receiver processor receives a message from another receiver processor, it inspects the timestamp, T_s, in that message. If it has not decided for the timestamp in the message, it counts the distinct processor identifiers in the message (let it be s) and checks the timeliness of the message in the following manner: if the message has been received at its clock time T_r such that $T_s - se \leq T_r < T_s + s(d+e)$, it is considered timely; otherwise, it is considered untimely and is ignored. If the received message is found to be timely, a decision for T_s is taken on the value contained in the message and, if s $\leq f$, the processor's identifier is appended and the message is sent to receiver processors whose identifiers are not present in the message.

Algorithm ACT-1

The sender executes the following algorithm.

```
sender:
  const own-id =....;
  var msg: M; local-value: value;
begin
   msg.v := local-value; msg.Ts:= clock.get;
   append-id-and-send(msg);
end.
```

The sender has its identifier stored in 'own-id'. By "append-id-and-send(msg)", it appends its identifier to the message and sends the message to all receiver processors.

The algorithm executed by a receiver processor accommodates $(f+1)$ rounds of message exchange. The first round allows a receiver processor to receive the sender's message directly, and the remaining f rounds are for the sender's value to be received through another receiver processor. The algorithm for a receiver processor is presented next:

```
const own-id = ....; maxm-rounds = (f+1);
 var
     msg: M; Tr: Time; s: integer;
function TC: boolean;
        begin TC := msg.Ts - se ≤ Tr < msg.Ts + s(d+e) end;
function decided(T:Time): boolean;
        begin if a value decided for T then decided:=true else decided:=false
        end;

begin
 cycle
1) receive(msg);
2) Tr:= clock.get;
3) s:= no-of-identifiers(msg);
4) if not decided(msg.Ts)
5)   then if TC
         then begin
6)          decide(msg.v, msg.Ts);
7)          if s < maxm-rounds
8)            then append-id-and-send(msg)
         end
 endcycle
end.
```

Explanation

The receiver processor's identifier is stored in 'own-id'. The constant, maxm-rounds, represents the maximum number of rounds of message exchange the algorithm accommodates and is taken to be $(f+1)$. The boolean function, TC, expresses the timeliness condition that a message received at time T_r (line 2) with s processor identifiers (counted in line 3) should satisfy to be accepted. The boolean function, decided(T), returns true, if a decision has already been taken for timestamp T; returns false, otherwise. For a message received in line 1, whether a decision has been taken for the timestamp in the message is checked in line 4. If not, and if the message is timely (line 5), then a decision is taken by executing "decide(msg.v, msg.T_s)" (line 6). If, in line 7, s is less than $(f+1)$, the message is relayed to appropriate receiver processors after the processor's identifier is appended to the message variable ids (line 8).

Correctness Of Algorithm

A processor with a consistent timing fault need not maintain its clock in bounded synchronism as required by A1. Therefore, a faulty processor may accept and relay a message that would have been considered as untimely by a non-faulty processor. Suppose that a non-faulty receiver processor, say, p, receives a message with timestamp T_s and s, s > 1, processor identifiers, at its clock time $T_s - \mathrm{s}e$. Let q, q≠p, be the processor that sent the message to p. Either of the following two conditions can be true with q: q is faulty and has accepted and relayed an untimely message that should have been ignored; or, q is non-faulty with its clock being faster than p's clock by e and has received a message with (s-1) processor identifiers at its clock time $T_s - (\mathrm{s}-1)e$, and p receives q's message in zero time. The processor p cannot ascertain whether q is faulty or non-faulty, and if it ignores q's message it may violate the unanimity condition. Similarly, by supposing that p receives q's message just before its clock time $T_s + \mathrm{s}(d+e)$, two scenarios can be constructed such that q is faulty in one scenario and non-faulty in the other one in which p's clock is faster than q's clock by e and p receives q's message in just less than d time. Thus, "$T_s - \mathrm{s}e \leq T_r < T_s + \mathrm{s}(d+e)$" is the necessary timeliness condition.

Theorem 3.1

Any execution of the above algorithm meets the unanimity and validity conditions for $\Delta = (f+1)(d+e)$ in the presence of at most f, $f < n-1$, distinct processors suffering consistent timing faults.

Proof

Under this fault model, no faulty receiver processor will alter the contents of the message it relays. Therefore, any processor that decides during the execution of the algorithm, will do so only on the sender's value which will be the same for all receiver processors. When the sender is non-faulty, all non-faulty receiver processors will find the sender's message with timestamp, say, T_s arriving in the specified time interval, due to assumptions A1 and A3, and will decide by their clock time $T_s + (d+e)$, $T_s + (d+e) < T_s + \Delta$. Hence the theorem, for a non-faulty sender. Next, the theorem is proved for a faulty sender by showing that it is not possible, at clock time $T_s + (f+1)(d+e)$, for one non-faulty receiver processor to decide, and another one not to decide, on the sender's value broadcast with timestamp T_s.

Since faulty processors fail only in timing manner, no receiver processor will send, and therefore will receive, a message with more than $(f+1)$ processor identifiers during any execution of the algorithm. So, no non-faulty receiver processor will decide for T_s no later than its clock time $T_s + (f+1)(d+e)$. Let p and q be any two non-faulty receiver processors. Let p decide by receiving a message with s, $1 \leq s \leq f$, processor identifiers. Processor p must have received the message at its clock time, say, T_p, $T_s - se \leq T_p < T_s + s(d+e)$. Since $s < (f+1)$, by A1 and A3, q will receive p's message at its clock time T_q, $T_s - (s+1)e \leq T_q < T_s + (s+1)(d+e)$. Let p decide by receiving a message with $s=(f+1)$. Since there can be at most f faulty processors, the processor whose message made p decide for T_s must be non-faulty and must have sent a message to q not later than q's clock time $T_s + f(d+e) + e$. Therefore, q must receive that message and decide on the value contained therein, not later than its clock time $T_s + (f+1)(d+e)$. Hence the theorem for $\Delta = (f+1)(d+e)$.

Observations

1. Consistency in timing failures.

In the proof, it can be observed that the algorithm has been developed without making use of consistency in processors' timing failures; in particular, the fact that a faulty processor's broadcast is received by all non-faulty processors is not utilised. The reasons for this are as follows:

In the design of any agreement algorithm with a known bound on execution time, a time interval, $I(T_s, s)$, of known and finite size has to be specified as the interval of acceptability for a message with s, $1 \le s \le f+1$, processor identifiers and with timestamp T_s. In the execution of the algorithm, a message should be accepted, only if it is received within the specified interval according to the receiving processor's clock. In ACT-1, $I(T_s, s)$ is $[T_s - se,\ T_s + s(d+e))$ and has a size of less than $s(d+2e)$. This size is the smallest necessary (as per assumptions A1 and A3) to ensure that no message sent or relayed by a non-faulty processor gets rejected at non-faulty destinations.

Recall that a faulty processor's clock may not be in specified synchronism with other non-faulty processors' clocks and that a faulty processor may suffer an unbounded length of delay in sending or relaying a message. Therefore, whatever be the specified interval $I(T_s, s)$, it is always possible for a faulty processor's broadcast messages to be received by some non-faulty processors within the specified interval and by others outside the interval. In other words, for any specified interval of message acceptability, it is possible for a faulty processor to fail in such a way that its messages, while being received by all non-faulty processors, are rejected by some non-faulty processors as untimely. So, the above algorithm has to be, and has been, developed without making use of the consistent nature of faulty processors'

timing failures. For that reason, the above algorithm will also be tolerant of timing faults where a faulty processor may fail by not sending its broadcast messages to some processors.

2. Time Optimality.

During a message broadcast, a faulty processor can respond to different processors in different timing manner - timely to some and untimely (either late or early) to others. Therefore, consistent failures become as severe as timing failures as far as reaching agreement in a bounded and known interval is concerned. Therefore, an agreement algorithm whose every execution can be guaranteed to terminate in less than $(f+1)$ rounds cannot be designed. Thus, the above algorithm is optimal with respect to the size of Δ which is $(f+1)(d+e)$.

One of the requirements for faulty processors in a distributed system to suffer consistent failures with respect to other processors will be to have a reliable communication medium with full connectivity. (See the section on component interconnections in chapter 2.) It is observed here that any attempts at achieving a fully connected reliable medium cannot make an agreement algorithm any faster, when processors suffer consistent timing failures.

3.5.2. Algorithm ACT-2 for Overloaded Processors

Time optimality of ACT-1 implies that a faster agreement algorithm cannot be developed, unless specific assumptions are made on failure modes of faulty processors. Processor overloading is a timing fault that can cause a processor to produce late responses. The second algorithm is developed with the only faulty processors in the system being overloaded processors which fail by producing consistently late responses. In developing this algorithm,

the following two assumptions are made in addition to the ones described in section 3.2.

Assumptions

assumption a1:

All processors have their clocks synchronised within the known and bounded difference of e.

This assumption requires that the execution of clock synchronisation algorithm in processors be carried out at a sufficiently low level and using high priority messages so that processing loads at high level have little impact on the execution of synchronisation algorithm. It extends A1 of section 3.2 to clocks of all processors in the system.

assumption a2:

If an overloaded processor receives and subsequently relays a message, the ratio of the delay the processor suffered in sending the message to the communication medium, to the delay it suffered in receiving the message delivered to it by the communication medium is bounded by a known quantity θ.

In a2, no assumption is made to quantify the delay an overloaded processor suffers in sending a message to the communication medium or the delay in receiving a message from the communication medium; also, there is no assumption to imply that an overloaded processor always receives an incoming message or that it always sends back a message to be relayed; what is assumed is a known relationship between the message relaying delays and message receiving delays.

Let w_r be the time (which can be measured according to any processor's clock, due to a1) a received message spends in a receive buffer of a processor

before being delivered to the destination process within the processor; let w_p be the message processing time and w_s be the time the message spends in an output buffer before being sent. Thus a message spends a total of $w_r + w_p + w_s$ time after reception and ultimately relayed by a processor. By a2,

$$(w_p + w_s)/w_r < \theta$$

will be true for any overloaded processor, if w_p, w_s, and w_r are finite.

The message processing required by the algorithm will be to verify the timeliness of the message, so w_p will be very small. The quantities w_r and w_s increase with processing loads. Thus, in general, a2 will be satisfied, if processing loads do not increase in the interval of length $w_p + w_r + w_s$ (which is usually in the order of milliseconds) beyond what was perceived in the estimation of θ.

Algorithm

This algorithm requires at most two rounds of message exchange to be carried out between processors. In the first round, the sender, as in the previous algorithm, broadcasts its value with a timestamp and its identifier. If a receiver processor finds the sender's message timely, it decides and relays the message to other receiver processors thus initiating the second and the last round. The timeliness check for messages from the sender is the same as in ACT-1.

If the sender is overloaded and if at least one non-faulty receiver processor finds the message timely, then all non-faulty receiver processors will reach agreement in the second round. If no non-faulty receiver processor finds the sender's message timely, then the second round of message exchange, if there is to be one, will be initiated only by an overloaded receiver processor. Using consistency in overloaded processors' failures and

assumptions a1 and a2, calculated will be an upper bound on the time interval within which all non-faulty receiver processors will receive an overloaded processor's relayed message. Given this bound, the failures of overloaded receiver processors are effectively reduced to consistent omission failures and the agreement is guaranteed at the end of the second round.

Suppose that the sender is overloaded and that all non-faulty receiver processors find the sender's message late in the first round. Since the sender's timing failures are consistent, the local clock time at which non-faulty receiver processors receive the sender's message will be within $(d+e)$ of each other. Overloaded receiver processors may suffer unduly long delays in receiving the message delivered to it by the communication medium. By a1, all processors have their clocks synchronised within e. Therefore, the local clock readings at which non-faulty processors receive, and overloaded processors can *potentially* receive, the sender's message will be within $(d+e)$ of each other. Let T_s be the message timestamp of the sender. Since no non-faulty receiver processor received the sender's message before its clock reading $T_s + (d+e)$, an overloaded processor cannot potentially receive the sender's message before its clock time T_s. While an overloaded receiver processor can suffer unduly long delays in receiving a message, it will however decide not to relay a message which it has received after its clock reading $T_s + (d+e)$. It can be established, based on a2, that if an overloaded processor receives the sender's message before its clock time $T_s + (d+e)$ and relays the message, then its messages will be received by all non-faulty processors by their clock time $T_s + (d+e) + \theta*(d+e) + (d+e)$. Thus, when Δ is taken to be $(2+\theta)(d+e)$, either all or none of the non-faulty receiver processors will receive an overloaded receiver processor's relayed message by the end of the second round.

In the following, the algorithm is presented and its correctness is proved for $\Delta = (2+\theta)(d+e)$.

The sender executes the following algorithm which is the same as in ACT-1.

```
sender:
  const own-id = .....;
  var msg: M; local-value: value;
begin
   msg.v := local-value; msg.Ts:= clock.get;
   append-id-and-send(msg)
end.
```

The algorithm for a receiver processor is:

```
const own-id = ...; maxm-rounds = 2;
 var
      msg: M; Tr: Time;
      s: integer;
 function TC: boolean;
         begin TC := (s=2 or msg.Ts - e ≤ Tr  < msg.Ts + (d+e)) end;
function decided(T:Time): boolean;
         begin  if a value decided for T then decided:=true else decided:=false
         end;

begin
 cycle
1)  receive(msg);
2)  Tr:= clock.get;
3)  s:= no-of-identifiers(msg);
4)  if not decided(msg.Ts)
5)    then if TC
            then begin
6)            decide(msg.v, msg.Ts);
7)            if s < maxm-rounds
8)              then append-id-and-send(msg)
               end
 endcycle
end.
```

Correctness Of Algorithm

Theorem 3.2

Any execution of the above algorithm meets the unanimity and validity conditions for $\Delta = (2+\theta)(d+e)$ in the presence of at most f, $f < n-1$, overloaded processors suffering consistently late timing failures.

Proof

Consider an execution of the algorithm in which the sender sends its message with timestamp T_s. When the sender is non-faulty, all non-faulty receiver processors reach agreement by their clock time $T_s + (d+e)$, due to assumptions A1 and A3. Hence the theorem is true for a non-faulty sender.

Suppose that the sender is overloaded. The theorem is proved for an overloaded sender by showing that either all non-faulty receiver processors decide by their clock time $T_s+\Delta$, or none of them ever decides.

Suppose that a non-faulty receiver processor decides for T_s by receiving a timely message directly from the sender. It decides by its clock time $T_s + (d+e)$ and, by A1 and A3, every other non-faulty receiver processor will be able to decide for T_s by its clock time $T_s + 2(d+e) < T_s + (2+\theta)(d+e)$.

Suppose that all non-faulty receiver processors find the message from the sender late; i.e., they all receive the sender's message at or after their clock time $T_s + (d+e)$. Since overloaded processors' failures are consistent, if any overloaded receiver processor relays the sender's message, all non-faulty receiver processors will receive, and subsequently decide on, the message. Thus, either all or none of the non-faulty receiver processors decide for T_s in the second round. Let r be such an overloaded receiver processor that relays the sender's message. Let p be any non-faulty receiver processor. Since no non-faulty receiver processor received the sender's message before its clock time $T_s + (d+e)$, r would not have been able to receive the sender's message

earlier than $T_s + e$ according to p's clock. When p's clock reads $T_s + e$, r's clock will read T', $T_s \leq T \leq T_s + 2e$, by a1. Therefore, r cannot receive the sender's message earlier than T_s according to its clock.

Since r is an overloaded processor, it may suffer an unduly long delay in receiving the sender's message delivered to it by the communication medium. That delay must be less than $(d+e)$, otherwise it could not have received the message before its clock time $T_s + (d+e)$ and would not have subsequently decided to relay the message. By a2, the delay r can suffer in relaying the message will be less than $\theta*(d+e)$. These relayed messages will take a transmission time of less than d and will be received by non-faulty processors by $T_s + (d+e) + \theta*(d+e) + d$ according to r's clock. Thus, every non-faulty receiver processor will receive r's message by $T_s + (d+e) + \theta*(d+e) + d+e$ according to its clock. Hence the unanimity condition and the theorem.

Observation

The proof of correctness is based on consistent nature of timing failures of overloaded processors. Therefore, ACT-2 will not be tolerant to overloaded processors which do not fail in consistent manner. This algorithm is faster than ACT-1, if $\theta < (f-1)$.

3.6. Consistent Emission Fault

Under the consistent emission fault model, the failure modes of a faulty processor are a union of consistent value and consistent timing failures. Algorithms will be derived from consistent timing fault tolerant algorithms by adding necessary measures to cope processor's value failures. Consistent timing fault tolerant algorithms involved more than one round of message exchange between processors. Under consistent emission fault model, a

faulty processor can attempt to alter the contents of the message it relays. Therefore, it is necessary for processors to sign every message they send or relay, and to authenticate every message they receive.

When signed messages are exchanged between processors, a faulty processor can undetectably forge another faulty processor's signature for a given message and can, therefore, undetectably alter the contents of another faulty processor's signed message it relays. By assumption A2, only the messages signed or over signed by a non-faulty processor are guarded against undetectable corruption. Therefore, when the sender is faulty, a receiver processor may receive, for example, two or more authentic messages containing different values but with the same timestamp. To detect such a state, every receiver processor maintains a value bag, denoted as V-bag, in which the sender's value and timestamp contained in authentic messages will be stored as a two element set. Another bag, called time bag and denoted as T-bag, is also maintained to hold only the timestamps of authentic messages. If there is more than one entry in V-bag for a given message timestamp stored in T-bag, a default decision will be made for the timestamp.

Two algorithms will be presented. The first algorithm is tolerant of consistent emission faults and the second will be for a special case of overloaded processors failing in consistently late emission manner. These algorithms will be derived from ACT-1 and ACT-2 respectively. The development of the second algorithm will require assumptions a1 and a2 of ACT-2 in the context of overloaded processors suffering consistent emission failures and another assumption a3 by which an overloaded processor cannot undetectably forge any other processor's signature for a given message. The following message structure will be assumed in the presentation of these two algorithms:

```
message structure:
type M record
    v: local-value; Ts: Time; signatures: string of char;
    end;
```

The message variable 'signatures' will have processors' signatures for the contents of the message. A receiver processor, on relaying a signed message, will sign the message by appending 'signatures' with its signature for message contents which are v, T_s, and the old value of 'signatures'.

3.6.1. Algorithm ACE-1 for Consistent Emission Faults

To cope with consistent timing failures, the algorithm, like algorithm ACT-1, allows at most $(f+1)$ rounds of message exchange between processors, and each message received should be checked for timeliness before being accepted. To cope with consistent value failures, messages are signed before being sent or relayed, and are authenticated before being accepted. The number of processor signatures in a received message, s, will indicate the number of processors that have sent or relayed the message. The timeliness check that is carried out on a received message is the same as the one in ACT-1.

Each receiver processor maintains a V-bag and a T-bag. The V-bag is a set of two-element sets. The first element of a two-element set will be a value and the second a timestamp. The T-bag is a set of timestamps. When a receiver processor receives a message during the execution of the algorithm, it verifies that the message contains a value and a timestamp not already stored as a pair in the V-bag. If so, the message is verified for its authenticity and timeliness. If the message is found authentic and timely, the value in the message and the message timestamp are entered as a pair into the V-bag; message timestamp is also entered into the T-bag, avoiding duplication; if the number of processor signatures in the message is less than $(f+1)$,

the message is signed and sent to processors whose signatures are not present in the message.

At any time during the execution of the algorithm, a non-faulty receiver processor may receive an acceptable message containing a value that is different from what it has stored in its V-bag for a given message timestamp. Therefore, unlike in ACT-1, a processor should defer its decision for a given message timestamp, say, T_s, until its clock time $T_s + (f+1)(d+e)$ which marks the end of execution for sender's message with timestamp T_s. If the V-bag, at that time, contains only one entry-pair for that timestamp, then the decision will be made on the value in the entry; otherwise, a default decision is made.

Since a receiver processor should defer its decision for a given message timestamp until its clock reads a particular value, its algorithm will be in two parts that are to be executed concurrently. In the first part, received messages are checked for acceptability and entries into the V-bag and the T-bag are made, if necessary; in the second part, a decision for a message timestamp in the T-bag is made by referring to corresponding entries in the V-bag at appropriate clock times. Concurrent execution of these two parts of the algorithm should share the V-bag and the T-bag in a mutually exclusive manner. In the following presentation of the algorithm, necessary implementation for this mutual exclusion is assumed, and the V-bag and the T-bag are shown as shared sets of data:

```
sender:
  var msg: M; local-value: value;
begin
   msg.v := local-value; msg.Ts:= clock.get;
   sign-and-send(msg)
end.
receiver:
 const Δ = (f+1)(d+e); maxm-rounds = (f+1);
 var
     V-bag: shared set of {v1: value, T1: Time};
     T-bag: shared set of Time; msg: M;
     default, v2: value; Tr, T2: Time; s: integer;
 function TC: boolean;
          begin TC := (1 ≤ s ≤ (f+1) and
                              msg.Ts - se ≤ Tr < msg.Ts + s(d+e))
          end;

cobegin
 begin
  cycle
1)    receive(msg);
2)    Tr:= clock.get;
3)    s:=no-of-signatures(msg);
4)    if ({msg.v, msg.Ts} not in V-bag)
5)      then if authentic(msg) and TC
              then begin
6)              store({msg.v, msg.Ts}, V-bag);
7)              if msg.Ts not in T-bag then store(msg.Ts, T-bag);
8)              if s < maxm-rounds
9)                then sign-and-send(msg)
             end
  endcycle;
 end
//
 begin
  cycle
10)     for any T2 in T-bag
11)       if clock.get = T2 + Δ
            then begin
12)            if {v2, T2} unique in V-bag
13)             then decide(v2, T2)
14)             else decide(default, T2);
15)             V-bag:= V-bag - { all {v2, T2} in V-bag};
16)             T-bag:= T-bag - { T2 }
            end
  endcycle
 end
coend.
```

Algorithm ACE-1.

Explanation:

The message variable 'signatures' in the sender's message will be initially null. The sender signs its message, msg, and sends it to receiver processors by executing "sign-and-send(msg)".

In the algorithm executed by a receiver processor, the constant, 'maxm-rounds', represents the maximum number of rounds of message exchange the algorithm should accommodate and is taken to be $(f+1)$. The operation "store({msg.v, msg.T_s}, V-bag)" stores the pair {msg.v, msg.T_s} in V-bag; similarly, the operation "store(msg.T_s, T-bag)" stores the message timestamp in T-bag.

The boolean function, TC, contains the boolean condition "$1 \leq s \leq (f+1)$" and the timeliness condition for a received message with s processor signatures. A faulty processor can undetectably forge another faulty processor's signature for a given message and can fail by putting not only its signature but also other processors' signatures onto the message it relays. Therefore, when messages are exchanged between processors, a non-faulty receiver processor may receive authentic messages with more than $f+1$ distinct processor signatures and for such messages "$1 \leq s \leq (f+1)$" will not become true.

The boolean function "authentic(msg)" (in line 5), returns true, if the message, msg, has only authentic signatures of distinct processors. If a message has unauthentic signatures and/or has more than one signature of the same processor, then the function will return false.

In line 4, a message received (in line 1) at time T_r (noted in line 2) with s processor signatures (counted in line 3) is checked whether its value and timestamp are already in V-bag. If not, in line 5, it is verified for its

authenticity and timeliness. On being found authentic with s, $1 \leq s \leq f+1$, distinct processor signatures and timely, the message is accepted and appropriate entries are made in V-bag and T-bag and if $s < (f+1)$ (line 8), it is over signed and sent (line 9) to appropriate processors.

Lines 10 to 16 represent the second part of the algorithm to be executed concurrently with the first part (lines 1 to 9). For any entry, T2, in T-bag, when the clock reads T2 + Δ (lines 10, 11), a decision is taken for T2 in lines 12 to 14. With T2 as timestamp, if {v2,T2} is the only entry in V-bag (identified by "unique" condition in line 12), a decision is taken on v2 for timestamp T2; otherwise, a default decision is made. In lines 15 and 16, entries corresponding to T2 are removed from V-bag and T-bag.

In the following, the algorithm is shown to be correct.

Correctness Of Algorithm

Theorem 3.3

Any execution of the above algorithm meets the unanimity and validity conditions for $\Delta = (f+1)(d+e)$ in the presence of at most f, $f < n-1$, distinct processors with consistent emission faults.

Proof

When the sender is non-faulty, all non-faulty receiver processors will obtain only one pair for every message timestamp, due to assumption A2. Hence the validity condition.

Consider now the case where the sender is faulty. We will say that a processor "obtains" {V, T_s}, when it stores {V, T_s} in its V-bag. As per the

conditions in TC, it is not possible for a non-faulty processor to obtain {V, T_s} at or after its clock time $T_s + (f+1)(d+e)$. Consider two non-faulty processors p and q. The unanimity condition is shown to be met by showing that for any {V, T_s}, when p obtains {V,T_s} before its clock time $T_s + \Delta$, q will also obtain {V, T_s} before its clock time $T_s + \Delta$.

Suppose that p obtains a {V, T_s} by accepting a message with s, $1 \le s \le f+1$ distinct processor signatures. The processor p accepts the message, only if the message is authentic and is received at its clock time T_p such that $T_s - se \le T_p < T_s + s(d+e)$. If $s \le f$, then q will find p's message timely, authentic and with distinct processor signatures and can thereby obtain {V, T_s} by its clock time $T_s + (s+1)(d+e) \le T_s + \Delta$. If $s = (f+1)$, since there can be at most f faulty processors, it is not possible for q not to have obtained {V, T_s} before its clock time $T_s + \Delta$. Thus, at their clock time $T_s + \Delta$, p and q will have identical entries for T_s in their V-bag. Hence the unanimity condition and the theorem.

Observation

Consistency in value failures.

It can be noted that the correctness of this algorithm, like that of algorithm ACT-1, has been established without making use of consistent nature of processors' emission failures. Therefore, this algorithm will also be tolerant of emission faults.

Consistent emission failures are a union of consistent timing failures and consistent value failures. It was shown, in the previous section, that an agreement algorithm tolerant of consistent timing faults has to be developed without making use of consistency in processors' timing failures. Therefore,

the development of any consistent emission fault tolerant algorithm cannot make use of consistency in timing failures. It is however possible to make use of consistency in processors' value failures in designing a consistent emission fault tolerant algorithm and such an algorithm will be:

a message received from the sender is accepted, decided on, signed, and relayed, if the message is authentic and timely; it is ignored, if it is unauthentic; it is preserved, if it is authentic and untimely. The preserved message is decided on, if a *decision* message - which is a timely and authentic message with s, $2 \leq s \leq f+1$, distinct processor signatures and with value and timestamp same as in preserved message - is received from another receiver processor in any of the remaining f rounds; it is ignored, if no decision message is received until clock time $T_s + \Delta$. If a decision message received has been signed by less than $(f+1)$ processors, it is signed and relayed.

In any execution of this algorithm, non-faulty receiver processors will be able to decide in less than $(f+1)$ rounds, if the sender is non-faulty or if less than $(f-1)$ receiver processors fail when a faulty sender fails by sending an authentic and untimely message. However, the value of Δ cannot be guaranteed to be less than that for ACE-1.

3.6.2. Algorithm ACE-2 for Overloaded Processors

Algorithm ACE-2 is developed to reach agreement in the presence of overloaded processors whose failures will be a union of consistently late timing failures (due to processing loads) and consistent value failures. It will be derived from ACT-2 which is for overloaded processors that fail only by producing consistently late responses. ACT-2 was developed with assumptions

a1 and a2 mentioned in previous section and required two rounds of message exchange between processors.

In ACE-2, the features of ACT-2 will be combined with the use of message signature and authentication mechanisms to cope with value failures. In addition to a1 and a2, required will be assumption a3 by which an overloaded processor cannot undetectably forge another processor's signature for a given message. Due to a3, an overloaded receiver processor cannot undetectably alter the contents of a signed message it relays and, therefore, it cannot make a non-faulty receiver processor obtain more than one value for a given timestamp. This eliminates a receiver processor's need, as in ACE-1, to maintain V-bag and T-bag and to defer the decision until particular clock time. The assumptions a1, a2, and a3 required for ACE-2 are stated below.

Assumptions

a3: An overloaded processor cannot undetectably forge another processor's signature for a given message.

Assumption a3 does not guarantee that an overloaded processor will not attempt to forge another processor's signature nor that it will always detect a message with unauthentic signature. It merely ensures that a non-faulty processor will be able to detect any forged signature contained in the message it receives. The underlying assumption in a3 is that the failures of an overloaded processor are not so serious that the signature and authentication mechanisms used will be effective enough for a3 to be true.

Assumptions a1 and a2 of ACT-2 are restated in the context of overloaded processors suffering consistently late emission failures:

a1: All processors have their clocks synchronised within the known and bounded difference of e.

a2: If an overloaded processor receives and subsequently relays a message, the ratio of the delay the processor suffered in sending the message to the communication medium, to the delay it suffered in receiving the message delivered to it by the communication medium is bounded by a known quantity θ.

The assumptions underlying a1 are: overloaded processors have non-faulty clocks; the execution of clock synchronisation algorithm in processors is carried out at a sufficiently low level and using high priority messages so that processing loads at high level have little impact on the execution of synchronisation algorithm; and, value failures do not affect clock adjustments performed during the execution of synchronisation algorithm. Assumed in a2 is that processing loads do not increase at a rate that is more than what was assumed in the estimation of θ.

Algorithm

The sender signs and sends its message to receiver processors. The algorithm executed by the sender is the same as that in ACE-1. The two-round algorithm for receiver processors is presented next.

```
const maxm-rounds = 2;
var
    msg: M; Tr: Time;
    s: integer;
function TC: boolean;
        begin TC := (s=2 or msg.Ts - e ≤ Tr < msg.Ts + (d+e))
        end;
function decided(T): boolean;
      begin if a value decided for T then decided:= true else decided:=false
      end;

begin
 cycle
1)    receive(msg);
2)    Tr:= clock.get;
3)    s:=no-of-signatures(msg);
4)    if not decided(msg.Ts)
5)      then if authentic(msg) and TC
              then begin
6)              decide(msg.v, msg.Ts);
7)              if s < maxm-rounds
8)                then sign-and-send(msg)
              end
 endcycle
end.
```

Correctness of Algorithm

Theorem 3.4

Any execution of the above algorithm meets the unanimity and validity conditions for $\Delta = (2+\theta)(d+e)$ in the presence of at most f, $f < n-1$, overloaded processors suffering consistently late emission failures.

Proof

When the sender is non-faulty, the theorem is true by theorem 3.3. Suppose that the sender is overloaded in an execution of the algorithm. By a1, the sender has a properly synchronised clock. Therefore, it will not carry out more than one broadcast with the same timestamp. Since its value failures are consistent, its replicated messages to all receiver processors will be of identical contents. This implies that the sender sends only one value

for a given message timestamp. By a3, an overloaded receiver processor, while relaying the sender's message, cannot undetectably alter the message contents. Therefore, if any non-faulty receiver processor decides for a message timestamp, say, T_s, during the execution of the algorithm, it will do so only on the value sent by the sender with timestamp T_s.

Let p and q be any two non-faulty receiver processors. The unanimity condition is shown to be met for Δ by showing that, for any message timestamp T_s, (i) it is not possible for p to decide and q not to decide; and, (ii) if p and q decide, they do so by their clock time $T_s + \Delta$.

A non-faulty processor, during an execution of the algorithm, may receive messages with more than two signatures. This can happen, if a faulty processor has attempted to put other processors' signatures onto the message it relays or sends. A non-faulty processor, by a3, will find such messages unauthentic and will ignore them. Thus, it reaches a decision only by receiving an authentic message that is either signed once by the sender or doubly signed.

Suppose that p decides for T_s by accepting a message that was signed only once (s=1). The processor p accepts the message, only if the message is authentic and received at its clock time T_p such that $T_s - e \leq T_p < T_s + (d+e)$. Processor q will find p's doubly signed message authentic and will decide for T_s by its clock time $T_s + 2(d+e) < T_s + \Delta$.

Suppose that p decides for T_s by receiving a doubly signed message. By theorem 3.2, p will receive the message before its clock time $T_s + \Delta$; and it is not possible for q not to have received the message p received, before its clock time $T_s + \Delta$. Since value failures of an overloaded receiver processor are consistent, the messages received by p and q will have identical contents. Since p decided on the message, q will also find the message authentic

and decide for T_s. Hence the unanimity condition and the theorem for $\Delta = (2+\theta)(d+e)$.

Observations

1. Assumption a3 and two rounds.

Consider an execution of ACE-2 without assumption a3. Suppose that the sender sends its message such that the message is timely to a non-faulty processor, say, p, and also to an overloaded processor, say, r and that the message is late to another non-faulty processor, say, q. If processor r relays the sender's message with contents undetectably altered, processor q will receive, in the second round, two authentic messages of different contents and cannot decide which of the two messages is original. If ACE-2 had accommodated at most $f+1$ rounds, then q could use the subsequent rounds to send p the message relayed by r and thereby reach agreement. Therefore, the two-round algorithm ACE-2 will not be correct without a3.

2. Consistency in failures.

Theorem 3.2 requires overloaded processors' timing failures to be consistent. Correctness of ACE-2 requires overloaded processors' value failures to be consistent and theorem 3.2. Therefore, ACE-2 will not be tolerant of emission faults in processors.

3.7. Omission Fault

The omission fault tolerant algorithm presented here is a variant of atomic broadcast protocol developed in [Crist85] under omission fault assumptions. Hence, it is briefly mentioned here to enable the reader to compare it with agreement algorithms developed under other fault models.

3.7.1. Algorithm AO for Omission Faults

Since processors with omission faults fail only by not producing expected outputs, the algorithm does not require the use of message signature and message authentication facilities, and of timeliness checks that are to be carried out before accepting a message. An omission faulty processor, while producing a replicated output, can however fail by not sending its messages to some of the concerned processors. Therefore, the algorithm should allow for at most $(f+1)$ rounds of message exchange so that the unanimity condition can be guaranteed, when the sender is faulty.

In the following presentation of the algorithm, the messages exchanged between processors are taken to be of the *record* structure that is same as the one used in consistent timing fault tolerant algorithms.

```
type M record
    v: value; Ts: Time; ids: sequence of identifier
    end;
```

Every processor is assumed to have a unique identifier that is known to every other processor. The message variable ids contains the identifiers of processors which have seen the message contents.

Algorithm

The sender sends all receiver processors its message which contains its value, a timestamp and its identifier. The execution of the agreement algorithm will require every receiver processor to behave in the following manner: On receiving a message containing sender's value, a receiver processor checks if it has decided for the timestamp in that message. If so, the message is ignored; otherwise, the message is accepted and a decision is taken. If the number of distinct processor identifiers present in an accepted message is less than $(f+1)$, then the receiver processor appends its identifier

to the message and relays the modified message to those processors whose identifiers are not in the message.

The algorithms executed by the sender and a receiver processor are given below. An explanation is omitted, as it follows from the explanation for ACT-1.

```
sender:
  const ownid =.....;
  var msg: M; local-value: value;
begin
   msg.v := local-value; msg.Ts:= clock.get;
   append-id-and-send(msg)
end.


receiver:
 const ownid = ...; maxm-rounds = (f+1);
 var  msg: M; s: integer;
 function decided(T:Time): boolean;
         begin  if a value decided for T then decided:=true else decided:=false
         end;
begin
 cycle
1)  receive(msg);
2)  s:=no-of-identifiers(msg);
3)  if not decided(msg.Ts)
      then begin
4)      decide(msg.v, msg.Ts);
5)      if s < maxm-rounds
6)        then append-id-and-send(msg)
       end
 endcycle
end.
```

Algorithm AO.

Any execution of the algorithm will guarantee that all non-faulty receiver processors reach agreement by their clock time $T_s + \Delta$, $\Delta = (f+1)d+e$. We refer the reader to [Crist85] for the correctness of the algorithm. To intuitively show that the size of Δ used here is necessary: since there can be at most f faulty processors, any execution of the algorithm should allow at least $(f+1)$ distinct inter-processor communication rounds to take place so that the unanimity condition is guaranteed, when the sender is faulty. Considering also the clock difference, Δ cannot have a

value less than $(f+1)d+e$.

Observation

The above algorithm can be modified to be faster, if it is given that a bounded number of omission faulty processors in the system fail only in consistent manner and that all processors are reliably connected by the communication network. Let a, $a>1$, be the maximum number of faulty processors that fail only in consistent omission manner. Thus at most $(f-a)$ faulty processors can fail in a manner that is not consistent. In such a system, an execution of AO with maxm-rounds$=f-a+1$, will guarantee agreement. To intuitively show this: consider an execution in which the sender is faulty and a non-faulty receiver processor receives a message with $f-a+1$ processor identifiers. Since no more than $f-a$ processors fail in a manner that is not consistent, at least one of those processors that have sent or relayed the message must be either non-faulty or with consistent omission faults. Therefore, every other non-faulty receiver processor must have received, and decided on, the sender's value.

3.8. Value Fault

Value faults subsume omission faults. This implies that the value fault tolerant algorithm should allow for at most $(f+1)$ rounds of message exchange so that the unanimity condition can be guaranteed to be met, when the sender processor is faulty. A processor with value faults, in addition to suffering omission failures, may attempt to alter the contents of a message it relays and, while producing a replicated output, may send messages of different contents to different processors. Therefore, the messages exchanged between processors during the execution of the algorithm should be signed before being sent and be authenticated before being accepted; and,

as in the algorithm for consistent emission faults, each processor should maintain V-bag and T-bag shared data structures and should defer its decision on sender's value until a particular time on its clock.

Under this fault model, faulty processors do not fail by producing untimely messages. Therefore, messages received during the execution need not be checked for timeliness. A faulty processor can undetectably forge another faulty processor's signature for a given message. It can fail by putting not only its signature but also other processors' signatures onto the message it relays. Therefore, when messages are exchanged between processors, a non-faulty receiver processor may receive authentic messages with more than $f+1$ distinct processor signatures and such messages should be ignored. So, the value fault tolerant algorithm becomes the same as the algorithm for consistent emission faults with $(f+1)$ rounds, ACE-1, except for the absence of timeliness condition on received messages and for a different value of Δ.

3.8.1. Algorithm AV for Value Faults

With the type of messages exchanged and the algorithm executed by the sender being the same as ACE-1, only the algorithm executed by receiver processors is given below. An explanation is omitted, as it follows from the explanation for ACE-1.

```
const Δ = (f+1)d+e; maxm-rounds = (f+1);
var
     V-bag: shared set of {v1: value, T1: Time};
     T-bag: shared set of Time; msg: M;
     default, v2: value; T2: Time; s: integer;

function TC: boolean;
  begin TC := ( 1 ≤ s ≤ (f+1) ) end;

cobegin
 begin
  cycle
1)     receive(msg);
2)     s:=no-of-signatures(msg);
3)     if ({msg.v, msg.Ts} not in V-bag)
4)       then if authentic(msg) and TC
5)             then begin
6)                 store({msg.v, msg.Ts}, V-bag);
7)                 if msg.Ts not in T-bag then store(msg.Ts, T-bag);
8)                 if s < maxm-rounds
9)                   then sign-and-send(msg)
                  end
  endcycle
 end

//

 begin
  cycle
10)     for any T2 in T-bag
11)       if clock.get = T2 + Δ
            then begin
12)             if {v2, T2} unique in V-bag
13)               then decide(v2, T2)
14)               else decide(default, T2);
15)               V-bag:= V-bag - {all {v2, T2} in V-bag}
16)               T-bag:= T-bag - {T2}
              end
  endcycle
 end
coend.
```

Correctness Of Algorithm AV

Theorem 3.5

Any execution of the above algorithm meets the unanimity and validity conditions for $\Delta = (f+1)d+e$ in the presence of at most f, $f<(n-1)$, distinct processors suffering value faults.

Proof

When the sender is non-faulty, the theorem is true by assumptions A1, A2, and A3. Suppose that the sender is faulty. As in theorem 3.3, we will say that a processor "obtains" $\{V,T_s\}$, when it stores $\{V,T_s\}$ in its V-bag. A non-faulty receiver processor obtains $\{V, T_s\}$ by receiving an authentic message with timestamp T_s and with s, $1 \le s \le f+1$ distinct processor signatures. The unanimity condition is shown to be met by showing that for any $\{V,T_s\}$, it is not possible for one non-faulty receiver processor to obtain $\{V,T_s\}$ before its clock time $T_s + \Delta$ and another non-faulty receiver processor not to obtain $\{V,T_s\}$ before its clock time $T_s + \Delta$.

Let p and q be any two non-faulty receiver processors. Suppose that p obtains a $\{V,T_s\}$ by receiving a message with s, $1 \le s \le f+1$, processor signatures and that q does not receive the message and does not contain $\{V,T_s\}$ in its V-bag. Since faulty processors do not fail by producing untimely responses, processor p must have received the message not later than $T_s + sd$ according to sender's clock (due to A3) and not later than $T_s + sd+e$ according to any receiver processor's clock (due to A1). If $s \le f$, q will obtain $\{V, T_s\}$ through the message signed and relayed by p before its clock time $T_s + (s+1)d+e \le T_s + \Delta$. If $s=(f+1)$, since there can be at most f faulty processors, the processor from which p received the message must be non-faulty. Therefore, it is not possible for q not to have obtained $\{V,T_s\}$ before its clock time $T_s + \Delta$.

Thus, for any message timestamp, say, T_s stored in T-bag, all non-faulty receiver processors will have identical entries in V-bag at their clock time $T_s + \Delta$. Hence the unanimity condition and the theorem.

3.9. Timing Fault and Emission Fault

In sections 3.5 and 3.6, it was observed that the algorithms tolerant of consistent timing faults and consistent emission faults have been designed without making use of consistency in processors' timing and emission failures respectively. Therefore, the algorithm for timing faults, AT, becomes the algorithm tolerant of consistent timing faults in processors, ACT-1; and, the algorithm for emission faults, AE, becomes that for consistent emission faults ACE-1.

3.10. Byzantine Fault

Faults of any type will form a proper subset of Byzantine faults that can cause a processor to fail in any manner. The Byzantine fault tolerant algorithm presented below as algorithm AB is a variant of Byzantine fault tolerant protocol presented in [Crist85]. It requires $(f+1)$ rounds of message exchange between processors. In each round, messages to be sent are signed or over signed and messages received are verified for authenticity and timeliness. Each receiver processor maintains V-bag and T-bag which are, as before, shared by two concurrent processes in a mutually exclusive manner. With Δ being $(f+1)(d+e)$, at clock time $T_s + \Delta$ a receiver processor inspects its V-bag for the value or values stored for timestamp T_s that has been stored in the T-bag. If only one value is present, decision is taken on that value; otherwise, default decision will be made.

3.10.1. Algorithm AB for Byzantine Faults

The type of messages exchanged between processors and the algorithm executed by the sender are the same as ACE-1. The algorithm executed by a receiver processor is given below.

```
receiver:
 const Δ = (f+1)(d+e); maxm-rounds = (f+1);
 var
     V-bag: shared set of {v1: value, T1: Time};
     T-bag: shared set of Time; msg: M;
     default, v2: value; Tr, T2: Time; s: integer;
 function TC: boolean;
         begin TC := ( 1 ≤ s ≤ f+1 and
                     msg.Ts - se ≤ Tr < msg.Ts + s(d+e))
         end;

cobegin
 begin
  cycle
1)    receive(msg);
2)    Tr:= clock.get;
3)    s:=no-of-signatures(msg);
4)    if ({msg.v, msg.Ts} not in V-bag)
5)      then if authentic(msg) and TC
               then begin
6)               store({msg.v, msg.Ts}, V-bag);
7)               if msg.Ts not in T -bag then store(msg.Ts, T-bag);
8)               if s < maxm-rounds
9)                then sign-and-send(msg)
               end
  endcycle
 end
//
 begin
  cycle
10)    for any T2 in T-bag
11)      if clock.get = T2 + Δ
          then begin
12)         if {v2, T2} unique in V-bag
13)         then decide(v2, T2)
14)         else decide(default, T2);
15)         V-bag:= V-bag - { all {v2, T2} in V-bag};
16)         T-bag:= T-bag - { T2 }
         end
   endcycle
 end
coend.
```

This algorithm is no different from algorithm ACE-1 that is tolerant of consistent emission and emission faults in processors. A Byzantine failure

that is not included in emission failures is a failure in which arbitrary messages are produced by faulty processors. So, Byzantine faulty processors can generate arbitrary messages during the execution of the algorithm and thereby can increase the message traffic between processors. An implementation of AB should take this increased message traffic into consideration by, for example, fixing a larger value of d in assumption A3 than the one chosen for implementing ACE-1. An increase in the value d will increase the value of e, if clock synchronisation is achieved without using any time service external to the system. For given values of d and e which are chosen to meet assumptions A1 and A3 in the presence of Byzantine faulty processors, algorithm AB can be proved to be correct using the approach developed for ACE-1 (see also [Crist85] for a proof).

3.11. The Generic Algorithm

In this section, we present the generic algorithm from which any of the algorithm developed in previous sections can be derived by substituting certain variables with appropriate parameters which are tabulated following the presentation of the algorithm. The variables to be replaced are tc, maxm-rounds, and Δ; The variable tc represents the conditions in the boolean function TC of an agreement algorithm. These three variables take different values for different algorithms and are indexed by α which will represent the name of the algorithm that is to be derived from the generic algorithm. For example, the generic algorithm will represent agreement algorithm for value faults, when the variables tc(α), maxm-rounds(α), and $\Delta(\alpha)$ in the generic algorithm are substituted by respective expressions provided in the table for α = AV. The generic algorithm and the table will help the reader to compare the requirements of agreement algorithms

developed in previous sections.

In algorithms for consistent timing, timing, and omission faults and for overloaded processors, it is possible for a non-faulty processor to be able to decide on a message with timestamp T_s before its clock time $T_s + \Delta$. An agreement reached will be said to be eventual, if one non-faulty processor can decide in round i and another one in round j, such that $i \neq j$. In some executions of these algorithms, agreement reached can be eventual. It can be observed that in these algorithms, the types of faults considered do not involve value faults, or, if value faults are involved, a faulty processor is assumed not to be able to undetectably forge another processor's signature for a given message; and that a receiver processor is not required to maintain V-bag and T-bag. In algorithms where V-bag and T-bag are maintained by receiver processors, the decision for T_s will be taken at clock time $T_s + \Delta$ and the agreement reached will be called immediate agreement in which all receiver processors decide at the same round. For detailed descriptions of eventual and immediate agreements, we refer the reader to [Dolev82a].

In developing the generic algorithm, the possibilities of a receiver processor being able to reach eventual agreement in certain algorithms have been ignored to achieve uniformity between all algorithms. For algorithms where early decision is possible, the generic algorithm will indicate the worst case time duration necessary to reach agreement.

The type of messages exchanged between processors is taken to be of the following *record* structure:

```
message structure:
type M record
     v: value; Ts: Time; signatures: string of char
     end;
```

The message variable 'signatures' is taken to represent message signatures of either of the following two types: (i) RSA type of message signatures which are generated and authenticated using schemes such as RSA scheme (Rivest, Shamir, and Adleman scheme) in [Rives78]; signatures of this type are signer dependent and contents dependent; (ii) ID type of signatures which are only signer dependent and are taken to be processors' identifiers. A processor's identifier, as in omission fault tolerant, and timing fault tolerant algorithms, will be unique and be known to every other processor in the system. When message signatures of ID type are used, the instruction "sign-and-send(msg)" will imply that the message, msg, is sent, after the sending processor's identifier is appended to the variable 'signatures' of the message; and, the boolean function "authentic(msg)" will always return true. The algorithm executed by the sender and a receiver processor are as follows:

```
sender:
  var msg: M; local-value: value;
begin
   msg.v := local-value; msg.Ts:= clock.get;
   sign-and-send(msg)
end.

receiver(α: algm-name);
 var
      V-bag: shared set of {v1: value, T1: Time}; T-bag: shared set of Time;
      msg: M; default, v2: value; Tr, T2: Time; s: integer;
 function TC: boolean;
          begin with msg do TC:= tc(α) end;
cobegin
 begin
  cycle
1)    receive(msg);
2)    Tr:= clock.get;
3)    s:=no-of-signatures(msg);
4)    if ({msg.v, msg.Ts} not in V-bag)
5)      then if authentic(msg) and TC
              then begin
6)              store({msg.v, msg.Ts}, V-bag);
7)              if msg.Ts not in T-bag then store(msg.Ts, T-bag);
8)              if s < maxm-rounds(α)
9)               then sign-and-send(msg)
              end
  endcycle
 end
//
 begin
  cycle
10)     for any T2 in T-bag
11)       if clock.get = T2 + Δ(α)
           then begin
12)          if {v2, T2} unique in V-bag
13)          then decide(v2, T2)
14)          else decide(default, T2);
15)          V-bag:= V-bag - { all {v2,T2} in V-bag};
16)          T-bag:= T-bag - { T2 }
           end
  endcycle
 end
coend.
```

The Generic Algorithm.

The table showing the expressions for tc(α), maxm-rounds(α), $\Delta(\alpha)$, and the type of message signatures for various agreement algorithms is presented next.

α	tc(α)	maxm-rounds(α)	$\Delta(\alpha)$	type of message signature
ACO (consistent omission)	true	1	$(d+e)$	ID
ACV (consistent value)	true	1	$(d+e)$	ID
ACT-1 (consistent timing)	$(T_s - se \leq T_r < T_s + s(d+e))$	$(f+1)$	$(f+1)(d+e)$	ID
ACT-2 (processor overloading)	$(s=2 \text{ or } T_s - se \leq T_r < T_s + s(d+e))$	2	$(2+\theta)(d+e)$	ID
ACE-1 (consistent emission)	$(1 \leq s \leq (f+1) \text{ and } T_s - se \leq T_r < T_s + s(d+e))$	$(f+1)$	$(f+1)(d+e)$	RSA
ACE-2 (processor overloading)	$(s=2 \text{ or } T_s - se \leq T_r < T_s + s(d+e))$	2	$(2+\theta)(d+e)$	RSA
AO (omission)	true	$(f+1)$	$(f+1)d+e$	ID
AV (value)	$1 \leq s \leq (f+1)$	$(f+1)$	$(f+1)d+e$	RSA
AT (timing)	$(T_s - se \leq T_r < T_s + s(d+e))$	$(f+1)$	$(f+1)(d+e)$	ID
AE (emission)	$(1 \leq s \leq (f+1))$ and $(T_s - se \leq T_r < T_s + s(d+e))$	$(f+1)$	$(f+1)(d+e)$	RSA
AB (Byzantine)	$(1 \leq s \leq (f+1))$ and $(T_s - se \leq T_r < T_s + s(d+e))$	$(f+1)$	$(f+1)(d+e)$	RSA

Table 3.1. Expressions For The Family Of Algorithms.

3.11.1. Discussions

Complexities of Algorithms

Complexities of agreement algorithms for faults of related types (related by "proper subset of" relation defined in chapter 2) can be compared by considering the following factors in the algorithms: the requirement for the use of message signatures and authentication, requirement to verify the timeliness of a received message (by referring to a clock), and the maximum number of rounds of message exchanges between processors required to guarantee agreement. Among the algorithms developed for each fault type, algorithms ACO and ACV, respectively tolerant of consistent omission faults and consistent value faults, stand out to be the simplest of all for the following reasons: the sender need not sign the messages it sends; no receiver processor will be required to authenticate, or to verify the timeliness of, the messages it receives; any execution of the algorithm guarantees agreement by time interval Δ which is as small as $(d+e)$ and involves just one round of message exchange between processors in which only the sender sends its messages. Under these fault models, a faulty sender does not respond to different processors in different manner. Therefore a simple broadcast by the sender, which is necessary in any agreement algorithm, is sufficient to guarantee agreement.

Under consistent timing and consistent emission fault models, a faulty processor was seen to respond to different processors in different (timing) manner - timely for some and untimely for others. Therefore, algorithms ACT-1 and ACE-1 required $(f+1)$ rounds of message exchange between processors. ACE-1 is more complex than ACT-1, since it requires the use of signed messages. ACT-2 and ACE-2 are developed for overloaded processors

and by making assumptions which restrict an overloaded processor's ability to respond in different timing manner to different processors. They turn out to be two round algorithms and are more complex than ACO and ACV which are one round algorithms. ACT-2 and ACE-2 will be faster than ACT-1 and ACE-1 respectively, if $\theta<(f-1)$. Under consistent fault models, it can be observed that if faults of type A are a proper subset of faults of type B, then the algorithm tolerant of type A faults is less complex than that tolerant of type B faults, except when A is consistent omission and B is consistent value.

ACT-1 and ACE-1 have been developed without making use of consistency in processor's failures and therefore become AT and AE respectively. The omission fault tolerant algorithm, AO, is less complex than AT and AE and is more complex than ACO, since it requires at most $(f+1)$ rounds of message exchange and does not involve signed messages and timeliness checks. AV is more complex than AO and less complex than AE.

Though the Byzantine fault tolerant algorithm, AB, is the same as AE, it was observed that a heavier message traffic may result during an execution of AB due to a Byzantine faulty processor's ability to generate arbitrary messages. In fact, it is not appropriate to compare the fastness of any two given algorithms only by considering the respective expressions for Δ, since message receiving and processing time varies depending on message traffic and on whether a given message is signed and/or is to be verified for timeliness. Due to the possibility of increased message traffic in the presence of Byzantine faulty processors, the value of d in assumption A3 may well be large and, therefore, AB may be slower than AE.

From these discussions, it can be observed that when faults of type A are a proper subset of faults of type B, then the algorithm tolerant of type A faults will be less complex than that for type B faults, except if A is consistent omission, consistent timing, consistent emission, and emission, when B is consistent value, timing, emission, and Byzantine respectively. In each of these four exceptional cases, type A fault tolerant algorithm is as complex as the algorithm tolerant of type B faults. These exceptional cases illustrate that an agreement algorithm does not necessarily become less complex, just because fault types considered are restricted to relatively "less severe" types.

Processors and Knowledge of Membership

The specification of the agreement problem "assumes" that every processor has identical knowledge of processors involved in reaching agreement. By this assumption, a processor was able to authenticate another processor's signed message, when signatures of RSA type are used; and, when signatures of ID type are used, a processor was considered to have a unique identifier that was known to every other processor involved in reaching agreement. Realising this assumption is easy in systems with a fixed configuration such as [Ezhil89] in which three processors are assigned to form a triple modular redundant (TMR) node until the end of the mission period. It becomes relatively difficult in systems where membership can change with respect to time as processors leave or join the system. In such systems, it should be ensured, following the occurrence of an event which affects the system membership, that all non-faulty processors have correct and consistent knowledge of the change in system membership in a bounded and known time interval. This problem is referred to as the group membership problem in [Crist88].

Given that non-faulty processors in a system have consistent knowledge of current system membership and can detect occurrences of events that affect system membership, group membership problem can be solved using a solution to the agreement problem. Using agreement protocols, three protocols are proposed in [Crist88] to solve membership problem in the presence of late timing faults in processors and communication medium.

3.12. Concluding Remarks

Agreement algorithms have been developed and presented for a range of processor fault types. When a given fault type is considered to be realistic for processors in a distributed system, the respective algorithm can be used to reach agreement. The algorithms presented here have been designed with one processor among n processors being designated as the sender. Extending these algorithms into agreement protocols where every processor can be a sender is a straightforward task.

In practical systems, a processor may be designed to have well-defined failure modes. For example, a fail-stop processor in [Schne84], is designed to restrict the failures to omission failures that are permanent and consistent. In designing processors with well-defined failure modes, the family of algorithms presented here illustrates any advantages, or lack of them, in choosing one particular failure type against another. For example, a choice of consistent timing failure type against timing failure type renders no simplification in the task of reaching agreement. However, the algorithms for overloaded processors indicate that agreement can be reached in a less complex manner (in at most two rounds, irrespective of f), if it can be guaranteed that processor overloading is the only consistent timing fault that can occur and that assumptions a1 and a2 can be realised.

Solving the agreement problem under different fault conditions, illustrating the relative complexities of various algorithms, and developing a generic algorithm are thus the main contributions of this chapter.

CHAPTER 4

EARLY STOPPING AGREEMENT ALGORITHMS UNDER OMISSION AND TIMING FAULT TYPES

4.1. Introduction

The agreement algorithms presented in the previous chapter were developed in the context of sender's broadcast time not known to receiver processors a priori. Algorithms tolerant of timing faults and omission faults require at most $f+1$ rounds of message exchange between processors to tolerate at most f faulty processors. In some executions of these algorithms, a non-faulty receiver processor was observed to be able to decide on the sender's value in less than $f+1$ rounds of message exchange between processors. During an execution, all non-faulty receiver processors decide on the sender's value in the first round when the sender is non-faulty, and decide before the $(f+1)$th round when the sender and fewer than $(f-1)$ receiver processors fail.

Dolev, Reischuk, and Strong [Dolev82a] were the first ones to investigate agreement algorithms whose execution can terminate taking fewer than the maximum number of rounds of message exchange required by the algorithms. They came up with two types of agreement that can be reached: *immediate* and *eventual.* This classification led the authors to the following definition of early stopping: an agreement algorithm tolerant of at most f

distinct processor failures is said to exhibit early stopping if all non-faulty processors are *guaranteed* to reach agreement in strictly less than the maximum number of rounds accommodated by the algorithm in all executions in which m, $m < f$, distinct processors fail. That is, every execution of an early stopping algorithm will guarantee early agreement, if the number of actually failed processors is less than the maximum expected.

In this chapter, we develop early stopping agreement algorithms under omission and timing fault types and in the context of sender's broadcast time being known to receiver processors a priori. When receiver processors know the sender's broadcast time beforehand, non-faulty receiver processors will be required to reach some decision, even when the sender fails to carry out the broadcast. Therefore, in the context of sender's broadcast time already being known to receiver processors, the omission fault tolerant, and timing fault tolerant algorithms of chapter 3 cannot guarantee agreement in less than $(f+1)$ rounds, when the actual number of faulty processors, m, is less than f. (Consider an execution in which the sender that is to send its value at some known time fails by sending no message to any processor. Non-faulty receiver processors will come to know that the sender did not broadcast its message, only at the end of the $(f+1)$th round, even if no receiver processor is faulty.) The problem of reaching agreement in the chosen context is described below:

A distributed system is considered to be made up of n, $n > 1$, potentially faulty processors that are capable of communicating with each other only by message passing. It is assumed that the communication medium is fault free and that faults occur only in processors. At most f, $f \leq n-2$, processors can be faulty. Any non-faulty processor cannot, however, ascertain

which other processors are faulty. Among these n processors in the system, one processor is designated as the sender and the others as receiver processors. The sender is to choose a value from many potential values and send it to all receiver processors at some time that is already known to all processors in the system. Agreement on the sender's value will be said to have reached, if

C1: all non-faulty receiver processors decide on the same value (unanimity); and,

C2: if the sender is non-faulty, all non-faulty receiver processors agree on the value the sender chose to send (validity).

In [Dolev82a], early stopping algorithms have been developed for Byzantine faulty processors capable of exhibiting arbitrary behaviour. In this chapter, we develop and present early stopping algorithms for omission and timing types of faults in processors. Under omission fault types, we will consider permanent omission faults (which cause a processor to stop functioning) and omission faults. Under timing fault types, we will consider timing faults and a particular case where overloaded processors fail in a consistently late timing manner. As in chapter 3, a known bound on message communication delays will be assumed and non-faulty receiver processors will reach agreement in a known and bounded time interval. Our permanent omission fault tolerant, and omission fault tolerant algorithms are found to be faster than those presented in [Hadzi84, Lampo84]. The timing fault tolerant algorithm and the consistently late timing fault tolerant algorithm are revised versions of algorithms presented in [Ezhil87, Ezhil86].

The algorithms presented here are developed with one processor being designated as the sender and other processors as receiver processors. Modifying these algorithms to the general context of every processor in the system being a sender and sending a value to all other processors is straightforward. An area of application for early stopping agreement algorithms will be distributed transaction commit where non-faulty processes on different sites have to agree unanimously on a commit or abort decision on the results of a transaction. These algorithms will enable the processes to reach agreement as quickly as possible with each non-faulty process knowing the time by which all other non-faulty ones will have reached the decision it has reached.

The rest of this chapter is organised as follows: The next section explains the assumptions involved in the design of the algorithms and the notations used in the presentation. Sections 4.3, 4.4, 4.5, and 4.6 respectively describe, and establish the correctness of, the permanent omission fault tolerant, the omission fault tolerant, the timing fault tolerant, and the consistently late timing fault tolerant algorithms. In each of these four sections, important observations about the respective algorithm are presented. In the development of consistently late timing fault tolerant algorithm, the assumptions presented in section 4.2 are weakened and the modified versions are stated before presenting the algorithm. Section 4.7 concludes the chapter.

4.2. Assumptions and Notations

We make two major assumptions in the design of our algorithms. These assumptions are identical to assumptions A1 and A3 of chapter 3 and are restated here for the sake of completeness.

Assumption A1:

At any given instant of real time, the observable difference between clock readings of any two non-faulty processors will be at most e.

A1 can be satisfied, when processors periodically execute some clock synchronisation algorithm such as [Crist86]. The term non-faulty in A1 is not necessary, when permanent omission faults and omission faults are considered for processors, since under these fault models, a processor fails only by not producing an expected response.

Assumption A2:

If, at time T, an event occurs in a non-faulty processor p and causes a message to be formed and sent to another non-faulty processor q, then that message will be received in q at time T_r, $T \leq T_r < T + d$, for some d, $d > 0$, where time is measured according to either processor's clock.

The assumptions underlying A2 are: non-faulty processors are reliably connected; d is fixed by considering the processing time taken when the occurrence of an event requires a processor to decide on sending messages, message routing and transmission in the communication medium.

Based on A1 and A2, the following can be stated:

If an event occurs in a non-faulty processor p at time T according to p's clock and causes a message to be formed and sent by p to another non-faulty processor q, then the message will be received in q at time T_r, $T - e \leq T_r <$

T + $d+e$, according to q's clock.

When processors are considered to have only permanent omission faults or omission faults, the term non-faulty is not necessary in A2. However, these assumptions need not be true for processors with consistent timing or timing faults. Under consistent timing fault assumptions, A1 will be extended to include even faulty (overloaded) processors. The assumptions in the design of consistently late timing fault tolerant algorithm are similar to, but weaker than, A1 and A2, and will be presented in section 4.6.

Without assumptions A1 and A2, a processor cannot decide whether a message has not been sent at all, or is yet to be sent by another processor. With processors not being able to solve this ambiguity, deterministic agreement algorithms cannot be designed [Fisch83b].

We explain a few notations that are frequently used in latter sections. The sender processor is denoted by s and its value to be broadcast to all receiver processors will be denoted by V. T_0 is the clock time when the sender has to broadcast its value, V. (T_0 is known to all receiver processors.) A processor denotes the set of all other processors in the system by P. N, F and S are any three subsets of P.

The messages exchanged by processors during an execution of an agreement algorithm are denoted by 'msg(v,i)' explicitly expressing two important items of their contents: v, the value a given message intends to deliver to a processor that receives it and i, an integer, is used to identify a given message to group it with other relevant messages.

In all the four types of faults concerned, a faulty sender does not fail by broadcasting different non-null values to different receiver processors and a

faulty receiver processor does not fail by trying to alter the contents of the message it relays. Hence the receiver processors, in any execution of the algorithms, have to decide either to agree on V or to conclude that the sender has failed to broadcast its value. We call the latter situation deciding on the 'default' value as the sender's value.

In presenting, and proving the correctness of, the algorithms, we make the following assumptions: a receiver processor has a "receive-buffer" that contains the messages received from other processors, and an object "clock" that represents its synchronised clock; a clock function 'get' will return the current clock value. A receiver processor is assumed to be able to establish the identity of the processor that sent the message it receives. It is also assumed to execute the statements of an algorithm in zero time. The last assumption will require an increase on the value of d to incorporate execution time overheads.

4.3. Permanent Omission Fault Tolerant Algorithm

A description of the early stopping algorithm tolerant of permanent omission faults is as follows.

In any execution of the algorithm, the sender processor, at its clock time T_0, is to broadcast its message msg(V,0) containing value V to all receiver processors. Each receiver processor waits for a message that contains a value to be decided on - i.e. either for a message msg(v=V,i) for some i, $0 \le i \le f$, or msg(v=default, i) for some i, $3 \le i \le f$. (The reason for the lower bound of 3 will soon become obvious.) On receiving such a message it decides on the value contained in the message and stops after

broadcasting msg(v,i+1) to other processors, if i $< f$. When such a message is not received, the processor continues to wait; while waiting, it executes the following steps:

S1 When the clock reads T_0 + i$d+e$ for i $\geq$ 1, it broadcasts to all other receiver processors a message msg('?',i) meaning "are you decided?" The set F_i, for any i, i$\geq$1, will contain processors that have been observed, at time T_0 +i$d+e$, to have failed. For i = 1 or 2, the set F_i will be formed to contain only the sender.

S2 Whenever it receives a message msg('?', j), for some positive integer j, it replies to the sender of msg('?', j) by sending a message msg('X', j+1) meaning "I have not yet decided".

S3 When its clock reads T_0 + i$d+e$, for i, i $\geq$ 3, it collects those receiver processors that replied by sending msg('X',(i-1)) in response to its message msg('?',(i-2)) into a set N and computes the set F_i = P-N.

S4 If $|F_i| \leq$ i-2 or $|F_i| = |F_{(i-2)}|$, it decides on the default value and stops after broadcasting msg(v=default, i) to all receiver processors. (The smallest value of i in a msg(default, i) is 3.)

If a processor cannot decide either by receiving a message containing another processor's decision or by computing appropriate F's until its clock reads T_0 + $(f+1)d+e$, it decides on the default value and stops the execution.

The different steps executed by a receiver processor are effectively combined in the following presentation of the algorithm in figure 4.1.

```
sender:
  begin
     wait clock.get = T0;
     broadcast(msg(V,0))
  end.


receiver-processor:
      i, j:integer;
  begin
     i: = 1;
     wait clock.get = T0 - e;
     flush receive-buffer;
     cycle
       while clock.get < T0 + i(d + e)
         do
1a)
         if (msg(v = V or default, j) in receive-buffer) then
           begin decide(v); if (j < f) then broadcast(msg(v,(j+1)); stop
           end;
         if (msg('?', j) in receive-buffer) then
              reply(msg('X',(j+1)), msg('?', j))
         od;
1b)
2a)
         if (3 ≤ i ≤ f) then
           begin N: = {processors whose msg('X',(i-1)) in receive-buffer};
             Fi:= P-N;
             if ( |Fi| ≤ (i-2) or |Fi| = |F(i-2)| ) then
               begin decide(default);
                   broadcast(msg(default,i)); stop
               end
           end;
         if (i ≤ 2 and i ≤ (f-2)) then Fi = {s};
2b)
3a)
         if i = (f+1) then
           begin decide(default); stop
           end;
3b)
         if i ≤ (f - 2) then broadcast(msg('?',i));
         i := i+1
     endcycle
  end.
```

Figure 4.1. The Permanent Omission Fault Tolerant Algorithm.

Explanation

By executing the algorithm, the sender broadcasts its value to all receiver processors at its clock time T_0 and each receiver processor, after initialising the variable i and emptying the receive-buffer, starts receiving messages msg(v, $j \geq 0$) at its clock time $T_0 - e$. In the first block of statements [from 1a to 1b], if a received message has v=V or v=default, decision is made on v; the execution is terminated by executing the "stop" statement after broadcasting msg(v, (j+1)), if $j < f$. Note that $j = f$ ($j > f$) would mean that the value of the message received has been relayed by (at least) f processors apart from the sender. Therefore, such a message need not be, and is not, relayed any further. If a received message is msg(v='?', j), then the statement "reply(msg('X', (j+1)), msg('?', j))" is executed to send a reply message, msg('X', (j+1)), to the processor from which msg('?', j) has been received. Appropriate mechanisms are assumed to have been implemented to avoid replying to the same processor more than once for a given value of j in msg('?', j).

In the second block of statements [from 2a to 2b], attempts are made at early stopping the execution for i, $3 \leq i \leq f$. The set F_i, for i = 1 or 2, is formed to contain only the sender, if $i \leq (f-2)$ i.e., $(i+2) \leq f$. An early stopping of the execution is not possible at clock time $T_0 + id + e$, for i = 1 or 2, and the F_i's formed at these clock time instants will not be needed at $T_0 + (i+2)d + e$, if $(i+2) > f$, since the execution is going to be terminated at clock time $T_0 + (f+1)d + e$. The third block of statements [from 3a to 3b] leads to termination of the execution in the worst case of the execution not having been terminated either by reception of a message with v=V or default or by computing appropriate F_i's.

If a receiver processor, following its activities at its clock time $T_0 + id + e$, $3 \le i \le f$, has not decided, it broadcasts msg('?', i), if $i \le (f-2)$, and continues the execution by incrementing the value of i. For $i > (f-2)$, msg('?', i) need not be broadcast, since the corresponding reply messages, if any, can only be guaranteed to be received after clock time $T_0 + fd + e$ at which time any attempt for early stopping would be of little use.

4.3.1. Correctness of the Algorithm

The algorithm is shown to be correct by establishing a series of lemmas. From here on, T_i is used to denote $T_0 + id + e$, for i, $i \ge 1$.

Lemma 4.1

Let p be an undecided non-faulty processor that computes F_i, for some i, $i \ge 3$, at its clock time T_i. Any processor that is an element of F_i must have halted functioning at some time T, $T < T_{(i-1)}$, according to p's clock.

Proof

Throughout this proof, we measure time according to p's clock. The F_i of p contains the sender processor which has obviously halted at some time T, $T \le T_0 + e < T_i$, $i \ge 1$. For every i, $i \ge 3$, the undecided non-faulty processor p broadcasts a message, msg('?',(i-2)), to all receiver processors at time $T_{(i-2)}$. By assumption A2, p's message msg('?',(i-2)) can be received by any other receiver processor at some time T_r, $T_r < T_{(i-1)}$ and the reply message msg('X',(i-1)), if sent, must be received by p before T_i. If any receiver processor in F_i had decided and stopped the execution without any failure by T_r, then p must have decided by T_i. But p is undecided. So, every processor in F_i must have failed and halted by T_r. Hence the lemma.

Lemma 4.2

Let p be an undecided non-faulty processor that computes F_i at its clock time T_i, $i \geq 3$. Any processor that is not in p's F_i must have remained undecided at time T, $T \leq T_{(i-2)}$, according to p's clock.

Proof

The lemma is obviously true for p which will not be in its F_i. Any other receiver processor that is not in p's F_i would have replied to p's message msg('?',(i-2)), only if it has been undecided at the time p's message was received. The p's message can be received by any other receiver processor, due to A2, at time T_r, $T_{(i-2)} \leq T_r < T_{(i-1)}$, according to p's clock. Hence the lemma.

Remark

The proof of this lemma makes use of two facts: a processor does not suffer any undue delay in replying to msg('?', j≥1) and does not produce messages containing incorrect information (such as indicating undecidedness when it is indeed decided). So, the above lemma will also hold true under omission fault assumptions.

Lemma 4.3

If an undecided non-faulty processor p finds $|F_i| = |F_{(i-2)}|$ at time T_i, according to its clock, for some i, $i \geq 3$, then no non-faulty processor could have decided, and can ever decide, on V.

Proof

Throughout this proof, it is supposed that p's clock will be used to measure time. Under permanent omission fault assumptions, a processor, on failing, halts for ever. Therefore, any F_i will be a subset of F_j for j, j > i ≥1 and if $|F_i| = |F_j|$, then $F_i = F_j$. According to the algorithm, p will have $F_1 = F_2 = \{s\}$. The condition that $|F_3| = |F_1|$ would imply that $F_3 = \{s\}$. This would mean that all receiver processors had, by lemma 4.2, remained undecided, when p's clock read T_1. Hence the sender must not have sent its message (by assumptions A1 and A2). Hence the lemma is true for i=3. With similar reasoning, the lemma can be shown to be true for i=4, though this situation can never occur, since the execution would be stopped for i=3 itself.

We prove the lemma for i, i≥5, by first showing that any processor, say q, that is in $F_{(i-2)}$ could not have sent a decision message containing either V or the default value to any processor, say r, that is not in $F_{(i-2)}$. Since q is in $F_{(i-2)}$, by lemma 4.1, it must have failed and halted at T, T < $T_{(i-3)}$. Assume that q has sent a decision message to r, before halting, that is before $T_{(i-3)}$. That message will be received by r before T+d, T+d < $T_{(i-2)}$. If r had been functioning until $T_{(i-2)}$, p should receive r's message and decide by $T_{(i-1)}$; but p is undecided at $T_{(i-1)}$. If, on the other hand, r had failed before $T_{(i-2)}$, then it would not have sent msg('X', i-1) in reply to p's msg('?', i-2); this means that r must have been counted in p's F_i. But $F_i = F_{(i-2)}$. Therefore, the hypothesis that q sent a message to r before $T_{(i-3)}$ cannot be true.

So, no processor in $F_{(i-2)}$ could have sent any message containing either V or the default value to any processor in P-$F_{(i-2)}$ at any time before $T_{(i-3)}$ and, by lemma 4.2, all processors in P-$F_{(i-2)}$, P-$F_{(i-2)}$ = P-F_i, must have

remained undecided at $T_{(i-4)}$ and until $T_{(i-2)}$. An undecided receiver processor can decide on V only by receiving a message containing V. Therefore, no processor in p's P-F_i can ever decide on V. A non-faulty receiver processor could not have decided on V before $T_{(i-4)}$, since p was undecided at $T_{(i-3)}$. Hence the Lemma.

Lemma 4.4

If a non-faulty undecided processor p finds $|F_i| \le$ (i-2) at its clock T_i, for some i, i $\ge$ 3, then no non-faulty processor could have decided, and can ever decide, on V.

Proof

When a receiver processor receives a message msg(V, j), j$\ge$1, that message must have been sequentially relayed by j distinct receiver processors after being sent by the sender. Under permanent omission (also omission) fault assumptions, faulty processors do not introduce any undue delay in relaying a received message. Therefore, by A1 and A2, when the clock of any non-faulty processor reads a value greater than or equal to $T_{(j+1)}$, j $\ge$ 1, no undecided processor can receive a message msg(V, j). By lemma 4.2, all processors in P-F_i must have remained undecided at p's clock time T, T $\le$ $T_{(i-2)}$. These processors can decide on V only by receiving a message containing V. If any processor in P-F_i is to receive msg(V, j), then that message has to come through processors in F_i (which also includes the sender) such that j $\le$ $|F_i|$ -1. But p finds $|F_i|$ $\le$ (i-2). This means j $\le$ i-3. Therefore, no processor in P-F_i can receive msg(V, j$\le$i-3), and can therefore decide on V, after p's clock time $T_{(i-2)}$. Hence the Lemma.

Lemma 4.5

When a non-faulty processor decides on the default value, no other non-faulty processor could have decided, and can ever decide, on V.

Proof

If a non-faulty processor can decide on the default value because either of the following conditions $|F_i| \leq$ (i-2) or $|F_i| = |F_{(i-2)}|$ has come true for some appropriate i, then, by lemma 4.3 or by lemma 4.4, the above lemma is true. Alternatively, a non-faulty processor can decide on the default value by receiving a message containing the default value from another receiver processor. Under permanent omission (also omission) fault assumptions, any response of a processor will be correct. Hence the other processor, while deciding on the default value, must have functioned like a non-faulty processor, correctly making, and broadcasting, the decision on the default value. Hence the lemma.

Theorem 4.1

In any execution of the algorithm, every non-faulty receiver processor reaches agreement on the sender's value in the presence of m, $m \leq f$, distinct processors out of n processors suffering permanent omission faults, and stops the execution not later than T_0 + min$\{(m+2)d+e,\ (f+1)d+e\}$ according to its clock after transmitting a total of O(nm) messages, where T_0 is the sender's clock time of the broadcast.

Proof

Consider an execution of the algorithm. If the sender is non-faulty, every non-faulty receiver processor would receive the sender's message by T_0+d+e according to its clock, due to assumptions A1 and A2. Hence the validity condition is realised. If the sender is faulty, it is impossible, by lemma 4.5, to have one non-faulty receiver processor deciding on V and another one deciding on the default value. Hence all non-faulty processors eventually decide either on V or on the default value. Thus the unanimity condition is met.

Suppose that the sender is faulty and that all non-faulty receiver processors decide on V. When a non-faulty receiver processor decides on V by receiving msg(V,i$\geq$1), at least i distinct processors (including the sender) must have failed. Therefore, all non-faulty receiver processors decide on V before their clock time $T_0 + (m+1)d+e$.

Suppose that all non-faulty receiver processors decide on the default value. Consider a non-faulty receiver processor that has $|F_i| \leq$ (i-2) at its clock time T_0 + i$d+e$, $3 \leq$ i $\leq f$. When i=3, the sender is faulty; if i>3, the processor must have had $|F_{(i-1)}| >$ (i-1) - 2, at its clock time T_0 + (i-1)$d+e$; thus it stops the execution at its clock time T_0 + i$d+e$, i $\leq |F_{(i-1)}| +$ 2 $\leq m+2$. Thus, when a non-faulty receiver processor has $|F_i| \leq$ (i-2), it stops the execution not later than its clock time $T_0+(m+2)d+e$. If it ever has $|F_{i-2}| = |F_i|$, it will stop the execution earlier than the time it would have stopped the execution, if it were to stop only by having $|F_i| \leq$ (i-2).

Consider a non-faulty receiver processor that decides by receiving a message containing the default value before its clock time T_0 +i$d+e$, i $\leq f+1$. If i$\geq$4, the processor must have had $|F_{(i-1)}| >$ (i-1) - 2 at its clock time T_0 + (i-1)$d+e$; thus, it decides before its clock time T_0 + i$d+e$, i $\leq |F_{(i-1)}| + 2 \leq m+2$; if i$\leq$3, then some receiver processor must have had F_3 containing only

the sender and therefore the sender must be faulty. Thus, when a non-faulty receiver processor receives a message containing the default value, it stops the execution not later than its clock time $T_0 + (m+2)d+e$. When a non-faulty receiver processor decides on the default value at its clock time $T_0 + (f+1)d+e$, it must have had $|F_f|>f$-2. This implies that $m>f$-2, and the processor stops the execution at its clock time $T_0 + (f+1)d+e$. Therefore, every non-faulty processor reaches agreement not later than its clock time $T_0+\min\{(m+2)d+e, (f+1)d+e\}$.

During an execution of the algorithm, a receiver processor is to broadcast either msg('?',i) or msg(V or default, i) to all other receiver processors at its clock reading $T_0+\mathrm{i}d+e$, $\mathrm{i}\geq1$, until a decision for agreement is reached. By this way, it broadcasts at most $(n-2)(\min\{m+2,f\})$ messages. Also, it has to reply at most $\mathrm{i}(n-2)$ messages of the form msg('X',i) before T_i, $\mathrm{i}\geq1$, until it stops. Hence it broadcasts a total of $O(nm)$ messages. Hence the theorem.

In the following, we make two observations about the algorithm. The first one concerns early stopping conditions used and the second one illustrates that this algorithm will not work in the presence of omission faults.

Observations

1. Early stopping conditions.

The above algorithm employs two early stopping rules: (i) $|F_i|\leq$ (i-2) and, (ii) $|F_i| = |F_{(i-2)}|$. Satisfying either condition leads to stopping the execution early. The first stopping rule verifies whether the size of F computed for every given i, i $\geq$ 3, is sufficiently small to decide on the default value. Therefore, during an execution, the smaller is the number of failed processors, the more quickly it is to be satisfied.

The second rule works by relating the size of F computed for a given value of i to that for (i-2), i $\geq$ 3. Hence, satisfying the second rule in an execution would require the number of failed processors to remain constant over a period of time i.e. no functioning processor should be observed to have failed during a given period of time. Therefore, during an execution, the sooner the faulty processors fail, the more quickly the second rule remains to be satisfied. In other words, if all faulty processors fail before $T_0 + id + e$, $i > 0$, according to a nonfaulty processor's clock, then the processor will stop the execution, by the second rule, not later than $T_0 + (i+4)d + e$, irrespective of the actual number of failed processors. To illustrate this, consider an execution with the following characteristics: exactly five processors (including the sender), named s, w, x, y, and z, fail; p is a non-faulty receiver processor and $f >> 6$.

Let the time be measured according to p's clock and $T_i = T_0 + id + e$, $i > 0$. In the execution, let s fail before T_0 without broadcasting its value and the other four receiver processors fail before T_1. The processor p will now compute all the five faulty processors in F_3, F_4, and F_5 respectively at T_3, T_4, and T_5. Hence, by the second rule, it stops the execution at T_5. If it were to stop by the first rule only, it should have stopped at T_7. If some of the faulty receiver processors, say, y and z, fail after T_1 and before T_2, then, by the second rule, p will stop the execution not later than T_6 with all the five faulty ones guaranteed to be computed in F_4, F_5, and F_6 respectively at T_4, T_5, and T_6.

Under permanent omission assumptions, a failed processor halts functioning for ever. In practical systems, it may often be the case that some processors have failed before the execution and no processor fails during the execution of the algorithm. In such cases, when $n > f >> 4$, the effect of the second stopping rule is more significant in bringing the execution to an

earlier stop (at T_5 of respective non-faulty processor's clock), irrespective of the number of processors that have halted functioning before the execution started.

2. The algorithm and omission faults.

In an execution of the above algorithm, processors are not guaranteed to reach agreement in the presence of omission faults.

Let us consider an execution in which q is another non-faulty processor in the context characterised in the previous observation. Let the five faulty processors, in this execution, fail before T_0, suffering omission faults and not reply to messages msg('?',i) of p and q at all. Let the sender's message be transmitted from s only to w, from w only to x, from x only to y, from y only to z, and from z only to q. Let q receive the message before its clock is to read T_5. If p does not receive q's message before its clock reads T_5, it will decide on the default value. So the agreement is not reached.

4.4. Omission Fault Tolerant Algorithm

It has been observed that the previous algorithm cannot be guaranteed to work in the presence of omission faults. A closer look into the early stopping conditions employed in the previous algorithm will reveal that only the early stopping condition, $|F_i| = |F_{(i-2)}|$, makes use of the permanent omission feature. The other one, $|F_i| \leq (i-2)$, works by recognising the situation in which the number of processors that did not respond at some time T_i, $i \geq 1$, during the execution, becomes so small and T_i is so late that no undecided processor will ever be able to receive a message from the failed ones. Thus, it can work in the presence of omission faults in processors. So the agreement algorithm for the omission fault type can be derived from the previous algorithm by simply removing the early stopping condition $|F_i| = |F_{(i-2)}|$. But for this, the omission fault tolerant algorithm is no different from the

previous one; in the following, the part of the algorithm containing the only appropriate early condition is presented:

```
2a)
            if (3 ≤ i≤ f) then
              begin N:= {processors whose msg('X',(i-1)) in receive-buffer};
                  Fi:= P-N;
                  if ( |Fi| ≤ (i-2))
                  begin decide(default);
                        broadcast(msg(default,i)); stop
                  end
              end;
            if (i ≤ 2 and i ≤ (f-2)) then Fi = {s};
2b)
```

4.4.1. Correctness of the Algorithm

Theorem 4.2

In any execution of the algorithm, every non-faulty receiver processor reaches agreement on the sender's value in the presence of m, $m \le f$, distinct processors out of n processors suffering omission faults, and stops the execution not later than $T_0 + \min\{(m+2)d+e,\ (f+1)d+e\}$ time according to its clock after transmitting a total of $O(nm)$ messages, where T_0 is the sender's clock time of the broadcast.

Proof

It has been observed that lemma 4.2 and lemma 4.4 are true under omission fault type as well. When "$|F_i| \le$ (i-2)" is the only early stopping condition in the algorithm, Lemma 4.5 will also be true for the omission fault tolerant algorithm. In the proof of theorem 4.1, only the early stopping condition "$|F_i| \le$ (i-2)" is used to establish that a non-faulty receiver processor decides on the default value not later than its clock time T_0 + $\min\{(m+2)d+e,\ (f+1)d+e\}$. Thus, by theorem 4.1, the above theorem will be true.

In the following, we make two observations about the above algorithm.

Observations

3. Fastness of the algorithm.

For the same m failures of respective types, the permanent omission fault tolerant algorithm will be either as fast as or faster than the above algorithm.

Given the same m failures of respective types, the executions of both the algorithms are to stop as early as $\min\{(m+2)d+e,(f+1)d+e\}$ after the start of the execution. The observation 1 illustrates that it is sometimes possible for the execution of the permanent omission fault tolerant algorithm to be stopped, due to the condition $|F_i| = |F_{(i-2)}|$ becoming true, earlier than it would have stopped, if it were to stop by the other condition. Thus, in some execution scenarios, the previous algorithm can be faster than the above one, for the same m failures of respective types.

Remark:

From the above observation, it can be noted that the permanent omission fault tolerant algorithm has been developed by making use of the special features of permanent omission faults over omission faults.

4. The algorithm and timing faults.

The above algorithm is not tolerant of timing faults.

The correctness of the above algorithm is based on the results of the lemma 4.4 in which it is stated that no processor can receive msg(v, $j \geq 0$) when, or after, the clock of any non-faulty processor has read T_0 + $(j+1)d+e$. Under the assumptions of timing faults, a faulty processor can be untimely in transmitting/relaying its messages. Therefore, the above statement, hence the lemma 4.4, will no longer hold true under timing fault

assumptions for processors.

4.5. Timing Fault Tolerant Algorithm

Since a faulty processor can fail by sending untimely messages, the messages received during an execution of the algorithm should be checked for timeliness and only the timely messages should be accepted by processors. Unlike in previous algorithms, a receiver processor will decide on the default value to stop the execution early, only by having the early stopping condition becoming true and not by receiving a message containing default decision of another processor. Upon receiving a timely message with the default value, a processor will only conclude that the processor that sent the message has stopped the execution of the algorithm. The reasons for this are as follows: a faulty processor may not have its clock in bounded synchronism as required by A1; therefore, during an execution, when a faulty receiver processor has its early stopping condition satisfied, a non-faulty receiver processor may be deciding on V. Due to this, any receiver processor should not decide on the default value just by receiving a message with the default value; however, it can conclude that the processor that has sent the message with the default value has stopped the execution. The algorithm is described below:

The receiver processors, while executing the algorithm, exchange messages in synchronised phases that are of uniform length $(d+e)$. With the sender processor broadcasting its message msg(V,0) at its clock time T_0, each receiver processor looks for a message, msg(V,i-1$\geq$0) or msg(default, i-1>0) to be received between $T_0+(i-1)(d+e)-e$ and $T_0 + i(d+e)$ according to its clock. On having received a msg(V,i-1), it decides on V at its clock time $T_0+i(d+e)$ and stops after broadcasting msg(V, i) if i $\leq f$. If the processor is undecided at its clock time $T_0 + i(d+e)$, then, it continues to execute the

algorithm by broadcasting msg('X', i) when i = 1 or by carrying out the following steps when i > 1.

S1: if $i = f + 1$, it decides on the default value and stops the execution; otherwise it executes S2 to S5.

S2: it puts those processors, if any, from which msg(default, i-1) has been received at clock time T, $T_0 + (i-1)(d+e) - e \leq T < T_0 + i(d+e)$, into the set S which will contain the set of all processors that are known to have stopped the execution;

S3: it collects those processors whose msg('X', i-1) has been received at clock time T, $T_0 + (i-1)(d+e) - e \leq T < T_0 + i(d+e)$, into the set N and computes F_i, F_i = P-N-S;

S4: if $|F_i|$ is less than i, then a default decision is made, a msg(default, i) is broadcast if $i < f$, and the execution is stopped;

S5: if $|F_i|$ is not less than i and if $i < f$, a message msg('X',i) is broadcast.

In the following presentation of the algorithm, a non-faulty processor is assumed to be capable of establishing the time of reception for every message that is in its receive-buffer.

```
sender:
  begin
      wait clock.get = T0;
      broadcast(msg(V,0))
  end.

receiver-processor:
    i: integer;
    M: set-of-messages; S: set-of-processors;

  begin
    i:=1; S := { };
    wait clock.get = T0 - e;
    flush receive-buffer;
    cycle
       wait clock.get = T0 + i(d+e);
       M:={msg(v,(i-1)) in receive-buffer and received at T,
                     T0+(i-1)(d+e)-e ≤ T < T0+i(d+e)};
1a)
       if (msg(v=V,i-1) in M) then
         begin decide(V);
             if (i ≤ f) then broadcast(msg(V,i)); stop
         end;
1b)
2a)
       if (2 ≤ i ≤ f) then
         begin
            S:= S ∪ {processors from which msg(default,(i-1)) in M was received};
            N:= {processors from which msg('X',(i-1)) in M was received};
            Fi:= P-N-S;
            if (|Fi| < i) then
              begin decide(default);
                   if (i < f) then broadcast(msg(default,i)); stop
              end
         end;
2b)
3a)
       if (i=(f+1)) then
         begin decide(default); stop
         end;
3b)

       if (i < f) then broadcast(msg('X',i));
       i:=i+1;
    endcycle
  end.
```

Figure 4.2. The Timing Fault Tolerant Algorithm.

Explanation

A receiver processor collects all timely messages in M at its clock time T_0 + i$(d+e)$, for i, 1 $\leq$ i $\leq$ $(f+1)$. In the first block of statements (1a to 1b), it attempts to decide on V and stop the execution. It executes the steps S2, S3, and S4 in the second block of statements. The processors whose messages containing the default value are in M, are added into, by "$\cup$" operator, the set S which is initially null. Thus the set S will contain all receiver processors that are known to have stopped the execution.

If the receiver processor remains undecided until its clock time $T_0+(f+1)(d+e)$, it decides on the default value and stops the execution in the third block of statements (step S1). A processor that could not decide at its clock time T_0 +i$(d+e)$, i $\leq$ f, continues the execution by incrementing the value of i; it also broadcasts msg('X', i), if i $<$ f.

4.5.1. Correctness of the Algorithm

In establishing the correctness of the algorithm, we let T_i = T_0 + i$(d+e)$, for i$\geq$0.

Theorem 4.3

In any execution of the above algorithm, every non-faulty receiver processor reaches agreement on the sender's value in the presence of m, $m \leq f$, distinct processors out of n processors suffering timing faults, and stops the execution not later than $T_0+(m+1)(d+e)$ according to its clock after transmitting a total of O(nm) messages, where T_0 is the sender's clock time of the broadcast.

Proof

Consider an execution of the above algorithm. When the sender is non-faulty, all non-faulty receiver processors receive msg(V,0) from the sender and stop the execution at T_1 after taking, and broadcasting, the decision on V. Hence, the validity condition is met.

Suppose that the sender is faulty. The unanimity condition is shown to be met by showing that in any execution of the algorithm, it is not possible for one non-faulty processor to decide on the default value and another one on V. Consider an execution in which p is a non-faulty processor that decides on the default value. When p decides at its clock time $T_{(f+1)}$, no other non-faulty processor can decide, or could have decided, on V, as there can be at most f faulty processors. Suppose that p decides on the default value at its clock time T_i, i < $(f+1)$, by having $|F_i|$ < i where F_i = P-N-S. F_i contains processors that have failed to send either a timely message to indicate their having stopped the execution or a timely message msg('X',i-1) indicating their undecidedness. The processors in S that have stopped the execution by broadcasting msg(default, j≤i-1) could not have, and will not, broadcast a message containing V. Faulty processors, under timing fault assumptions, do not fail by producing messages of incorrect information and will execute the algorithm correctly in value aspects. Therefore, any faulty processor in N must be undecided at the time of sending its msg('X', (i-1)) and will not accept any message msg(V, j <i-1) from that time onwards. As $|F_i|$ is less than i, a processor in F_i cannot broadcast msg(V, j≥i-1) and can, if at all, broadcast messages msg(V, j) only for j <i-1 which will be ignored by processors in N. Therefore, no non-faulty processor in N can decide on V. The set S cannot contain any non-faulty processor that had decided on V,

otherwise p would not be computing F_i at its clock time T_i. Thus, no non-faulty processor could have decided, and can decide, on V, when p decides on the default value.

Suppose that all non-faulty processors decide on V in an execution. When a non-faulty processor decides by accepting msg(V,i>0), there must be at least i faulty processors and therefore it decides at or before its clock time $T_{(m+1)}$. Suppose that a non-faulty processor decides in an execution on the default value at its clock time T_i, $2 \le i \le (f+1)$. If i=2, the sender must have failed. If i is greater than 2, it must have had $|F_{(i-1)}| \geq$ (i-1) at its clock time T_{i-1}; otherwise it would have stopped the execution at $T_{(i-1)}$ itself. This means that i $\le (m+1)$. Thus, all non-faulty receiver processors reach agreement no later than their clock time $T_{(m+1)}$.

It can be seen that a non-faulty receiver processor, during any execution, has to broadcast, until it stops, msg('X',1≤i<f) or msg(v=default, 2≤i<f) or msg(v=V, 1≤i≤f) to all other receiver processors at its clock time T_i, for every i, i>0. Hence the number of messages broadcast will be no more than $(n-2)(m+1)$. Thus the theorem is proved.

Observation

For the same m, $m>3$, failures of respective types, the permanent omission fault tolerant, and omission fault tolerant, algorithms are faster than the above algorithm.

Any execution of the above algorithm will be stopped as early as $T_0+(m+1)(d+e)$ according to a non-faulty processor's clock. Whereas the execution of the previous two algorithms can be expected to stop by T_0 +

$\min\{(m+2)d+e,(f+1)d+e\}$. Dolev and Halpern [Dolev84] established that e would be at least as large as $d/2$. Assuming that $e=d/2$, the first two algorithms can be seen to be faster than the third one, when $m > 3$.

Remark

The design of early stopping algorithms presented in [Lampo84, Hadzi84] is not effectively influenced by the no-untimely-response nature of permanent omission and omission failures. As a result, the permanent omission fault tolerant, and omission fault tolerant algorithms presented there turn out to be as slow as the timing fault tolerant algorithm presented here.

4.6. Consistently Late Timing Fault Tolerant Algorithm

This algorithm is developed by considering faulty processors to be overloaded processors which fail in consistently late timing manner. The assumptions A1 and A2 are modified to a1 and a2 respectively.

4.6.1. Assumptions

Assumption a1:

At any given instant of real time, the observable difference between the clock readings of any two processors will be at most e.

Assumption a1 requires the following: overloaded processors have non-faulty clocks; and, the periodic execution of clock synchronisation algorithm in processors be carried out at a sufficiently low level using high priority messages so that the higher level processing loads have little impact on the

execution of the algorithm.

Assumption a2:

In A2, only non-faulty processors are assumed to be reliably connected. Here, we assume all processors are reliably connected by the communication network. The message communication delay between two non-faulty processors is assumed, as in A2, to be bounded by d:

If, at time T, an event occurs in a non-faulty processor p and causes a message to be formed and broadcast, then a non-faulty processor q will receive it at time T_r, $T \leq T_r < T + d$ - where time is measured according to any processor's clock.

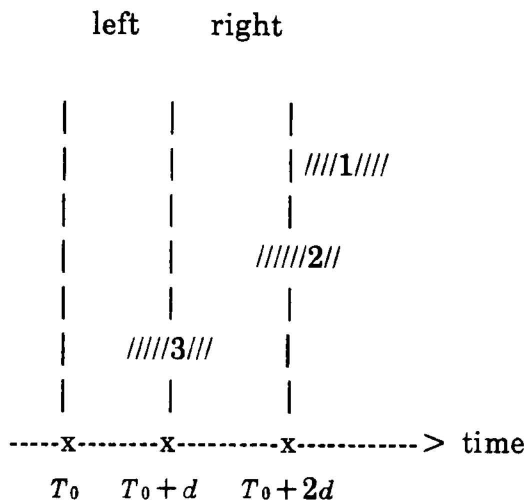
The message communication delay includes the skew interval (specified for replicated responses in section 2.3) and the bound d can be measured according to any processor's clock due to a1.

4.6.2. Development of the Algorithm

The algorithm is developed by exploiting the fact that an overloaded processor's late timing failures are consistent and therefore the processor's broadcast messages will be skewed within the specified interval. Suppose that the sender suffers a late timing failure in broadcasting a message at its clock time T_0 and that a non-faulty receiver processor, say, p receives the message at its clock time T_p. Since the sender's failure is consistent, any other receiver processor, say, q can *potentially* receive the sender's message at p's clock time T, such that $|T_p - T| < d$. If q is non-faulty, it will receive the message at T; if it is overloaded, it may suffer an unduly long delay in receiving the message, thus T represents the earliest time q could ever

receive the message.

Assuming, for a moment, that $e = 0$, we will refer to the diagram below. The three shaded bands represent three of many possible time intervals in which messages from an overloaded sender can be potentially received by receiver processors. They are of length at most d. The left window $[T_0, T_0 + d)$ is referred to as the acceptable time window during which the sender's message should be received to be regarded as timely. The interval $[T_0, T_0 + 2d)$ will be called the two-window time interval.



Referring to the above diagram, the following two properties can be stated:

Property pr1:

If a non-faulty processor does not receive the sender's message during the two-window time interval, then no other processor will receive it during the acceptable time window.

The first shaded interval (numbered 1 in the diagram) represents, and the second shaded interval can represent, a scenario where a non-faulty receiver processor does not receive the sender's message within the two-

window time interval.

Property pr2:

If a non-faulty receiver processor receives the sender's message during the two-window time interval, $[T_0, T_0+2d)$, then some receiver processor may well have received it during the acceptable time window.

In the diagram, the third shaded interval represents, and the second shaded interval can represent, a scenario where a non-faulty receiver processor receives the sender's message during the two-window time interval. When processors receive the message during the second shaded interval, no processor will have received the message during the acceptable time window.

Accounting for non-zero e, the acceptable time window and the two-window time interval in $pr1$ and $pr2$ can be redefined according to any processor's clock as $[T_0 - e, T_0 + (d+e))$ and $[T_0 - e, T_0 + 2(d+e))$ respectively.

An execution of the algorithm presented here proceeds in rounds of message exchange between processors and at most $f+1$ such rounds can take place. The sender's message initiates the first round. Any receiver processor that received the sender's message during the acceptable time window will relay the message at its clock time $T_0+(d+e)$ and the relayed message will initiate the second round. For the message of the second round, properties *pr1* and *pr2* will be valid, with acceptable time window and two-window time interval defined by replacing T_0 by $T_0+(d+e)$. In general, if a receiver processor receives a timely message during the ith round, $1 \leq i \leq f$, and initiates the (i+1)th round at its clock time $T_0 + i(d+e)$, its message will satisfy the two properties, for all $i < f$, when acceptable time window and two-window time interval are defined respectively as $[T_0 + i(d+e) - e,$

$T_0 + (i+1)(d+e))$ and $[T_0 + i(d+e) - e,\ T_0 + (i+2)(d+e))$. Based on this, the algorithm is developed here to ensure that, in the presence of at most f distinct processor failures, all non-faulty receiver processors

(i) decide on the default value, when all or some of them do not receive the sender's message during the two-window time interval; and

(ii) decide either on the sender's value or on the default value, when all of them receive the sender's message during the two-window time interval.

In (i), by *pr1,* no receiver processor can receive a timely message from the sender and therefore every non-faulty receiver processor's decision has to be on the default value. In (ii), if, as implied by *pr2,* there is a receiver processor that has received the sender's message during the acceptable time window, then that receiver processor will be considered to play the role of the sender in the second round; thus, an execution of the algorithm can unfold until at most $(f+1)$ rounds are carried out. At the end of each round, attempts will be made by receiver processors, using property *pr1,* to decide on the default value: a processor's clock time reaching $T_0 + i(d+e)$, $2 \le i \le f$, marks the end of the acceptable time window for the ith round and the end of the two-window time interval for the (i-1)th round; a processor decides on the default value at its clock time T_i, if it has received neither an ith round message nor an (i-1)th round message.

4.6.3. The Algorithm

The sender processor broadcasts its message, msg(V,i=0), at its clock time T_0. The algorithm to be executed by a receiver processor can be understood in three parts. In the first part, if a receiver processor receives msg(V,0) at its clock time T, $T_0 - e \le T < T_0 + (d+e)$, it accepts the

message, decides on V, and stops after broadcasting msg(V,i=1). If it does not receive msg(V,0) until its clock time $T_0 + (d+e)$, it continues to execute the algorithm. The second part of the algorithm describes the activities of such an undecided receiver processor from clock time $T_0 + (d+e)$ upto $T_0 + f(d+e)$.

For all i, $2 \leq i \leq f$, an undecided receiver processor waits for either msg(V,i-1) (a timely message in the ith round) or msg(V,i-2) (a late broadcast which could have been timely for other processors in the (i-1)th round) to be received until its clock reads $T_0 + i(d+e)$. If msg(V,i-1) is received before $T_0 + i(d+e)$, then the processor decides on V and stops the execution after broadcasting msg(V,i). If it has not received any msg(V,i-1) until its clock time $T_0 + i(d+e)$, it can decide either to continue or to stop the execution: if it has ever received msg(V,i-2) at any time T, $T_0 + (i-1)(d+e) \leq T < T_0 + i(d+e)$, then it continues the execution; otherwise, it decides on the default value and stops the execution.

When a receiver processor remains undecided by its clock time $T_0 + f(d+e)$, it executes the third part of the algorithm: if it receives msg(V,f) at clock time T, $T < T_0 + (f+1)(d+e)$, it decides on V and stops the execution; else it decides on the default value and stops the execution.

The three parts of the algorithm executed by a receiver processor are effectively combined and presented in figure 4.3.

```
sender:
begin
    wait clock.get = T0;
    broadcast(msg(V,0))
end.

receiver-processor:
   i: integer;
begin
    i:=1;
    wait clock.get = T0 - e;
    flush receive-buffer;
   cycle
     wait clock.get = T0 + i(d+e);

     if (msg(V,i-1) in receive-buffer) then
       begin decide(V);
         if (i ≤ f) then broadcast(msg(V,i));
           stop
       end;

     if (2 ≤ i ≤ f) then
       if (msg(V,i-2) not in receive-buffer) then
          begin decide(default); stop
          end;

     if (i = f+1) then
       begin decide(default); stop
       end;

     i := i+1
   endcycle
end.
```

Figure 4.3. The Consistently Late Timing Fault Tolerant Algorithm.

4.6.4. Correctness of the Algorithm

Lemma 4.6

In any execution of the algorithm, if the sender fails during its broadcast, then it is not possible for one non-faulty processor to decide on the default value and another non-faulty processor to decide on V.

Proof

Throughout this proof, the clock time $T_0+i(d+e)$ will be denoted as T_i, for $0 \le i \le (f+1)$. Let p and q be any two non-faulty receiver processors in an execution of the algorithm in which the sender is overloaded. Let p decide on the default value at its clock time T_i and $i=(f+1)$. Since there can be at most f overloaded processors, q could not have decided on V during the execution by accepting a msg(V,i$\le f$).

Suppose that p decides on the default value at T_i, $2 \le i \le f$. The lemma is shown to be correct by showing that it is not possible for q to decide on V during the execution. Processor p decides on the default value, if it does not receive either msg(V,i-2) or msg(V,i-1) at any time T $<$ T_i for i, $2 \le i \le f$. Since p did not receive msg(V,i-1) until its clock time T_i, q could not have decided on V by its clock time T_{i-1}. The fact that p did not receive msg(V, i-2) until its clock time T_i implies that no processor could have received msg(V,i-2) before its clock time T_{i-1}. (This implication was stated as property *pr1)*. Assume, to the contrary, that a receiver processor, say, r, r $\ne$ p, receives msg(V,i-2) at its clock time T, T $<$ T_{i-1}. Since overloaded processors' timing failures are consistent, p must receive msg(V,i-2) before $T+d$ according to r's clock. By a1, p should receive msg(V,i-2) before its clock time $T+d+e$, $T+d+e < T_i$. But p did not. Therefore, no receiver processor that was undecided at its clock time T_{i-2} would receive a timely msg(V,i-2). Any overloaded processor that receives msg(V,i-2) after its clock time T_{i-1} will identify the message to be late, due to a1, and will subsequently ignore the message. Thus, no receiver processor that was undecided at its clock time T_{i-2} will accept a msg(V,i-2) and therefore a message msg(V, j>i-2) will not be broadcast during the execution. So, if q is

undecided at its clock time T_{i-2}, it cannot decide on V later in the execution. Thus, it is not possible for q to decide, or to have decided, on V during the execution, when p decides on the default value at its clock time T_i, $2 \le i \le (f+1)$. Hence the lemma.

Theorem 4.4

In any execution of the algorithm, every non-faulty receiver processor reaches agreement on the sender's value in the presence of m, $m \le f$, distinct processors out of n processors suffering consistently late timing failures, and stops the execution by no later than $T_0 + \min\{(m+2),(f+1)\}(d+e)$ according to its clock after carrying out no more than one message broadcast, where T_0 is the sender's clock time of the broadcast.

Proof

Consider an execution of the algorithm. If the sender is non-faulty, all non-faulty receiver processors receive the sender's message before clock time $T_0 + (d+e)$, due to assumptions a1 and a2. According to the algorithm, all of them decide on V and stop the execution after broadcasting a message to every other receiver processor.

If the sender is overloaded, it is impossible, by lemma 4.6, to have one non-faulty receiver processor to decide on V and another one to decide on the default value. Hence all non-faulty processors eventually decide either on V or on the default value and thus reach agreement.

Suppose that all non-faulty processors decide on V. When a non-faulty processor decides on V by accepting a timely message msg(V,$i \geq 1$), there must be at least i overloaded processors that have failed during the execution. Therefore, it decides on V no later than its clock time $T_0 + (m+1)(d+e)$.

Suppose that all non-faulty processors decide on the default value in an execution. Let p be any non-faulty processor that decides at its clock time $T_0 + i(d+e)$, $2 \leq i \leq (f+1)$. When $i=2$, the sender must have failed. When $2 < i \leq (f+1)$, it must have received a late msg(V,i-3) before its clock time $T_0 + (i-1)(d+e)$ (otherwise, it would have stopped the execution at its clock time $T_0 + (i-1)(d+e)$). A receiver processor broadcasts a message containing V only after receiving a message with V from another processor; it stops the execution after broadcasting a message with V. Therefore, (i-2) distinct processors (including the sender) must have failed during the execution, for p to have received a late msg(V,i-3). Thus, p decides on the default value no later than its clock time $T_0 + \min\{(m+2)(d+e),(f+1)(d+e)\}$.

When a non-faulty processor decides on V by accepting a message, msg(V,$0 \leq i < f$), it stops the execution after broadcasting a message containing V; when it decides on the default value, it simply stops the execution. Thus every non-faulty processor carries out no more than one message broadcast during an execution. Hence the theorem.

Remark

The total number of messages sent by processors in an execution of the algorithm is zero if the sender does not broadcast, and at most $n(n-2)$ otherwise; it is the smallest compared with the other three algorithms. Due to

this low number of messages exchanged, the message traffic in the network may be light resulting in a smaller value of d - the bound on message communication delay between two non-faulty processors. If d is sufficiently small, for a given set of processors failing in an execution, the consistently late timing fault tolerant algorithm may turn out to be faster than the timing fault tolerant algorithm. When Byzantine faults were considered in [Dolev82a], $O(m(t^6+nf^2)+nf^2)$ was the message complexity of the algorithm.

4.7. Concluding Remarks

Early stopping algorithms for reaching agreement in the presence of permanent omission, omission, timing, and consistently late timing faults have been presented. In the consistently late timing fault model, faulty processors were considered to fail only due to processing loads and were assumed to have their clocks in bounded synchronism with non-faulty processor clocks. We observed that the algorithm tolerant of permanent omission faults can sometimes be faster than the omission fault tolerant algorithm, for the same number of failures of respective types. Since a permanent omission faulty processor fails by halting for ever, the earlier all faulty processors fail, the earlier the execution of the algorithm tends to stop. With omission fault assumptions, a processor, once failed, can later produce correct responses. So, stopping the execution of the omission fault tolerant algorithm gets delayed in direct proportion to the number of processors that failed. We also observed that the omission fault tolerant algorithm is faster than the algorithms tolerant of consistently late timing, and timing faults due to a timing faulty processor's failure of producing untimely responses. When failures are consistent, receiver processors need

to carry out no more than one message broadcast to reach agreement. The early stopping capabilities and message requirements of these algorithms reveal the fact that each algorithm has been designed by exploiting the distinct features that characterise the respective type of faults to be tolerated. This has been made possible by the fault classification developed in chapter 2. We believe that the algorithms presented here make a significant contribution to the area of early stopping algorithms for reaching agreement in distributed systems.

CHAPTER 5

PERFORMANCE EVALUATION

5.1. Introduction

The agreement algorithms presented in the previous two chapters have been designed with one processor in a distributed system being designated as the sender. Extending these algorithms into agreement protocols where every processor in the system can be a sender is a straightforward task. Agreement protocols can provide a fundamental service in systems with replicated processing. In this chapter we consider the problem of evaluating the performance of systems with replicated processing and majority voting. Performance evaluation of such systems taking into account of failure probabilities, failure modes, overheads of agreement and order protocols, etc., is a challenging task. We present some initial steps in this direction by considering a special case - that of a pipeline system. This architecture considerably simplifies the development of analytical models. We believe our techniques can be extended to more general architectures.

Replicated processing with majority voting - N modular redundant (NMR) processing - provides a powerful means of constructing highly reliable computing systems. A given computational task will be carried out concurrently in N, $N \geq 3$, processing modules which fail independently of each other. The results produced by these modules will be subject to a majority vote to obtain the final result. A majority vote is possible and the final

result obtained will be correct, if at least majority of the processing modules are functioning correctly and if correctly functioning modules produce identical results. NMR processing is of particular relevance to real time systems requiring a very high degree of reliability for two reasons: (i) the capability of masking the effects of failures of processing modules by majority voting means that there need not be a sudden degradation in response times due to failures; and (ii) majority voters are capable of tolerating arbitrary behaviour of failed modules.

A common form of NMR processing in practical systems is triple modular redundant processing (TMR processing) where three processors will be used to process the computational tasks concurrently. In this chapter we consider a distributed replicated system that is made up of a collection of TMR processing nodes connected in tandem. We present an analytical and simulation study of the performance of such a pipeline TMR system. In particular we examine the influence of majority voting times and processor failure rates on the response times of jobs processed by nodes in the system.

No performance evaluations of distributed replicated systems have been reported in the literature, although single node systems have been evaluated [York83, Pitte89]. In [York83], the authors evaluate the impact of voting on throughput, both analytically and experimentally. In [Pitte89], a TMR database system has been evaluated experimentally to examine its throughput. An algorithm is presented in [Abrah74] to evaluate the reliability of a distributed system of TMR nodes connected in any arbitrary manner. An empirical study of the performance of such a system will be the topic of our future work and the work presented here for a pipeline TMR system will, we hope, serve as a step towards that.

The type of system we are studying here is of practical interest. Special purpose multiple processor fault tolerant architectures with processors connected in the form of pipelines, rings, two and multidimensional arrays are finding widespread applications in avionics, image processing and process control fields (e.g. [Theur86, Harpe88, Iacop89, Napol89]). Thus there is every reason to evaluate the performance of such systems, examining in particular the impact of critical system parameters on the response times of jobs.

The chapter is organised as follows. Some interesting features of replicated processing systems are discussed in general terms in section 5.2. The factors which influence performance are mentioned there. The particular pipeline models that are considered for the purpose of performance evaluation are described in section 5.3. Two evaluation methods are employed: analytical approximations and computer simulations. The analysis, which is quite simple but nevertheless effective, is presented in section 5.4. Section 5.5 reports on a number of experiments where the models are simulated for different values of the parameters and the simulation results are compared to the analytical approximations. The latter are shown to be capable of predicting performance with sufficient degree of accuracy. Conclusions from this work are drawn in section 5.6.

5.2. Distributed Replicated Processing Systems

We start by considering a pipeline distributed processing system without replication. We assume that such a simplex system consists of a number of processors connected by a communication subsystem. Each processor is capable of performing one or more system functions (for example, in an avionics system, such functions could be sensor related processing,

flight path related processing and so forth). The environment of the system consists of a set of initiators (the entities that demand services from the system at arbitrary times). A service request from an initiator gives rise to a job which requires processing at the processors in sequence; at any time there could be several such requests being processed by the system. Suppose that a job requires processing at processors $P_1, P_2, \ldots, P_N$, in that order, before it completes. Then the end-to-end delay for that job - its sojourn time in the system - is composed of waiting and service at P_1, followed by a transmission delay from P_1 to P_2, etc., until service is completed at P_N. A message passed from P_i to P_{i+1}, containing the relevant state information necessary for processing a job at P_{i+1}, will be termed a 'task message'.

The replicated version of the above system is assumed to work as follows. Each system function will be performed by an ensemble of 3 processors. This ensemble of processors will become a triple modular redundant node, or TMR node for short. If at least two processors of a TMR node are functioning correctly and are producing identical results, then their results would constitute a majority. Subsequently, at most one processor failure can be tolerated within a node.

Consider now the processing of a job in a replicated system where the nodes 1 to N are visited in that order. Each of the 3 processors in the first node receives a separate version of the job and works on it independently of the others. Those versions will be referred to as 'siblings'. When a sibling is completed by a processor, 3 copies of the resulting task message are sent to node 2. Thus, each processor in node i, ($i>1$), receives 3 messages from node $i-1$. These messages are majority voted by the voter process of the processor; if a majority can be formed, then the voted sibling is processed and 3 copies of the task message are sent to node $i+1$ (except in the case of the

last node, where a single task message is sent by each processor to a final voter). The structure of each processor within a node is shown in figure 5.1.

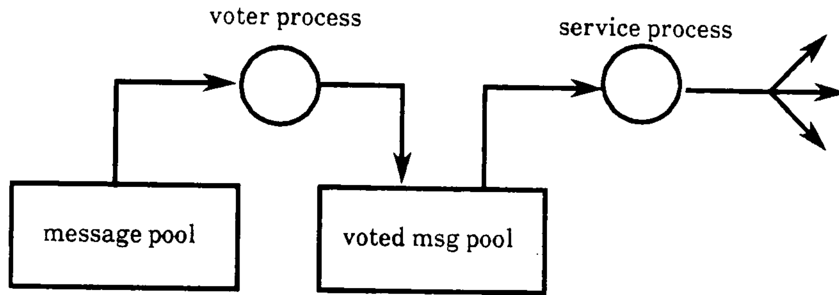


Figure 5.1. Voting and job processing at a processor of a TMR node.

Both the voter process and the service process of the processor maintain pools of buffers for storing incoming messages. The voter process performs voting as soon as it can form a majority on a given set of messages received from a sending node. The voted messages are stored in the voted message pool. The service process picks up a voted message from the pool and processes the request associated with it; if further processing at a subsequent node is required, then 3 copies of the task message are sent on, as stated earlier.

In general, it should be assumed that processors maintain states which affect the execution of jobs, and that the execution of a job by a processor can modify its state. Job processing is assumed to be deterministic, in the following sense: if processes of non-faulty processors have identical states and then process copies of a task message, then the final states of the processes will be identical. Assuming that service processes of all non-faulty processors of a node have identical initial states before any job

processing begins, we require that all these processes process voted messages in an identical order. This sequencing requirement is necessary in order to ensure that non-faulty processors of a node produce identical results. It is relatively easy to meet in specialised pipelined systems. However, when more general replicated systems are considered, some form of protocol is required to meet this requirement. For example, the processors of a node could execute an agreement protocol for selecting messages from the voted message pool in an identical order. Such agreement protocols can be easily developed from the agreement algorithms presented in chapter 3.

From the above discussion, we can identify a number of factors which may have an impact on the sojourn time of a job within a replicated system:

(i) Voting times: Voting consumes processing resources. If the time taken to reach a majority decision is relatively large compared to the actual processing time, then the sojourn time for a task is likely to be substantially larger than the corresponding time for the simplex system.

(ii) Processor failure rates: In a simplex system, a processor failure cannot be masked (the affected job will not complete). Whilst a replicated system can tolerate a bounded number of failures, such failures can affect sojourn times. Consider, for example, the progress of a job through two consecutive nodes, i and $i+1$. Moreover, suppose that the loads on the processors in these nodes are not identical (e.g., in addition to replicated processing, each processor in node i may have other, unreplicated processing functions to perform). Thus, the delay times of the job's siblings in node i may well be different. As a consequence, the task messages associated with those siblings arrive at a given voter in node $i+1$ at different times (these differences may be exacerbated by variations in message transmission delays). If there are no failures in node i, a voter

in node $i+1$ can form a majority as soon as 2 copies of the task message arrive. On the other hand, processor failures in node i can delay a voter in node $i+1$ by obliging it to wait for the slowest processor of node i to respond. Thus, even if failures are masked, this can be at the expense of increases in sojourn times.

(iii) Extra message traffic: A replicated system can generate more messages than its unreplicated counterpart. The impact of this increase will depend upon the network bandwidth, topology and architecture. For example, if nodes are connected by 3-redundant busses, a processor need only send a single task message on each of the 3 busses - thus a given bus will experience the same message traffic as in the simplex system, so the extra message traffic will have little impact on the sojourn time in this particular case.

(iv) Sequencing overheads: As stated earlier, processors of a node must be kept in step to prevent sequence failures. If a sequencing protocol is required, it can consume both processing and communication resources, thereby contributing to the sojourn time of tasks.

In this chapter we investigate the impact of first two factors on average sojourn time of a successfully completed job (also referred to as the system response time).

We shall assume that there is enough communication bandwidth available, so that factor three is of little significance. By making the following assumptions, the impact of factor four on system performance will also be removed: the system will be assumed to receive service requests from only one initiator; the initiator will be assumed to be reliable and its service requests will be assumed to arrive at the processors of the first TMR node in

zero time. Moreover, the task messages will be assumed to arrive at a destination node in the order in which they are sent by the source processor. These assumptions, together with the fact that we will assume our distributed processing system to be a pipeline, with a unique route followed by all jobs, implies that there is no need for special sequencing protocols.

In practical applications, the system can receive service requests from several replicated initiators. Under these circumstances, processors of the first TMR node can execute protocols to agree on, and order the requests. Execution of these protocols will provide an abstraction of a single "logical" initiator that delivers identical requests in an identical order to the non-faulty processors of the first node.

With assumptions to eliminate the need for the use of sequencing protocols, the message pools in figure 5.1 will be treated as FIFO queues. The voters will be assumed to have mechanisms to identify the siblings of a job, so that the siblings can be matched for voting.

5.3. Model Definition

We study pipeline systems consisting of N nodes, $1, 2, \ldots, N$. These nodes are visited by every job, in the order in which they are numbered. In the case of a simplex, unreplicated system, each node contains a single processor and an unbounded queue where jobs wait in order of arrival. In the replicated case, all nodes have degree of replication 3. The structure of a TMR node is illustrated in figure 5.2.

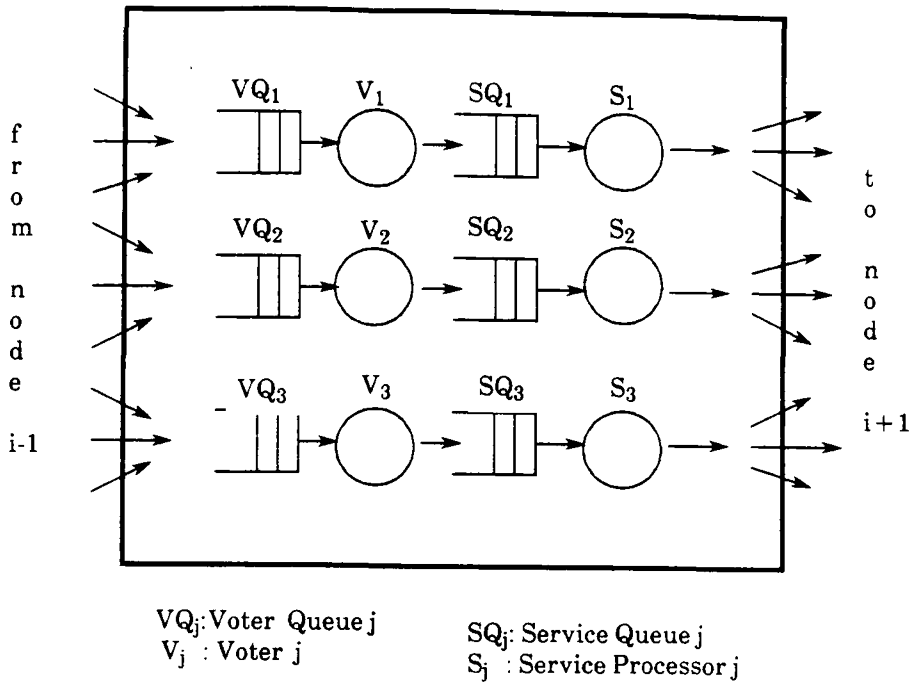


Figure 5.2. Model of a TMR node.

The voting and computational functions of each replicate are separated and are carried out by two independent servers. The latter are referred to as the 'voter' and 'service processor', respectively. There is an unbounded queue for each voter and each service processor. A voter queue in node i ($i>1$), receives job siblings (or, rather, task messages corresponding to job siblings) from each service processor in node $i-1$. As soon as 2 siblings of a job are present in the queue, the voter may attempt to vote on them. If that vote results in an agreement, then the job is passed on to the service queue; otherwise, the third sibling is awaited and the procedure is repeated. If, after all 3 siblings are present, there is still no agreement, then the job is effectively discarded.

Each service processor services the jobs in its queue in FIFO order. After a service completion, the processor sends 3 siblings (task messages) to

the 3 voter queues in node $i+1$. If the processor is non-faulty, those siblings will agree with others produced by non-faulty processors in node i; otherwise they will not.

There are two exceptions to the model in figure 5.2, namely nodes 1 and N. In node 1, there are no voters and voter queues; jobs coming into the system from the outside are replicated on arrival into 3 siblings which immediately join the 3 service queues there. On the other hand, the service processors in node N produce a single result after each service completion (rather than 3). There is a single final voter, with its queue, which arbitrates over the output of node N.

Jobs arrive into the system in a Poisson stream with rate a. Service times at the voters and service processors in node i are exponentially distributed with means $1/v_i$ (v_i is the average voting rate) and $1/s_i$ (s_i is the average service rate), respectively (the Poisson and exponential assumptions are of course not necessary when the system is simulated, but are needed for the analysis). The average transit time between nodes i and $i+1$ (for $i<N$) is $1/t_i$; it is independent of how many messages are being transferred in parallel. The distributions of the transit times are immaterial (but see the simplifying approximations in section 5.4). The transit times of service requests from the initiator to the processors of the first node, and of messages from the processors of the Nth node to the final majority voter are taken to be zero.

Fault assumptions.

The system reliability assumptions are as follows. Communication subsystem, the initiator, and the final majority voter are fault free; faults occur only in processors (where the term 'processor' is interpreted in the sense of

figure 5.1). Permanent and consistent value faults are the types of faults suffered by processors. The fault assumption on processors can be justified as follows: if processors are connected by (triplicated) busses, then processor failures can be consistent (see section 2.4). Since processors are assumed to maintain states, once a processor fails, its state gets corrupted, and the probability of the processor with corrupted state producing correct results is assumed to be negligibly small.

Faults occur in different processors - whether in the same node or in different ones - independently of each other. The intervals of non-faulty operation of processors in node i, called 'up-times', are exponentially distributed with mean $1/u_i$ (u_i represents the average processor failure rate). With respect to repairing faulty processors, two types of systems will be examined:

(i) Having once failed, a processor remains so until the end of the observation period.

(ii) A failed processor in node i is repaired, or is replaced by a new non-faulty one, after a delay, called 'down-time', whose average length is $1/d_i$.

The models resulting from these two assumptions correspond to systems without repair and systems with repair (cf. [Carte71, Avizi71]); they will be referred to as 'model 0' and 'model 1', respectively. In both the cases, it should be emphasised, the behaviour of a faulty processor differs from that of a non-faulty one only in that the former produces either no or consistently incorrect results.

One immediate consequence of the above assumptions is that the condition for stability, i.e. for the existence of a steady-state, does not involve the

up-time and down-time parameters. The transit parameters are not involved either, because there is no queueing for transmissions. Thus, the system is stable if

$$a < v_i \text{ and } a < s_i \text{ , } i = 1, 2, \ldots, N.$$

These conditions are assumed to hold.

The performance measures in which we are interested are: (a) The average sojourn time, W (interval between arrival into and departure from the system), for jobs that are completed successfully; and, (b) the distribution of the system operative state. The latter has a different interpretation in models 0 and 1, which will be clarified in section 5.4.

5.4. Analytical Approximations

To represent completely the state of a system employing N-modular redundancy, one would have to specify not only the numbers of jobs in each service and voting queues, and in transit between nodes, but also the individual identities of the jobs in all waiting, service and transit positions. This is necessary in order to keep track of the siblings of any given job, so as to account for matching delays at the voters. A representation of this type is of course possible. Given a suitable set of assumptions, it would lead to a vector-valued Markov process which could, in principle, be solved numerically. However, the size and complexity of the problem are such that, in practice, an exact solution is generally unattainable. The equations do not exhibit any 'nice' structure, such as local balance, that can be exploited in solving them. Simulations, on the other hand, while perfectly feasible, may be very expensive in both computer utilisation and elapsed time.

It is desirable, therefore, to devise an approximate solution method whereby estimates of performance measures can be obtained cheaply and quickly, albeit with some loss of accuracy. This is our objective in the present section. Certain simplifying steps are taken in the approximation. The first of these is to assume that all siblings of a job arrive at the service queues in a node at the same time. This is of course true in node 1, but not necessarily in subsequent nodes. However, since the voters act as synchronisation points for siblings, and since voting times and transit times between nodes are usually small compared to the service times, the assumption is not unreasonable. In fact, we shall see that even when voting and transit times are not small, the accuracy of the approximation is acceptable.

The second simplifying step concerns the distribution of the interval between the arrival of a sibling at a service queue, and its arrival at the following voter. That interval, which includes waiting, service and transit (excepting the latter in the case of node N), will be referred to as the 'passage time' for the given node. Now, it is well known (e.g., see [Mitra87]) that if jobs arrive into a single-server queue in a Poisson stream with rate a, and have exponentially distributed service times with mean $1/s$, then in the steady-state their response times are exponentially distributed with mean $1/(s-a)$. We shall assume that the addition of the subsequent transit time does not destroy that exponentiality (which it does, in general). The passage times for node i will be treated as exponentially distributed random variables with mean

$$P_i = \frac{1}{s_i - a} + \frac{1}{t_i} , \qquad (5.4.1)$$

where s_i is the service rate at node i, a is the (common) job arrival rate and $1/t_i$ is the average transit time from node i to node $i+1$. Note that $1/t_N = 0$ by

definition.

Assuming that at least 2 of the processors in a node are operating correctly, the following voter will be able to carry out a successful voting on a job when the first 2 of the job's correctly executed siblings complete their passage times through the node. To estimate the average time until the occurrence of that event, we shall use the following known result:

Let $X_1, X_2, \ldots, X_n$ be a sample of n i.i.d. (independent and identically distributed) random variables distributed exponentially with parameter μ. Consider the order statistics of that sample, $Y_1, Y_2, \ldots, Y_n$. In other words, Y_1 is the smallest of the X's, Y_2 is the second smallest, etc. Then the expectation of Y_k is given by $E(Y_k) = (1/\mu)H_{n,k}$, where

$$H_{n,k} = \sum_{j=0}^{k-1} \frac{1}{n-j} \text{ , } k=1, 2, \ldots, n. \tag{5.4.2}$$

Suppose that node i has c operative processors ($c=2,3$). Denote by $w_i(c)$ the average associated delay time, i.e. the average period between the arrival of a job's siblings at the service queues of node i and their arrival at the service queues of node $i+1$ (or the departure of the job from the system if $i=N$). Since the delay time consists of the second smallest passage time of the job's correctly executed siblings, plus the subsequent queueing and voting time, we can use the approximation

$$w_i(c) = P_i H_{c,2} + \frac{1}{v_{i+1}-a} \text{ ,} \tag{5.4.3}$$

where P_i and $H_{c,2}$ are given by (5.4.1) and (5.4.2) respectively, and v_{i+1} is the service rate of the voters in node $i+1$ (or of the final voter, if $i=N$).

Note that in the case of a TMR node with all three processors operating correctly, we have $H_{3,2}=5/6$, so that the first term in (5.4.3) is actually

smaller than the corresponding average passage time in an unreplicated system. If voting times are small compared to service and/or transit times, then the TMR system will perform better than the unreplicated one. This is a reflection of the fact which (5.4.2) quantifies for the exponential distribution, namely that the second best out of three realisations of a random variable tends to be better than a single realisation. On the other hand, if only two out of the three processors are operative, then $H_{2,2}=3/2$, and the replicated system has a worse performance than the unreplicated one. These phenomena will be illustrated by the experimental results in section 5.5.

Thus, the average delay time associated with a replicated node depends on the operative state of that node, i.e. on how many of its processors produce correct results. A node is said to be 'fully operative' if all three of its processors operate correctly. If only two of them do so, then the node is said to be 'partially operative'. The entire system is said to be 'operative' if every node is either fully or partially operative.

Let $\Omega=\{1,2, ..., N\}$ be the set of all nodes, φ be an arbitrary subset of Ω and $\Omega-\varphi$ be the complement of φ with respect to Ω. We shall say that the 'working state' of the system is φ, if the nodes in φ are fully operative and those in $\Omega-\varphi$ are partially operative. Denote by $q(\varphi)$ the conditional probability that the system is in working state φ, given that it is operative.

If the probabilities $q(\varphi)$ are known, then the average response time of a successfully completed job, W, which is the interval between the job's arrival into and departure from the system, can be approximated by

$$W = \sum_{\varphi\subset\Omega} q(\varphi)[\sum_{i\in\varphi} w_i(3) + \sum_{i\in\Omega-\varphi} w_i(2)] \ , \tag{5.4.4}$$

where $w_i(c)$ is given by (5.4.3). The first summation extends over all subsets of Ω. An alternative form of this expression is obtained by exchanging the order of summations:

$$W = \sum_{i=1}^{N} [w_i(3) \sum_{\varphi_i \subset \Omega} q(\varphi_i) + w_i(2) \sum_{\varphi^i \subset \Omega} q(\varphi^i)] \ . \tag{5.4.5}$$

Here, φ_i and φ^i vary over all subsets of Ω which do, and do not, contain node i, respectively.

Note, that expressions (5.4.4) and (5.4.5) rely on equilibrium being reached within each operative state of the system, i.e. on intervals between server breakdowns being much larger than the job interarrival, service, transit and voting times. This is usually true in practice.

Our object now is to determine the probabilities $q(\varphi)$. To simplify the development, we shall assume that the breakdown (and repair) rates for all processors are equal: $u_i = u$; $d_i = d$, $i = 1,2,...,N$. This implies that $q(\varphi)$ depends only on the size of φ, and not on its membership. Such an assumption is usually justifiable in practice, since the same (or similar type of) hardware is likely to be used at all nodes. Moreover, we shall see that it can be generalised, at the expense of considerably increasing the computational complexity of the solution.

With the assumption of equal breakdown (and repair) rates, it is sufficient to find the conditional probabilities, q_j, that there are j fully operative and $N-j$ partially operative nodes, given that the system is operative ($j = 0,1,...,N$). In terms of those probabilities, and denoting by $|\varphi|$ the number of nodes in φ, we can write

$$q(\varphi) = \frac{q_{|\varphi|}}{\binom{N}{|\varphi|}} \ . \tag{5.4.6}$$

Next, substituting (5.4.6) into (5.4.5) and counting the number of subsets of a given size which do, and do not contain node i, we obtain, after some algebra,

$$W = \frac{m}{N}\sum_{i=1}^{N} w_i(3) + \frac{N-m}{N}\sum_{i=1}^{N} w_i(2) , \tag{5.4.7}$$

where m is the conditional average number of fully operative nodes, given that the system is operative:

$$m = \sum_{j=1}^{N} j q_j$$

We are thus left with the problem of finding the distribution q_j and hence the mean m. In solving that problem, models 0 and 1 will be considered separately, since they require different treatment.

5.4.1. Operative State Distribution for Model 0

Recall that in model 0 the system starts in working state Ω (all nodes fully operative), and whenever a processor breaks down, it remains broken for ever. Eventually, a processor breakdown occurs in a node which is only partially operative; at the first such instant, the entire system becomes inoperative. However, we are interested in the conditional distribution of the working state, given that the system is operative. It is appropriate, therefore, to consider a 'modified' system which is equivalent to the original, but is never inoperative. That is, whenever the original system would become inoperative, the modified one re-enters working state Ω.

Clearly, the instants when the modified system enters working state Ω are regenerative points for the working state of that system. The operative period of the original system corresponds to one regenerative period of the

modified system. Therefore, the conditional probability $q(\varphi)$ in the original system is equal to the steady-state probability that the modified system is in working state φ.

With the above in mind, we define a Markov process which is in state j when, in the modified system, there are j fully operative and $N-j$ partially operative nodes, $j=0,1,...,N$. From state j, the process moves to state $j-1$ with rate $3ju$ (if one of the processors in the fully operative nodes breaks down), and to state N with rate $2(N-j)u$ (if a breakdown occurs in a partially operative node, triggering a regeneration). The corresponding state diagram is shown in figure 5.3.

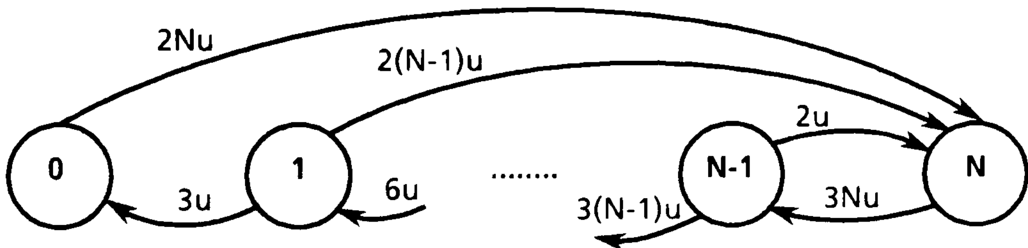


Figure 5.3. The State Diagram.

The probabilities q_j satisfy the balance equations

$$3jq_j = (2N-1+j)q_{j-1} \, , \; j=1,2,...,N \; , \tag{5.4.8}$$

together with the normalising equation

$$\sum_{j=0}^{N} q_j = 1 \, . \tag{5.4.9}$$

The solution of the recurrences (5.4.8) is of the form

$$q_j = q_0 \frac{1}{3^j j!} \prod_{k=0}^{j-1} (2N+k) \ , \tag{5.4.10}$$

while q_0 is obtained by substituting (5.4.10) into (5.4.9).

Another quantity of interest is the average operative period, A, of the original system. This is equal to the average first passage time from state N to state N of our Markov process. Since the average holding time in state N is $1/(3Nu)$, we can write

$$A = \frac{1}{3Nuq_N} \ . \tag{5.4.11}$$

The approach described here clearly generalises to the case where the processor breakdown rates are different for different nodes (but the same within a node). One would then have to consider a vector-valued Markov process $(b_1,b_2,\ldots,b_N)$, where b_i is 1 if node i is fully operative, 0 if partially operative. It would be easy to write a set of equations for the steady-state distribution of this process. However, solving those equations numerically would be a non-trivial matter, due to the large (2^N) number of states.

It should be emphasised that, in order to be able to apply the above results, the observation period, T (the mission time), should be large compared to A. When that is not true, it is necessary to estimate the transient performance of the Markov process. This can be done by employing the following approximation.

Let A_j be the average holding time in state j, i.e. the average uninterrupted interval during which there are j fully operative and $N-j$ partially operative nodes. These intervals can be found by noting that $q_j = A_j/A$, and therefore $A_j = q_j A$, $j = 0,1,\ldots,N$.

Now, remember that during an operative period, the process passes through the holding times A_N, A_{N-1}, ..., A_0, in that order. In order to account for the possibility that the mission time, T, expires during one of those holding times, define the quantities,

$$B_j = min(T, \sum_{k=0}^{j} A_{N-k}) \ , \ j=0,1,...,N \ . \tag{5.4.12}$$

Then the probability $\tilde{q}_j$, interpreted as the fraction of the mission time during which there are j fully operative and $N-j$ partially operative nodes, can be estimated from

$$\tilde{q}_{N-j} = \frac{B_j - B_{j-1}}{B_N} \ , \ j=0,1,...,N \ , \tag{5.4.13}$$

where $B_{-1}=0$ by definition.

5.4.2. Operative State Distribution for Model 1

In model 1, when a processor breaks down, it is repaired (or replaced by a new identical one), after a delay called the 'down-time'. Down-times are i.i.d. random variables with mean $1/d$. In this model, neither the up-times nor the down-times need to be exponentially distributed. However, we still assume that equilibrium is reached between consecutive changes in the system operative state.

In the long run, any given processor is operative for a fraction of time, α, given by

$$\alpha = \frac{1/u}{1/u + 1/d} = \frac{d}{u + d} \ . \tag{5.4.14}$$

Hence, the probability that a given node is fully operative, p_0, is given by $p_0 = \alpha^3$. The probability that the node is partially operative, p_1, is

$p_1 = 3\alpha^2(1-\alpha)$.

The probability that the system is operative, q, is obviously equal to

$$q = (p_0 + p_1)^N = \alpha^{2N}(3 - 2\alpha)^N. \qquad (5.4.15)$$

Finally, the conditional probability that there are j fully operative nodes and $N-j$ partially operative ones, given that the system is operative, is of the Binomial type:

$$q_j = \frac{1}{q} \binom{N}{j} p_0^j \, p_1^{N-j} \, , j=0, 1, \ldots , N. \qquad (5.4.16)$$

This analysis generalises easily to the case when the breakdown and repair rates are different at different nodes.

It is perhaps worth emphasising again that these are long-run results. They are valid only when the observation period, T, is large compared to the up-times and down-times. To analyse the performance of the system in the short run, under assumptions of exponentially distributed up-times and down-times, one would have to study the transient behaviour of the Markov process $\{J_t; t \geq 0\}$, where $J_t = j$ if at time t there are j fully operative nodes and $N-j$ partially operative ones ($j=0, 1, \ldots , N$). Another (absorbing) state, say '-1', would be added to represent an inoperative system. Then, for instance, the first passage time from state N to state -1 would correspond to the interval during which the system is operative. While not really difficult, such an analysis is not considered here.

Despite their simplicity and roughness, the approximations described here give quite accurate estimates of system performance measures. This will be illustrated in the following section.

5.5. Experimental Results

Experiments are carried out to assess the accuracy of the approximations involved in analytically estimating the system sojourn times. Both model 0 and model 1 are examined and the results are presented here. In an experiment, sojourn times obtained by simulations are compared with those obtained through analytical approximations. Comparisons between the performance of TMR and simplex (unreplicated) systems are also made at the same time.

An experiment normally involves fixing all parameters except the average voting time, and then simulating the simplex and the TMR systems for that set of parameters. Simulations of TMR system are repeated for different values of average voting time. These simulations do not assume any of the approximations described in the previous section but will correspond to the simplex or TMR system whose model is described in section 5.3. For model 0, a 'simulation' consists of 10 independent runs which differ only by the random number streams. In model 1, a single long run is made, divided into 10 portions with equal number of jobs completed in each portion. These samples of observations are used to obtain point estimates and confidence intervals for the average sojourn time, and point estimates for the fraction of time that the system is operative. The confidence intervals are not shown in the figures; their half-width is always less than 5% of the point estimate.

When the system simulated is a TMR one, the sojourn time for the concerned set of parameters can be estimated analytically using approximations described in section 5.4. The simplex system is of course a special case, with voting times equal to 0. In fact, the 'approximation' is then the exact steady-state result for queues in tandem:

$$W = \frac{N}{s - a} + \frac{N-1}{t}. \tag{5.5.1}$$

The relative error of the approximation, expressed as a percentage, is denoted by e:

$$e = \frac{W' - W}{W'} 100 , \tag{5.5.2}$$

where W' is the point estimate by simulation and W is the approximated one.

For $r=1$ (simplex system), $e \approx 0$ indicates that the system has reached steady-state during the simulation run. When $e<0$, the analytical approximation overestimates the average response time; otherwise it underestimates. In the following, unless specified, the value of e will be for $r=3$ (TMR systems).

In order to avoid having to deal with too many parameters, we have examined systems where all nodes are statistically identical:

$$s_i = s, 1 \le i \le N; \; t_i = t, 1 \le i < N; \; v_i = v, 1 < i \le N+1. \tag{5.5.3}$$

The actual choice of these parameters was influenced by the form of the expression for the passage times in (5.4.1). We considered cases where transfer times dominate queueing delays ($1/(s-a) < 1/t$), where the two are equal ($1/(s-a) = 1/t$), and where queueing delays dominate transfer times ($1/(s-a) > 1/t$). This is done by choosing different values of $1/s$ such that $1/s < 1/a$. In all simulations performed here, $1/a$ is fixed at 2 and $1/s$ is varied from 1.5, 1.0, and 0.5 to represent heavy, medium, and light processing loads at service processors respectively.

In each experiment, simulations of TMR system are carried out for at most eight values of average voting time. Thus an experiment will contain

at most nine simulations - one for the simplex system and the rest for the TMR system.

5.5.1. Results for Model 0

Two groups of results are presented for model 0. In the first group, the simulation time (or observation time in the simulations) is chosen to be longer than the operative periods of the system. Thus, these results are for a system which breaks down before the end of the simulation time. The second group of results is for a simulation period during which a TMR system generally remains operational, i.e., every node in the system is fully or partially operative. In the following, results of group 1 are presented.

5.5.1.1. Group 1

The results for a 5-node system are summarised in table 1. Simulation experiments are carried out with $1/u$ taking 20000, 15000, and 10000 and with simulation time being fixed at 20000. In all simulations, the system breaks down before the mission time of 20000 time units elapses. For this simulation time and for the range of values chosen for $1/u$ and N, the TMR system operates long enough for its steady state behaviour to be observed. In each experiment, simulations of TMR system are carried out for eight different values of mean voting time which vary from 0 to at least 50% of the average service time considered.

It can be observed in table 5.1 that the magnitude of e becomes smaller, when $1/s$ gets smaller in each of the cases for $1/(s-a)$ being less than, equal to, and greater than, $1/t$; it also decreases, as $1/t$ increases for a given $1/s$. This is explained in the following manner:

Model 0 (group 1): 1/a = 2; Simulation Time = 20000; N = 5.

1/u / {1/s,1/t}	20000	15000	10000
1/(s-a) < 1/t:			
{1.5,10}	$-8.0\% \le e \le -2.0\%$	$-8.0\% \le e \le -5.0\%$	$-8.8\% \le e \le -4.7\%$
{1.0,4.0}	$-4.5\% \le e \le -3.3\%$	$-4.9\% \le e \le -3.7\%$	$-5.2\% \le e \le -3.7\%$
{0.5,2.0}	$+1.8\% \le e \le +2.5\%$	$+1.9\% \le e \le +2.3\%$	$+1.8\% \le e \le +2.4\%$
1/(s-a) = 1/t:			
{1.5,6.0}	$-9.5\% \le e \le -6.6\%$	$-12.2\% \le e \le -3.0\%$	$-12.3\% \le e \le -3.4\%$
{1.0,2.0}	$-5.2\% \le e \le -4.3\%$	$-5.9\% \le e \le -4.6\%$	$-6.4\% \le e \le -4.7\%$
{0.5,0.67}	$-4.1\% \le e \le -3.5\%$	$-5.1\% \le e \le -2.9\%$	$-4.5\% \le e \le -3.0\%$
1/(s-a) > 1/t:			
{1.5,0.5}	$-18.3\% \le e \le -7.2\%$	$-19.9\% \le e \le -12.0\%$	$-17.3\% \le e \le -8.7\%$
{1.0,1.25}	$-7.6\% \le e \le -5.3\%$	$-7.5\% \le e \le -5.2\%$	$-8.1\% \le e \le -4.6\%$
{0.5,0.0}	$-5.6\% \le e \le -2.3\%$	$-6.8\% \le e \le -2.5\%$	$-4.4\% \le e \le -2.5\%$

Table 5.1. Accuracy of Approximations in Model 0 (Group 1).

In analytically estimating the TMR sojourn times, it is assumed that the passage times of task siblings at a TMR node are independent of each other. While the transfer times of task siblings are indeed independent of each other, their queueing times at the servers are not. However, that dependence is reduced when the load on the servers becomes lighter. Also its effect becomes less noticeable when the transfer times begin to dominate.

The results of those experiments with parameters $\{1/s=0.5,\ 1/t=2,\ 1/u=15000\}$, $\{1/s=1.0,\ 1/t=2,\ 1/u=15000\}$, $\{1/s=0.5,\ 1/t=0.67,\ 1/u=15000\}$, and $\{1/s=1.5,\ 1/t=0.5,\ 1/u=15000\}$ are shown in figures 5.4, 5.5, 5.6, and 5.7 respectively. The graphs indicated by $r=3$ represent TMR average sojourn times for different values of average voting time. The value of e for simplex system was so small in some experiments that both the estimates are shown by a single line (indicated by $r=1$).

In figures 5.4, 5.5, and 5.6, the simulation estimates of TMR sojourn time for zero voting time are larger than those of the simplex sojourn time. Recall that the average task completion time (the service time + transfer time) in one TMR node is analytically estimated to be 16.6% less, and 50% more, than the average task completion time in a simplex node, when that node is fully, and partially, operative respectively. When the simulation is run until the system breaks down, the value of m/N in equation (5.4.7) is 0.654 which is also the probability that a TMR node is fully operative given that the system is operative. Thus, when there is no voting in TMR system, the sojourn time can be expected to exceed the simplex sojourn time by

$$(1 - 0.654) * 50 - 0.654 * 16.6 = 6.44\%, \tag{5.5.4}$$

where the % difference is expressed with respect to the simplex sojourn time. In figure 5.4, the simulation estimate of TMR sojourn time is 8.8%

Model-0: 1/s = 0.5; 1/a = 2; 1/t = 2; up-time = 15000; Simulation Time = 20000; N = 5.

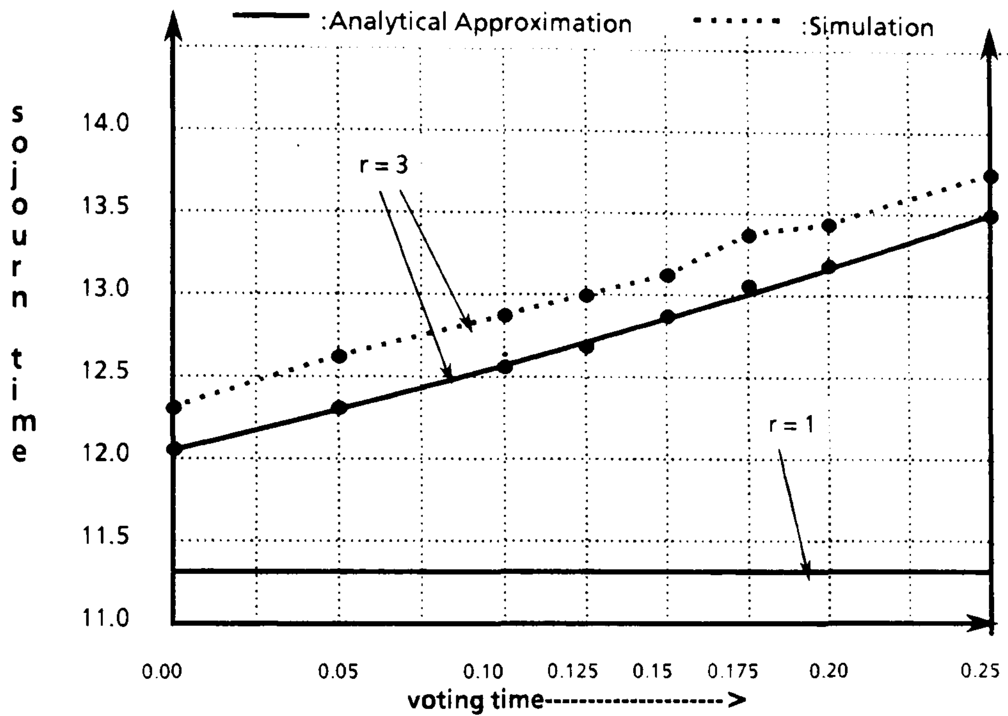


Figure 5.4. Sojourn Time Vs. Voting Time For 1/(s-a) < 1/t.

Model-0: 1/s = 1.0; 1/a = 2; 1/t = 2.0; up-time = 15000; Simulation Time = 20000; N = 5.

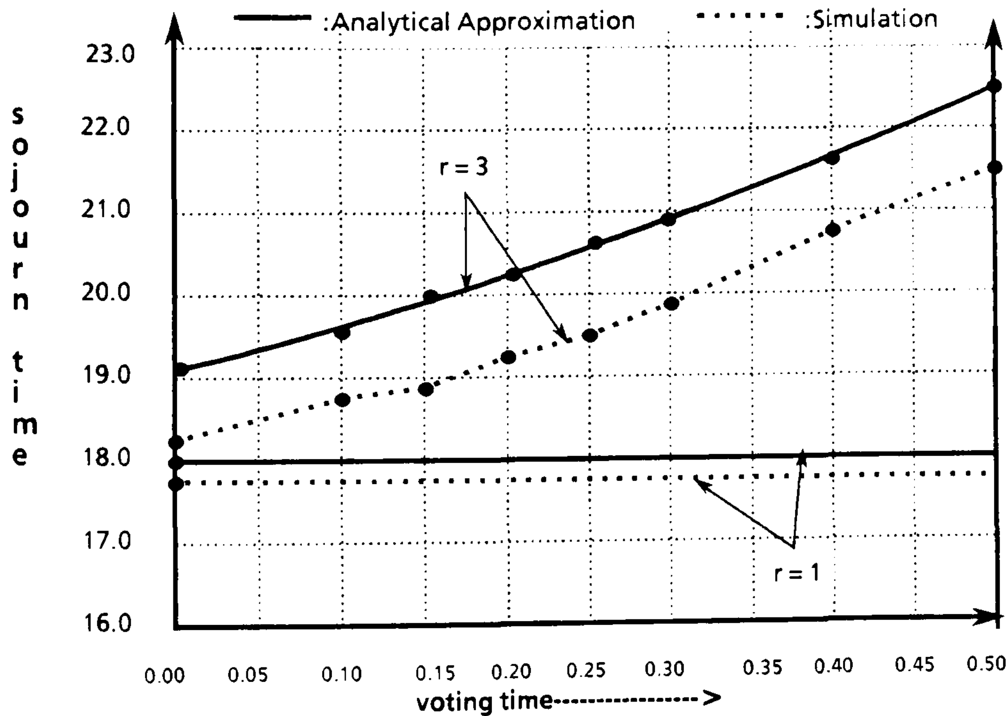


Figure 5.5. Sojourn Time Vs. Voting Time For 1/(s-a) = 1/t.

Model-0: 1/s = 0.5; 1/a = 2; 1/t = 0.67; up-time = 15000; Simulation Time = 20000; N = 5.

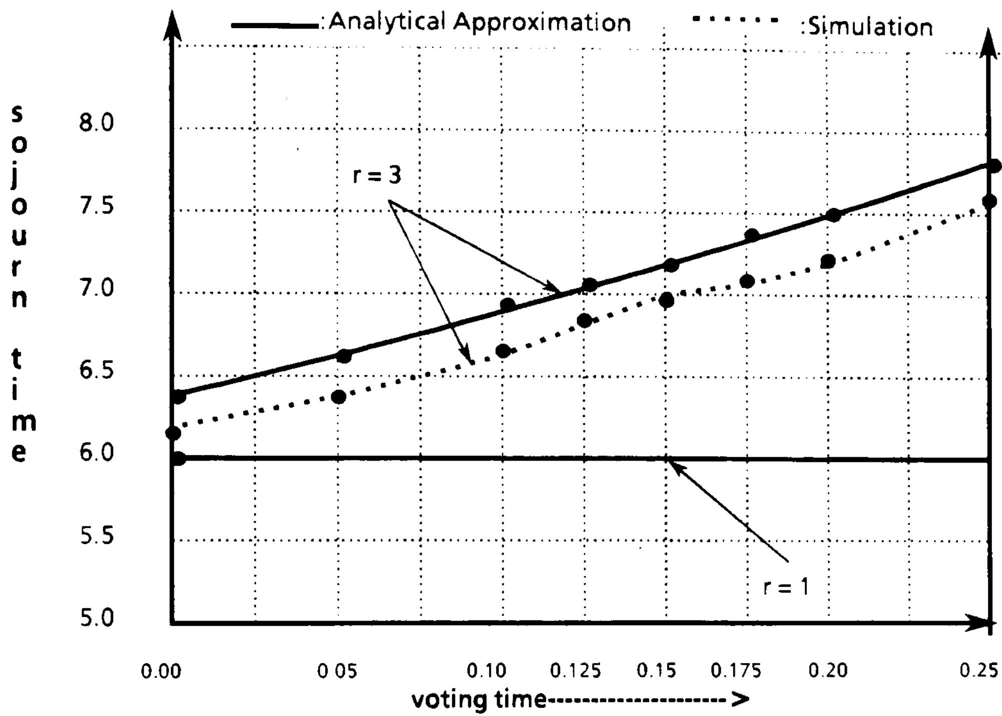


Figure 5.6. Sojourn Time Vs. Voting Time For 1/(s-a) = 1/t.

Model-0: 1/s = 1.5; 1/a = 2; 1/t = 0.5; up-time = 15000; Simulation Time = 20000; N = 5.

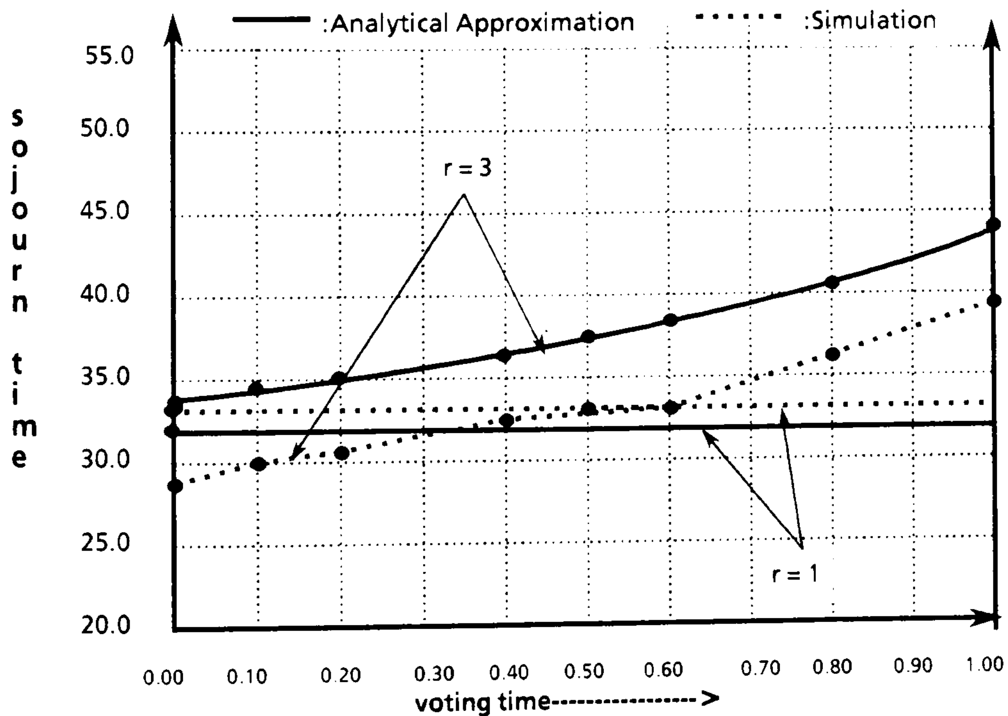


Figure 5.7. Sojourn Time Vs. Voting Time For 1/(s-a) > 1/t..

more than the simplex sojourn time. This percentage difference between the simulation estimates of TMR and simplex sojourn times becomes smaller in figures 5.5 and 5.6, and becomes less than zero in figure 5.7.

To observe the accuracy of approximations for different values of N, experiments were carried out for $N = 10$ and 3, and with the following sets of parameters: $\{1/s = 0.5,\ 1/t = 2,\ 1/u = 15000\}$, $\{1/s = 1.0,\ 1/t = 2,\ 1/u = 15000\}$, $\{1/s = 0.5,\ 1/t = 0.67,\ 1/u = 15000\}$, and $\{1/s = 1.5,\ 1/t = 0.5,\ 1/u = 15000\}$. When $1/s = 0.5$, e was positive and less than 2.8% for $N = 10$ and less than 4.5% for $N = 3$, and when $1/s = 1.0$, it was near zero (varying between -0.6% and 1.5%) for both values of N. For $1/s = 1.5$, the magnitude of e became larger: $-15.4\% \le e \le -8.7\%$ and $-7.5 \le e \le +2.6$ for $N = 10$ and 3 respectively, but still it is less than that for N=5. This observation, that the accuracy of the approximations improves for large and small number of nodes, is not intuitively obvious and we have no satisfactory explanation for it.

5.5.1.2. Group 2

In all simulation experiments in this group, simulation time is fixed at 2000. The values of $1/u$ considered for a 5-node system are 100000, 50000, and 25000. For $1/u = 100000$ and 50000, the TMR system suffered no failures, and tolerated failures, until the end of simulation, respectively. When $1/u$ was 25000, it broke down before the end of simulation period in only 2 out of 10 'simulation runs' of a simulation. Table 5.2 summarises the results of simulation experiments for a 5-node system.

The values of e in table 5.2 are largely positive implying that the approximations underestimate the TMR sojourn times. When it is assumed that all siblings of a job arrive at the service queues in a node at the same time, the approximation is optimistic in ignoring the variabilities in passage

Model 0 (group 2): 1/a = 2; Simulation Time = 2000; N = 5.

1/u \ {1/s, 1/t}	100000	50000	25000
1/(s-a) < 1/t:			
{1.5,10}	+4.5% ≤ e ≤ +7.0%	+4.5% ≤ e ≤ +9.5%	+5.8% ≤ e ≤ +8.6%
{1.0,4.0}	+6.6% ≤ e ≤ +8.9%	+8.2% ≤ e ≤ +9.9%	+7.8% ≤ e ≤ +9.4%
{0.5,2.0}	+7.0% ≤ e ≤ +8.1%	+8.5% ≤ e ≤ +9.6%	+8.3% ≤ e ≤ +9.3%
1/(s-a) = 1/t:			
{1.5,6.0}	+3.4% ≤ e ≤ +7.3%	+1.0% ≤ e ≤ +7.0%	+2.3% ≤ e ≤ +9.0%
{1.0,2.0}	+5.6% ≤ e ≤ +9.0%	+6.2% ≤ e ≤ +9.9%	+6.1% ≤ e ≤ +8.6%
{0.5,0.67}	+9.3% ≤ e ≤ +6.9%	+10.4% ≤ e ≤ +8.1%	+9.7% ≤ e ≤ +8.3%
1/(s-a) > 1/t:			
{1.5,0.5}	-2.2% ≤ e ≤ +3.4%	-4.1% ≤ e ≤ +1.8%	-8.0% ≤ e ≤ +5.6%
{1.0,1.25}	+3.8% ≤ e ≤ +6.8%	+3.8% ≤ e ≤ +7.3%	+4.9% ≤ e ≤ +7.2%
{0.5,0.0}	-1.3% ≤ e ≤ -0.1%	+0.3% ≤ e ≤ +2.3%	+0.08% ≤ e ≤ +1.7%

Table 5.2. Accuracy of Approximations in Model 0 (Group 2).

Model-0: 1/s = 0.5; 1/a = 2; 1/t = 2; up-time = 50000; Simulation Time = 2000; N = 5.

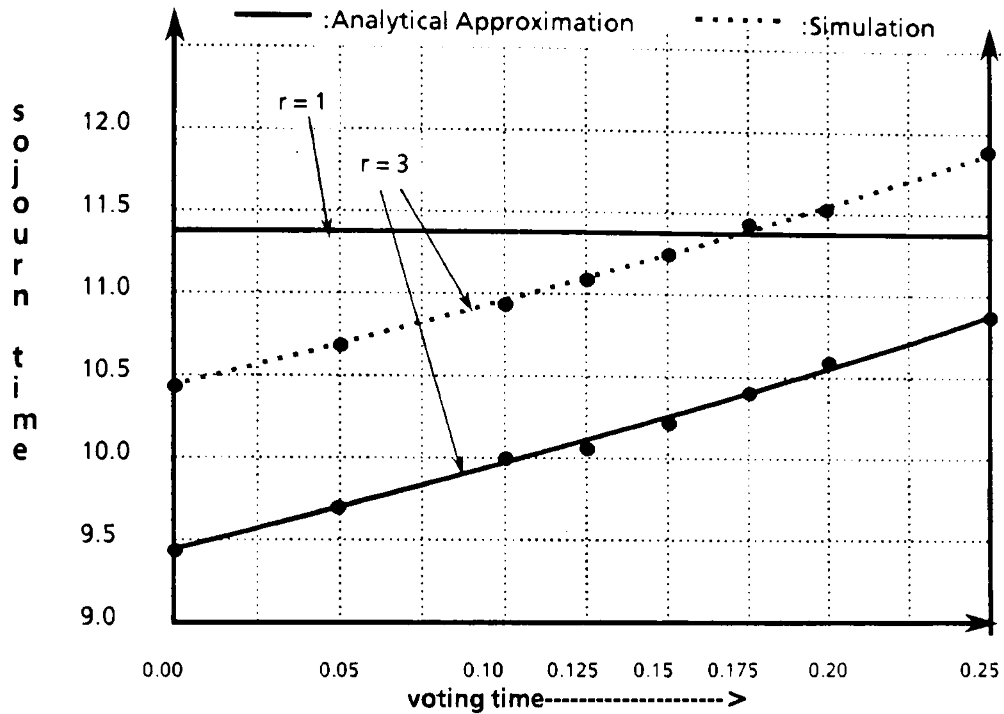


Figure 5.8. Sojourn Time Vs. Voting Time For 1/(s-a) < 1/t.

Model-0: 1/s = 1.0; 1/a = 2; 1/t = 2.0; up-time = 50000; Simulation Time = 2000; N = 5.

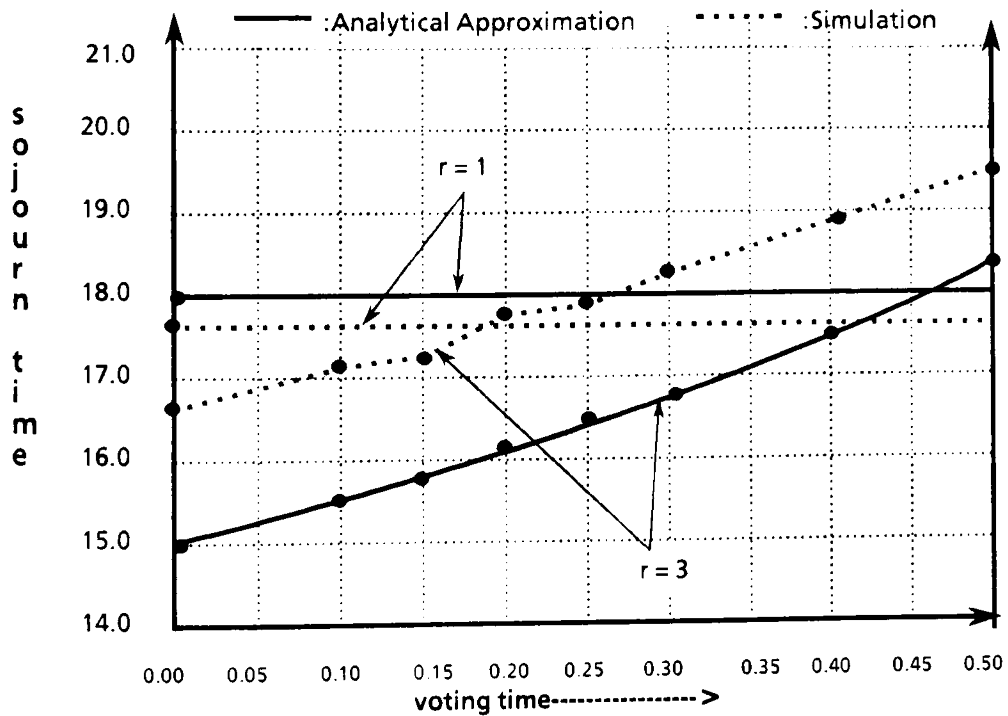


Figure 5.9. Sojourn Time Vs. Voting Time For 1/(s-a) = 1/t.

Model-0: 1/s = 0.5; 1/a = 2; 1/t = 0.67; up-time = 50000; Simulation Time = 2000; N = 5.

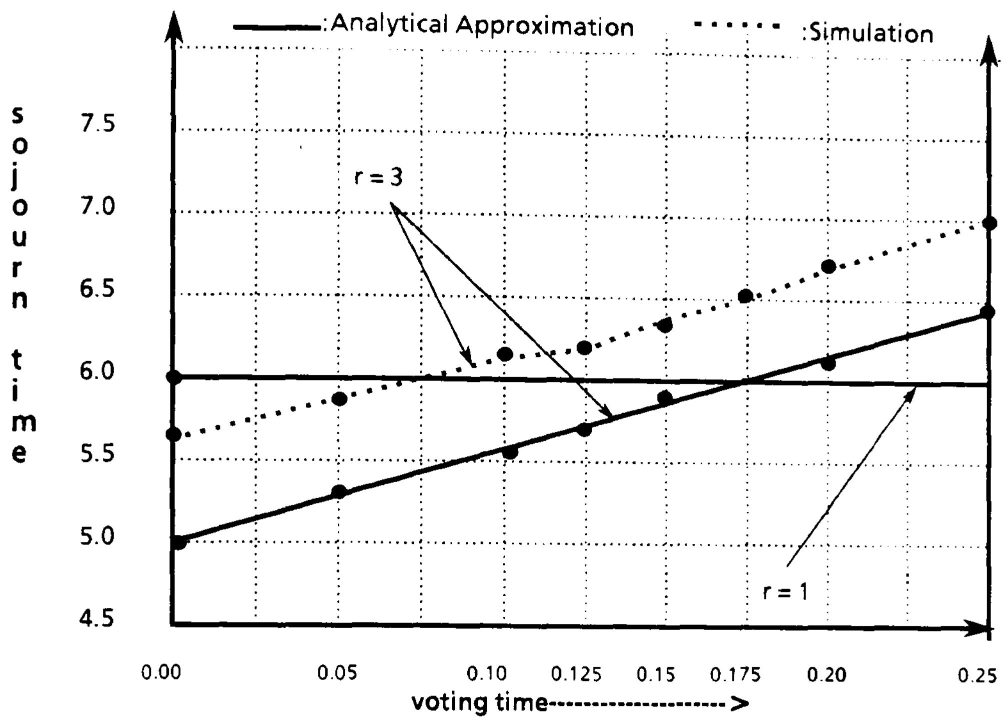


Figure 5.10. Sojourn Time Vs. Voting Time For 1/(s-a) = 1/t.

Model-0: 1/s = 1.5; 1/a = 2; 1/t = 0.5; up-time = 50000; Simulation Time = 2000; N = 5.

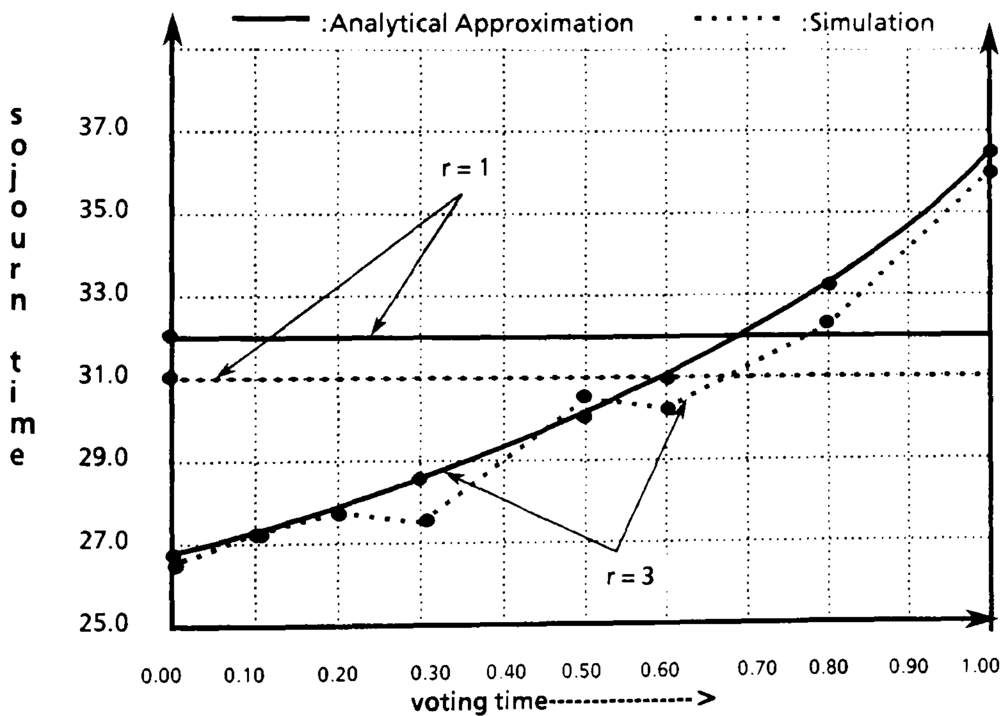


Figure 5.11. Sojourn Time Vs. Voting Time For 1/(s-a) > 1/t.

times of task siblings. On the other hand, the approximation becomes pessimistic, when the system is considered to reach steady state instantaneously following a failure. These optimistic and pessimistic trends in approximations counteract each other in reality. In these experiments, no failures occur or a few failures occur less frequently and the approximations turn out to underestimate the TMR sojourn times. In the experiments for the previous group, system is observed till it breaks down and the approximations are seen to overestimate the sojourn times.

The assumption that the sum of exponentially distributed queueing delays and transfer delays is also exponentially distributed tends to be more accurate when one delay is larger than the other. The worst cases of e are seen for $1/(s-a) = 1/t$. The assumption becomes real, when one of the delays is zero. Consequently, when $1/t$ is zero, the magnitude of e is the least.

For $1/u = 50000$, the results of experiments for $\{1/s=0.5,\ 1/t=2\}$, $\{1/s=1.0,\ 1/t=2\}$, $\{1/s=0.5,\ 1/t=0.67\}$, and $\{1/s=1.5,\ 1/t=0.5\}$ are shown in figures 5.8, 5.9, 5.10 and 5.11 respectively. In these figures, the TMR sojourn times for zero voting time are less than the simplex sojourn times. Since a few failures occur in TMR system for $1/u = 50000$, the simplex system is outperformed by the TMR system for small values of $1/v$.

We also carried out experiments to study the accuracy of analytical approximations for different values of N which was taken to be 10 and 3. When the accuracy of approximations for $N=10$ ($N=3$ respectively) was as good as that for $N=5$, it was worse for $N=3$ ($N=10$ respectively).

5.5.2. Results for Model 1

For all simulation experiments in this model, the values of $1/d$ were chosen to be 1%, 5%, and 10% of the average uptime $1/u$ which is fixed at 1000. A batch of 2000 jobs are successfully completed during each of the 10 portions of a simulation run. The results for a 5-node system are summarised in table 5.3.

In most of the experiments, e is positive implying that the approximation underestimates the TMR sojourn time. When e is negative, its magnitude increases for a larger value of $1/s$. The magnitude of e for $1/d = 100$, is less than that for $1/d = 10$ and 50 in respective experiments. This is because, as $1/d$ becomes larger, there is more time for the system to reach steady state following a failure.

The results of selected experiments are presented in figures 5.12 to 5.15. In all of these figures, the TMR sojourn time for zero voting time is less than the simplex sojourn time. Analytical approximations show that the TMR sojourn time with no voting is 85.3%, 92.0%, and 98.7% of simplex sojourn time, when $1/d$ is 10, 50, and 100 respectively. In almost all experiments carried out, the TMR system was observed to be no slower than the simplex system at zero voting time.

In the experiments carried out with $N = 10$ and 3, e was almost in the same range as that obtained for $N=5$, except for $\{1/s = 1.5,\ 1/t = 0.5\}$ in which case e was negative and greater than -13.3% and -5.2% for $N=10$ and $N=3$ respectively.

Model 1: 1/a = 2; 1/u = 1000; Number of jobs completed = 20000; N = 5.

1/d {1/s, 1/t}	10	50	100
1/(s-a) < 1/t:			
{1.5,10}	+3.8% ≤ e ≤ +7.6%	+4.8% ≤ e ≤ +7.4%	+2.4% ≤ e ≤ +4.7%
{1.0,4.0}	+5.2% ≤ e ≤ +6.7%	+5.6% ≤ e ≤ +6.1%	+4.1% ≤ e ≤ +5.2%
{0.5,2.0}	+5.2% ≤ e ≤ +6.2%	+5.5% ≤ e ≤ +6.4%	+4.9% ≤ e ≤ +5.6%
1/(s-a) = 1/t:			
{1.5,6.0}	+3.8% ≤ e ≤ +7.5%	+4.1% ≤ e ≤ +7.3%	+1.3% ≤ e ≤ +4.3%
{1.0,2.0}	+5.1% ≤ e ≤ +6.7%	+3.5% ≤ e ≤ +7.1%	+3.1% ≤ e ≤ +4.0%
{0.5,0.67}	+4.5% ≤ e ≤ +6.6%	+5.0% ≤ e ≤ +6.3%	+3.3% ≤ e ≤ +4.5%
1/(s-a) > 1/t:			
{1.5,0.5}	-3.9% ≤ e ≤ +4.5%	-8.4% ≤ e ≤ +0.6%	-8.5% ≤ e ≤ +1.9%
{1.0,1.0}	+2.2% ≤ e ≤ +3.8%	+2.0% ≤ e ≤ +3.2%	+0.3% ≤ e ≤ +2.3%
{0.5,0.0}	-2.4% ≤ e ≤ -1.7%	-2.4% ≤ e ≤ -0.7%	-2.3% ≤ e ≤ -0.6%

Table 5.3. Accuracy of Approximations in Model 1.

Model-1: 1/s = 0.5; 1/a = 2; 1/t = 2; up-time = 1000; 1/d = 50; N = 5.

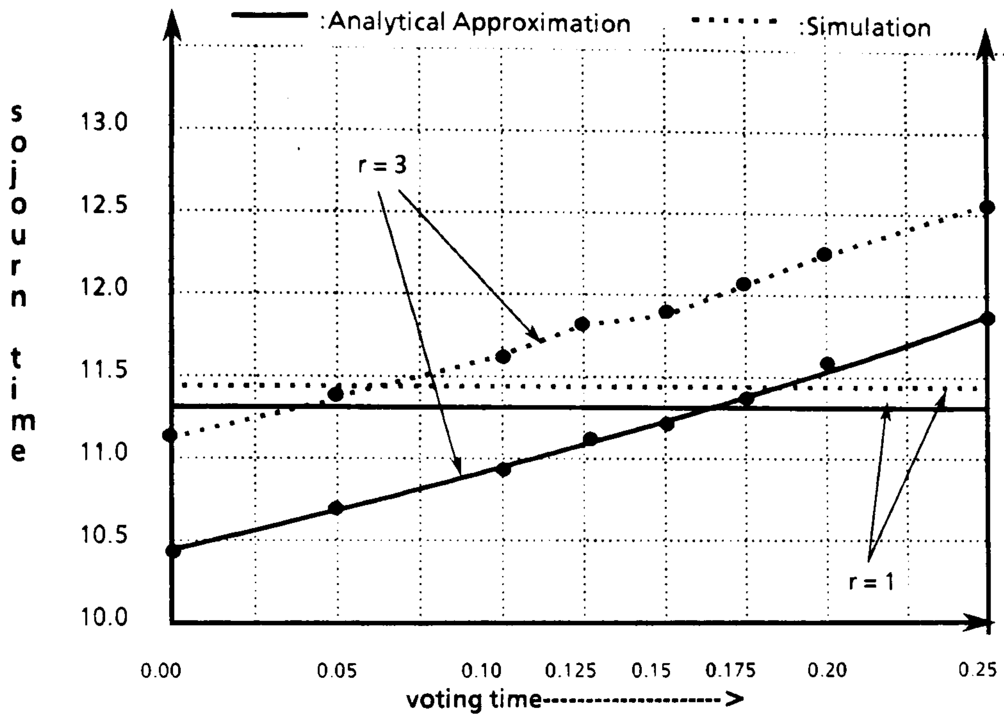


Figure 5.12. Sojourn Time Vs. Voting Time For 1/(s-a) < 1/t.

Model-1: 1/s = 1.0; 1/a = 2; 1/t = 2.0; up-time = 1000; 1/d = 50; N = 5.

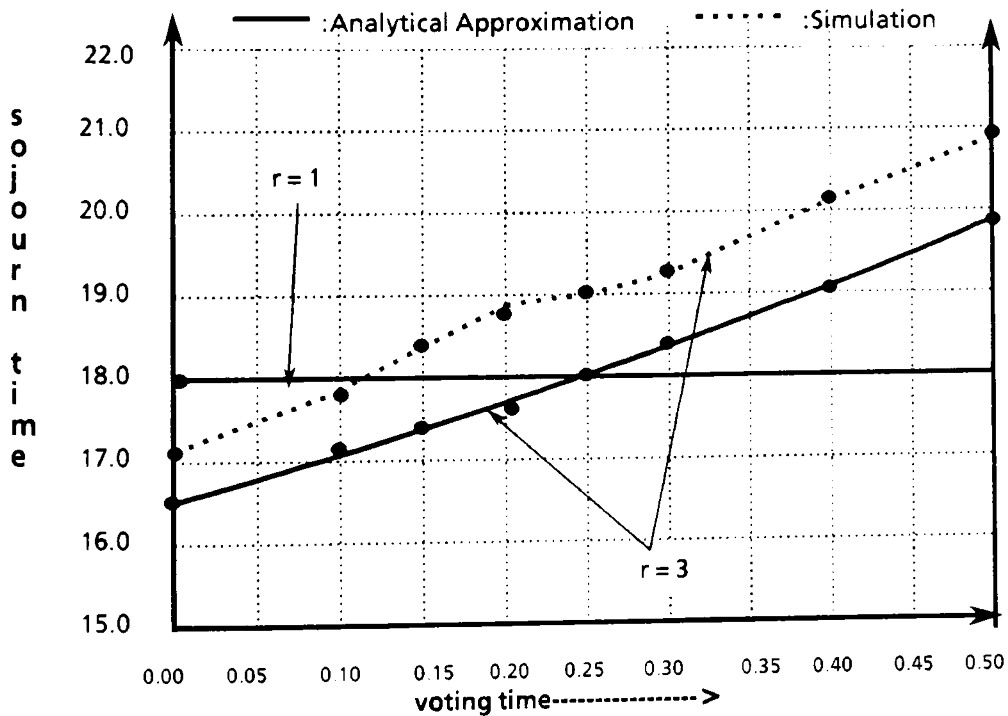


Figure 5.13. Sojourn Time Vs. Voting Time For 1/(s-a) = 1/t.

Model-1: 1/s = 0.5; 1/a = 2; 1/t = 0.67; up-time = 1000; 1/d = 50; N = 5.

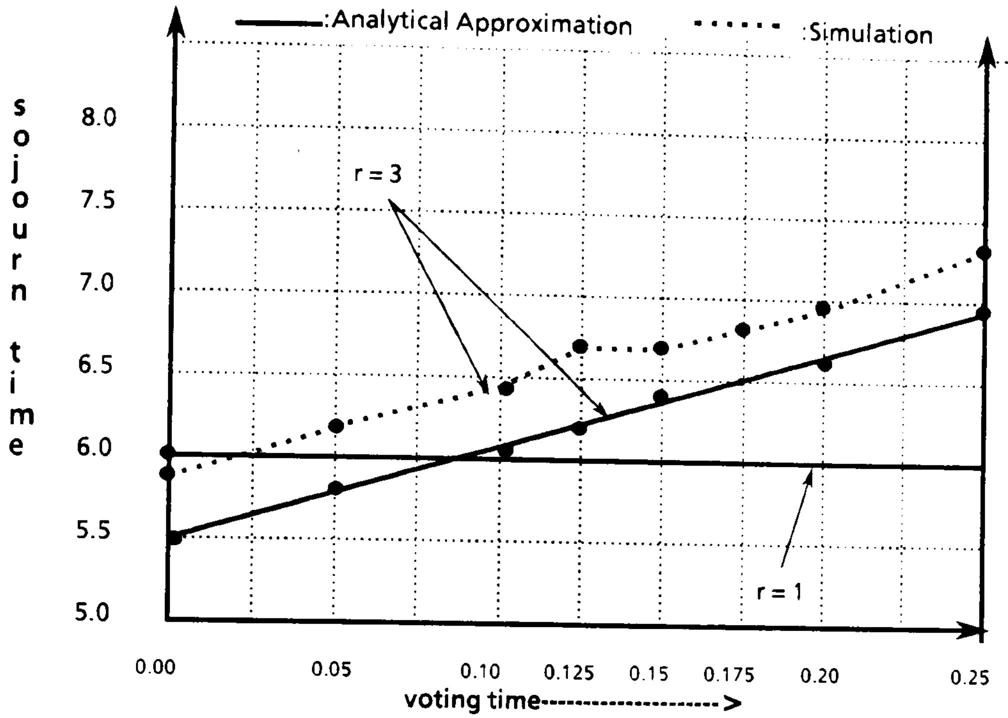


Figure 5.14. Sojourn Time Vs. Voting Time For 1/(s-a) = 1/t.

Model-1: 1/s = 1.5; 1/a = 2; 1/t = 0.5; up-time = 1000; 1/d = 50; N = 5.

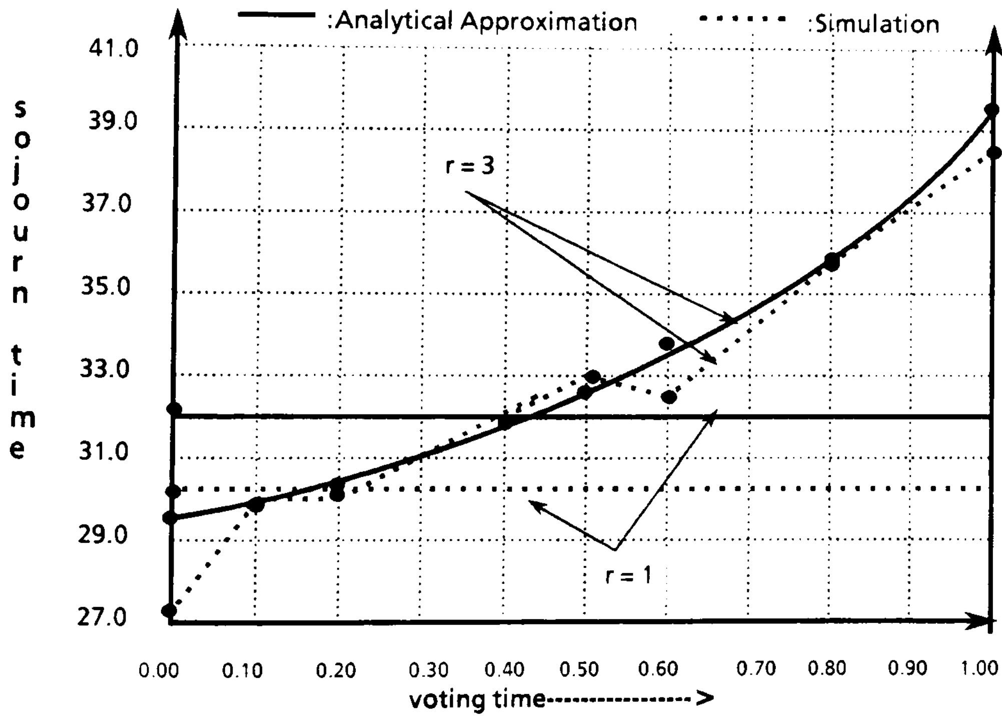


Figure 5.15. Sojourn Time Vs. Voting Time For 1/(s-a) > 1/t.

5.5.3. Computer Time for Simulations and Analytical Estimations

The CPU time needed for simulations and analytical estimations were also measured (the CPU was an Amdahl 5860). The CPU time to run a simulation for a given set of parameters increased with queueing delays, transit times, voting times and the number of nodes in the system. In model 0 and group 1, it increased for large values of $1/u$, since the system was operative for longer periods. An increase in $1/d$ for a given $1/u$ increased the CPU time in model 1. The CPU time for analytical estimations did not show much variations in the respective models and was less than 60 milliseconds for the replicated system in both models. It should be recalled that a simulation for a given set of parameters was made up of 10 'simulation runs'. In the following, CPU times per simulation run for the replicated system are given for some selected sets of parameters used in the experiments of the previous subsections. ($1/a$ and simulation time in respective cases are the same as in the experiments.) The values of e are also given in an attempt to indicate the loss of accuracy against saving in computer time and hence the cost in using analytical approximations instead of simulations.

When $1/u = 10000$ and $N = 5$ in model 0 and group 1, the CPU time per simulation run was 13 and 10 seconds for $\{1/s = 1.5,\ 1/v = 0.0,\ 1/t = 10\}$ (for which e was -6.5%) and $\{1/s = 0.5,\ 1/v = 0.0,\ 1/t = 2.0\}$ (for which e was 1.8%) respectively; for the second case, when $1/u$ was increased to 20000, the CPU time increased to 18.5 seconds and e was 1.9%. When $1/s = 0.5$, $1/t = 0.0$, $1/u = 50000$ in group 2, a simulation run required 6.5 and 6.6 seconds of CPU time for $1/v = 0.0$ and 0.25 respectively. For these cases, e was respectively 0.8% and 0.3%. With $1/v = 0.0$, when N was increased to 10, the CPU time requirement became 14 seconds and e was -21.5%. In model 1, when $1/s = 0.5$, $1/v = 0.0$, $1/t = 0.0$, $N = 5$, and $1/u = 1000$, the CPU time per run was

13 and 17 seconds when $1/d$ = 10 and 100 respectively; e was -2.4% and -1.6% respectively. The CPU time was 37 seconds and e was -4.7% when $1/d$=10 and N = 10. The CPU time for simulations can be reduced drastically by employing powerful multiprocessors and advanced concurrent simulation techniques.

5.6. Concluding Remarks

The factors capable of affecting the performance of a TMR system in relation to the simplex system were discussed in section 5.2 and were identified to be (i) voting times, (ii) processor failure rates, (iii) extra message traffic, and (iv) sequencing overheads. In this study, we have assumed a specialised pipeline architecture with high bandwidth communication which enables us to ignore the effects of extra message traffic and of sequencing overheads. The model of such a system was presented in section 5.3. The performance of such a system was then evaluated both by computer simulations and by analytical approximations. Despite their simplicity and roughness, the approximations developed here have been shown to estimate the system performance measures fairly accurately - the analytical estimates are within 10% of simulation estimates in 90% of the simulation experiments carried out. Thus, when simulation experiments are time consuming and expensive to carry out, analytical approximations can provide an attractive alternative.

We have compared the mean sojourn times of jobs in replicated and unreplicated systems. A rather surprising result has been that when voting times are small and processor failures are less likely, a replicated system can provide better response times. This is because when redundant processors in each node are considered to be unevenly loaded, the presence of extra

processors brings performance benefits by exploiting the earliest service completions of siblings. These performance measures have been discussed in section 5.5.

The particular assumptions that we have made are not the only ones that can be handled by our approximation approach. For example, rather than assigning the voting and computational functions to separate (processing) units within a processor, one could use a single processor, with appropriately modified service times, to carry out both voting and processing. It is also possible to consider a system configuration in which there is a single voter for all three processors of a TMR node. If the voters in such a system are reliable, the system will be more cost-effective than a system of the type considered here (cf. [Carte79]). In such a configuration, it is no more an approximation to consider that voters act as synchronisation points for messages. Also, the assumption that congestion has no effect on transit times can be relaxed. These generalisations of the models would provide a relevant topic of further research.

CHAPTER 6

CONCLUSIONS

Design and development of algorithms for fault tolerant distributed systems has been our chosen topic. Under this topic, this thesis has presented the following: A classification of faults in systems was presented and fault tolerant algorithms for a particular system function were developed. Reaching agreement was the system function considered and agreement algorithms tolerant to processor faults of different classes were presented. The problem of evaluating the performance of fault tolerant distributed systems that require the use of agreement algorithms was considered. Analytical methods (algorithms) were developed to evaluate the performance of a particular type of distributed replicated systems. These methods were derived based on some approximations, the accuracy of which was examined using computer simulations.

Given that a component can have many failure modes, the design of a fault tolerant algorithm for any given system function requires making fault models of components and specifying precisely the assumed behaviour of faulty components. This requirement led us to investigate the two types of faults often considered in the literature: omission and Byzantine faults. It was observed that these two fault types represent the two extreme cases of the most restricted and the unrestricted types. Fault types of intermediate restrictions were identified and defined using "expected-value" and "timeliness" as the two specified properties of a component's response. These fault

types are value, timing and emission. A value (timing) fault causes a component's response to be incorrect only in the value (time) domain. An emission fault causes a response to be incorrect in either domain or in both the domains.

This classification has been extended to apply to components that are required to produce replicated responses. For such components, nine fault types have been identified based on the notion of consistent failures in producing a replicated response. A transitive relationship, "a proper subset of" or "more restricted than", between fault types has been established. These fault classifications for components with replicated or unreplicated responses, have been applied to specify the behaviour of selfchecking components and to analyse the faulty behaviour of a composite component in terms of the fault types of constituent components. One such composite component considered was a processor with a clock and the subsequent fault analysis led to some interesting observations: for a clock that responds simply in response to the passage of real time, value failures and timing failures cannot occur independently of each other; a clock that fails in a manner other omission can seriously affect the failure modes of the processor; a fast clock, for example, can make the processor fail both in the value and the timing domains. A distributed system was subject to fault analysis by considering it to be made up of processors connected by a communication subsystem. It was observed in the analysis that when a processor fails in a manner other than Byzantine, a communication subsystem capable of providing a reliable broadcast service is necessary for considering a consistent fault model for the processor.

The fault classifications presented in chapter 2 provide a convenient means for the development of increasingly more sophisticated algorithms to

solve a given problem tolerating faults of increasingly general types. We have chosen to solve the agreement problem. The problem was defined with one processor in a distributed system being designated as the sender. The sender can be faulty or non-faulty. Extending a solution to this problem in a general context where every processor in the system can be a sender is a straightforward task. The agreement problem has been solved by considering processors of a distributed system to be synchronous and by assuming a known bound on message communication delays between processors. An upper bound on the number of processors that can possibly fail is also assumed. The assumptions made in solving the problem are stated and they are essential for reaching agreement in a bounded and known time interval.

In the context where processors will not know a priori the sender's broadcast time, deterministic agreement algorithms have been developed for each of the fault types defined for components with replicated responses. For some fault types, special cases have also been considered to develop algorithms. The resulting agreement algorithms constitute a family of algorithms which is presented in chapter 3. A generic algorithm is also presented in that chapter to represent the family of algorithms collectively. Based on the generic algorithm, the complexities of algorithms tolerant to faults of different types are compared. With some exceptions, an algorithm is found to be less complex than another one, if the faults for which the first algorithm is developed are a proper subset of the faults considered in the development of the second algorithm. In the exceptional cases, no change in complexity is observed. These algorithms can be further developed into broadcast protocols that are essential in systems with replicated processing. Agreement algorithms developed under different fault types and an illustration of their relative complexity are the main contributions of chapter 3.

In chapter 4, the agreement problem is solved in the context where processors are assumed to know a priori the sender's broadcast time. Only a few selected fault types are considered. Agreement algorithms tolerant to permanent omission faults, omission faults, timing faults, and consistently late timing faults have been developed and presented. These algorithms are early stopping algorithms which attempt to reach agreement faster, when the number of actually failed processors is less than the upper bound assumed. Applications of early stopping algorithms in distributed transaction commit are cited. The early stopping conditions in the permanent omission fault tolerant algorithm take into account of the fact that some faulty processors may have failed and stopped functioning before the execution of the algorithm starts. Consequently, this algorithm can be faster than the omission fault tolerant algorithm for the same number of failed processors in an execution. The omission fault tolerant algorithm can be faster than, and the consistently late timing fault tolerant algorithm has less message complexity than, the timing fault tolerant algorithm. Under consistently late timing fault model, faulty processors were considered to be overloaded processors and the agreement algorithm has been developed using assumptions that are weaker than, but similar to, assumptions made in the development of the other three algorithms.

The algorithms of chapter 4 are substantially different from the corresponding ones presented in chapter 3. The reason is that they are developed with different assumptions about the processors' a priori knowledge of the sender's broadcast time. When processors do not have a prior knowledge of the sender's broadcast time (as is the case in chapter 3), they can skip reaching a unanimous decision on the sender's value, if the sender is faulty. However, when they know the sender's broadcast time a

priori, they have to reach some decision unanimously for every broadcast of the sender. One of the most difficult scenarios to cope with is the sender not carrying out any broadcast at all. Thus, when the sender does not carry out a broadcast that it should have performed, the processors executing an algorithm of chapter 4 have to exchange more number of messages and take longer time to reach agreement (on the default value). On the other hand, the processors executing an algorithm of chapter 5 will not exchange any messages and will not decide on any value.

A pipeline TMR system has been considered in chapter 5 for performance evaluation of a distributed replicated system. Faulty processors in the system are assumed to fail in a permanent and consistent value manner. Two system models with respect to recovery of faulty processors are considered. In model 0, no mechanism for recovery was assumed and a faulty processor remained faulty till the end of the mission period. In model 1, failed processors were assumed to be repaired within a finite and random delay. Analytical methods have been derived to evaluate system response times. Derivation of these methods involve simplifying approximations and, as a result, these methods are simple to use.

The accuracy of analytical approximations was examined by computer simulations. The performance estimates obtained by analytical estimation and by simulation were compared for different values of system parameters. The differences between these two estimates for every given set of system parameters considered were tabulated for both models. For the first model, two cases were considered: the TMR system is operative, and inoperative at the end of the mission period of given length. For the simulation experiments carried out, the analytical estimates were within 20% of the simulation estimates in the worst cases, and were within 10% in ninety percent of

the experiments conducted. Analytical methods derived in chapter 5 make two important contributions: they can provide an alternative to simulations, in particular, when simulations are expensive; they can be a step towards an empirical study of the performance of systems with more complex architectures.

The main contributions of this thesis can be summarised to be: the fault and failure classification, the use of the classification in the development of a family of fault tolerant agreement algorithms and a collection of early stopping agreement algorithms, and development of analytical techniques for evaluating the performance of distributed replicated systems.

6.1. Directions For Further Research

A fault analysis of a composite component in terms of fault types of constituent components has been presented in chapter 2. A fault analysis in the reverse direction can be carried out in a particular system context. Consider, for example, a distributed system where processors maintain their clocks in synchronism. Let a processor be considered to be made up of three components: clock, computational unit and network interface. For a given fault assumption for such a processor, the fault assumptions required on the processor's components can be analysed. Such an analysis will reveal the requirements on components' behaviour so that the processor can fail only in a particular manner. Suppose that the processor is assumed to fail in an omission manner. This means that the network interface can omit sending any number of messages; however, it cannot omit receiving more than certain number of messages when a clock synchronisation algorithm is being executed; otherwise, the clock may not be synchronised and consequently the processor can fail in a timing manner. Similarly, when the processor

maintains some state information, its computational unit should not omit processing a message which would result in changes in the state information; otherwise, in subsequent message processing, the processor can produce incorrect values. It should be noted that the results of such an analysis will vary with the system context in which a particular fault assumption is considered. If a processor does not maintain a synchronised clock and all its processings are considered to be stateless, then assuming the omission fault type for the processor would mean that the network interface can omit receiving any number of messages and the computational unit can omit processing any message.

Only one type of processor fault has been considered in developing each of the agreement algorithms presented here. A faulty processor was assumed to fail in the "worst" possible manner that is permitted within the chosen fault model. One can consider the possibilities of developing algorithms to reach agreement in the presence of faults of different types and with a bound on the number of faults of each type. Two such algorithms have been developed in the literature: timing and Byzantine faults were considered in [Meyer87]; in [Thamb88], Byzantine faults, consistent omission faults and *malicious symmetric faults* that come closely under our definition of consistent value faults were considered. We have considered consistent omission faults and omission faults in section 3.7. When it is given that at most $(f-f')$, out of at most f, omission faulty processors fail in a consistent manner, the algorithm has been observed to require just $(f'+1)$ rounds instead of $(f+1)$ rounds. In fact, every algorithm in chapter 3 which requires $(f+1)$ rounds in the presence of at most f faults of the type, say A, considered will require just $(f'+1)$ rounds, if $(f-f')$ and f' are assumed to be the new bounds on the number of processors that have faults of consistent omission and type A respectively. The family of algorithms and their

complexity analysis can provide useful insights into developing efficient algorithms tolerant of faults of multiple types.

The development of the permanent omission fault tolerant algorithm in chapter 4 leaves us with an impression that it may not be possible to develop a faster algorithm. A permanent omission fault is a special case of an omission fault that is a proper subset of faults of every other type that is not consistent. If our impression can be formally proved, then it will establish a lower bound on the execution time of any early stopping algorithm that is tolerant to faults that are not consistent.

Our work on performance evaluation is just an initial step and can be extended for systems with more general architectures and processor faults of more serious types.

We are currently building a TMR pipeline system using transputers. The architecture of the system has been presented in [Ezhil89]. The transputers of a TMR node execute agreement protocols to meet ordering and agreement requirements. Using this system, we plan to study the performance of agreement algorithms of chapter 3. The accuracy of analytical approximations of chapter 5 will also be examined. It is also our plan to construct a fail-silent node (a node whose failure mode is restricted to permanent omission) out of two transputers. Its performance will be compared with the fail-silent node being built by Ferranti Computer Systems Ltd., using special purpose hardware to achieve clock synchronisation, agreement and order. Our fail-silent nodes are intended to support the object oriented distributed systems (ARJUNA) [Shriv89] developed at Newcastle. Thus the theoretical work reported here will be complemented by extensive experimental work concerned with the building of real systems.

References

Abrah74.

J. A. ABRAHAM AND D. P. SIEWIEOREK, "An Algorithm For the Accurate Reliability Evaluation of Triple Modular Redundancy Networks," *IEEE TC*, vol. C-23, pp. 682-692, July 1974.

Ander81.

T. ANDERSON AND P. A. LEE, *FAULT TOLERANCE: Principles And Practice,* Prentice-Hall International, Inc., 1981.

Avizi71.

A. AVIZIENIS, G. C. GILLEY, F. P. MATHUR, D. A. RENNELS, J. A. ROHR, AND D. K. RUBIN, "The STAR (Self-testing and repairing) Computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE TC*, vol. C-20, no. 11, pp. 1312-1321, November 1971.

Babao85.

O. BABAOGLU AND R. DRUMMOND, "Streets Of Byzantium: Network Architectures For Fast Reliable Broadcasts," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-11, no. 6, pp. 546-554, June 1985.

Birma87.

K. BIRMAN AND T. JOSEPH, "Reliable Communication in the Presence of Failures," *ACM TRANSACTIONS ON COMPUTING SYSTEMS*, vol. 5, no. 1, February 1987.

Carte71.

W. C. CARTER, D. C. JESSOP, A. B. WADIA, P. R. SCHNEIDER, AND W. G. BOURICIUS, "Logic Design for Dynamic and Interactive recovery," *IEEE TC*, vol. C-20, no. 11, pp. 1300-1305, November 1971.

Carte79.

W. C. CARTER, "Hardware Fault Tolerance," in *Computing Systems Reliability*, ed. T. Anderson and B. Randell, pp. 211-263, Cambridge University Press, 1979.

Crist85.

F. CRISTIAN, H. AGHILI, R. STRONG, AND D. DOLEV, "Atomic Broadcast: From Simple Message Diffusion To Byzantine Agreement," *DIGEST OF PAPERS, FTCS-15*, pp. 200-206, Ann Arbor, Michigan, June 1985.

Crist86.

F. CRISTIAN, H. AGHILI, AND H. R. STRONG, "Clock Synchronisation in the Presence of Omission and Performance Faults, and Processor Joins," *PROC. 16TH SYMPOSIUM ON FTCS*, pp. 218-223, Vienna, Austria, July, 1986.

Crist88.

F. CRISTIAN, "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems," *PROCEEDINGS OF THE 18TH FAULT TOLERANT COMPUTING SYSTEMS (FTCS-18)*, Tokyo, 1988.

Daly73.

W. M. DALY, A. L. HOPKINS, AND J. F. MCKENNA, "A Fault-tolerant Digital Clocking System," *PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING (FTC/3)*, pp. 17-21, Palo Alto, CA., 1973.

Davie78.

D. DAVIES AND J. F. WAKERLY, "Synchronistion and Matching in Redundant Systems," *IEEE TRANS. ON COMPUTERS*, vol. c-27, no. 6, pp. 531-539, June 1978.

Dolev82a.

D. DOLEV, R. REISCHUK, AND H. R. STRONG, "'Eventual' Is Earlier Than 'Immediate'," *PROC. 23RD SYMPOSIUM ON FOUNDATIONS OF COMP. SCIENCE*, pp. 196-203, Chicago, Illinois, 1982.

Dolev82b.

D. DOLEV AND H. R. STRONG, "Requirements For Agreement In A Distributed System," *PROC. 2ND INTERNATIONAL SYMPOSIUM ON DISTRIBUTED DATABASES*, pp. 115-129, Berlin, September 1982.

Dolev83.

D. DOLEV AND H. R. STRONG, "Authenticated Algorithms For Byzantine Agreement," *SIAM JOURNAL OF COMPUTING*, vol. 12, no. 4, pp. 656-666, Nov. 1983.

Dolev84.

D. DOLEV, J. HALPERN, AND H. R. STRONG, "On The Possibility And Impossibility Of Achieving Clock Synchronisation," *PROC. 16TH ANNUAL ACM STOC*, pp. 504-511, Washington D.C., April, 1984.

Ellin73.

C. E. ELLINGSON AND R. J. KULPINSKI, "Dissemination Of System Time," *IEEE TRANS. ON COMMUNICATIONS*, vol. COM-21, no. 5, pp. 605-624, May 1973.

Enslo78.

P. H. ENSLOW, "What is a Distributed Data Processing System?," *COMPUTER*, pp. 13-21, January 1978.

Ezhil86.

P. D. EZHILCHELVAN AND S. K. SHRIVASTAVA, "A Characterisation of Faults in Systems," *PROC. 5TH SYMPOSIUM ON RELIABILTY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, pp. 215-222, Los Angeles,

CA., January 1986.

Ezhil87.

P. D. EZHILCHELVAN, "Early Stopping Algorithms For Distributed Agreement Under Fail-stop, Omission And Timing Fault Types," *PROC. 6TH SYMPOSIUM ON RELIABILTY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, pp. 201-212, Williamsburg, VA., March 1987.

Ezhil89.

P. D. EZHILCHELVAN, S. K. SHRIVASTAVA, AND A. TULLY, "Constructing Replicated Systems Using Processors with Point-to-Point Communication Links," *PROC. 16TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, pp. 177-184, Jerusalem, Israel, May 1989.

Fisch83b.

M. J. FISCHER, N. A. LYNCH, AND M. S. PATERSON, "Impossibility Of Distributed Consensus With One Faulty Process," *PROC. 2ND SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS*, Atlanta, GA, 1983.

Fisch83a.

M. J. FISCHER, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)," *PROC. INTERNATIONAL CONFERENCE ON FOUNDATIONS OF COMPUTATION THEORY*, Borgholm, Sweden, August, 1983.

Garci86.

H. GARCIA-MOLINA, F. PITTELLI, AND S. DAVIDSON, "Applications of Byzantine Agreement in Database Systems," *ACM TRANSACTIONS ON DATABASE SYSTEMS*, vol. 11, no. 1, pp. 27-47, March 1986.

Giffo79.

D. K. GIFFORD, "Weighted Voting for Replicated Data," *PROC. 7TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pp. 150-161,

1979.

Gray79.

J. GRAY, "A Discussion Of Distributed Systems," *IBM RESEARCH REPORT*, no. RJ2699, Sept. 1979.

Hadzi84.

V. HADZILACOS, "Byzantine Agreement Under Restricted Types Of Failures," Technical Report, Aiken Computation Laboratory, Harvard University, Cambridge MA, 1984.

Halpe84.

J. Y. HALPERN, B. SIMONS, H. R. STRONG, AND D. DOLEV, "Fault Tolerant Clock Synchronisation," *PROC. 3RD ACM SYMPOSIUM ON PODC* , pp. 89-102, Vancouver, Canada, Aug. 1984.

Harpe88.

R. E. HARPER, J. H. LALA, AND J. J. DEYST, "Fault Tolerant Processor Architecture Overview," *PROCEEDINGS OF THE 18TH FAULT TOLERANT COMPUTING SYSTEMS (FTCS-18)*, pp. 252-257, Tokyo, 1988.

Hopki78.

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, Oct. 1978.

Iacop89.

M. J. IACOPONI AND D. K. VAIL, "The Fault Tolerance Approach of the Advanced Architecture On-board Processor," *PROCEEDINGS OF THE 19TH FAULT TOLERANT COMPUTING SYSTEMS (FTCS-19)*, pp. 6-12, Chicago, 1989.

Kopet85.

H. KOPETZ, "Resilient Real-Time Systems," in *Resilient Computing Systems*, ed. T. Anderson, pp. 91-101, Collins, 1985.

Kopet87.

H. KOPETZ AND W. OCHSENREITER, "Clock Synchronisation in Distributed Real-Time Systems," *IEEE TC*, vol. C-36, no. 8, pp. 933-940, August 1987.

Lampo82.

L. LAMPORT, R. SHOSTAK, AND M. PEASE, "The Byzantine Generals Problem," *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, vol. 4, no. 3, pp. 382-401, July 1982.

Lampo84.

L. LAMPORT AND M. FISCHER, "Byzantine Generals And Transaction Commit Protocols," Opus 62, SRI International, Menlo Park, CA., 1984.

Lampo85.

L. LAMPORT AND P. M. MELLIAR-SMITH, "Synchronising Clocks In The Presence Of Faults," *JACM*, vol. 32, no. 1, pp. 52-78, Jan. 1985.

Lapri85.

J. C. LAPRIE, "Dependable Computing and Fault-tolerance: Concepts and Terminology," *PROC. 15TH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING (FTCS-15)*, pp. 2-11, Ann Arbor, Michigan, USA, June 1985.

Lyons62.

R. E. LYONS AND W. VANDERKULK, "The Use of Redundancy to Improve Computer Reliability," *IBM JOURNAL*, April 1962.

Mathu70.

F. P. MATHUR AND A. AVIZIENIS, "Reliability analysis and architecture of a hybrid-redundant digital system: generalised TMR with self-repair," *SPRING JOINT COMPUTER CONFERENCE*, vol. 36, pp. 375-383, 1970.

Meyer87.

F. J. MEYER AND D. K. PRADHAN, "Consensus with Dual Failure Modes," *PROCEEDINGS OF THE 17TH FAULT TOLERANT COMPUTING SYSTEMS (FTCS-17)*, pp. 48-54, Pittsburgh PA, July 1987.

Mitra87.

I. MITRANI, *Modelling of Computer and Communication Systems*, Cambridge University Press, 1987.

Mohan83.

C. MOHAN, R. STRONG, AND S. FINKELSTEIN, "Method of Distributed Commit and Recovery Using Byzantine Agreement Within Clusters of Processors," *PROC. 2ND ACM SYMP ON PRINCIPLES OF DISTRIBUTED COMPUTING*, pp. 89-103, Montreal, August 1983.

Napol89.

L. M. NAPOLITANO, D. D. ANDALEON, K. R. BERRY, P. R. BRYSON, S. R. KLAPP, AND J. E. LEEPER, "Fault Tolerance in a High-speed 2D Convolver/Correlator: STARLOC," *PROCEEDINGS OF THE 19TH FAULT TOLERANT COMPUTING SYSTEMS (FTCS-19)*, pp. 80-87, Chicago, 1989.

Okamo88.

T. OKAMOTO, "A Digital Multisignature Scheme Using Bijective Public-Key Cryptosystems," *ACM TOCS*, vol. 6, no. 8, pp. 432-441, November 1988.

Pease80.

M. PEASE, L. LAMPORT, AND R. SHOSTAK, "Reaching Agreement In The Presence Of Faults," *JOURNAL OF ACM*, vol. 27, no. 2, pp. 228-234, 1980.

Pitte89.

F. PITTELLI AND H. GARCIA-MOLINA, "Reliable Scheduling in a TMR Database System," *ACM TOCS*, vol. 7, no. 1, pp. 25-60, Feb. 1989.

Popek81.

G. POPEK, B. WALKER, J. CHOW, C. KLINE, G. RUDISIN, AND G. THIEL, "LOCUS: A Network Transparent, High Reliability Distributed System," *PROC. 8TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pp. 169-177, 1981.

Powel88.

D. POWELL, G. BONN, D. SEATON, F. WAESELYNCK, AND P. VERISSIMO, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," *DIGEST OF PAPERS, FTCS-18*, Tokyo, June 1988.

Rives78.

R. L. RIVEST, A. SHAMIR, AND L. ADLEMAN, "A Method For Obtaining Digital Signatures and Public Key Cryptosystems," *CACM*, vol. 21, no. 2, pp. 120-126, Feb. 1978.

Schli83.

R. D. SCHLICHTING AND F. B. SCHNEIDER, "Fail Stop Processors: An Approach To Design Fault Tolerant Computing Systems," *ACM TRANS. ON COMP. SYSTEMS*, vol. 1, no. 3, pp. 222-234, Aug. 1983.

Schne84.

F. B. SCHNEIDER, "Byzantine Generals In Action: Implementing Fail-stop Processors," *ACM TRANS. ON COMP. SYSTEMS*, vol. 2, no. 2, pp.

145-154, May 1984.

Schne86.

F. B. SCHNEIDER, "Abstractions For Fault Tolerance In Distributed Systems," *PROCEEDINGS OF IFIP CONGRESS '86*, pp. 727-733, North Holland, Sept. 1986.

Shriv89.

S. K. SHRIVASTAVA, G. N. DIXON, G. D. PARRINGTON, F. HEDAYATI, S. M. WHEATER, AND M. C. LITTLE, "The Design and Implementation of ARJUNA," Technical Report 280, Computing Laboratory, University of Newcastle upon Tyne, UK, 1989.

Thamb88.

P. THAMBIDURAI AND Y. PARK, "Interactive Consistency with Multiple Failure Modes," *PROC. 7TH SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS*, pp. 93-100, Columbus, OH., October 1988.

Theur86.

N. THEURETZBACHER, "VOTRICS: Voting Triple Modular Computing System," *PROCEEDINGS OF THE 16TH FAULT TOLERANT COMPUTING SYSTEMS (FTCS-16)*, pp. 144-151, 1986.

Veris89.

P. VERISSIMO, "A Fault Model for Reliable Fail-Controlled Communication Systems," *A DELTA-4 DOCUMENT*, INESC, Lisbon, Portugal, July 14, 1989.

Wensl78.

J. H. WENSLEY, L. LAMPORT, J. GOLDBERG, M. W. GREEN, K. N. LEVITT, P. M. MELLIAR-SMITH, R. E. SHOSTAK, AND C. B. WEINSTOCK, "SIFT: Design And Analysis Of A Fault Tolerant Computer For Aircraft Control," *PROCEEDINGS OF THE IEEE*, vol. 66, no. 10, pp. 1240-

1255, Oct. 1978.

York83.

G. YORK, D. SIEWIOREK, AND Z. SEGALL, "Asynchronous Software Voting in NMR Computer Structures," *IEEE PROC. 3RD SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, pp. 28-37, Florida, Oct. 1983.