A FAULT TOLERANT DISTRIBUTED COMPUTER CONTROL SYSTEM

by

Orlando Gomes Loques Filho

February, 1984

A thesis submitted for the degree of Doctor of Philosophy of the University of London and for the Diploma of Membership of the Imperial College.

Department of Computing
Imperial College
London, S.W.7

Abstract

Distributed systems offer a number of potential advantages over conventional systems in response time, availability, extensibility, and performance. For this reason distributed computer systems are increasingly being used for the implementation of process control applications. Some of these applications have stringent reliability requirements which can be met by the use of fault tolerance techniques. In most systems fault tolerance is obtained through the use of hardware redundancy, but software redundancy techniques are also required. This thesis proposes techniques for the use of "hot" and "cold" standby redundancy of software modules, based on standard stations connected by a communication network.

The hot and cold standby redundancy are supported by a common set of system modules, which provide for detection of failures and for reconfiguration of the modules of the application. Cold standby modules are created and activated by the system in order to replace failed modules, but no state information is preserved. Hot standby modules do preserve state information, and transparent failure recovery is supported. In this case the application system can continue its operation without any loss of control capability. Extra mechanisms are needed in order to support the hot standby scheme by performing the transfer of the application state to the back-up module(s) and, in case of a module failure, by allowing its back-up to take over its functions.

The fault tolerance support mechanisms are intentionally simple, and are implemented by standard modules. Their simplicity permits them to be used in microcomputer systems. Their modularity permits a system designer to use them selectively in order to meet the reliability requirements of the application. Another quality of the approach is that the support mechanisms are built using facilities provided by an existing distributed system: CONIC, developed at Imperial College, for distributed computer control applications. This allows applications having mixed reliability requirements to be supported by the same software architecture. Also, application modules need not be written with fault tolerance in mind , but can be transformed, at a later stage, to provide the required degree of reliability. In addition, the mechanisms can be readily transported to any hardware system already supporting this software architecture. The relevant features of the fault tolerant system were implemented and tested in a working prototype.

Acknowledgements

The ideas leading to this thesis were aided greatly by discussions with the members of the Distributed Computing Group at Imperial College. Also, without the experience of closely following the development of the Conic project, and the use of the Conic concepts and system to test my ideas, much of the practical results of this thesis would be missed. In particular I should thank: Jeff Kramer, my supervisor, who by reading earlier drafts of this report much contributed to the structuring and clarity of my presentation of a difficult subject; Morris Sloman, who by insisting on simplicity of concepts and practical applicability has instigated the refinement and crystallization of my ideas; Jeff Magee, who by patiently interpreting my English, providing clear comments on my proposals, and clarifying the details of the Conic implementation, has much helped my acquaintance with a new area.

Next, I would like to thank my coleagues for their friendship and encouragement. In particular, Sebastian Danicic for his fraternal spirit and carefull proof-reading under a tight deadline.

Finally, I would like to thank CAPES -- an agency from the Brazilian Ministry of Education, for their financial support to my postgraduate studies.

To Roselia, and Gabriel

CONTENTS

LIST of FIGURES

CHAPTER I

INTRODUCTION

The use of computers in control applications is increasingly common. Combined factors such as uninterrupted control capability, small incremental expansion cost, and enhanced modifiability and flexibility has led to many of these systems being implemented by sets of computers. These computers (termed stations) are physically interconnected in order to support the application control function. The control function is naturally divided among processing entities supported by the computers. These processing entities are logically connected in order to cooperate to perform the global control function. The result is a distributed computer control system (DCCS).

There is a need for a comprehensive support environment for building real-time process control applications. This environment should provide the basic architecture and the tools for the development and implementation of different applications according to their specific requirements. Control applications inherently have strong reliability requirements which require the provision of fault tolerance capability. Thus the support environment has to support the development of applications which require fault tolerance.

The aim is to design a fault tolerant distributed computer system for providing varying degrees of reliability requirements for process control applications. In this direction we propose the integration of features for specification and implementation of fault tolerant systems into a standard DCCS. In general, achieving a fault tolerance capability is non-trivial, since all levels of the system should be considered. In order to reduce the size of the task we have selected a programming environment specially designed for the support of distributed control applications -- Conic, developed at Imperial College [Kramer 83]. Although the design of the fault tolerance support mechanisms was based on Conic, basic principles for the construction of fault tolerant systems were identified; they are useful for the construction of fault tolerant systems based on other programming environments.

This approach is different to that adopted in most of the fault tolerant system proposals that can be found in the literature, e.g., [Geitz 81, Hopkins 78, Kaiser 78, Katsuki 78, Wensley 78]. In those systems the fault tolerance capability is obtained by special purpose designs of the whole system. This could be justified in order to supply specific requirements of their intended application. However, there are many applications where simplicity of design and ease of implementation are mandatory requirements. Also, different parts of the system can have different reliability requirements; thus fault tolerance should be made available only where required. Our approach is intended for this class of applications.

1.1 Design Goals

In this section the main goals we have set up for the design of the fault tolerant system are presented and justified.

Simplicity

The computer control systems we devise are likely to be composed of a large number of computer stations. Typically each station will be concerned with the local control of a plant device. Because of cost considerations most of these stations have to be implemented by a single microcomputer which may not have very sophisticated or large local resources. We are interested in mechanisms that can be used by any station in a typical application. Hence these mechanisms should be simple enough not to add any unreasonable resource demand to the stations. A reduced number of more powerful stations can be provided in order to support special services to be shared by a group of stations.

Also, keeping the mechanisms simple helps the achievement of reliability: They should be easier to design and implement, and thus design faults within the own mechanisms are less likely. They should be easier to understand by application designers, hence the risk of misuse is reduced.

Transparency

It should be possible to design and test an application system without having fault tolerance in mind. At a second stage of design, fault tolerance capability would be added where necessary, but without having to redesign the application modules. This permits standard techniques and development tools to be used for the specification, verification, and testing of applications having different reliability requirements. In addition transparency is required for modularity: application modules can be independently designed and assembled together to implement different application systems. In order to achieve this goal the fault tolerance support mechanisms should be orthogonal to the application functions.

Modular Design

The fault tolerance capability should be achieved through mechanisms built upon the basic architecture provided by the base system (e.g., standard Conic). This allows the same architecture to be used for implementing several applications according to their particular reliability requirements without having to redesign the basic supporting system. It should be necessary only to include the required reliability support modules. An additional quality of this approach is that application modules having different reliability characteristics can cooperate together in the same application system.

This does not means that a different mechanism is needed for supporting each particular reliability requirement. Instead, the support mechanisms should be structured in order to lead to a minimum number of different modules. An approach that helps fulfil this goal is to find structures where each level treats as uniformly as possible the objects under its responsibility.

Hardware Independence

In addition to the redundancy necessary to provide the fault tolerance capability the mechanisms should be as far as possible independent of special hardware features. This would allow their use in systems composed of available hardware components.

In addition to these design goals, a working prototype of the fault tolerant system should be provided. This should ensure that all design details have been considered. It also would provide a platform for further refinements and extensions.

1.2 Why use Conic for Designing a Fault Tolerant System ?

Distributed control applications have a multiplicity of reliability requirements, even within a specific application. Conic provides a programming and configuration environment for the implementation of distributed computer control applications, but no features particularly aimed at fault tolerance. Thus, there is a need to extend Conic to allow the implementation of fault tolerant systems. Conic was designed with the goal of providing sufficient flexibility for its area of application by the use of the modularity concept [Magee 84]. Basic support mechanisms are provided and additional modules are used to configure the system in order to meet the requirements of each application, these modules are programmed in the Conic language and supported by the basic mechanisms. In this way, a capability such as fault tolerance can be obtainned by providing standard system modules for its support. In addition this system provides an ideal environment for supporting systems where fault tolerance is obtained through re-configuration. Its interesting characteristics can be summarised as following:

Communication System Architecture. The Conic environment directly provides a communication system. The application modules can run in different processing stations and communicate with each other. The physical separation of the processors is an important fault isolation factor.

Replaceable Units. Application modules can be separately designed and compiled. Module instances can be assigned and created in stations at run-time, without interfering with the operation of other modules. This matches the concept of a basic replaceable unit and hence directly supports system reconfiguration and repair activities.

Message Passing. Intermodule communication is performed via message passing without using shared data objects. This is an important factor in confining errors within a failed module.

Indirect Naming. The interface of a module is characterized by the messages which can be sent and received by the module. Exitports are used to send messages out of a module, whereas entryports are used to receive messages into a module. Thus a message is directed not to an entryport of a receiving module but to an exitport of the sending module. A separate binding operation is used to link local ports to those of another module. This helps system reconfiguration in case of failure.

Strong Typing. Ports are strongly typed objects. This contributes to the removal of faults at the design stage.

It should be pointed out that we aim to identify general principles for the design of fault tolerant DCCSs. The above qualities of Conic help the achievement of this objective, thus the language itself does not become part of the problem, but it is a part of its solution. The relevant details of the Conic approach are presented in appendix A. In the remaining chapters of this thesis it is assumed that the reader is reasonably acquainted with the Conic concepts.

1.3 Thesis Contents

The contents of the rest of the thesis are summarized below:

Chapter II: A presentation of the basic concepts used in the thesis. It also defines the level of the system at which fault tolerance is provided and the two classes of application systems that are supported.

Chapter III: A discussion of fault tolerance techniques which we have considered and adopted in the design of the system.

Chapter IV: A description of the fault tolerant system and of the the two reliability services that it supports. These services are intended to provide the reliability requirements of each class of application systems which are identified in chapter II.

Chapter V: A presentation of the recovery technique we propose for the support of one of the class of application systems identified in chapter II. The language level mechanisms associated with this technique are also presented.

Chapter VI: A presentation and discussion of the implementation details of the fault tolerance support mechanisms.

Chapter VII: Relevant aspects of the application of the approach for the construction of fault tolerant application systems are discussed.

Chapter VIII: Conclusions and Suggestions for Further Work.

Appendix: The relevant details of Conic are presented. Its purpose is to provide a concise summary of the Conic concepts used in this thesis. The appendix is structured in three sections. Section A.1 presents the Conic programming approach, section A.2 presents the Conic configuration language, and section A.3 presents the support mechanisms which provide the Conic run-time environment.

CHAPTER II

BASIC CONCEPTS

In this chapter basic concepts for the development of the thesis are established. In sections 2.1 and 2.2 relevant concepts and the associated terminology required for the discussion of fault tolerance and reliability are presented. In section 2.3 we discuss how to achieve fault tolerance in a distributed computer system and propose a classification of system types which simplifies the study of the related issues, and the presentation of the thesis. Finally, in section 2.4 we discuss the whole system and introduce some of the assumptions that are adopted in the thesis.

2.1 Concepts and Terminology

A careful reading of the literature of fault tolerant computing indicates that several sets of conflicting basic concepts have been used. The importance of using standardized concepts, which can be understood in a uniform way is recognised by the researchers working in the area. Ideally this set of concepts should be simple so that it is easily understood, and have general applicability in that it can be used for the discussion of all aspects of a computer system. Several contributions in this direction were presented at the last fault tolerant computing symposium [Avizienis 82, Kopetz 82b, Laprie 82, Robinson 82, and Anderson 82], and a special group has been working with the same purpose [Lee 82]. There appears to be a trend to a general agreement on the basic concepts. Major disagreements have arisen in relation to the terms to be employed to name these concepts, the cause of this probably being that authors tend to allow their particular area of interest to influence in their terminology proposals. It is not in the scope of this work to propose a new set of concepts and terminology. Thus we adopt the basic concepts and terminology proposed in [Anderson 81, Anderson 82]. They are very general, and are directly applicable to different aspects of our proposals. A more informal view of fault tolerant computing concepts is available in [Hopkins 80].

2.1.1 The Concept of System

Any identifiable entity that maintains a pattern of behaviour at an interface with its environment can be considered a system. In general systems can be hierarchically subdivided into components. Thus the definition: a system consists of a set of components which interact under the control of a design. The design is itself a system, which has special characteristics: it supports and controls the interaction of the components. Thus the design also defines the behaviour of the system.

The interface is the place a system interacts with its environment. The environment is itself a system, and consequently an interface is the place of interaction between two systems. The external behaviour of a system can be described in terms of a finite set of external states and a function that defines transitions between states. The environment produces inputs as stimuli, and perceives the system state transitions at discrete instants of time. To each of the components of a system can be attached a state, the state of a component results from its interaction with the other components. The state of all the components of a system defines the **internal** state of the system. The external behaviour of the system can be described in terms of a function that maps the internal state to the **external** state.

Unfortunately, due to several reasons, the behaviour of a system may depart from that expected. In order to identify when this happens a specification of the system behaviour is required. There are many discussions about the characterization and even the possibility of obtaining a complete and correct specification for a complex system, but they do not seem to be very helpful and in order to make progress it is assumed that there is an **authoritative specification** of behaviour. This specification can be applied as a test in any situation to determine whether the behaviour of a system should or should not be considered acceptable.

2.1.2 Failure, Fault and Error

A **failure** of a system is said to occur when its behaviour deviates from that required by the specification of the system. It is essentially an event that can only be observed at the system interface with the environment. In order to define fault and error we have to refer to the internal state of the system.

An erroneous state is an internal state which could lead to a failure, which can be attributed to some aspect of that state. An **error** is that part of an erroneous state that can lead to a failure. A **fault** is the mechanistic cause of an error. Thus a fault is the cause of an error, and an error is the cause of a failure.

It should be noted that the only difference between a fault and an error is related to the system structure: A fault in a system is an error in a component of the system. But, the difference between error and failure is not only related to the system structure but also reflects the difference between a state and an event. A system is in error when its state is erroneous, whereas a system failure is the event of not producing behaviour as specified. The failure event occurs and may be observed only when the erroneous state is made visible at the system interface.

If the design of a system is assumed correct, then there is only one way for a failure to occur: due to the failure of one of the system components. However, this may lead to a contradiction because the failure of the component can also be due to a design fault, in that a component with inadequate specification was used. In the real world it is not always feasible to control all the factors which can lead a system component to fail and some probability of failure is inevitable.

2.2 Reliability and Fault Tolerance

Reliability is a word that has been used in the computing world with two basic meanings: In a narrow sense as a quantitative measure that aims to quantify the quality of service provided by a system with respect to error occurrence and effect. In a broader sense it refers to a wide range of approaches and techniques which are used in the construction of a system in order to ensure that it will operate according to its specifications, i.e., to achieve a reliable system.

2.2.1 Reliability as a Measure

A system is normally designed in order to provide a specified service. The situation in which a system fails to provide its service is termed a system failure. Reliability is usually characterized by a function of time, which provides a measure of the probability that no system failure will have occurred at a given time. In the practical usage the reliability function is used to estimate the mean time to failure (MTTF) of a system, or the probability that a failure of a system will have occurred at a given time. The subject is covered in many books and publications, e.g., [Shooman 68, Osaki 80].

The reliability measure is not sufficient to express the basic facts a system designer needs to know about a system. For example, considering that a system failure has occurred, a different measure -- **maintainability**(or repairability) is used to express the length of time the service is interrupted. And, considering the alternation of service provision and service interruption another measure can be defined -- **availability.** Although these measures are not independent each of them can usually quantify a desired quality for a system service.

Most systems provide a number of different services. Also, in general within the same system, several error modes can be defined. Hence these measures may be generalized in order to consider this fact [Laprie 82].

2.2.2 Reliability as an Achievement

In order to achieve reliability two complementary approaches are normally employed [Avizienis 76].

The first -- **Fault Prevention,** attempts to ensure that no faults will occur in a system. This approach, used in the development phase of the system, has two basic interactive steps: (1) **Fault avoidance** techniques are used in order to minimize the introduction of faults during the system conception, e.g., specification and design methodologies, and quality control. (2) **Fault removal** techniques are used in order to eliminate faults which still exist in the system, e.g., validation and testing techniques. Extensive coverage of the subject is available in [Adrion 82, Shooman 83]. The second -- **Fault Tolerance,** used in the operational phase of the system, uses a diversity of techniques that attempt to intervene and prevent faults from causing system failures.

A system in which fault tolerance techniques are employed is termed fault tolerant. In this work we aim to design a fault tolerant distributed computer control system. In general these systems will be composed by a large number of computers. Thus the probability of having failures is in some way increased. This is one of the reasons for application of fault tolerance techniques.

2.2.3 Redundancy

All techniques for achieving fault tolerance are based on the employment of redundancy. That is, extra elements that would not be required in a system which could be guaranteed to be free from faults.

Redundancy has been employed to provide fault tolerance within a computer unit. For example, extra bits are used to provide error correction or detection capability in memory systems. Now the technology allows whole computer units to be made at very low cost, and in very small chips. This fact has made economically and physically feasible the use of redundant computer units. Also our area of application has particular characteristics which make attractive the use of redundant computer units to achieve fault tolerance. Thus we aim to fault tolerance techniques based on redundancy available at the computer unit level.

2.2.4 System Specification and Reliability

The reliability of a system always involves a distinction between acceptable and unacceptable behaviour. For this to be possible a system specification which describes the behaviour of the system must be available. Systems specifications have two important characteristics. The first is **exactness**, without which it cannot be used as a test on the system reliability. The second is **multiplicity**, which means that different aspects of the system behaviour may be subject to different specifications. These specifications may be related in such a way that still permits us to define desirable behaviour of a system which has failed according to some of its specifications. Thus there is an overall specification that defines the standard of reliability which is demanded of the whole system, and other specifications that are related to specific aspects of a system. It is important to distinguish a failure of the system according to the overall specification from a failure with respect to each component specification. The former is obviously much more significant. This characteristic removes any contradiction that could result from the reading of the next sections.

2.3 Fault Tolerant Distributed Computer Control Systems

A fault tolerant system is one that includes internal mechanisms which provide the system with the ability to recover by itself from failures of its components. In order to design a system like this the first decision to be made is concerned with the level where faults will be assumed to occur. The second is concerned with the level at which failures will be recovered. A control application is implemented by a set of application modules which are supported by a distributed computer system. The distributed computer system consists of microcomputer

stations interconnected by a communication network. These systems can be constructed in such a way that a fault at a station does not affect the other stations of the system. Thus a fault at a station can directly affect only the application modules which are supported by this station. These modules can fail and consequently threaten the reliability of the application system.

The station failure independence can be used to achieve reliability for the application system. On the one hand the remaining stations can be used to provide fault tolerance for the application modules. On the other hand the remaining stations can support the necessary fault tolerance mechanisms. They have to detect failed application modules and intervene in order not to allow their failure to lead to an overall application system failure.

Our aim is to provide fault tolerance for the application system. In order to design the fault tolerance support mechanisms it is necessary to consider the relationship of the reliability of a module with the reliability of the overall application system. This is done in the next section.

2.3.1 Types of System

The systems of interest are classified in two abstract types. The classification intends to capture the relation between the failure of any of the constituent modules of a system and the failure of the overall specification of the system. The system types are characterized as follows.

Weak Failure Dependency: The system design is such that the standard of reliability required by the application that it implements is not broken if one or more of its constituent modules fail. That is, the service capability provided by the system is in some way degraded because of the failure of a module, but this is still sufficient to meet the application reliability requirements.

Strong Failure Dependency: The system design is such that the standard of reliability required by the application is broken if any of its constituent modules fail. That is, the failure of a module leads to a system failure.

In the context of the above classification the design is a result of a system specification. The specification is a result of the application requirements. The final product of the system development is a set of modules which implement the system specification. Ideally this set of modules should constitute a weak failure dependency system. In practice the result of the development process can result in a strong failure dependency system. The specific reasons for this fact to occur are very dependent on the particular application, and within its context also on design decisions.

Most existing systems fit the weak failure dependency type. However In the literature many practical examples of architectures for the support of strong failure dependency systems can be found, e.g., [Wensley 78, Hopkins 78, Sklaroff 76], and there are also works which are concerned with particular aspects of this type of system, e.g, [Scheneider 81, Lamport 78, Hecht 76]. The support of this type of system requires the use of special techniques and mechanisms in order to deal with their specific requirement. Although some common mechanisms can be used for the support of common requirements for both types of system. This thesis proposes an architecture for the support of both types of system. The abstract classification presented here is intended mainly to facilitate the study of related issues and the presentation of the support mechanisms. However, it should be pointed out that subsystems of both types may be found within the same application system.

2.3.2 Redundancy and System Types

In this section the motivation for using redundancy in each type of system is discussed.

2.3.2.1 Weak Failure Dependency

Here the main reason for the use of redundancy is to re-establish the quality of service provided by the system before a failure. This may be required for a number of reasons. For example, the interruption of the service provided by a failed module will not, by itself, immediately result in any irrecoverable damage to expensive equipment or endanger human life. While the service interruption initially does not have any catastrophic consequence this may happen if the interruption persists. A system like this has a dominant maintainability requirement for its modules. The simple interruption of the service does not mean a system

failure. But a system failure can occur if the module is not repaired within a specified time. The existence of redundant computer resources in this system is essential to repair the application system and meet the standard of reliability of the overall application.

Other factors, of an economic nature, can make the use of redundancy attractive. One of them is the cost due to lost production caused by the service stoppage. In some installations it is not possible or it is difficult to realize on-site maintenance. Also, in general, it is difficult to predict the time necessary to make the repair. Thus the use of redundancy can allow the service to be re-established quickly and reduce the cost resulting from the failure. Another factor is the cost of the maintenance itself, which can be very high and even dominate the cost of a system if its whole life cycle is considered. The use of redundancy allows the system to continue working in spite of failures. The failed modules can be taken to a centralized repair facility. This leads to a reduction of maintenance personnel and associated test equipment with a consequent reduction of costs. Quantitative support for this argument is available in [Souza 80].

Finally it should be considered that systems are built to be used, and are always expected to provide some service. The absence of this service will normally cause problems for their users. The cost of computing resources tends to be very low and constitutes a very small part of the cost of a system. Hence the use of redundancy has become attractive for a larger class of systems.

2.3.2.2 Strong Failure Dependency

In this type of system, redundancy is mandatory as a condition to meet the overall relibility requirement. In order to illustrate the idea we give two examples.

(1) A module can have its behaviour dependent on the history of its interaction with the other modules of the system and/or with its environment. The result of this interaction is represented by the content of some state local to the module. Thus if the module fails and its state is lost the system fails.

(2) The behaviour of the whole system can depend on a co-ordinated cooperation of its constituent modules. This co-ordination is essential for performing the specified service and thus obtaining the reliability characteristic. If one of these modules fails the necessary co-ordination is lost and the system failure occurs.

Fault prevention cannot be complete, thus in order to meet the requirements of this type of system the modules failures must be masked within the system itself. That is, the modules must be repaired in such a way that the other modules in the system will not notice the failure. Thus, in addition to the use of redundant resources some special technique must be applied in order to achieve the capability of masking the failure of a module.

2.3.3 Mechanisms for Fault Tolerance

From the discussion above two sets of support mechanisms which are necessary in the fault tolerant system can be identified.

One set of mechanisms is necessary to provide the capability of repairing individual modules in the application system. After a failure the entity implementing this mechanism will use the remaining computing resources and attempt to recreate the original set of modules of the system. Considering that a module failure is caused by a fault in a computing resource an additional entity is necessary in order to identify faulty resources. Thus the repair entity must cooperate with this entity in order to avoid the use of failed resources. This set of mechanisms would suffice in the support of weak failure dependency systems.

Another set of mechanisms is needed in order to achieve the reliability characteristic required for strong failure dependency systems. For reasons that will be clarified in the next chapters, the use of replicated modules is proposed in order to obtain the transparent recovery from failures of individual modules of a system. In this way a module at the system level would in fact be implemented by a set of replicas. Thus special techniques and corresponding support mechanisms are needed in order to support the use of these replicated modules and obtain the masking capability.

It is interesting to point out that the system repair capability is also necessary for strong failure dependency systems. Another system of modules can be defined if the replicated modules are considered as individual modules. In this system each of them makes a contribution to the achievement of reliability. The failure of a module of this system has no direct effect on the behaviour of the fault tolerant system, but indirectly it reduces its reliability. Thus a series of failures can lead the system to fail. Here the repair capability is needed to re-establish the original structure of modules of the system.

2.4 The Whole System

In this section a view of the whole system is presented. In order to simplify the discussion it is assumed that only one application module can run at a station.

2.4.1 The Application System

The application system consists of a set of application modules. Each application module is a distinct entity designed to perform specific functions or services. In order to perform these services the modules communicate through well defined interfaces. It is our fundamental assumption that modules communicate exclusively by exchanging messages. The interface of a module is then defined by the messages it can accept and by the messages it can generate. Thus the behaviour of a module can only be judged by checking the messages it generates. It is also assumed that the application modules are correctly designed and in consequence no erroneous messages are generated if no other faults occur (we do not treat the thorny issue of algorithmic faults in the software). In this case the messages generated by a module can fail either because of faults at the station where the module is running, or because of faults affecting the messages themselves while they are being transported between modules. Thus it is necessary to make assumptions about how these faults can affect messages. This is done in the next sections.

2.4.2 The Communication System

For many reasons it is attractive to identify within a distributed computer system a separate entity -- **The Communication System.** This system is in charge of transporting messages exchanged among the application modules. It is composed of a distributed part that exists at every station and by a global part represented by the communication network. For now consider that the stations are reliable and messages can fail because faults affecting the communication network, e.g., messages can get corrupted or be lost. But communication systems can be also fault tolerant and recover (with a high probability of success) from most of the errors. The design of such kind of system is itself a subject of intensive research, e.g., [Morgan 77, Wensley 78, Wolf 79, Kleinrock 80, Powell 82]. In order to limit the scope of this thesis we assume mostly that the communication system reliability is an issue that can be resolved separately. In the rest of the thesis we make more

specific assumptions about the qualities required for the communication system and discuss specific points related to our design. For now, it is assumed that the communication system reliability is enough to meet the reliability requirements of the application system.

It should be pointed out that the distributed part of the communication system is a part of the station. Thus faults at one station can also affect the messages being received at or transmitted from this station. The consideration of this fact is essential for the achievement of the reliability of the communication system.

2.4.3 The Station

Considering the communication system as reliable, the failure of an application module can only be caused by faults affecting the station where the module is running. Faults at a station can occur either in the hardware or in the design (algorithmic faults) of the base machine which supports the application modules (see appendix a.3 for an outline of the Conic Machine). In relation to the second type of faults we hope they do not exist, that is, the base machine is correctly designed. This assumption is essential, otherwise the fault tolerance support mechanisms could not be relied upon, since they also run on stations. Thus if a undiscovered fault exists in the basic machine design it has to be removed by a design change, which requires external intervention. In consequence, the only possibility of an application module failure is due to a fault in the hardware of a station. Hardware faults can be transient or permanent. Some errors due to transient faults may be recovered by retry, e.g., [Ciacelli 81], some errors due to permanent faults may be masked inside the station where they occur, e.g., by error correcting codes in memory. The station can make use of these techniques, but the effects of some hardware faults cannot be recovered at the station. This can affect the application module in many different ways. However, the module failure can only be observed by other modules through the messages which are generated by the module at fault. This permits the failure modes of a module to be classified into two basic categories.

(1) Error Confinement: The fault affects the module in such a way that it stops all its activities. Hence the module also stops generating any message. Thus no erroneous mesages are delivered by the module at fault, i.e, all the errors are confined within the module.

(2) Error Propagation: The fault affects the module in such a way that the module may continue operating. However, the messages the module may generate may not be according to its specification, i.e., errors can propagate outside the module because of erroneous messages.

These two failure modes are sufficient to describe the possible effects of a fault of the station hardware in the behaviour of an application module. It should be noticed that a failure in error propagation mode can change to a failure in error confinement; what is important is that in the mean time some erroneous messages may have been generated. In fact this difference is very important for the design of the fault tolerance support mechanisms.

2.4.4 Discussion

If error confinement is assumed the first consequence of a failure will be the interruption of the service performed by the affected module. There is no way this failure can directly lead other application modules to fail. But the failure can always be noticed when communication with the module is required, which can lead other modules to fail. If error propagation is assumed other modules can be directly affected by the fault. This happens because it is difficult to predict how the fault affects the module behaviour and its messages, e.g., a failed module can generate apparently correct messages, which can cause the system to fail. Some of the fault effects could be predicted and the application modules could have specific programming in order to recover from the corresponding errors. This is called forward error recovery and requires a special design for each different application system, [Randell 78]. It is also possible to use techniques that allow the fault tolerance support mechanisms to be orthogonal to the application system, thus being reusable. It is interesting to note that error confinement helps failure detection: A module can surely detect the failure of another module if it has been waiting too long time for a response.

The fault tolerance support mechanisms are implemented by modules very similar to the application modules. Thus, in their design it should be assumed that they have the same failure mode as the modules of the application system. At this point it should be noted that error propagation is more important for the design of the fault tolerance support mechanisms for strong failure dependency systems. It is reasonable to consider that some time after a fault the consequent

errors will propagate to other modules and be detected. For weak failure dependency systems this could be recovered by stopping the application modules, and restarting them without using the failed station. For strong failure dependency systems the failures must not be noticed by the other modules. Thus the fault tolerance support mechanisms must in some way mask the consequences of a module failure. In the next chapter we present a review of application independent techniques which can be used for the support of strong failure dependency systems. There, we also introduce the technique used in our design and discuss the failure assumption on which it relies.

2.5 Summary of the Chapter

This chapter presents the basic concepts for the development of the rest of the thesis. A precise set of concepts and terminology for fault tolerant computing is summarized: A **system** is an entity that mantains a pattern of behaviour at an interface with an environment. A **failure** of a system is said to occur when its behaviour deviates from that required by the specification of the system. An **error** is that part of the state of the system that can lead to a failure. A **fault** is the mechanistic cause of an error.

In a narrow sense **reliability** refers to a measure which expresses the probability that no system failure will have occurred at a given time, during the system operation. In a broader sense reliability refers to a wide range of approaches and techniques which are used in order to achieve the behaviour required by the specification. In order to achieve high reliability two complementary approaches are used: **Fault prevention,** applicable in the development phase, uses a diversity of techniques which intend to ensure that no faults will exist in a system; **Fault tolerance,** applicable in the operational phase, uses a diversity of techniques that attempt to intervene and prevent faults from causing system failures. A **fault tolerant system** is one that includes internal mechanisms which provide the system with the capability to recover by itself from failures. The specification of real systems has multiple aspects, thus there is an overal specification that defines the standard of reliability which is demanded of the whole system and other specifications that are related to specific parts of the system.

Our aim is to provide fault tolerance for the the application system. In order to design the fault tolerance support mechanisms it is necessary to consider the relationship of the reliability of a module with the reliability of the total application system. Thus two abstract types of application systems are defined according to the way the reliability requirement of the total system depends on the reliability provided by each module of the system: **strong failure dependency systems** and **weak failure dependency systems.** The motivation for using redundancy for achieving fault tolerance and thus reliability in each type of system is discussed. Two sets of mechanisms needed for the achievement of fault tolerance are identified. The first set provides the capability of repairing failed modules of the system and is required for the support of both types of system. The second provides the capability of masking failures of the modules and is required for the support of strong failure dependency systems.

Finally, a view of the whole system is presented. The communication system is assumed to be reliable, and the application modules and the base machine, which supports them, are assumed to be correctly designed. In consequence, application modules can only fail because of hardware faults at the stations. Faults can be assumed to lead modules to fail in two different modes: **error confinement,** and **error propagation.** Either failure mode can be separately adopted as the basic assumption for the design of fault tolerant systems. The failure mode adopted has direct influence on the design of the fault tolerance support mechanisms, specially those needed to support strong failure dependency systems.

In the next chapter we present a review of techniques for providing fault tolerance for strong failure dependency systems. Its objectives are to evaluate their use in our fault tolerant system and to provide a means of comparison with the technique we have adopted for our design.

CHAPTER III

FAULT TOLERANCE TECHNIQUES

In this chapter, a discussion of fault tolerance techniques based on massive redundancy and checkpointing is presented. The main objective of the discussion is to evaluate and select an application independent technique to be used for the support of strong failure dependency systems. However, the results of this discussion are also relevant in the design of the support mechanisms of the whole fault tolerant system; this point will be clarified throughout the thesis.

3.1 Massive Redundancy Techniques

These techniques are basically extensions of the classical triple modular redundancy scheme used in hardware, e.g., [Avizienis 71, Wakerly 76]. Their main attraction is that they can be used to mask failures at the module level. In order describe some issues that should be solved in order to extend this scheme for a distributed system, we refer to the figure 3.1. Each application module is implemented by three replicated instances of the same module type. In addition to failure independence of both the application and voter modules, two other requirements are needed for the correct functioning of the scheme:

(1) The clock skew is bounded and known.

(2) All fault-free modules will process identical inputs and generate corresponding outputs in approximately the same amount of time.

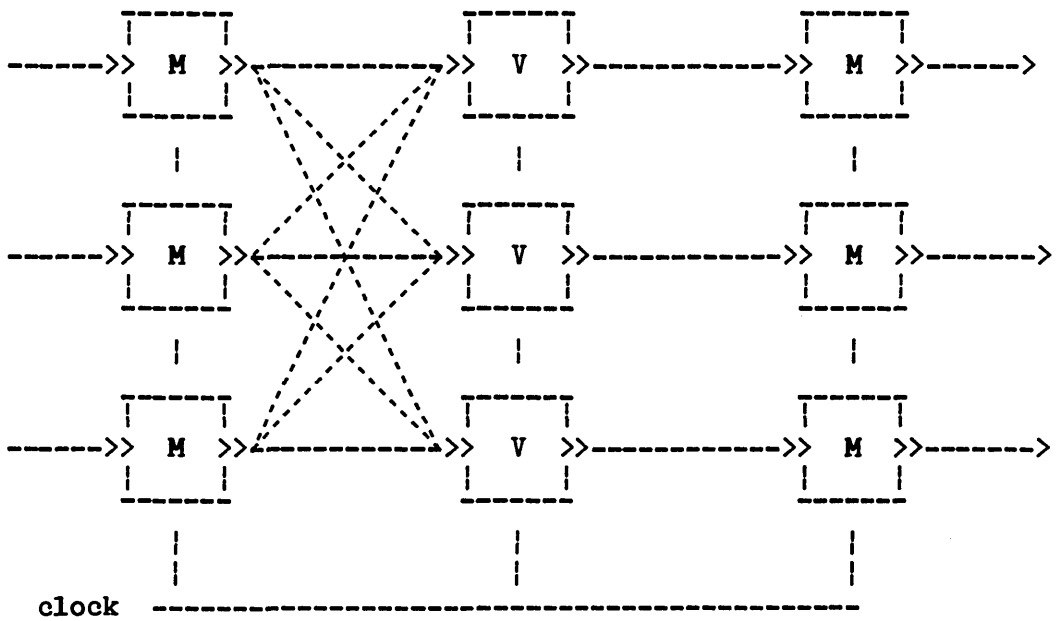
Requirement (1) ensures that a new set of inputs is available for the modules within a known time interval. Requirements (1) and (2) permits the calculation of the time interval after which the modules will have stabilized their outputs, i.e., the next voter inputs. Then, by a proper specification of the clock signal, fault tolerance can be achieved. If either requirement is not met, it is possible for the voters to be activated while their inputs are not compatible, because they are not synchronized. Incompatible outputs can then be emitted by the voters, leading the system to fail.

Now, consider that each module can accept input data from a number of different sources. We assume that the modules are deterministic in the sense that if they consume identical inputs in the same order, they produce identical outputs in the same order. Under this asssumption, it is obvious that if all replicas are fed with identical values of input data, they will produce identical streams of output data. Hence, in order to use the scheme in a distributed system, two requirements are necessary:

(1) **Synchronization of voter inputs:** Each replicated set of inputs must be voted at the right time.

(2) **Ordering of the replicated modules input data streams:** In order to produce consistent inputs for the voters all the replicas must receive identical streams of input data.

These requirements must be imposed independently at each station which supports a replicated module. In normal operation or in the presence of a single module failure, all remaining modules must take identical decisions, otherwise the system fails. Reference [Fischer 82a] gives an idea of the complexity of the problem. There, it is formally proved that even assuming reliable communications and error confinement for the modules, it is impossible to get consensus if a module fails at a critical time during their interaction. The only solution for the problem is to include some time reference in the system. In the next sections we consider two different approaches for the consideration of time.



V -- Majority Voter M -- Application Module

Fig. 3.1 Classical Triple Modular Redundancy Scheme.

3.1.1 Global Time Reference

In this approach a global time reference(GTR) is available for all stations in the system. Thus it is possible to relate all the events in the system to this reference. The approach was first proposed by the designers of the SIFT system [Wensley 76, Wensley 78, Goldeberg 80]. The main characteristics of this system are summarized below.

3.1.1.1 The SIFT system

The design of SIFT basically implements in software the characteristics of triple modular redundant circuits implemented in hardware. Every application module has its activation synchronized with the GTR, and these activations are periodically repeated. A fixed activation period is specified for each module. Replicated modules in the same set are activated at the same time instant, but run at different stations.

Conceptually, the structure of SIFT based systems can be seem as consisting of the application modules and of a special data module. This data module contains all the state information of the application being controlled. At each activation the application modules get from the data module, the data needed to perform their calculations, and the result of the calculations are also returned into the data module. In practice, the outputs of the application modules are not directly deposited into the data module, but each application module broadcasts, through the communication system, the results of its calculations to all target modules. Immediately before a module has to be activated, the results generated by replicated modules are voted upon in order to obtain a reliable input for processing in that activation. The synchronization of the voter inputs is achieved provided that both the time a module takes to process its inputs, and the transport delay of the communication system are bounded. The sum of these times must be such that a consistent set of replicated outputs is available for voting at the time a module needs to use them.

The ordering of inputs is automatically set by the fixed activation time of each module. Thus there is no need for any protocol to decide which input will be consumed by each replica. The application designer has to make sure that the data to be consumed at each module activation is present at the station where the module is running at the time immediately before the activation. This depends on the relative allocation of the modules activation time in relation to the GTR, and on

the factors described in the paragraph above. It is worth noting that the scheme does eliminate the possibility of any nondeterminism in the choice of inputs to be consumed. This fact seems not to be a problem in the class of applications SIFT addresses, i.e., sampled data control applications.

3.1.1.2 Interactive Consistency

If three replicated modules are used, the success of the majority voting scheme depends on an initial assumed condition that at least two modules of each replicated group have produced correct and identical results. Thereafter three replicated voters and modules, with at most one faulty voter or module per group, are sufficient to mantain the condition of at least two correct and identical results. The team working in the design of SIFT has verified that this condition cannot be always met in some cases by the use of only three channels [Pease 80]. These cases are: (1) The input of information from an unreplicated source, e.g., an unreplicated sensor or an unreplicated module; and (2) the case where a consensus must be reached on different values generated by independent sources, e.g., data generated by replicated sensors, clock readings generated by different stations used for the (re)synchronization of their clocks, and error reports generated by each station. In these cases a single module failing in a "malicious" mode can defeat the TMR scheme. This is examplified in figure 3.2, adapted from [Frison 82]: Consider that the modules operate in a SIFT like fashion. Each sensor module periodically produces an output which is sent to all processing modules. The processing modules pick-up the median value of the outputs produced by the sensor modules and use this value as output. As a consequence of the malicious behaviour of the faulty sensor module inconsistent outputs are produced; it can be proved that inconsistent outputs can occur regardless of the algorithm the processing modules can use to select their outputs. It should be pointed out that this is a consequence of the error propagation environment. The general solution of this problem requires at least $3f+1$ modules to tolerate up to f failures; and a special protocol is needed to ensure the initial condition. This protocol relies on an interactive message interchange among replicated modules, hence the name interactive consistency. This special case will be further discussed in the next section.

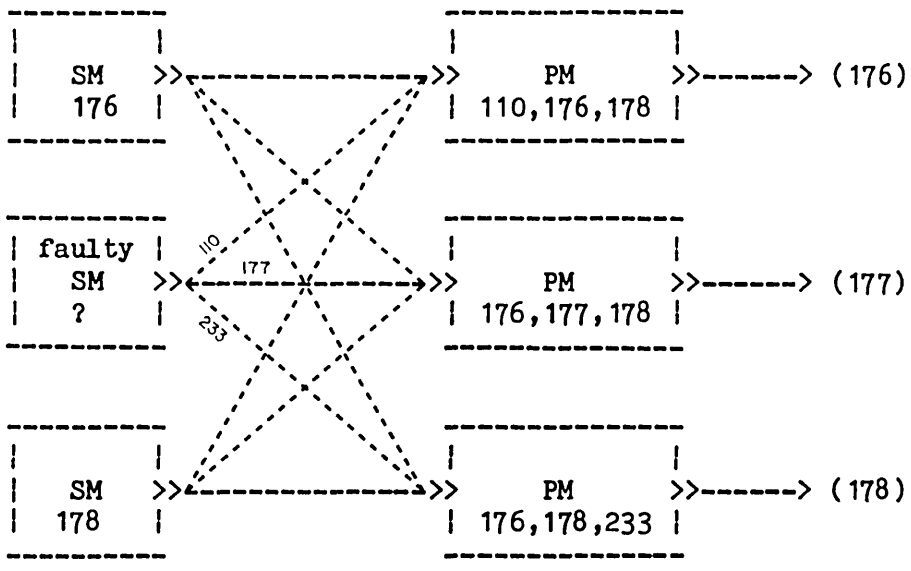


Fig. 3.2 Loss of Consistency in a TMR Scheme

3.1.2 Local Time Reference

This approach for obtaining fault tolerance by using N-modular redundancy is proposed in [Leung 80]. It intends to provide fault tolerance for the hardware modules of a data-flow multiprocessor. However, in principle, the design can be used in any system based on message passing. The approach allows fault tolerance to be obtained by decisions, based on time references, taken locally at each station without using any global-time reference; and without any assumption dependent on the other modules of the system. The principal attraction of the approach is that it allows application modules to be independently designed, thus meeting the modularity requirement.

3.1.2.1 The Approach

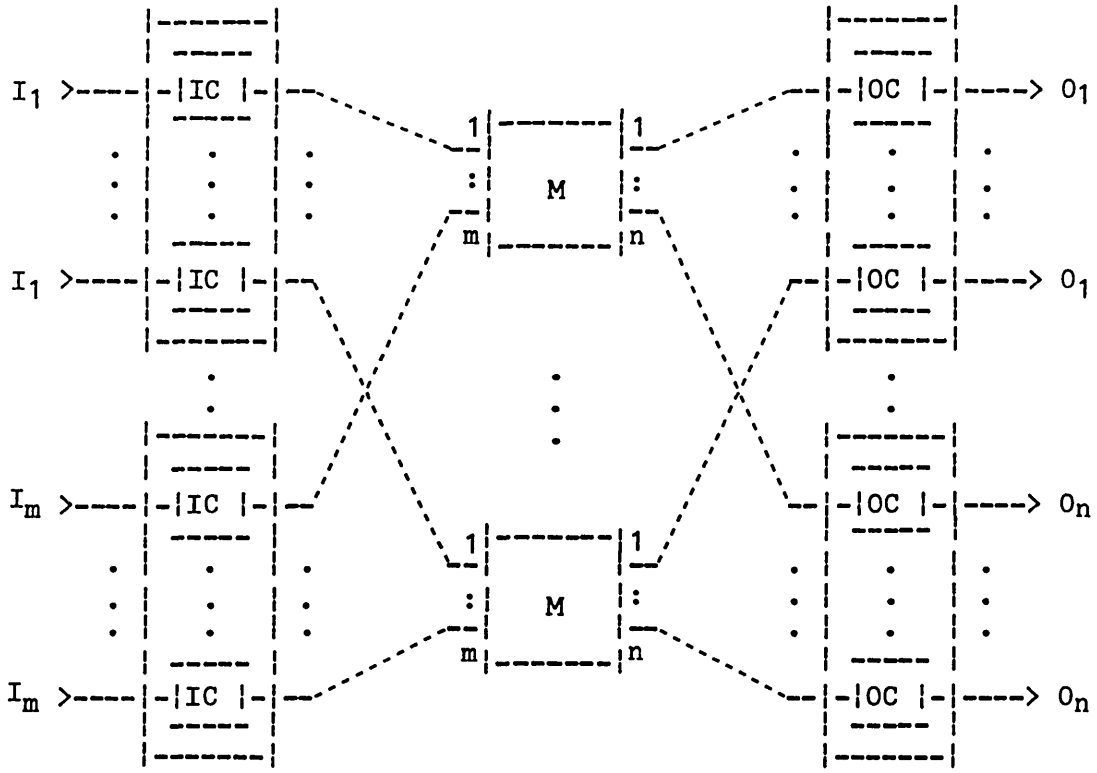
The fundamental requirements for fault tolerance: synchronization of voter inputs, and the ordering of inputs of replicated modules, are obtained by the use of special standard control modules. These control modules use in their design the interactive consistency protocol proposed by Pease et al [Pease 80], for exchanging information among modules in a distributed system. A minimum of four replicated modules are required in order to tolerate a single failure. Two rounds of communication are performed. In the first round the modules exchange their private values (messages). In the second round they exchange the values obtained in the first round. The exchange having been completed, majority voting can guarantee that:

(1) The message transmitted by a fault-free module will be known to every fault-free module, and

(2) The fault-free modules will agree on the contents of messages transmitted by faulty modules.

In general the algorithm requires at least **3f+1** modules to tolerate up to f module failures among them. A negative result which states that this fault tolerance capability cannot be achieved with less than 3f+1 modules is also provided in [Pease 80]. It was also proved that any algorithm that assures interactive consistency in the presence of f faulty modules requires at least **f+1** rounds of communication, and that the amount of message values that need to be interchanged and stored in each protocol execution is aproximately (3f+1)**(f+2) [Fischer 82b]. These values quantify the cost of any protocol to establish interactive consistency. They are valid on the assumption that communications are reliable.

Figure 3.3 represents a general replicated module structure proposed by [Leung 80]. A set of input control modules(**ICs**), and a set of output control modules(**OCs**) are required to protect respectively each input and output of the application module. In receiving an input message an IC module executes the interactive consistency protocol with the other ICs which are protecting that port. As a result of the interaction, a reliable input is forwarded to each replicated module. The protocol also guarantees that these inputs are generated within a given time interval. The OC modules also perform a similar role. Assuming that the application modules can process input messages within a specified time interval it is guaranteed that the OC modules will also generate their output messages within a specified time interval. This permits the ICs of a target module to time-out when awaiting inputs from failed modules, thus avoiding interference to the execution of the algorithm, and allows the required synchronization to be kept in the system.



IC -- Input Control OC -- Output Control M -- Application Module

Fig. 3.3 Structure of a Replicated Module.

In data-flow applications the application modules can, in most cases, be designed in such a way that their outputs depend only on the set of inputs received. However, in some cases the order in which messages are accepted will influence the output messages (In Conic this is the most general case). In order to cater for these cases it is proposed to introduce another control module, called the merge module, which has the role of ordering the input messages to be consumed. The replicated merge modules receive messages coming from two different ports and delivers them in the same order for consumption. If more than two inputs need to be merged these modules can be cascaded. This introduces overheads that can only be reduced by adopting a special style of programming.

The same algorithm proposed by Pease et al [Pease 80] is used to obtain the interactive consistency required in some cases for GTR based systems. However, in SIFT, the fact that every event can be related to the GTR simplifies the design of the control modules. In general, 3f+1 modules are also required in order to tolerate f failures. Although, if some more restricted failure modes are assumed a reduction is possible [Frison 82].

3.1.3 Discussion

Systems based on the global time approach have to meet tight synchronization requirements. A special design for all the system modules is required in order to meet these requirements. In addition the modules have to use a special style of communication in order to simplify the implementation of the system. The difficulty and resources for obtaining interactive consistency also have to be considered. It is possible to support a SIFT like architecture in Conic. However, it is an open question as to whether or not the incurred programming style is suitable for the implementation of process control applications in general.

In relation to local time based approach, the main problems are the large number of messages required and the amount of information that needs to be stored. This makes its use unpractical in systems where it is necessary to obtain interactive consistency for every message consumption. However, the approach can be used when a lesser degree of interaction is required, e.g., in the Space Shuttle computer system [Sheridan 78] the same application system is replicated in four computers, and is designed in such a way that they can provide consistent outputs provided that their external inputs are consumed in the same order. This reduces the places where the protocol needs to be used. Another interesting example is available in [Ihara 78].

The study of massive redundancy approaches provides a clear view of the problems and costs of designing fault tolerant systems under the error propagation assumption. It is worth pointing out that the assumption of intermediate failure modes does not simplify the problems. It would be necessary to interchange more information, keep more state information, and consider problems that can be caused by the view each participating module has of this state. The potential advantage of these approaches is that the computer modules are not required to have any built-in error detection mechanism in the hardware, since this can be provided by high-level mechanisms, which can be implemented in software. However, there is a trend to incorporate built-in error checking mechanisms at the hardware level. This allows simpler and less expensive use of replication for obtaining fault tolerance. This is further discussed in section 3.3.

3.2 Checkpointing Techniques

Checkpointing techniques aim to preserve enough information about the state of a system so that the system can be restarted after a fault has occurred. These techniques have been widely used to provide a degree of fault tolerance for uniprocessor systems. In the context of multiprocess based system checkpointed information can be used to implement backward error recovery [Randell 75, Randell 78]. This type of recovery involves abandonning work which may have been performed erroneously due to a fault, and then repeating the work, hopefully correctly, after the faulty component has been removed from the system. In the next section we discuss backward error recovery in a distributed system.

3.2.1 Checkpointing in Distributed Systems

In a distributed system checkpoints are taken for the individual modules at relevant points of their operation. The checkpoint operation is performed independently by each module. When an error is detected it is necessary to obtain from the checkpointed information a set of consistent states for the modules of the system. This is needed in order to put the system in a consistent state which could validly have existed at some time whilst the system was functionning correctly. Due to error propagation, more than one module can be in an erroneous state. Also, when a module has its state restored, all communications that it has performed, after reaching that state, should be revoked. Consequently these modules, with which communication has occurred, must also have their state reset in order to impose a valid system state. A special protocol has been proposed in order to solve the general case of establishing a set of consistent module states [Merlin 77]. A simplified version of this protocol has been proposed in [McDermid 80]. But, in general it cannot be guaranteed that a consistent set of states different from the trivial initial state of the system will be reached. This is called the "domino" effect.

One solution for this problem is to arrange that no information will be propagated from a group of modules which need to interact in order to perform an action. If any error occurs during the action execution it is only needed to restablish the modules to the state they had immediately before starting their interaction. This makes any intermediate states invisible for the rest of the modules, i.e., the action is atomic. A special protocol, named two phase commit protocol is

used in order to reliably coordinate the termination of an action [Gray 78]. Another possible strategy is to restrict the communication of the modules in order to make the system immune to the domino effect [Kim 79, Russel 80].

Assuming that a consistent state has been reached, a second problem arises: As a consequence of the re-execution of the modules, it is not guaranteed that the outputs at the system interface will be repeated. Considering that some outputs could have been generated in the first execution, and that different outputs are generated in the second, the coherence at the system interface can be lost. This does not allow the use of the approach in applications where this coherence is required. One way of dealing with this problem is to design the distributed program in such a way that it always produces the same results when started from the same state. Although this requires restrictions on the structure and communication of the modules [Bernstein 79].

A third problem is caused by the possibility of error propagation beyond the system interface. An error detected at a later stage can invalidate a previous output. However, an output delivered to the environment cannot always be recovered. Thus before delivering an output it is necessary to be sure that no errors have occurred during its calculations. This is difficult to obtain because of error propagation and the distributed environment. These problems limit the application of backward error recovery in applications directly interacting whith the environment. This was also noticed in [Kopetz 83]. The only solution for fault tolerance (under the error propagation assumption) is to provide forward error recovery. This requires the provision of a compensation procedure to undo the effects of outputs [Shrivastava 79], which is complicated and does not meet the requirements of some applications.

Two additional issues can be enumerated: First, in order to allow the reallocation of modules to other stations in the system the checkpointed information must be accessible to the other stations in which the modules can run. The integrity of this information should also be preserved. This introduces the requirement of a reliable data base. Second, it is necessary to ensure that the set of checkpoints stored does not grow too large. It is difficult to devise algorithms for removing checkpoints amenable for distributed execution. Also this requires some assumptions about error propagation to be made; because in principle it is always possible to imagine an error for which one of the deleted checkpoints was essential for recovery.

3.2.2 Discussion

Most of the complexity in supporting backward error recovery and the related issues pointed out in the previous section is a result of the error propagation environment. If it is assumed that the modules fail in error confinement mode, the implementation of backward error recovery is considerably simplified. Also its related problems can be eliminated. Only one recovery point needs to be kept for each module. After the detection of a module failure it is sufficient to restart the module with the state taken at this recovery point. The module re-execution and a simple protocol can ensure that all the communications which were being performed by the module before the failure are consistently completed. This allows module failures to be masked; with the exception of some performance degradation. It is also possible to ensure that any output generated by the system of modules is repeatable; thus the control capability is not disrupted because module failures. These qualities can be obtained for a system of Conic modules without incurring any unreasonable restrictions on the style of programming. Also they require the support of very simple mechanisms. Our approach for providing fault tolerance will be presented in the next two chapters. The next section discusses techniques to enforce error confinement.

3.3 Error Confinement

The validity of the error confinement assumption is directly dependent on the coverage of the error detection mechanisms. In the following paragraphs we discuss available options for obtainning error detection capability.

Executable assertions were originally added to software for proof of correctness of a program. However they can be also used to detect errorscaused by faults affecting the hardware [Randell 75, Andrews 79]. Recently assertions were proposed as a means to be used to confine errors within a module in a message passing system for control applications [Koptez 82a]. In this proposal an assertion is evaluated just before a message is sent from a module. In detecting an error the module stops sending messages. The advantage of this technique is that it does not depend on hardware support. The disadvantage is that an assertion must be explicitly programmed for each case. Another option would be to run an application independent error checking routine in place of the assertion. Some systems use a special error checking module which runs periodically or whenever that the processor is idle. In

general, it can be said that these approaches validate, with some probability, the error confinement assumption. However, more guarantees can be provided by hardware based techniques.

Most commercial computers in use today incorporate some form of error detection. The most common approach is to use code based techniques, of which the most simple is parity for single error detection. Although interface circuits for Hamming coded memories, providing single error correction and double error detection are currently offered by several semiconductor manufacturers. Error detection capabilities are currently being offered at the microprocessor chip level, e.g., the Motorola 68000 microprocessor can detect internally invalid operation codes and ill formed addresses; its memory management unit can detect abnormal memory accesses such as fetch of data in program area, fetch of instructions in data area, write in ROM, etc. These capabilities together with coding techniques can be used to implement efficient error detection capability [Marchal 82, Schmid 82].

The most promising approach seems to be the use of totaly self-checking logic circuits; which guarantees that any internal fault is detected and that any erroneous output is signalled at the circuit interface. In this direction two approaches are possible. The first is to develop specific self-checking modules which allow error detection to be obtained by the use of standard microprocessors [Rennels 78, Chavade 82]. The second is to incorporate it directly in processors and components [Sedmak 80, Disparte 81]. The use of these approaches allows a near 100% error detection capability without needing to duplicate all the system components, e.g., a totally self checking microprocessor requires 58% more area in the chip [Disparte 81].

The set of error detection techniques to be used depends on the reliability requirements of the application. Hardware based techniques are more attractive because they are very efficient and do not require programming effort. The current price of the hardware makes them cost effective. In general, the capability of error confinement simplifies the achievement of fault tolerance for both types of systems, particularly for strong failure dependency systems. In addition it also simplifies the programming of the software for the location of faulty components as noticed by [Goetz 78]. These points will be demonstrated by the simplicity of the fault tolerance support mechanisms which are proposed in this thesis.

3.4 Summary of the Chapter

Massive redundancy techniques can be classified in global time reference and local time reference. They can provide very high reliability at a cost of at least three times the number of resources needed in a normal system. The global time reference based system is simpler than local time based reference system but the effective use of the technique requires restrictions in the style of programming and do not allow a system to be built out of independently designed modules: The activation of each module has to be synchronized with the global time reference, this is necessary for the proper functioning of the majority voters used to mask module failures. This fact has to be taken into account when designing the modules of a system. The local time reference based approach requires 3f+1 modules to tolerate f module failures. In addition it uses a large number of messages and requires the storage of a large amount of information for each decision related to the consumption of an input message. Its only advantage over the global time based technique is that modules can be independently designed. However, for both techniques the transparency goal is difficult to achieve. Massive redundancy approaches are specially designed to work under the error propagation assumption, if only the error confinement assumption is considered less expensive systems can be designed by the use of checkpointing techniques.

Checkpointing techniques aim to preserve enough information about the state of a system so that the system can be restarted after a fault has occurred. After a failure, the state of the modules of a system is restored to a previous, hopefully correct, state and execution is continued. This is called backward error recovery, a special protocol is needed to enforce this behaviour. However, this activity is prone to a "domino effect" by which the initial trivial state of the system is reached, this problem cannot be eliminated for a general system structure. In addition, as a consequence of the re-execution of concurrent modules, different outputs can be generated at the system interface, thus a control sequence interrupted by a failure cannot be re-taken. Also, it is difficult to stop the propagation of errors beyond the system interface, i.e., an erroneous output delivered to the environment cannot be recovered. These drawbacks make backward error recovery innappropriate for programming distributed control applications. However, their cause is the error propagation failure mode

assumption. If it is assumed that modules fail in error confinement mode, backward error recovery can be simplified and its drawbacks eliminated.

In this thesis, we adopt the error confinement assumption and use a checkpointing based technique. This option requires the simplest fault-tolerance support mechanisms, uses less resources than those required by massive redundancy techniques, and can meet the requirements for supporting strong failure dependency systems. The approach is presented in chapter V. It permits a system of Conic modules to tolerate failures without any apparent interruption of the capability of controlling the application. The degree of reliability of the approach depends on the validity of the error confinement assumption. Available techniques to ensure the validity of this assumption were discussed in section 3.3.

CHAPTER IV

THE FAULT TOLERANT SYSTEM

In this chapter an overview of the fault tolerant system is presented. The main goals are to present our proposal for providing reliability requirements for the two types of application system that were identified in chapter II: weak and strong failure dependency systems, and to introduce the system modules which are needed to provide the required fault tolerance. Some relevant aspects of the design are discussed; however not all aspects are considered in this chapter: Language level mechanisms and a recovery technique used for the support of strong failure dependency systems are presented in the next chapter. The design and implementation details of the fault tolerance support mechanisms are presented in chapter VI.

4.1 Assumptions

In the following we summarize all the assumptions which were introduced in the previous chapters. They are essential for the the correct functioning of the fault tolerance techniques that are proposed in this thesis.

a. **Correct Design:** There are no design faults (algorithmic) either in the system provided facilities or in the application defined modules.

b. **Error Confinement Mode:** This assures that the failure of a module will be consistently seen by any other module of the system, and that the failure of a module does not cause other modules to fail through error propagation. The validity of this assumption is directly dependent on the coverage provided by the error detection mechanisms at each station. The techniques that ensure the validity of this assumption were discussed in section 3.3.

c. Reliable Communication System: Two explicit assumptions are required: (1) If a message M is sent and received, then the received message is precisely M. (2) A message issued from a station will reach its target station within a specified time interval. Assumption (1) can be met with high probability of success by the use of standard error detection/correction techniques [Hamming 50, Peterson 72]. Assumption (2) requires redundant communication paths and special fault tolerance techniques. By this assumption, we are excluding the possibility of a partitioned communication system. This is fundamental for the correct functioning of the mechanisms proposed in this thesis.

d. Bounded Response: The system can assure that some specific modules can always respond to incoming messages within a predictable time interval: i.e., if they are operating normally. In Conic this can be achieved by assigning a proper priority for the module tasks and by a proper module design. This assumption is required for failure detection.

In addition to these essential assumptions, we make a "convenient" assumption: any fault at a station will stop completely all activities in the station; i.e., all the modules in the station are stopped; the alternative would be to consider that a fault can affect only some of the modules of the station. This assumption simplifies the presentation and implementation of a prototype of the fault tolerant system. The subject is is further discussed in section 4.7.

4.2 Definitions

The following definitions intend to establish a basic vocabulary to be used in the remaining parts of this thesis.

4.2.1 The Application System

The application system is constituted by a set of Reliable Modules, Rmodules for short, which cooperate in order to provide the application functions. The application system is defined at a logical level by the application designer. It should be pointed out that the application system for a large control application will probably be composed of several application subsystems; each of them being associated to a particular function. They may cooperate in order to execute an overall control function. In this thesis we assume that the reliability requirements of each subsystem can be met separately.

4.2.2 Rmodule

At the logical level, an Rmodule is similar to a normal Conic module. In addition, one specific reliability service is associated with the specification of an Rmodule. The particular reliability service specified for a given Rmodule will determine the reliability requirement met by the Rmodule, and determine how it will actually be implemented in the running application system configuration.

4.2.3 Reliability Services

The fault tolerant system provides two reliability services, which meet the reliability requirements of the two types of systems identified in chapter II.

Cold Standby: Which meets the requirements of weak failure dependency systems.

Hot Standby: Which meets the requirements of strong failure dependency systems.

The application designer specifies for each Rmodule the type of reliability service needed to meet the application requirements. Standard development tools can automatically perform part of the translations needed to implement each service, but additional run-time support is also required. The services and their support are presented in the next sections.

4.3 Cold Standby

Each Rmodule using this service is implemented by one Conic module instance. The designer can specify a set of alternative stations where the Rmodule instance can run. He can also specify a preferential order for using these stations.

At the application system start-up, the system chooses one of the user specified stations and performs all the actions necessary for the normal operation of the Rmodule instance there. During the operation, the station hosting this Rmodule instance can fail. When such a failure is detected, a new instance of that Rmodule can be created; provided that there is a station available in its station set. The system performs all the actions necessary to instantiate the new Rmodule instance in that station. It is worth pointing out that:

a. No state information is preserved between two successive activations of Rmodule instances; i.e., the service only provides a volatile storage for the Rmodule. This is illustrated in figure 4.1.

b. A non-negligible time interval is necessary for the system to perform all the actions required to re-integrate a standby Rmodule instance in the application system, since the Rmodule instance has to be created and its ports linked. This time interval can increase if the same fault, or a concurrent one, causes the system services which are used to re-integrate the Rmodule to fail.

c. An implicit requirement for the use of this service is that the other Rmodules that can co-operate with a given Rmodule have provision to continue working in the case that this Rmodule fails. Also, after the recovery of a Rmodule, they should begin to co-operate normally.

d. A standard Conic module is equivalent to an Rmodule which has only one station in its station set.

The points enumerated above restrict the use of this reliability service to the class of application systems having weak failure dependency. The service only assures the creation of a new instance of the Rmodule; all other activities concerning recovery are left to the application program, e.g., recovery of the instance state. However, a mechanism that can help in the implementation of application dependent recovery strategies is discussed in section 7.1.

Fig. 4.1 Cold standby Rmodule

4.4 Hot Standby

An Rmodule using this service is implemented by two identical module instances. As in the cold standby case, the designer specifies a set of stations for each Rmodule, but at least two stations are required at a time. The system allocates each instance to run in a different station in that set. Each of these instances is fully prepared to perform application processing without requiring any further action from the system modules, e.g., port linkage. However, at any one time they will be performing different roles: One of them will be actually processing messages and cooperating with other Rmodules of the system. This instance is called the **active instance**. The another one, is called the passive instance, and does not perform any processing. Its role is to keep an updated copy of the application state information being produced by the active instance. This state information is transferred to the passive instance at relevant points in the operation of the active instance. This transference is done atomically: either the entire state information is completely transferred, or the passive instance state is not changed at all. Thus a failure during the transference does not leave the state of the passive instance inconsistent.

At the configuration start-up, the operating system creates and starts the two instances, ensuring that one takes the active role and the other takes the passive role. During system operation either the active instance or the passive instance may fail. In the first case the passive instance, after detecting the failure, changes its role to active and continues application processing. In the second case the active instance processing is not affected by the failure. The system can create a new instance to replace the failed one; this instance will automatically take the passive role. This is assured by an underlying mechanism, which will be presented in section 6.2. It is worth noting that:

a. The Rmodule state information is preserved when the active instance fails and the available passive instance takes the active role; i.e. the passive instance provides a stable storage for the Rmodule, [Gray 78, Lampson 81]. This is illustrated in figure 4.2.
b. The passive instance is fully prepared to perform application processing. Thus, the time interval needed to repair the subsystem is bounded by the time taken to perform the active instance failure detection and activate the passive instance. This repair time is an useful measure in evaluating the real-time response of Rmodules using this service (See section 7.2 for related discussion).

c. An implicit assumption for the reliability of the approach is that during any period of time at most one of the two instances has failed. In practice, this period of vulnerability is minimized if another instance is created to replace a failed one. In this thesis we consider that duplication is enough to meet the reliability requirements of most applications; however, it is possible to allow for more than one passive instance. This requires a more complicated protocol to keep the state of all passive instances consistent, and a protocol to elect an active instance in case of failures. The problem is similar to that treated in [Garcia 82].

The points enumerated above allow the support of strong failure dependency systems. In principle, the state information preserved in stable storage is enough for the achievement of fault tolerance. However, this requires application dependent recovery algorithms, which should be provided by the programmer; the subject is further discussed in section 6.2.1. In the next chapter we present an application independent recovery technique, for use by hot standby Rmodules, for the programming of strong failure dependency systems. This technique allows the transparent recovery from failures. The application of the hot standby service in the construction of application systems is discussed in chapter VII.

Fig. 4.2 Hot standby Rmodule

4.5 Support for Reliability Services

Each reliability service needs specific support in order to be used in a practical way in the implementation of several applications. Some activities are performed off-line in a development system, while others are performed on-line when the system is operating.

4.5.1 Off-line Support

Off-line support is performed in two phases:

4.5.1.1 Configuration Specification

In Conic, the system designer specifies the logical configuration of the application system through a separate configuration language [Kramer 83]; a summary and an example are also provided in appendix A.2. In order to allow the specification of fault tolerant applications, this basic language is extended in order to allow:

(1) The specification of the reliability service required by each Rmodule.

(2) The specification of configuration rules which are needed to dynamically control the instantiation of the Rmodule instances.

Capability (1) is readily acquired by associating its required reliability service to each Rmodule name. In general, capability (2) cannot be specified in a simple way: -- it depends on the capabilities of each station of the system and on the particular use the application designer wants to make of them. However it can be observed that in control applications the designer needs to allocate specific module instances to specific stations, e.g., because of I/O interface or communication delay requirements. Thus, a configuration rule is always needed to specify the set of stations where instances can run. In general, additional rules are needed to fully specify the configuration control algorithm; this is discussed further in section 4.6. For now, we assume that (1) and (2) are specified by a slightly modified create declaration:

```
CREATE   Rmodulename(parameterlist):moduletype, servicetype
         AT  stationset;
```

Servicetype specifies the required reliability service of the Rmodule, and stationset specifies a set of station names where the instance can run. Parameterlist defines the instance parameters as usual for a standard Conic module. The physical address corresponding to a given station name is assumed to be available from a data base in the development system.

With the exceptions of the extended create declaration, and of a special port primitive (that will be presented in the next chapter), the application logical configuration is specified as in the standard Conic configuration language.

4.5.1.2 Configuration Translation

The configuration specification is processed by a translator program which performs validation of the specification and translates it to a symbol table form, which is kept in a configuration description file. This file also contains the addressing information required by the run-time support mechanism to produce load images for the Rmodule instances. Rmodules using the hot standby service require specific support in the development system. This will be fully clarified along the thesis.

4.5.2 On-line Support

On-line support is provided by two system modules: the **configuration manager** and the **status collector**. They provide the capability of controlling the instances of the application system configuration at run-time. For the sake of brevity, we say that they provide the **configuration management service**. Fault tolerance capability for the configuration management service modules is obtained by using the hot standby service. This and other implementation details are presented in section 6.1. The operation of the configuration management service modules is described in the next subsections.

4.5.2.1 Configuration Manager

The running configuration of the system is controlled by a system module called the **configuration manager**. This module has access to the configuration description and to the configuration rules. It can start, stop, and control the configuration in order to support the reliability services. These activities depend on the status of the stations of the system. The set of stations defines the **physical configuration** of the system. The set of module instances corresponding to the application system specification defines the **logical configuration**. In order to start a configuration, the configuration manager acquires the status of the physical configuration and according to the configuration rules chooses a mapping of the logical configuration to the available stations. Next it performs the configuration control operations to start the logical configuration. It does so by sending messages to the operating system at each station. During its normal operation, the configuration manager receives station status reports from the status collector. When a change in the status of a station is noticed configuration operations may have to be performed; this decision is based on the configuration rules. The approach we have adopted for configuration control is discussed in section 4.6.

In addition to the functions described above, the configuration manager can answer queries about the status and/or report the status changes of Rmodules. This is further discussed in sections 6.1.3 and 7.2.

4.5.2.2 Status Collector Module

This module periodically collects the status of every station in the physical configuration. Changes of status occur either when a station fails or when a station joins the system, e.g., after being repaired. Any change noticed in the status of a station is reported to the configuration manager. Figure 4.3 presents a view of the interaction of the on-line support modules. The capability of station failure detection relies on assumptions c and d (section 4.1); i.e., reliable communication system and bounded response. The techniques used for detection of station failures and for assuring consistency of status reports are discussed in section 6.1.2.

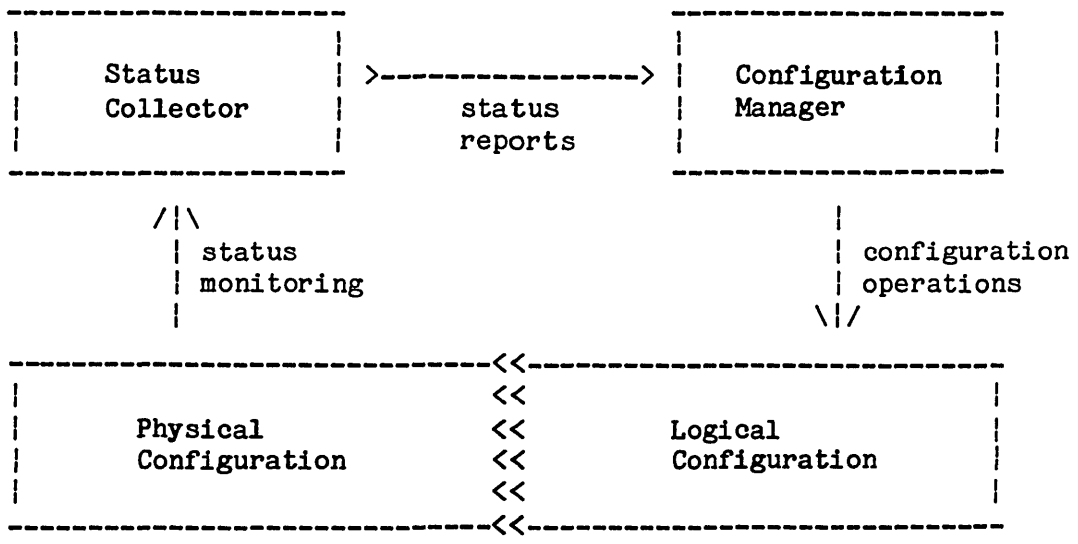


Fig. 4.3 View of the on-line support mechanisms.

4.5.2.3 Treatment of Rmodules

At run-time Rmodules using the cold standby service are completly supported by the configuration management service. At the configuration start-up time, the configuration manager creates one instance for each Rmodule using the cold service. A station supporting one of these instances can fail; this is noticed by the status collector. The station failure is communicated to the configuration manager which can re-allocate the instance to another station in that Rmodule station set. If the re-allocation is possible, the configuration manager executes the configuration control actions needed to recreate the instance and hence re-establish the logical configuration. Otherwise, when it is not possible, the configuration manager marks the state of the corresponding Rmodule as failed; however the failure can be reported to other instances of the system.

For Rmodule instances using the hot standby service the configuration manager gives a similar treatment to that described for the cold standby service. However, the configuration manager cannot assign the two instances corresponding to an Rmodule in hot standby to the same station, since this creates a common failure point. In order to obtain the full characteristics desired for the hot standby service extra run-time mechanisms are necessary. They support the management of the role performed by each instance, the transference of state information to the passive instance, and the technique for transparent recovery of failures. These mechanisms are presented in section 6.2.

At any time stations can be repaired and rejoin the physical configuration. Initially, only the station operating system is activated. A module of the station operating system informs the status collector module that the station is in an operational state. The status collector module informs the configuration manager about the change of status of the station. If there is an Rmodule instance which can run in that station the required configuration operations are then performed by the configuration manager. The approach used to control the allocation of Rmodule instances to the stations is discussed in the next section.

4.6 Configuration Control

The configuration manager functions can be divided into two groups: One is in charge of the execution of the configuration control operations, which is implemented by simple interfaces to the Conic operating system modules (see appendix A.3). The other is in charge of deciding which configuration control operations should be executed. Configuration control activities may be needed when a change in the physical configuration occurs. The configuration decisions depend on the current status of the stations, the current logical configuration, and the configuration rules specified by the application designer. The enforcement of the decision-making capability is discussed below.

4.6.1 Approaches

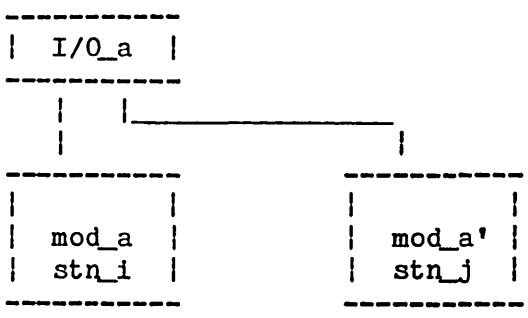
There are two approaches to enforce the configuration rules.

The first approach is simply to pre-determine for each application system all useful mappings of modules to stations allocation. This information can be assembled in an allocation table, with entries for each possible physical configuration status. When a station status change occurs, the current logical configuration is compared with the target configuration which can be supported in the new physical configuration, and the corresponding configuration control operations are determined, and performed. The main disadvantage of this approach is that it can be expensive to keep the configuration control information if the system has a large number of stations and modules. The problem is aggravated if partial failures of stations are considered, since the details and state of the resources of each station should be kept [Loques 83]. However, in practical situations, the fact that modules have to be allocated to specific stations allows the identification of partitions (subsystems) in the logical configuration which can be independently controlled. This allows considerable reduction of stored information.

The second approach is to use some algorithm which can determine the operations to execute at run-time, according to the changes in the physical configuration, and the configuration rules. It is not the main intention of this work to study these algorithms and configuration rules; but we have found that for some typical redundant structures commonly used in control systems [Brown 83, Kaiser 78, Tillman 82, Toy 78, Wood 80] it is possible to take the configuration decisions by the

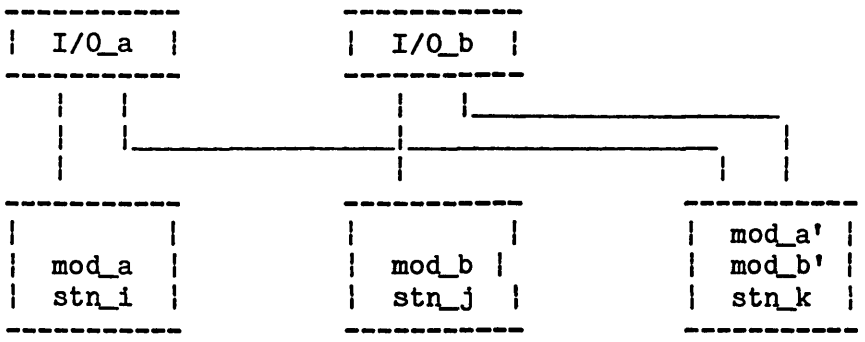
use of very simple application independent rules and straightforward algorithms. Some of these redundant structures are presented in figure 4.4: Structure A is a typical standby redundancy. It does not matter which reliability service is used by an Rmodule; in either case the configuration decisions can be easily taken by using the station set specified for the Rmodule. In structure B station k can be used as standby for Rmodules a and b. If both Rmodules use the hot standby service, there can be one instance of each permanently allocated to station k. In this case, the order in which the stations are specified, in the station sets defines an ordering for the allocation of the instances. The configuration manager can control the configuration such that the active instances of Rmodules a and b are allocated to stations i and j respectively; this can be useful in a situation where load distribution is required for efficiency. In structure C, any of the stations can accept any of the instances; up to two station failures can be tolerated. It should be pointed out that even for structures where the allocation of module instances to stations cannot change, e.g., as that in fig. 4.4.a, the use of an algorithm instead of allocation tables allows the reduction of stored configuration description data.

In general it should be considered that stations can rejoin the physical configuration. As a consequence instances may have to be reallocated in order to establish a more useful logical configuration. This requires more application dependent configuration rules, storage of information describing the resources required by each Rmodule and the resources available at each station. This complication can be avoided if the stations which are shared in the station sets of different Rmodules can separately support either (1) exactly one of these Rmodule instances, or (2) all of them. For the first case a simple rule: a priority associated to each Rmodule, can decide conflicts if there are more than two Rmodule instances to be allocated to a same station. Under these constraints a simple algorithm can be used to control any variation of the structures presented in figure 4.4.



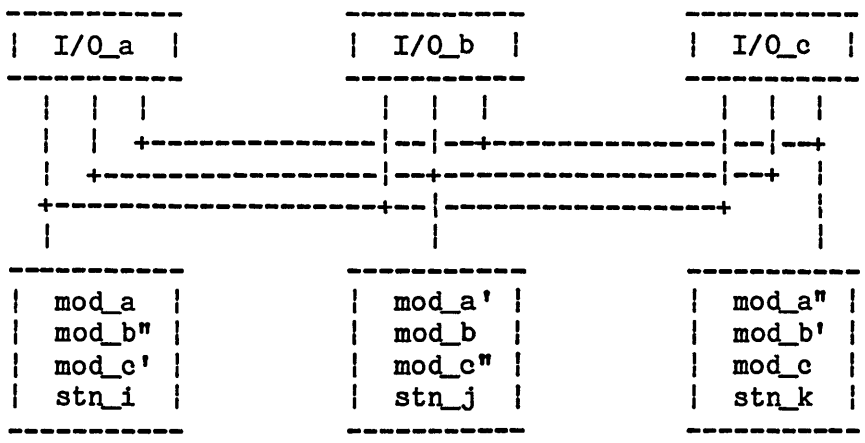
mod_a: stn_i, stn_j;

A. typical standby structure



mod_a: stn_i, stn_k; mod_b: stn_j, stn_k;

B. shared standby structure



mod_a: stn_i, stn_j, stn_k;

mod_b: stn_j, stn_k, stn_i;

mod_c: stn_k, stn_i, stn_j;

C. multiple shared standby structure

Fig. 4.4. Typical Redundant Structures

In our prototype system we have adopted the above described simplifications, and the algorithm has been integrated in the design of the configuration manager. This has allowed changes in the station sets of each Rmodule to be made very easily, and has been useful for the tests we have performed on configuration management. Although the approach is suitable for controlling typical redundant structures it does not offer general flexibility. More general configuration control strategies could be implemented by separating the decision-making functions from the executive functions in the configuration manager design. It is also possible to perform dynamic configuration changes of the actual application system specification in order to reconfigure the system after failures. This is a subject of current research [Magee 83b].

4.6.2 Discussion

The main characteristic of our configuration control approach is that it has a centralized design. Other works available in the literature propose distributed designs for this function.

[Kain 80] proposes various schemes for configuration control. The most robust is based on an algorithm which distributes the configuration control information accross all the modules in the system. Also, each of these modules has authority for performing configuration control; this is intended to provide fault tolerance. In order to implement the approach all the modules have to perform a protocol to obtain the configuration control information and the configuration control rules. In addition they have to support the control algorithms, and have to have interfaces with the operating system of the stations and with the failure detection entity. This is expensive and requires special design for each application module. [Barigazzi 82] proposes a scheme by which every station can independently decide its logical configuration. The configuration decisions are based on arbitrary priorities assigned to the stations. Each station that can support a module broadcasts to all other stations "choice" messages, which contain the priority of the station to support that module. The station having the highest priority supports the module. This algorithm assures the consistency of the configuration decisions taken by all the stations, however it does not take into consideration the allocations already made for the other modules. In addition, because the priorities are arbitrary the resulting mapping of the modules to stations is nondeterministic. Thus a global

strategy for configuration cannot be enforced. [Zielinski 83] studies the configuration control problem with the objective of optimizing the real-time performance of a system. The distributed algorithm proposed is intended to be used at run-time. However, it requires the resolution of complicated matrix equations, which consume computing resources. We chose the centralized approach for the following reasons:

* Typical process control applications can have a large number of stations. Thus in order to keep costs low, these stations should be implemented by simple and cheap microcomputers. Adding to each station the resources needed for configuration control would increase the complexity and cost of each station.

* In our approach, fault tolerance is obtained by specifying the hot standby service for the configuration manager module. This does not require the existence of redundant resources in all stations. Considering that failures are rare, this would be an unjustified overhead.

* In a centralized design, the algorithm for configuration control is not constrained by the need to independently assure consistency of configuration decisions as is the case in a distributed design. Also, considering that the algorithm and corresponding resources do not need to be replicated they could be as sophisticated as required, e.g., algorithms as proposed by Zielinski could be used.

* Independent configuration managers can be used to control different subsystems in a large application.

4.7 Station-Module Failure Relationship

We have adopted the "convenient" assumption that a fault at a station will stop the activities of all module instances running in the station. One of the reasons for adopting this assumption is the simplification incurred for the re-configuration of the logical configuration. This allows a simpler design for the prototype of the fault tolerant system and is also suitable for the support of many applications. If required this assumption can be relaxed. In the following, we present other points related to the issue.

* Total station failures can always occur. Thus the convenient assumption must be considered for fault tolerance. Also, it allows the complete control of the logical configuration by simple configuration rules and algorithms as discussed in the previous section. It is worth pointing out that the convenient assumption does not preclude transparent recovery from failures by mechanisms internal to the station.

* In many cases a station and its module will be a unique entity in the sense that its hardware was specially designed for the module function, e.g., a complex sensor. If partial failures are tolerated they are better treated inside the station.

* An error can be quickly detected by the buit-in error detection mechanisms, but it is difficult and time consuming to identify all the errors caused by a fault. Hence, if the station is allowed to continue operation, other modules will probably fail. It is safer and simpler to stop all application activity, e.g., the bounded response assumption could be difficult to validate and implement. After the error is detected diagnostic tests can be performed in the station. If the fault is diagnosed as transient, the station can be returned to service. This procedure is adopted in other fault tolerant system proposals, e.g., [Katsuki 78, Geitz 81].

* In the case that partial failures of stations are assumed, the decision making function requires explicit knowledge of the status of the resources available at each station and of the resources required by each module. Depending on the refinement of the station model, and the number of stations, a large number of states could result. For the static allocation approach, the corresponding table will have a size proportional to the product of the number of states of each station. Also a run-time allocation algorithm would be complicated, because it would have to consider that stations have variable resources. It is not clear if the benefits of the inclusion of partial failures are worth the related effort.

It should be pointed out that Conic provides mechanisms for error detection and reporting (see appendix A.3). In our present proposal their use should be restricted to debugging purposes. They can be used to extend the basic fault tolerant system if the convenient failure assumption is relaxed.

4.8 Summary of the Chapter

In this chapter an overview of the fault tolerant system was presented. Its main characteristic is the provision of two reliability services: the **cold** and **hot standby**. Modules using these services are called **Rmodules**, they are logically equivalent to standard Conic modules but satisfy specific reliability requirements.

The cold standby service is intended to support application systems having weak failure dependency. Rmodules which use this service are implemented by one module instance. The service provides a volatile storage, thus if the instance fails its state information is lost. The operating system can create another instance of the Rmodule. However this can take a non-negligible time interval. Also activities for recovering the Rmodule operational conditions should be explicitly programmed. A mechanism which can help in the structuring of recovery activities for weak failure dependency systems is discussed in section 7.2.

The hot standby service is intended to support application systems having strong failure dependency. Rmodules which use this service are implemented by two module instances. Only one of the instances, called active, is performing the application processing at any one time. The other one, called passive, is ready to to be activated but does not perform any processing; it provides a stable storage for the Rmodule. At appropriate moments, application state information is transferred from the active instance to the stable storage provided by the passive instance. Failures can occur at any moment during the instances operation. If the active instance fails the passive instance is automatically activated and can continue performing application processing. In the next chapter we present a recovery technique that, for Rmodules using the hot standby service, allows automatic recovery from failures without any apparent interruption of the processing. The language mechanisms for the use of this technique are integrated in Conic.

The support of the reliability services is partly achieved through activities performed off-line in a development system. A slight extension of the Conic configuration language allows the specification of the reliability service required by the application modules and the set of stations where the corresponding Rmodule instance(s) can run. The application system configuration specification is processed by a translator program. This translator program peforms validation of the configuration specification and produces a configuration description, which is used at run-time by the support mechanisms, to instantiate Rmodule instances. Rmodules using the hot standby service require specific support in the development system, this will clarified in the next chapters.

On-line support is partly provided by two system modules. The configuration manager is needed to control the logical configuration of the application system. It has access to the configuration description and the configuration rules which specify the mapping of instances to stations. Configuration control activity is performed at the system start-up or when a change of the physical configuration status occurs. At the system start-up time, the instances of the logical configuration are activated. During the system operation, in the case that a station fails, the configuration manager can instantiate new Rmodule instances in order to replace those which were assigned to the failed station. Configuration activities can also be performed when a station joins the physical configuration. The status collector is in charge of collecting statuses from all stations in the physical configuration. Any change in the status of a station is reported to the configuration manager. These two modules provide configuration management capability which is sufficient for the support of the cold standby service. For the hot standby service additional mechanisms are necessary for the support of the automatic recovery capability. The relevant details of the design and implementation of the fault tolerance support mechanisms will be discussed in chapter VI.

In this chapter, the configuration manager operation and design were also discussed. The configuration manager uses a simple algorithm to automatically decide the configuration operations that should be executed, thus the storage of precomputed tables containing allocation information is not required. This configuration control algorithm provides the needs of typical redundant structures used in control systems. The configuration manager has a centralized design, other proposals advocate a distributed design for this function, mainly for providing fault tolerance. A centralized design keeps simple the rest of the stations of the system, allows the use of more sophisticated configuration control algorithms, and fault tolerance can be achieved by specifying the hot standby service for the configuration manager Rmodule.

Finally, for completeness, the station-module failure relationship was discussed. We have assumed that any fault at a station will cause all the activities in the station to be stopped. This is mainly justified because this kind of failure can always occur. In addition this assumption keeps the design of the support mechanisms simple. It is not clear if the benefits of the consideration of partial failures of stations are worth the associated effort. However if required they can be treated by extending the basic fault tolerant system.

CHAPTER V

LANGUAGE SUPPORT for STRONG FAILURE DEPENDENCY SYSTEMS

In this chapter we present our approach for providing language support for programming strong failure dependency systems. The related language mechanisms are only available to Rmodules using the hot standby service. This enables relevant application state to be kept in the stable storage provided by the passive instance associated to Rmodules using that service. Thus the state information can survive the active instance failure. In addition the use of an application independent recovery technique allows the failure to be transparently recovered without any interruption of the activities being performed by the application Rmodules. Language level support is provided by an extension of the Conic port concept -- **reliable ports**, and by a primitive -- **save**, which is used to invoke transference of state to stable storage. Transactions performed through reliable ports are guaranteed to complete even if a failure occurs during the transaction execution. This is assured by the recovery technique and associated run-time support mechanisms.

5.1 Computation Model

This model is presented in order to introduce the recovery technique. For brevity of exposition it is considered that there is only one task per module and that the port connections are one-to-one. This will be generalized later in the thesis. Also, for simplicity consider that any message produced is available at the consumption time in some message queue implemented in the communication system.

The basic unit of the model is an action. An action execution can be seen as the unit within which communication through state changes can occur. A task is defined by a **local state**, a **set of entryports**, a **set of exitports**, and a **set of action execution specifications**. The computation performed by a task takes place in a series of action executions. Each action execution specification specifies the form of a task computation by stating:

-- An entryport for the action. The input message is received by the task from this port.

-- The output local state and a set of output messages as a deterministic function of the input local state and the message received. To each input local state is associated one or more action execution specifications. The action to be executed depends on the availability of a message in an entryport and of the entryport itself. The specific message choice is made by an nondeterministic mechanism.

The execution of actions can be seen as executed by an interpreter. This interpreter maintains a local state for each task and has access to the queue of messages for each entryport. One cycle of this interpreter selects a task, and from its set of entryports selects one that contains a message to be executed, and carries out the corresponding action execution. After the action is executed the interpreter changes the local state of the task to the resulting output state of the action execution. Note that one or more messages can be produced as a result of the action execution.

5.1.1 Effects of Failures in the Model

According to the error confinment assumption when a task fails it ceases all its activities. In this case the system may be left in an inconsistent state. This happens because some messages might have been generated whilst others have not, depending on the point of the execution the failure has ocurred. This can be prevented by using the stable storage available to Rmodules in hot standby. Before executing an action, the task local state and the selected input message are transferred to the stable storage. The action is also marked as executing. This transference is atomic in the sense that either all the information is transferred or no transference is performed if a failure occurs before the transference is completed. When the action completes execution, the interpreter transfers the resulting output state to stable storage and marks the action as executed. In the case of failure it is assumed that after some time the task and its interpreter will be recovered (In the implementation, this occurs when the passive instance takes the active role). On recovery, the interpreter verifies if there is any action marked as executing, and in this case re-executes the action, otherwise it proceeds as in the normal case.

5.1.2 Replicated Messages

Since the input message and the local input state are the same as before the failure, and that the function corresponding to an action is deterministic, the results produced in any re-execution will be identical. Thus, in terms of the external interface of the task, the only effect of re-execution is possibly to generate replicated messages. The atomic update of the task input state and message, and the re-execution of the action assures that the task reaches a consistent local state, but the state consistency for all the system can be violated if any task consumes the same input message more than once. This can happen because replicated messages are generated on recovery.

A system of reliable communicating tasks can be obtained by associating a sequence number to each port. Specifically, to each exitport we associate a transmit sequence number(TSN) and to each exitport we associate a receive sequence number(RSN). TSNs and RSNs are stored as part of the task local state. Before transmitting a message through a port, the interpreter increments its TSN. The value of the TSN is used as a sequence number(SN) for the message. Considering that addition is a deterministic function, any re-execution of an action will produce messages with repeated TSNs. When trying to execute an action, the interpreter compares the SNs of the available messages for a port with the RSN associated to this port. The message is accepted only if SN = RSN + 1. After accepting the message, the interpreter makes its associated RSN equal to the SN of the accepted message. Before executing the action, a stable storage update is performed; this assures the re-execution of the action and the repeatability of any produced result including the SNs of messages. The filtering of replicated messages assures that no system inconsistency occurs as a result of the consumption of replicated messages.

5.2 Conic Application

The model presented can be used for the implementation of reliable systems; but for this it would require the use of a style of programming where each action is explicitly defined. Also, the interpreter would have to duplicate most of functions which are already provided by the Conic Kernel. Fortunately, a very elegant implementation is possible for the model. It has the following characteristics:

* It allows the use of the standard message and task management functions provided by the Conic system kernel. Thus no overheads are introduced.

* It does not introduce any unreasonable restriction in the style of programming the application modules. In principle, it is possible to design modules without any special consideration and afterward perform an automatic transformation to make them fault tolerant.

* It provides more efficiency in terms of transaction execution time than that which would be provided if the model was implemented directly.

Conic provides two communication primitives which support different message transactions (see appendix A.1 for summary). The first transaction, -- **request-reply** is specially designed for use when a confirmation of the acceptance of the output message is required. After sending a message, the task execution is suspended. In the normal, and most useful case, the task execution is activated after receiving the associated reply message. The second transaction -- **notify** does not provide any assurance on the fate of the output message. In fact it is designed in order to provide maximum flexibility. Specific guarantees can be enforced by user or system provided service modules, e.g., buffered virtual circuit modules [Sloman 83].

Our approach for supporting strong failure dependency systems hinges on providing a special reliable request-reply transaction which is used for communication between tasks of modules using the hot standby service. This transaction is reliable in the sense that it always completes and provides a **uniform exactly-once semantics** in a single instance failure case (see section 5.5 for related discussion). Thus, a system of tasks communicating purely by reliable request-reply transactions is automatically made fault tolerant. If required, some form of reliable notify transaction can be readily implemented by using a service module in hot standby having its interface defined by reliable request-reply ports.

Figure 5.1 shows the general pattern of the transaction for a source-target system. There will be only one active module instance per pair at any time. An active or passive instance can fail at any time, in particular, during a transaction. The failure of the passive instance does not interfere with the transaction. In the case of active instance failure, the passive instance takes over and completes the transaction. This can involve the re-transmission of messages. For request-reply transactions there can be only a single pending message associated to each pair of ports performing a transaction. This fact allows a very efficient design for a request-reply transaction. According to the model, the effect of failures is limited to the generation of replicated messages. This applies both to request messages and to the reply messages. In practice, failures can cause other subtle critical situations which can affect the reliability of the transaction. This and other details of the transaction implementation are discussed in section 6.2.3. For now, consider that reliable communication can be obtained through a protocol that uses the reply message (with an RSN) as an explicit acknowledgement to the request message, and that when needed, an underlying mechanism can pick up a reply message already produced in the task data space and send it back to the exitport task side. This underlying mechanism also deals with the transport of request and reply messages between the module instances.

In addition to a primitive to specify reliable request-reply transactions, another primitive is required to specify transference of state information to stable storage. They are both presented in the next section.

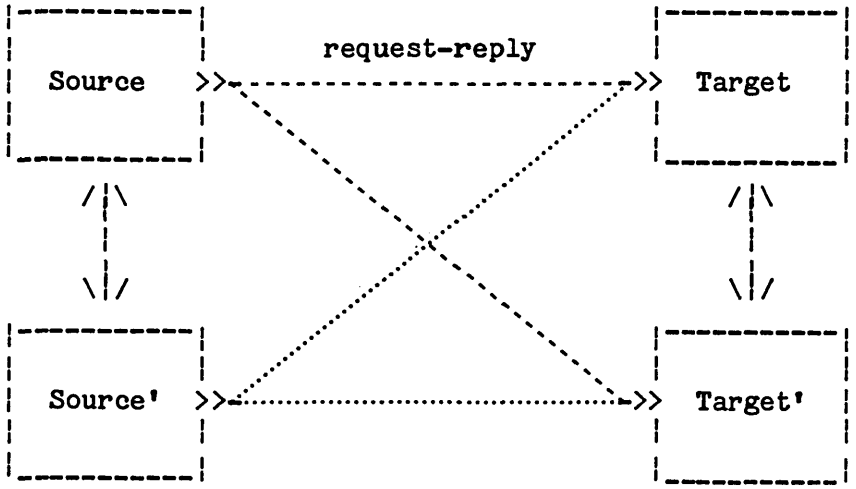


Figure 5.1 General pattern of the request-reply transaction.

5.3 Language Primitives

Language support for the programming of strong failure dependency systems is provided by adding two new primitives to Conic. The first introduces the concept of a **reliable port**: A reliable port primitive is used to specify communication performed through reliable request-reply transactions. The second supports the concept of stable storage: A save primitive is used to specify transference of state information to stable storage at relevant moments of a task operation. These primitives are only available to Rmodules using the hot standby service.

5.3.1 Reliable Port

A reliable port is logically identical to the standard request-reply port used in Conic. Thus, it provides a local name and a message type holder for the port, and can have its connection described by a standard link declaration. In addition to these standard characteristics it also provides a local recipient for the end-to-end control sequence numbers that are required to enforce the transaction semantics; this is further discussed in section 6.2.3.

Reliable ports can be used during the task operation to automatically control the execution of saves. This is obtained without any explicit declaration of actions at the application programming level. Thus making the actions completely transparent. Consequently no restrictions of programming style are introduced. The technique also allows a minimum number of save operations to be performed. The save control technique is presented in section 5.4.

5.3.2 Save

At some points in a task operation it is necessary to transfer state information to the stable storage provided by the passive instance. To execute a state change, the task program invokes the save operation. The actual state transference is performed by an underlying mechanism, which takes a copy of the current state and transfers it to the passive instance through a message. The fact that messages are either received or not received at all ensures that the state update is done atomically: either the entire state information is transfered to the passive instance or the effect is as if the save operation had not been started. Thus a failure in the middle of a save does not leave the state of the task in the passive instance inconsistent. If there is no passive instance available, the control is just returned to the task

performing the save. For now, consider that the state information transferred is enough to assure that the re-execution of any action that can be interrupted by a failure will produce repeated results. The achievement of this characteristic for a save operation, and schemes to minimize the amount of information transfered in a save are discussed in section 6.2.1.

5.4 When to perform a Save

In the model, it was shown how the careful use of stable storage can be used to provide reliability for a system of tasks. Here, we define when it is really necessary to transfer state information to stable storage, i.e., to perform a save operation. This is achieved using three rules which are sufficient to enforce the exactly-once semantics specified for transactions peformed through reliable ports.

5.4.1 Rule I

It is necessary to perform a save when a request message is accepted in a reliable entryport. Besides transfering the state information, this save also works as an implicit lock of the task interface, i.e., no other message can be accepted, during the corresponding action execution, in case of failure. This makes the execution of the action repeatable, including the generated reply message, if no nested action execution is performed. It is also necessary to guarantee repeatability when a nested action execution is needed. The message requesting the nested action is repeatable. When this message is accepted for processing in a recipient task, a save operation is also performed. Consequently this nested action execution is also repeatable, which also includes the generation of the reply message. The transaction support mechanisms ensure that, on recovery, corresponding pairs of request-reply messages will be matched, and that any interrupted transaction will be completed. Their design takes advantage of the reply message availability in the stable storage provided by the entryport task (see section 6.2.3 for implementation details). This fact permits a simple rule to be defined:

> **Rule I: A save must be performed after a message is accepted in a reliable entryport and before any result of the processing of this message is sent out of the task.**

It is not necessary to perform a save when a reply message is consumed in the reliable exitport side. Although the reply corresponds to a new input; the repeatability of the action (which would be defined by the acceptance of this reply, according a direct interpretation of the model) execution is assured by the stable storage provided by the entryport side, and by the additional rules.

Figure 5.2 shows the optimized approach to enforce the model. Consider that all transactions use reliable request-reply ports. It can be seen, that in any single failure case, the action execution started by req_1 (or req_2, or req_3) will complete consistently without any further saves. For example, consider that task_2 fails after performing the save. On recovery the task will execute again and generate req_3, which could have been already generated in the previous execution. If req_3 is a duplicate, and has already been accepted, and has its associated reply produced, the underlying mechanism picks up this reply and sends it back to task_2. Otherwise req_3 has to compete at task_2's interface for consumption. Also, before the failure, rep_2 could have already been generated and consumed by task_1. In any case, duplicates are discarded by the underlying support mechanism.

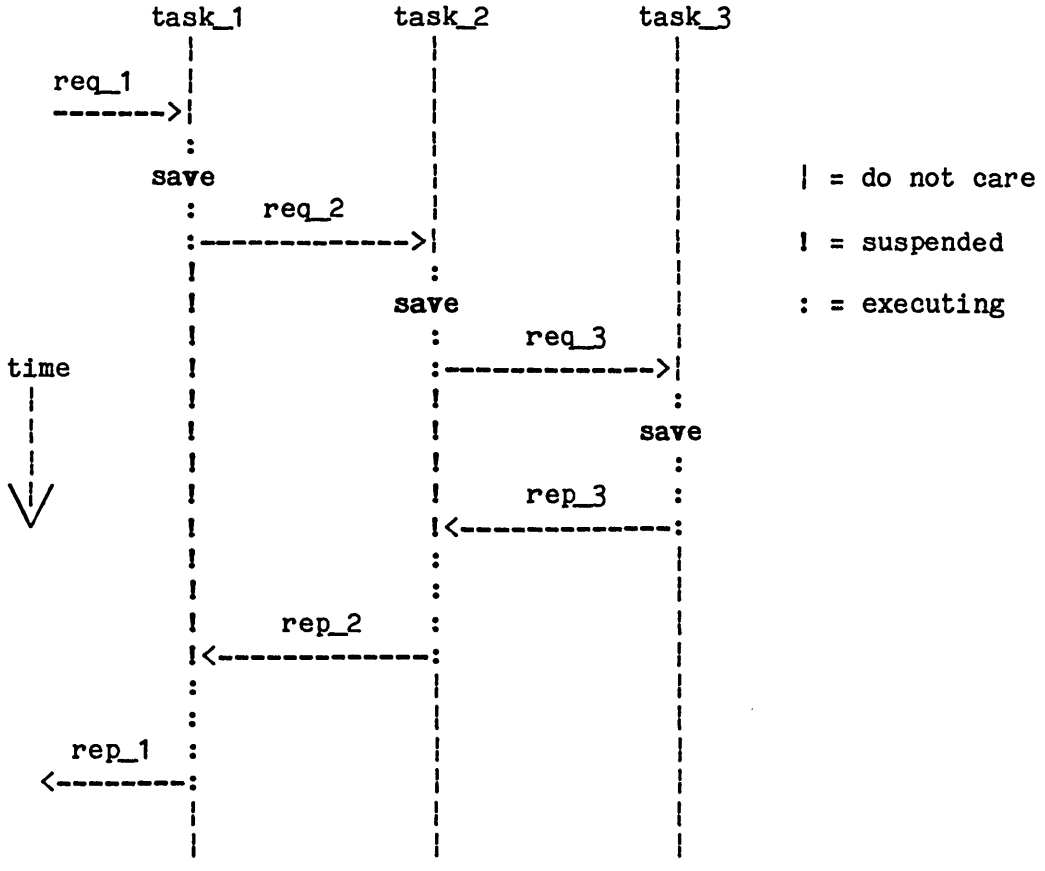


Fig. 5.2 Optimized Approach

The model could be enforced directly, as described in section 5.1. Figure 5.3 extends figure 5.2 in order to show the extra saves used by the direct approach to enforce the model. In this case, the capitals written saves would be required, whilst in the case of the optimized approach, only the lower written saves would be required. Thus at least one save operation can be economized for each action. More saves are economized depending on the exact definition of the actions, e.g., a new action could be defined by the receiving of the reply message. Saves take time to be performed, hence the optimized approach provides more efficiency in an "optimistic" environment, i.e., where few failures occur.

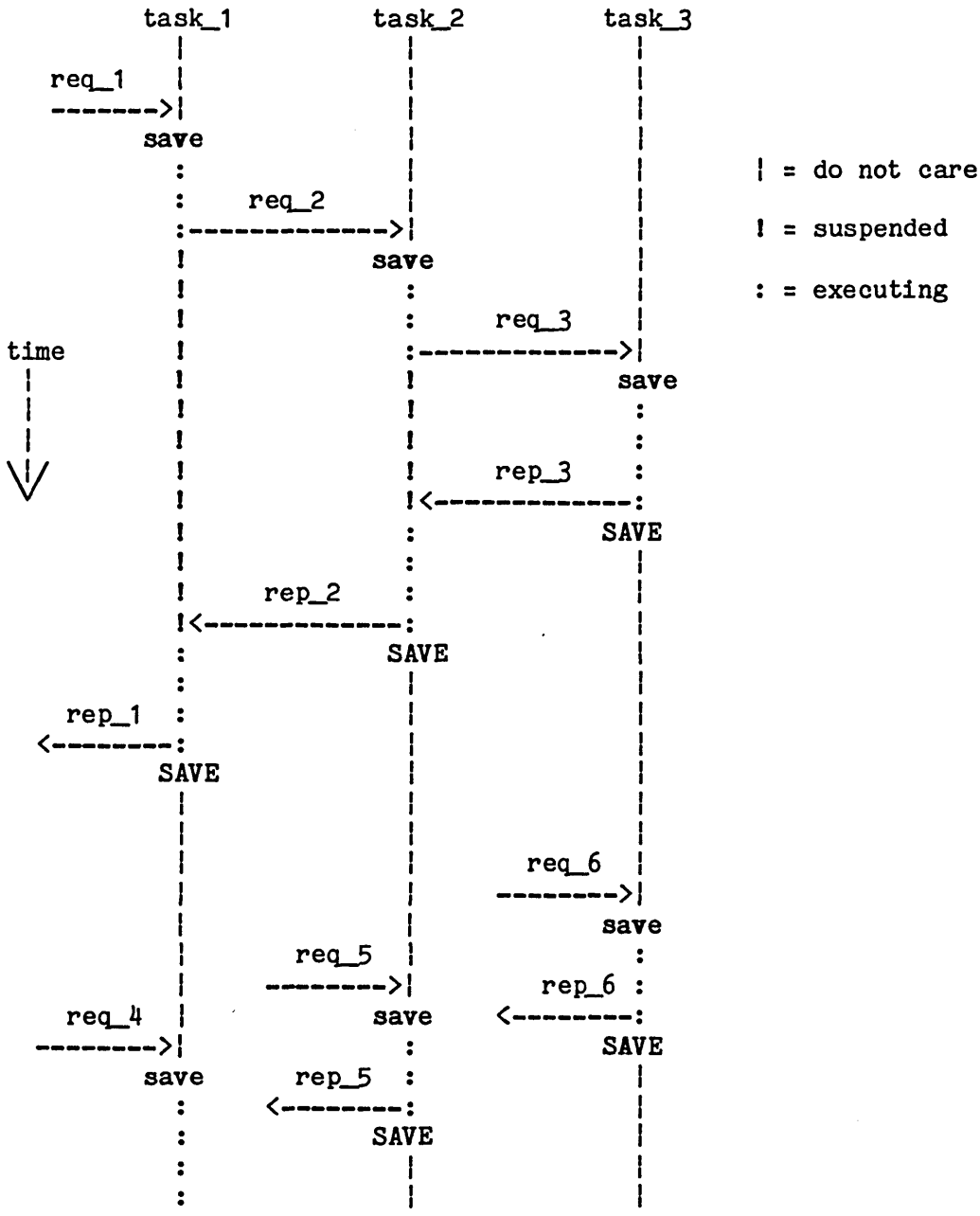


Fig. 5.3 Direct Approach

It is worth noticing that in some failure cases all the execution of the action has to be repeated, e.g., if task_2 fails after issuing rep_2, but before a save is performed. This is true either for the direct or the optimized approach. A partial solution is to use intermediate saves, e.g., after the acceptance of rep_3.

5.4.2 Rule II

The rule already presented is not enough to ensure consistency in all possible cases. In figure 5.4 consider that req_2 and req_3 are issued through the same reliable exitport within the execution of an action. Also consider that task_1 fails after req_3 is accepted and saved by task_2, and that rep_3 is already produced, e.g., at point **y**. In this situation, rep_2 cannot be recovered by task_1. This happens because the reply message variable, in task_2, was overwritten and contains the value of rep_3. In order to cater for this case, and to leave open the use of this option by the programer, a new save rule is needed, otherwise a restriction in the style of programming would have to be introduced.

Rule II: A save must be performed before a message is sent out through a reliable exitport if no save has been performed after a previous message has been sent out through this exitport.

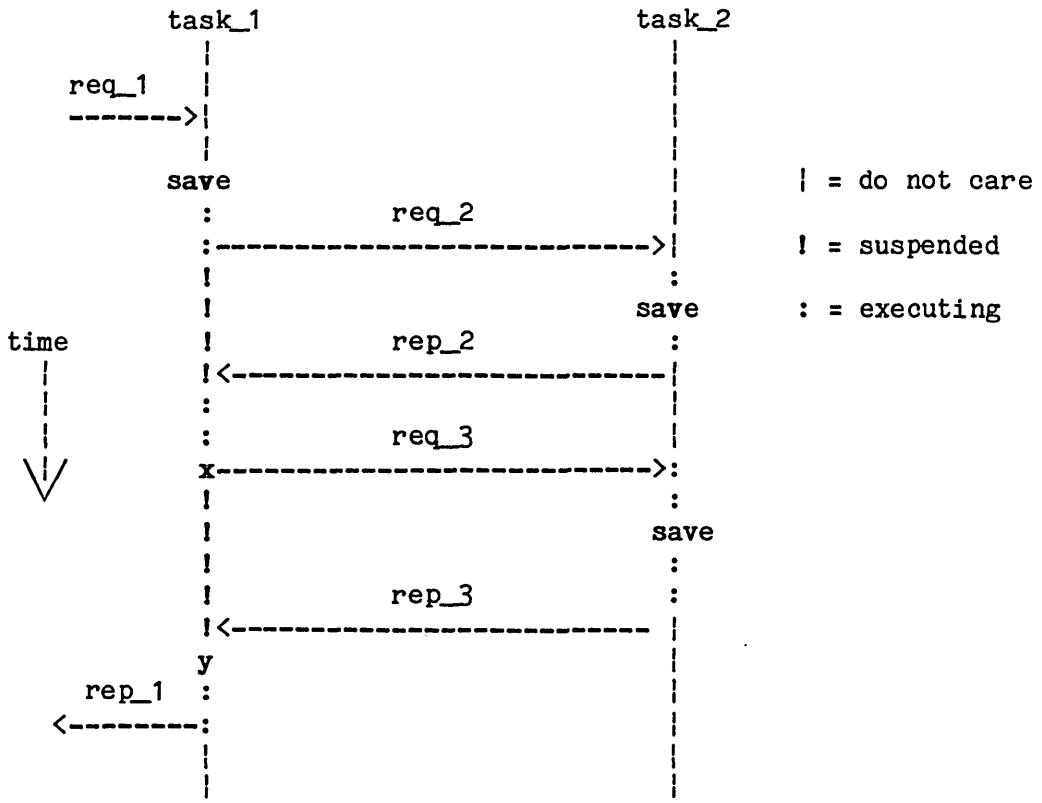


Fig. 5.4 Loss of the Reply Message

In figure 5.4, a save has to be performed before point x. Rule II can be generalized if the entryport task keeps in stable storage copies of a number of reply messages associated to previous transactions.

5.4.3 Rule III

Up to this point, we have considered that inter-task communication is peformed only through reliable ports. This restriction can be removed in order to allow inter-task communication to be performed also through standard Conic ports: activity which for brevity we call unreliable communication. This facility increases the flexibility of our approach.

When unreliable communication is used, the production of repeated results may not be guaranteed if a re-execution is performed because of a failure. In consequence, inconsistency can arise in a reliable transaction, e.g., a different request message could be generated, but the reply message associated to the request message generated in a previous execution can already be available and be consumed. Figure 5.5 helps to illustrate the situation, consider that the pair req_2/rep_2 is associated to standard ports and that req_3/rep_3 is associated to reliable ports. In this context, assume that task_1 fails after issuing req_3, which is consumed by task_3. On recovery task_1 repeats the transaction req_2/rep_2, but considering that task_2 can be in a different state the contents of rep_2 can be different from that of the previous transaction. In consequence the contents of req_3 can also be different. However, the previous instance of req_3 had already been consumed and the available rep_3 is a result of this execution. The underlying mechanism picks up rep_3, which can cause inconsistency, since task_1 will act according to the contents of rep_3, which may not reflect the state of the system. The problem is solved by performing a save after rep_2 is accepted and before req_3 is sent out by task_1 solves the problem, i.e., a save must be performed up to point x.

It is difficult to characterize all the situations that can arise when unreliable communication is performed. For example, the action requested by req_2 could be idempotent, in which case the use of a save might not be required. Thus, a rule which forces a save to be performed after any use of unreliable communication would be an overkill. In order to provide maximum flexibility, we leave to the programmer the responsability of ensuring the repeatability of messages associated to reliable request-reply transactions. This is done through the explicit use of the save primitive, although this requires the save primitive to be available at the language level.

There is another case where a save primitive can be useful: when an action takes a long time to be executed. For instance, an action requiring many items to be operated upon in a repetitive fashion. In this case, the programmer can use a save to transfer partial results after some amount of the work has been done. The effect of this save would be to speed up recovery if a failure coincides with the action execution. This would be effective if the time taken to perform the "sub-action" is big in relation to the time spent in a save operation. Arising from the above discussion, a last rule is defined:

Rule III: A save must be performed when it is explicitly invoked by the task program.

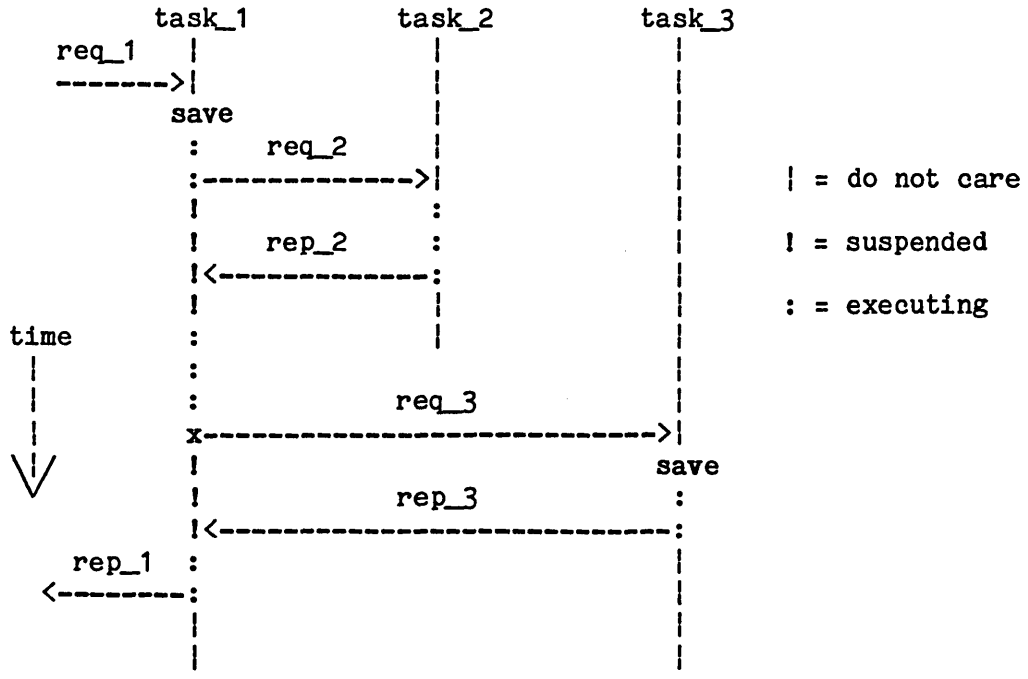


Fig. 5.5 Unreliable Communication and Consistency

5.4.4 Comments

(1) The above defined rules are enough to ensure the repeatability of any output sent out through a reliable port. This is an essential condition to assure consistent and transparent recovery. Rules I and II can be automatically enforced by a transparent mechanism activated by pieces of code embedded in the task program. These pieces of code are associated to reliable ports and can be inserted in the task code by a simple program transformation that can be performed by a preprocessor or compiler. Rule III is also easily enforced because it is associated to a save explicitly defined in the task program. However, in this case manual intervention for editing the program is required. It is also possible to leave the responsability of enforcing the three rules to the programmer by using explicitly defined saves. An example of the application of the approach is presented in section 7.5.

(2) Reliable ports can be declared in a special definition file, associated to each Rmodule in hot standby. At the time, the system configuration specification is processed in the development system, this file is used to instruct the preprocessor or compiler; thus the support for reliable ports is made available. Reliable exitports can only be linked to reliable entryports; the standard Conic link declarations can be used to check this constraint.

(3) A fault tolerant system is achieved if all the rules are enforced by all its tasks; this allows the support of strong failure dependency systems. It is interesting to note that if only pure reliable communication is used, i.e., the interfaces of the tasks of all modules are entirely defined by reliable ports, fault tolerance capability can be automatically obtained.

(4) We assume that a double failure is a very rare event and that duplication is enough to meet the reliability requirements of most applications. However, it should be pointed out that the rules are independent of the number of available passive instances. Thus the rules still apply even if more than one passive instance is supported.

5.5 Exception Handling

In this section we discuss issues related with exceptions that can be associated with Conic request-reply exitports. Two exceptions are defined: The first is activated when a transaction is attempted through an unconnected exitport. The second is activated when an application defined timeout interval expires. The first exception should not occur in a correctly designed system; its use is provided for testing and debugging of application systems. Thus we restrict our discussion to the timeout exception.

Timeouts are generally incorporated in distributed programs for one of two reasons: (1) to assure minimum performance -- if an action takes more than some time to be executed, it times out and the program can do another job, (2) to detect failures. When they are used for efficiency reasons, it may be necessary to provide a means for the exiport instance to deal with the transaction that was timed out. Many schemes can be conceived in order to support this facility, e.g.:

(1) A special service interface can be provided: the task may repeat the same transaction request using the same sequence number. This is intended to ensure exactly-once-semantics.

(2) The task may repeat the same or initiate a new logical transaction with a new sequence number.

The problem with schemes like these is that no standard handling procedure is defined for a timeout exception; thus all responsibility is left to the application programmer. The ADA [USA-DOD 80] designers have attempted to solve this problem by associating a timeout to the request message acceptance at the target instance side. Thus before a timeout exception is activated the system must certify that the request message will not be accepted by the target. In a distributed system, this has to be enforced by an protocol executed by an underlying support mechanism.

Efficiency timeouts can be useful in general resource sharing networks that have an unpredictable number of users competing for scarce resources. In control systems, resources are used by a controlled number of disciplined users; this can eliminate the need of using efficiency timeouts. For this purpose, resources would be designed in order to meet some minimum response requirement. This could mean a quick execution of the service or an explicit indication by the reply message about the

reason of service deferment. This option is a good approximation for an "ideal" software component, as proposed by [Anderson 81, Wulf 75], and we believe it represents a better style of programming. In addition it does not require any extra support mechanism and keeps the underlying system simple; however, it requires a careful choice of the timeout value (The subject is further discussed in section 7.3). According to the above discussion, the use of efficiency timeouts is not recomended; although, if required, they can be supported. In this case, a save should be performed immediately after a timeout exception has occurred: The treatment of the timeout splits the program path; the save guarantees that the same path is taken when failures occur; this ensures consistency of the task outputs.

Timeouts can be used in entryports in order to simplify the style of programming. Considering that in this application they do not interfere with transactions, we do not make any restriction on their use.

5.6 Related Work

Our approach can be related to other works proposed in the literature in some different ways.

5.6.1 Reliable Transactions

The Conic request-reply transaction is very similar to a procedure invokation in the sense that the request message specifies the input values and the reply message the result values of the procedure execution. A current topic of research is the extension of the procedure concept to distributed computer systems. In this area, the proposals of Nelson [Nelson 81] and Liskov [Liskov 81] are particularly significant. The main goal of them has been to specify a consistent and uniform semantics for procedure calls. The subject is interesting because of the distributed environment and the possibility of processor and communication failures.

Nelson proposes to extend the procedure semantics provided in traditional uniprocessor systems for the whole distributed system. Hence in normal operation an exactly-once semantics is assured: The system guarantees the computation associated to a call to be executed only once: If any processor supporting a distributed program fails, all partial results of computations are abandoned and the system backtracks to a previously checkpointed state. This is called last-one semantics and in fact is equivalent to a system reset.

Liskov proposes to extend the CLU language with a procedure call primitive to support distributed programs [Liskov 79]. In normal operation the procedure call primitive offers what is called at-most-once semantics: If the caller receives a reply, the system guarantees that the call was acted on exactly once, which is is in fact identical to Nelson's proposal. It is also proposed the use of stable storage and an automatic scheme to transfer relevant state to this storage; but the intended use of stable storage is different from that in our proposal, this being due to the class of applications to be supported: distributed data bases. The system keeps in stable storage copies of data objects which are changed in a procedure execution. If the procedure does not complete, either because a failure, or because the caller lost interest, these objects are restored to the state they had before the procedure call. The system also supports synchronization properties for the procedure call in order to allow concurrent changes to the objects without leading to inconsistency. The synchronization properties are obtained by locking the use of objects according to some fixed rules; although this can lead to deadlock. In order to breack deadlocks and allow progress some executing calls can be aborted and in this case the objects' states are also restored. Another characteristic of this proposal is that distributed computations may be held up while any node supporting them is failed if the failure has occurred at a critical point. It is proposed as a solution to the problem to implement replication on top of the basic mechanism; although it could also be solved by implementing it on top of the mechanisms we propose. A synthesis of the approach is presented in [Moss 81].

Both approaches do not intend to provide consistency at the interface with the environment. Nelson's last-one semantics in the failure case is incompatible with this goal. In the other proposal, this would be obtained if no outputs are released until the whole transaction completes; a restriction that is not adequate for control applications.

According to Nelson's terminology, our reliable transaction proposal specifies exactly-once semantics in normal operation and in single failure cases. Thus failures will not result in interruption or any loss of the ability to control the application process, as would happen in a system based on his proposal. Extra resources and special, but simple, support mechanisms are required in order to match this specification. This is worthwhile in the DCCS context in order to meet the reliability requirements of demanding applications. In relation to

Liskov's proposal, we have assumed that control applications have different requirements, -- it is more important to provide guaranteed response than to pander to impatient users. In addition most control programs are composed of a static set of modules which interact in a predefined fashion through fixed communication channels; this allows the elimination of possible deadlocks at the design stage. These two facts allow a considerable simplification of the mechanisms needed to support our approach. Although our approach might be generally applied we concentrate efforts to Conic based distributed systems; this has allowed a simple solution for providing language support for programming reliable application systems.

5.6.2 Actions

Actions and transactions have interrelated recovery properties. Recently Liskov has proposed to implement her procedure call as a sub-action which is defined at the language level [Liskov 83]. The concept of action were also explored in [Lomet 77, Randell 75], although in different contexts. Scheneider and Schilichting propose an approach based on actions to program distributed fault tolerant control systems. Stable storage is implemented by a number of processors and used to make the actions restartable and produce repeated results, which in principle is similar to our approach. However, their actions can co-operate only through shared variables which are kept in stable storage, although an underlying mechanism, which uses a special message protocol is proposed to keep the consistency of replicated copies of the shared variables. In their first proposal [Scheneider 81], actions were equivalent to a single process and a special restartable semaphore [Dijkstra 68] was provided to synchronize access to shared variables. A restartable semaphore allows a process p to re-enter a critical section if p has already entered that critical section but never done a V operation to exit. This situation can occur in the case that p fails and recovers. In spite of an ingenuous scheme to implement stable storage updates, and of the restartable semaphores, their first proposal cannot guarantee what they have intended: repeatable outputs. Although the flaw was not mentioned, it was corrected in their second proposal [Schlichting 82], by allowing nesting of actions within a process, which has required the explicit definition of actions. Some comments can be made on their approach: The use of shared variables does not allow modularity, and makes the programming of systems and the proof of their correctnes difficult [Zave 79]. It is also not justified in a distributed system

where message passing is a more natural concept and must be used at some level in the implementation, as they did. In addition, in their proposal, the processes are assumed to be cyclically activated, which requires an underlying mechanism, which they do not specify, that must also be fault tolerant. Moreover, nested actions require a careful implementation in order to provide recovery capability. This implementation is also not specified.

Our approach does not require explicitly defined actions, which makes the programming easier, and conforms with Conic. Also in the same context as theirs, this allows the use of automatic techniques to transform modules not originally designed with fault tolerance in mind. These modules can then be assembled together in order to achieve a fault tolerant system. In addition, our approach is integrated in a much more flexible software structure, which supports modularity and provides more natural and powerful interfaces. Moreover, the implementation of the corresponding support mechanisms is very simple and completly specified. It is interesting to note that the same techniques that they propose to verify fault tolerant systems [Schlichting 80] are also applicable to our approach.

5.6.3 Fault Tolerant Systems

The Tandem system [Levy 78,Bartlet 81] uses duplicated processes to obtain fault tolerance. However, it does not provide a standard technique to provide recovery from failures, which should be enforced by the programmer. Also the input interface of a process is defined by a single message queue, which allows neither a non-deterministic choice of input, nor the use of guards, as allowed in Conic. This lack of flexibility can make the programming of some applications difficult. In addition, communications are performed by referencing in the source process the name of the target process , which does not allow modularity. Even so, this system has been used in the implementation of elaborate fault tolerant applications, which confirms the usefulness of duplicated processes [Borr 81].

Another system using duplicated process is presented in [Kaiser 78, Gaude 80]. Fault tolerance is obtained through a combination of a special operating system, special hardware, and special style of programming. The main quality of the approach is that it requires very little information to be transfered from the active to the passive

computer, which is intended to provide efficiency. This does not seem to be entirely justified because of the low speed of the application to which it is directed: a welding shop of a car manufacturer, and the peformance data provided by Gaude. In addition there are applications were fault tolerance is more important than efficiency. Thus the flexibility and simplicity offered by our design can be an advantage.

5.7 Summary of the Chapter

In this chapter our approach for providing language level support for the programming of strong failure dependency systems was presented. The presentation is based on a computation model which requires the definition of explicit actions for task executions and makes use of the stable storage abstraction. The actions that can be executed by a task are made repeatable, i.e., they produce the same results if re-executed, by transfering information identifing the action and the message that has activated it to stable storage. When a failure occurs, during its execution, the action is re-executed. The external effect of the re-execution is limited to the generation of replicated messages, which can be filtered out by the use of sequence numbers. The computation model shows that no state inconsistency occurs when an active instance fails. However the direct implementation of the model would lead to unnecessary state transferences to stable storage and require the explicit definition of actions, which changes the normal Conic programming style. Fortunately, these drawbacks can be eliminated through a special integration of the model in Conic.

For this purpose two special primitives are provided: The first -- save is used to specify transference of state information to stable storage. The second introduces the concept of reliable ports, which are used to specify communication performed through reliable request-reply transactions. These transactions are assured to complete in spite of failures of the instances involved. For this it is enough to enforce three rules for using the save primitive, given in section 5.4. By controlling the transfer of state to stable storage they assure the repeatability of any message sent out through a reliable port, which together with a transparent request-reply transaction support mechanism assures consistent state recovery. In this way a system of tasks of hot standby modules can be made fault tolerant. It should be pointed out that this does not require any explicit definition of action(s), thus no restriction on the style of programming is incurred. Rmodules which have

their interface defined exclusively through reliable request-reply ports can be made fault tolerant through an automatic program transformation, although manual intervention is required to deal with application dependent situations. In addition the rules allow a reduction in the use of stable storage with consequent efficiency improvements. Also, the approach allows the use of the standard message and task management functions provided by the Conic kernel, thus no overheads are introduced. The critical points of our approach are the design of the mechanisms which support the reliable request-reply transaction and the save operation. They will be discussed in the next chapter.

In this chapter the issue of exception handling in reliable request-reply transactions was also discussed. A reliable request-reply transaction cannot complete because of an efficiency timeout or a double failure. Efficiency timeouts are considered a bad style of programming in control applications. We have assumed that the probability of a double failure is very low and should not be considered for most applications. Thus in the normal use timeout exceptions should not occur. However, if required, they can be readily supported. In this case explicitly programmed saves have to be used to assure repeatability of the task outputs.

In section 5.6 our approach was related to relevant works in the area. The discussion is already concise and is not summarized; however, in general, it underlines the simplicity and flexibility offered by our approach in its area of application: distributed control.

CHAPTER VI

THE SUPPORT MECHANISMS

In this chapter, the design and implementation details of the support mechanisms which make the system fault tolerant are presented. The chapter is organized in two main sections. In the first, the Configuration Manager and Status Collector modules are presented; they provide the capability of configuration management which is enough to support weak failure dependency systems. In the second, the mechanisms associated to the hot standby service are presented; they extend the basic system in order to support strong failure dependency systems.

6.1 Configuration Management

The configuration management service has already been presented in chapter IV; here the discussion is concerned only with the design and implementation issues of the system modules, viz., the configuration manager(CM) and status collector(SC), which together provide this service (Figure 6.1).

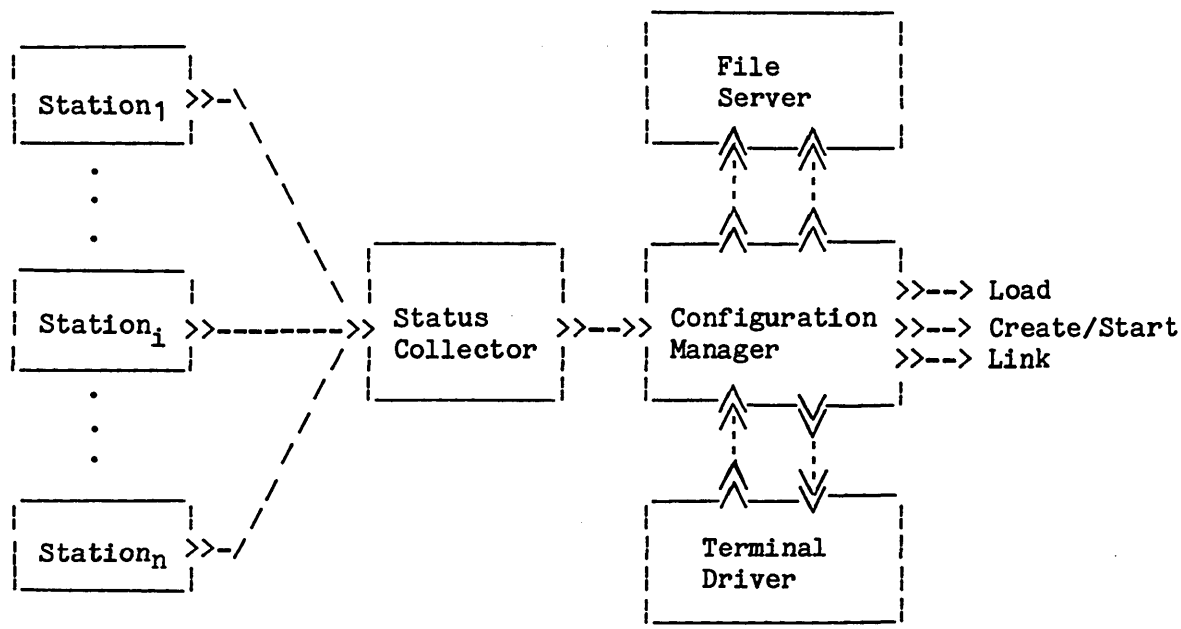


Fig. 6.1 Configuration Management Service

6.1.1 Configuration Manager

This module has access to a fileserver where the configuration description file is available. The configuration description file has been generated in the development system as discussed in section 4.5. The CM does allocation of module instances to physical stations at run-time according to the physical configuration status and the configuration control algorithm discussed in section 4.6.

6.1.1.1 Configuration Manager Reliability

The CM must be fault tolerant itself in order to support the reliability services for the control subsystem modules. In this section, we discuss some points related to this issue. The implementation can either use the hot or cold standby service in order to make this module fault tolerant.

If the hot option is adopted, the configuration state information can be redundantly stored in the system. In the case of failure of the station supporting the currrently active CM instance, the passive CM instance automatically takes the active role (this is inherent to hot standby Rmodules). As the configuration state information is kept, any reconfiguration required can be quickly performed. Another atractive point of this scheme is that the recovery capability for the configuration manager instances is also automated. This can be obtained if the CM Rmodule is included in the own configuration that it controls; thus CM instances can be automatically recreated.

The hot option is useful to implement a master CM, or to provide fault tolerance for a system having one single CM. In the master configuration manager case, the other CMs reliability could be obtained by having them configured in a hierarchy in which the responsibility over a CM is placed on its superior CM.

Adopting the cold option would require that, in case of failure, the CM recovers the dynamic part of the configuration information from the system. This requires a query to every station in order to find its state and the module instances that may be running there. This could take a relatively long time if the configuration has a large number of modules; although a backing store could be used to keep the configuration state information. During the CM recovery time, the reconfiguration capability would not be available; this reduces the reliability of the approach. Also the recovery process has to be initiated by an external agent: an operator or another CM.

The choice of one of these options depends on the particular system requirements. We have choosen the hot option because it is useful as a practical example of the use of this service.

6.1.1.2 Configuration Manager Operation

The CM deals with a single subsystem configuration each time. It provides two basic functions: start-up of the configuration and shut-down of the configuration. In addition it can answer queries on the status of the Rmodule instances and report changes of their status. In the prototype system, these functions can be accessed through replicated terminals. The configuration operations are performed through messages sent to the station operating system modules; these are briefly described in Appendix A.3. The CM operation is divided into two main phases:

(1) Start-up: In this operation phase, the CM executes the necessary configuration control operations to start-up the application system module instances. The activities in this phase are performed as follows:

Creation of module instances: Following the allocation of instances their module type object code is loaded to the stations (only if required; object codes could be stored in ROM at stations), and the instances are created there. The type loading and instance creation are performed through messages generated by the CM and sent to the station module manager and the subsystem loader modules.

Linkage of module instances: The module interface ports connection specification is translated to a system compatible format. This addressing information is them passed by the CM, through standard link messages, to each station link manager. This activity is performed serially for every exitport of each module instance in the configuration.

Start-up of module instances: In this step, the CM sends a module start message to each host station module manager. The configuration start-up phase is then finished.

This sequence of operations assures that the instances are fully prepared when started. During this phase, the CM does not try to receive any station status report from the Status Collector. However, while performing the operations to start up a given instance a failure of this instance host station could happen. As a consequence, the CM detects an error, i.e., a timeout occurs. In this case, the CM puts the instance in failed state and avoids the command of any further operation involving that instance. When the startup phase finishes, the CM can receive station status reports and treat the failure. The CM could also be designed to recover from failures concurrently with the start-up activities. This would require a more complex configuration control algorithm. The solution adopted is based on the assumption that station failures are not very frequent, at least during this phase, and also intends to simplify the CM design.

(2) Steady State: In this phase the CM activities are initiated by the receipt of status information coming from the status collector module. The Configuration manager checks wether the station that had its state altered is supporting any module instance at that time. If the station is not being used only its status is updated. Otherwise, reconfiguration activity is required in order to replace the failed module instance(s). This is similar to the start-up phase; the only difference is that the exitports connected to module instances being replaced must also be relinked.

An optimization is possible by forcing identifiers, used to address module instances which refer to instances associated to the same Rmodule, to be unique within the subsystem. In this case it is not necessary to perform relinkage of exitports because the addressing information does not change during the operation of the system. This is also convenient for implementing reliable transactions efficiently, as will be discussed in section 6.2.3.4.

The CM can also shut-down a configuration being controlled. This is obtained by stopping and deleting all the Rmodule instances. This operation should be executed when the application system is in a quiescent state.

6.1.1.3 Concurrent Failures

If an active configuration manager instance fails during a configuration activity, some uncertainty about the operations that were already performed will happen. Even using the hot standby service it is not possible to eliminate this problem. This is due to the impossibility of determining whether messages corresponding to configuration operations, have been sent out when a failure occurs.

This can be easily solved if the operating system modules implement every configuration operation in an **idempotent** fashion. For example, a linkage operation is not performed if the exitport is already connected to the same entryport as contained in the new link request command. Similar behaviour can be programmed for all other configuration operation commands. Taking advantage of this feature, the CM does not need to execute any special procedure on recovery, e.g., it does not need to query each station to ascertain the current configuration. It simply restarts and continues performing the configuration operations after recovery. Idempotent configuration operations can be achieved through very small changes in the standard operating system modules of Conic.

6.1.2 Status Collector

This module periodically collects the status of every station in the physical configuration. Any change noticed in the status of a station is reported to the configuration manager module.

6.1.2.1 Implementation

The primary mechanism for status collection is implemented by an operating system module resident at each station. This module periodically sends a status message to the SC module. Every time a status message is received by the SC, a bit indicating the event is set. Periodically the SC module checks and resets these bits. A bit not set indicates a station failure, which is reported to the CM. The reliability of this decision is assured by the bounded response and communication system assumptions and by the proper specification of the periodicity of the station operating system module and SC module. The SC algorithm can also notice when a station joins the physical configuration, e.g., after being repaired; this event is also reported to the CM. The SC Rmodule uses the hot standby service; this is required in order to provide automatic fault tolerance for the configuration management service.

We have assummed that on the first signal of station failure all application processing is stopped. In some cases, processing capability can still remain at the station. This remaining capability can be used to communicate the occurrence of the failure to the SC.

The design of the SC is intended to provide the minimum functionallity required, and is independent of the particular communication system. If required it can have an optimized implementation, which depends on the particular communication system. For example: (1) In a carrier sense multiple access medium (CSMA), the status collector can detect "on the fly" all the messages transmitted from each station. A service module at each station can transmmit a dummy message if no real message is generated within a given time interval, or the SC could query the station in the absence of a real message. This allows a reduction of the messages used for failure detection. (2) In a communication system using token based access control an inherent failure detection capability is available; this can provide for status collection without need to duplicate the same function.

6.1.2.2 Consistent Failure Detection

It is necessary to make sure that all station status changes are consistently noticed, otherwise the CM cannot perform the required configuration operations. This is straightforward except in a case where stations can recover after a failure so quickly that the SC does not notice the change. This can happen if the fault was transient and the diagnosis algorithm performed in the station is very fast, and/or in case of concurrent failure of the station and SC occurs. We consider that the diagnosis of the failure will normally take a long time, which would allow the failure to be identified. However to ensure that this happens, the time the station stays unavailable can be artificially extended: the operating system module in the station simply waits enough time to allow consistent failure detection before making the station operational again. This ensures that station failures are consistently detected and is supported by a very simple mechanism. We found other ways of solving the problem: (1) To use a special lock-step protocol, based on the same principles as the two-phase commit protocol [Gray 78], for registering the start up of the station with the status collector. This protocol allows in any combined failure case the identification of a station failure. (2) To use a unique identifier generator, e.g., a

system time. On start up a station takes a new identifier, which when compared with its previous identifier allows the detection of a failure. These two schemes are independent of any timing consideration but require more complicated support mechanisms.

6.1.3 Comments

The configuration manager uses a strategy that ensures that configuration operations are completed once started; the mechanism for status collection assures that every station status change is consistently communicated to the configuration manager. Thus, if failures do not occur very quickly in sequence, a stable configuration is reached. Other features which extend the configuration management service capabilities are discussed below.

Management Interface. The management service keeps information about the state of the Rmodules of the system. Thus it can answer queries about the status of Rmodules and report changes of their status. Queries about the status of Rmodules can be directly answered by the configuration manager. The set of events to be communicated to a given module can either be determined at run-time or specified through the configuration language. The first approach requires some run-time interaction with the configuration manager in order to specify the name of the Rmodule to be informed, and the events this Rmodule wants to know about. In this case special programming is required, and the identity of the events would have to be known by the application program and sent in messages. This is against modularity and is not directly supported by the current Conic language proposal. An option would be to restrict the events to the state of Rmodules to which a given Rmodule is connected; this does not require the explicit naming of other modules. The second approach requires a simple extension of the current Conic configuration language, which is subject of current research [Magee 83b]. This interface provides a general capability for implementing error recovery strategies, which is similar to that available in other distributed system proposals, e.g., [Lantz 80]. The use of the management interface is discussed in section 7.3.

Distributed configuration management. We have assumed that the reliability requirements of a sub-system can be met within this subsystem. Thus, a system composed of many reliable subsystems is also reliable. Each of these subsystems can be controlled by its own configuration manager. However, modules of different systems may need to

communicate. In this case, the configuration manager modules need to co-operate in order to establish the connections of these modules at run-time.

Operator Interface. The configuration management activity is automatically performed. This may be mandatory in order to meet the requirements of some applications, e.g., real-time or unnattended operation. However, an operator can participate in the configuration activity, e.g., to defer a re-configuration activity to a latter stage of the system operation.

Diagnosis Capability. Most of the failures are caused by transient faults, which occur at least one order of magnitude more frequently than permanent failures [McConnell 79, Ohm 79]. The reliability of the system is directly dependent on the availability of stations. Thus, after a failure, diagnostic tests can be performed, and if the tests reveal transient faults, the station can be again available for service. We consider that the configuration management activity and the diagnosis activity are operationally independent. However they are complementary activities for the achievement of reliability.

6.2 Hot Standby Support

Support for the hot standby service is performed at two different stages: The first stage is performed in the development system as explained in sections 4.5 and 5.4.4, the specific activities will be clarified along this section. The second one is performed at run-time by the configuration management service, as described in the previous section, and other additional mechanisms. This section presents the details of the additional mechanisms. The run-time support mechanisms deal with the following basic functions:

1. Save primitive. A mechanism is needed to transfer the state information from the active instance storage to the passive instance storage at each invokation of the save primitive.

2. Instance Management. A mechanism is needed to manage module instances which implement an Rmodule. This mechanism is in charge of determining the role to be performed by each instance, and performing the initialisation of new instances.

3. Reliable transaction. A mechanism is needed in order to support the communications performed through reliable ports.

The first two functions are implemented by a special management task. This task is standardized and can be automatically included in the Rmodule code at the system development stage, without any change in its programming. In the next section we discuss the relevant details of the design of the save primitive. Instance management and the reliable transaction support are discussed in separate sections.

6.2.1 Save Primitive

In order to implement the save primitive, it is necessary to specify its semantics.

6.2.1.1 Save Semantics

-- When a save is invoked **state information** is transferred to the passive instance storage. The transfer is atomic: either all the state is transferred, or the effect is as if the save had not been invoked. The passive instance can fail during a save operation. The task that invokes the save is suspended until the save operation completes or the failure of the passive instance is detected.

-- If the active instance fails, the passive instance is made active. The state information transferred is enough to assure the repeatability of any output that may have been delivered by the task in a previous execution.

-- If no passive instance is available, the save returns immediately.

6.2.1.2 State Variables

In order to discuss the implementation of the save primitive, it is convenient to classify the task variables in two types: state variables and auxiliary variables (non-state variables).

State variables are those that exist and must maintain their values across task actions.

Auxiliary variables are those whose existence and value depends on the particular task action, e.g., the local variables of actions within a task, and the execution control information associated to these actions.

For recovery after failures, only the state variables must be kept in stable storage. That is, by inspecting the state variables the action the task was performing before the failure can be retaken. This can have some drawbacks which will be discussed later in this section.

6.2.1.3 Implementation Options

Here the discussion is concerned with the issue of assuring the repeatability of the outputs resulting from a task execution. Repeatability is related to the state information tranferred in each save and can be implemented in a number of ways. We assume that the data and control information representation in the stations where the instances run are compatible.

Implementation I

All the task variables, state and auxiliary, are stored in the task stack. The save is invoked through a procedure call. At this moment a copy of all the words being used in the task stack is taken. This information is transferred by the underlying mechanism and copied into the corresponding task stack in the passive instance. On recovery, the task is made ready to execute. The task continues execution from the first instruction after the last save that has been invoked by this task in the previously active instance, just as if it had invoked the save (the contents of the task stack are assumed to be storage position independent). This assures the repeatability of the execution.

This implementation allows a great deal of flexibility for the use of a save. Saves can be introduced at any point in the task program, and provided that the rules set up in chapter V are enforced, no inconsistency will arise. However, redundant information will be tranferred in each save. An optimization is considered in the next implementation.

Implementation II

An optimization can be made if we have a closer look at the way state and auxiliary variables are allocated. State variables can be allocated statically in the store used for the task data. Auxiliary variables will be allocated at run time according to the path taken by the task program. Not all state variables need to be transfered in every save: only those that have changed since the last save must be

transfered. A run-time mechanism could identify and transfer state variables which have changed. The fact that state variables location is known allows a simple implementation of this mechanism. Mechanisms based in a similar principle are proposed in [Randell 75, Kant 83], although in a different context. The effectivennes of the approach is dependent on both the time taken to assemble and disassemble the state transfer messages and on the communication system delays. Thus if the size of all the state variables is shorter than a given threshold (system dependent) it does not result in any benefit. In this case, all state variables can be transfered in every save. The option to be used can be taken at compilation time.

Auxiliary variables include local variables of procedures called within the task and associate execution control information. It would be more difficult to select which ones must be saved at run time. It is easier to save all of them without attempting any optimization.

Implementation III

A further optimization is possible if a particular style of programming is adopted: The task programming is organized in a decision table style. In this case the only control information that needs to be saved is an identifier in the table of the action being performed. The other auxiliary variables can be recreated when the task re-executes. The state variables are transfered as in the previous implementation. This implementation allows maximum efficiency at a cost of a special style of programming. This should not be very restrictive if we realize that any task that loops around a Select receive statment uses a decision table style of programming (see appendix A.1).

Simpler Save

It is also possible to define a simpler save operation: the save transfers only the state variables. This would require the application to define the algorithms to allow the program to proceed after a failure; the state variables must also be explicitly specified. The problem with this approach is that it does not impose the use of standard mechanisms to deal with failures. Thus recovery activities would depend on the programmer's discipline. The save semantics we propose can be combined with standard mechanisms for communication and allow an automatic provision of fault tolerance. A simpler save can be specified and implemented if required.

6.2.1.4 Save Operation

The save operation is performed through a request-reply transaction with the management task. The task invoking the save is suspended while the operation is performed. Thus the management task can directly access the variables to be transfered without any conflict. The transfer is effected through a message sent to the management task of the passive instance which assures the failure atomicity of the operation. Also, according to the error confinement assumption, no erroneous information will be present in this message. The tranfer operation is idempotent; thus a simple protocol can be used. If the state variables cannot be contained in a standard size message, a proper protocol should assure the failure atomicity of the save operation.

6.2.2 Instance Management

The instance management functions are performed by the management task (fig. 6.2). It is assumed that the save uses implementation I, described in the previous section.

6.2.2.1 Active Instance Operation

(1) **Transference of state** -- The variables to be transfered in a save are copied from the application task stack in a message and sent to the passive instance management task. When this message is acknowledged the save completes. If there is no passive instance available, the state information is only locally updated, and the save operation completes.

(2) **Detection of the passive instance failure** -- The passive instance management task periodically attempts communication with its active instance counterpart. A failure is detected when this message is not received within the specified time interval. A failure is also detected if the passive instance does not acknowledges the state transference message.

(3) **State initialisation** -- After receiving a notification of the existence of a passive instance, the management task initialises the state of that instance. It does so by sending its local state to its counterpart. Thus the initialisation does not interfere with the activities of the application tasks.

6.2.2.2 Passive Instance Operation

(1) Transference of state -- The management task receives state transference messages, transfers the contents to the task stack, and acknowledges the message confirming the execution of the operation.

(2) Detection of the active instance failure -- A failure of the active instance is detected if the message periodically generated by the passive instance is not acknowledged. After the failure is detected the instance is made active, and its application tasks are activated.

(3) State initialisation -- A new instance is started in the passive role. After starting, the management task notifies its counterpart. Next the active instance management task initialises the state of the passive instance. If no active instance is available or if the state initialisation is not completed a double failure has occurred. In this case, state information has been lost and the instance is put in an failed state.

6.2.2.3 Operation Details

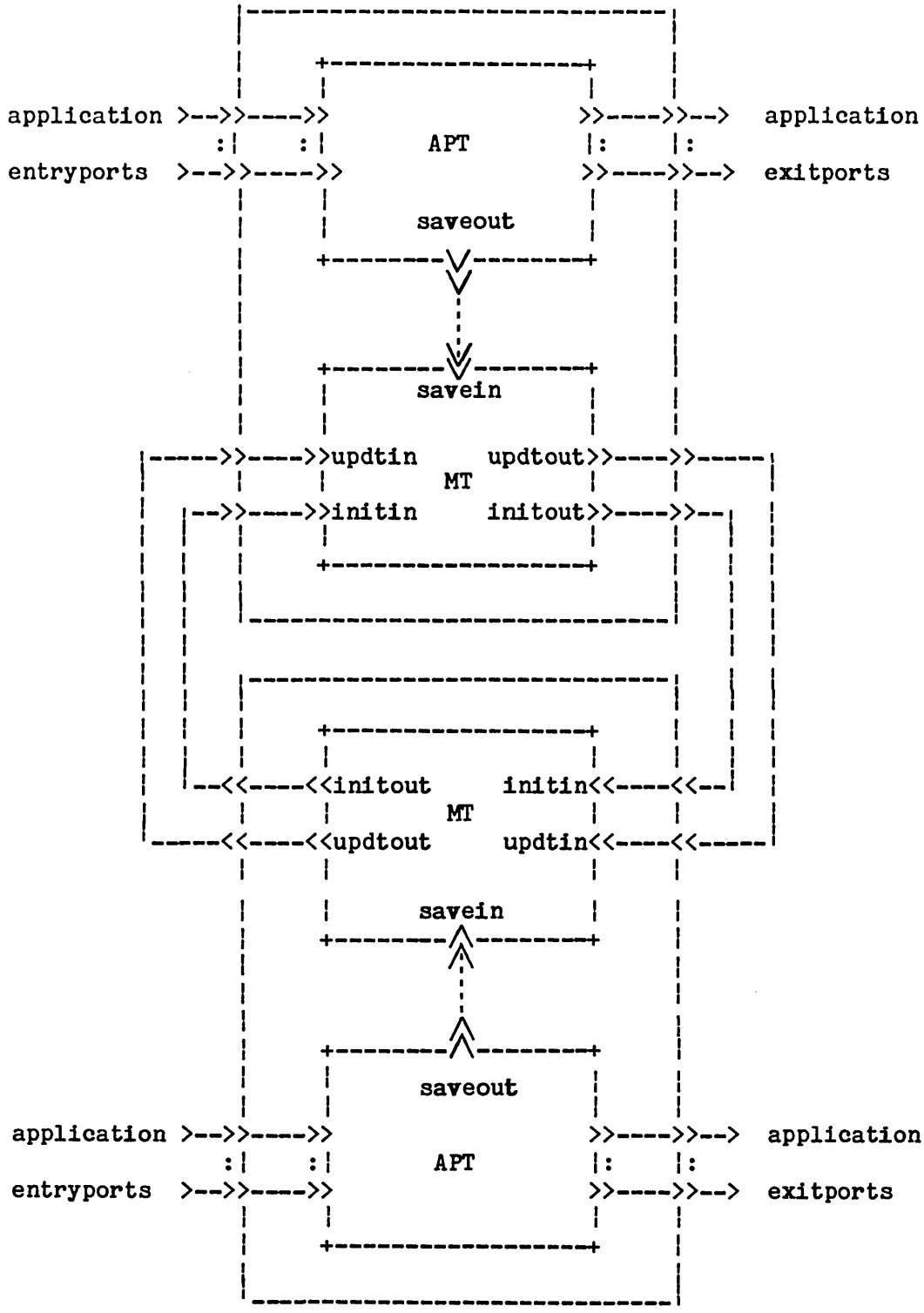
Figure 6.2 shows the two instances which implement a hot standby Rmodule, the management tasks, and their connections. For simplification, only one application task is represented. On system start-up the configuration manager creates one instance as active and another as passive. Saves in the task programming are transformed in a procedure which performs a request-reply transaction through the saveout port. The management task of the active instance transfers the state to the passive instance by sending a message through the notify port updtout. The transfer is confirmed through an acknowledgement received from updtin. The passive instance management task periodically sends the message required for failure detection through updtout. This message is acknowledged by the active instance; however, for optimization the acknowledgement can be "piggy backed" on a state transfer message.

The state initialization is requested by a new passive instance through a message sent through the notify port initout. The active instance can use one or more messages to perform the state initialisation. However, only after all the state is transferred, the initialisation is considered completed. If the active instance fails

before the initialisation is completed the passive instance is put in a failed state and its tasks are not activated. This ensures that no inconsistent behaviour occurs. A module can contain multiple application tasks, thus the implementation can be optimised: the saves invoked by different tasks can be pipelined or combined in a message in order to increase the speed of the state transfer operation; also saves can be performed concurrently with the initialisation operation. For fast execution of state transfer, a special communication channel can be used, e.g., high priority messages, or the message can be transmitted through a separate physical channel.

The instance failure detection capability relies on the bounded response and communication system assumptions (see section 4.1), and is achieved independently by the management task of each instance. This capability could be implemented by the configuration management service. In this case a third entity is required to ensure the recovery of the configuration management service module instances (see discussion in section 6.1.1.1). The current implementation of failure detection allows simple hot standby configurations to be implemented without a configuration management service. Thus we were able to simulate CM failures in order to test its automatic recovery. The integration of failure detection capability within the communication system is also possible. This is similar to that already discussed in section 6.1.2.1 for the status collection function.

The management task controls the execution of the application tasks by direct access to the kernel routines. It is worth pointing out that the state of the kernel data structures do not need to be saved. All information needed for recovery is present in the stack of each task, and the required kernel state is recovered by re-executing the tasks. Thus the standard kernel message and task management functions can be used and the implementation is thereby simplified.



MT - management task

APT - application task

Fig. 6.2 Hot Standby Implementation

6.2.3 Reliable Transaction

In chapter V it was demonstrated how the effect of failures is limited to the generation of replicated messages. Here the discussion is restricted to the support mechanisms that ensure the transaction semantics required for state consistency. A reliable transaction is performed between a reliable exitport and reliable entryport. A specific mechanism is needed to deal with the events pertaining to each port type. In the following discussion we determine the requirements for the support mechanisms.

6.2.3.1 Scenario

The pattern of the transaction is illustrated in fig. 5.1. A transaction is initiated when the source task sends a request message through an exitport. Every request contains a sequence number, this sequence number is always greater than the last one used in that port connection. The transaction finishes when a reply message with the same sequence number is received from the target task. It is worth pointing out that the sequence numbers are state variables, hence they too are preserved in case of failures.

In the presentation below, it is assumed that the communication system provides a "best effort" service for message delivery and does not get partitioned. This facilitates the identification of the minimum requirements for the support mechanisms. The integration of the transaction support mechanisms in a particular communication system, and the issue of addressing the transaction messages are discussed in section 6.2.3.4. The transaction support mechanisms must assure reliability in case of Rmodule instances or communication failures. Failures can occur at any time during the transaction duration. The possible cases are considered below:

(1) **The request or reply message fails:**

This can be recovered if the exitport mechanism keeps transmitting the request message until a reply with the same sequence number is received. For example, if a first request has failed a retried one can be accepted, and the corresponding reply would be produced; if it was the reply that had failed, the retried request informs the entryport side that the reply is still not available. The support mechanism in the entryport side can then pick up a copy of the reply message and send it back to the exitport side.

(2) The active instance owning the entryport fails. There are two cases:

a. The failure occurs before the request message is consumed and saved. The passive instance takes over, but does not know anything about the request message. This is equivalent to a request message failure. This is recovered by retrying the request; this message has to compete normally with other messages to be consumed at the target instance interface.

b. The failure occurs after the request message has been consumed and saved. On recovery, the task eventually generates the reply message. This reply can be a duplicate since the previous active instance could have already produced it. This repeated reply has to be filtered out by the exitport side support mechanism. The filtering is possible because the reply contains the same sequence number.

(3) The active instance owning the exitport fails. There are two cases:

a. The failure occurs before the request message can reach the entryport side. On recovery, a replicated instance of the request message is generated. This message is equivalent to a new request message for the entryport side.

b. The failure occurs after the request message has reached the entryport side. On recovery, an identical request message is again generated. It does not matter if the entryport side has accepted the previous request or not; this is dealt by the entryport side mechanism.

(4) A passive instance fails:

Passive instances do not receive or send application messages. Thus, the failure of a passive instance does not affect the transaction execution.

6.2.3.2 Requirements

The requirements for the support mechanisms can be summarized as follows:

Exitport support mechanism:

-- Retransmit periodically the request message until a matched reply message is received (or the timeout expires).

-- Filters out replicated reply messages.

Entryport support mechanism:

-- Filters out replicated request messages.

-- and, if the corresponding reply message is already produced it takes a copy of this message and sends it back to the exitport side.

On recovery, there is only one way for the exitport instance to know if a request message has been accepted by the entryport instance: Namely, to use the sequence number of this message in order to query the entryport instance about the fact. Obviously it is better to send the request message again and let the entryport instance filter out this request if necessary. The same reasoning is applied to the entryport instance and reply message. It is important to notice that transfer of control information to stable storage does not completly solve the problem and is more expensive than retransmission (specially in environments where message transport is reasonably reliable, e.g., any state-of-the-art local area network).

6.2.3.3 Implementation

The first attempt to meet the transaction requirements was by the use of program stubs inserted in the task programming. This requires each message to be always accepted in order to check its sequence number and discard the duplicates. In some cases this is not a problem, but guards can be used in order to control the acceptance of messages through a entryport. In this case the guard has to be evaluated and if false, the message should either be discarded or stored within the task data space in order to be available later on. Another problem occurs in the case that a reply message is lost. In this case, the exitport will retry the request in order to recover from the situation; but in the meantime the entryport can consume and process other messages. The retried request will have to compete at the instance interface and wait for the other messages to be processed, consequently the reply will have its recovery delayed, which also delays the transaction completion. In addition this implementation can cause deadlocks, in the case of failures, if no restrictions on program structure are imposed: e.g., a task T1 performs a (request-reply) transaction R1 with a task T2 and waits a response from T2 through a (request-reply) transaction R2. If T1 fails after R1 is accepted deadlock occurs since on recovery R1 cannot complete and proceed because it needs the associated reply, and T2 cannot recover the reply associated to R1, because it is waiting for the reply associated to R2, which cannot be recovered by T1.

In order to solve the problems mentioned above, a mechanism that works concurrently with the application task is required. This mechanism could be easily implemented by a standard service module. However, it is possible to take advantage of the Conic implementation and avoid this overhead. In Conic, remote transactions are supported by an interprocess communication service which is implemented by two service modules -- IPCIN and IPCOUT, showed in fig 6.3, taken from [Sloman 83]. IPCOUT builds the message frame for the transaction and holds it in an internal buffer until the transmission is completed in order to avoid further copy operations. IPCIN provides destination buffering for remote message transactions. These modules can have their programming easily changed in order to meet the requirements of reliable request-reply transactions. IPCOUT is modified in order to retransmit request messages, while IPCIN is modified in order to check and discard replicated request or reply messages, and also to pick up any already available reply message in the task data space and send it back to the entryport side. These modifications do not introduce any overhead in the processing of the standard Conic transactions. This approach allows the use of the standard Conic kernel interface procedures for operating on reliable ports. It also avoids any inefficiency or problem that is associated with the the use of program stubs.

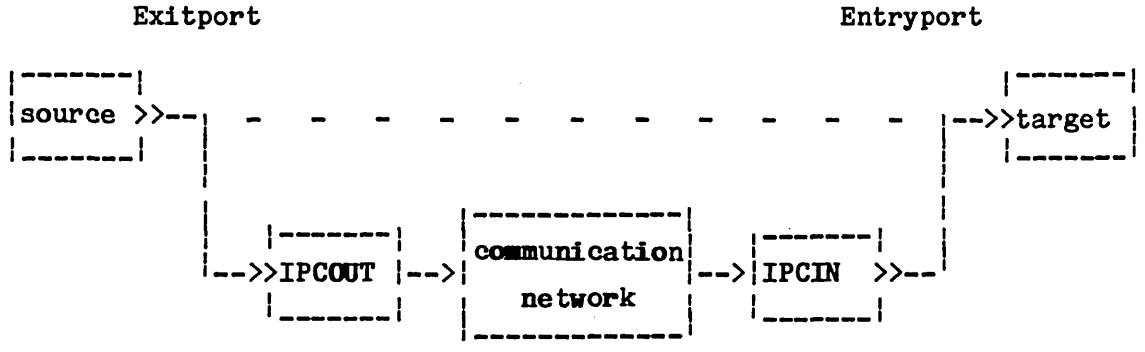


Fig. 6.3 Support of Remote Transactions

Both local (intra-station) and remote (inter-station) reliable request-reply transactions have the same semantics. Local transactions can can have their semantics enforced by a standard service module, as IPCIN, inserted in the message path, although some ineficiency is introduced. In the standard Conic system, local transactions are performed by directly copying the message from the sender to the destination task data space; this is performed by the kernel. Since the amount of processing necessary to enforce the semantics is very small, it can be performed totally in the kernel. This allows efficiency improvement and economy of resources at a cost of a small change in the Conic kernel.

The proposed design for the request-reply transaction uses a simple request-response (RR) protocol, and is intended for maximum efficiency and simplicity of implementation (see next section). N-to-one connections for reliable ports can be implemented as sets of n one-to-one connections; one for each of the n exitports. In this case it is necessary to keep a copy of each reply message at the entryport side. A request-response-acknowledgement (**RRA**) can be used for the implementation of N-to-one connections. In this case, the acknowledgement message is generated only after a save is performed at the exitport side, thus ensuring that the transaction is not repeated in case of failure. This economizes storage, since a copy of the reply message for each connection does not need to be kept at stable storage at the entryport side. However the RRA option slows down the completion of the transaction and requires more complicated support. This option can be used if the reply message is long, since if it is short, the incurred overheads do not make it worthwile. An RRA protocol can also be useful for the implementation of the interface of a server.

The use of an RRA protocol also makes it possible to transfer only the sequence number of the request message to stable storage at the entryport side. Thus, on recovery, the message content would have to be recovered from the source of the request . This strategy avoids the initial saving of the received message, but requires a more complicated implementation. Typical messages in control applications have short length [Prince 81] and the sequence number has to be saved anyway. Thus in a high speed communication network little efficiency improvement would be achieved.

Finally, it should be noticed that the transfer of state, through saves, is not explicitly related to the execution of actions. Thus no explicit state for actions needs to be considered to perform end-to-end control. All that is needed to enforce the transaction semantics is the checking of sequence numbers; which are independently saved. This simplifies the implementation of the transaction.

6.2.3.4 Communication System

The proposed support mechanisms are enough to provide the reliability required for the transaction, provided that the communication system does not get partitioned. In a practical application their reliability and efficiency depends on a special design of the communication system [Kleinrock 80, Powell 82, Wensley 78, Wolf 79, Smith 75]. Redundant communication paths must be used and fast fault detection and routing mechanisms are required. Also, the end-to-end control required for reliable transactions is essential for reliability and any lower level end-to-end control mechanism is not logically necessary, although it can be used for efficiency. Thus their use require a careful analysis of each case as pointed out by [Saltzer 81]. Some examples are presented below.

Example 1: In the model, we assumed the availability of large sequence numbers. This can be relaxed if the transport service can guarantee that messages are received in other stations in the same order they were generated in their source station. This could be inherent to particular implementation technology, e.g., some loops or rings, or imposed by a mechanism of the communication system. The size of sequence numbers also can be bounded if a maximum message lifetime is assured [Watson 81].

Example 2: Reference [Boyd 81] considers the use of the "token control" access technique to allow link switch-over without losing or corrupting in transit messages. There, it is also stated that the time needed to regenerate a lost tokens meets stringent real-time requirements. In this case, no message needs to be retransmitted to recover from message failures. The influence of the token control technique in the status collection was also discussed in section 6.1.2.1.

Example 3: A number of state-of-the-art local area networks offer a very low failure rate for broadcast message transmission , e.g., [Shoch 80]. The design can take advantage of this fact in order to implement transactions efficiently, as proposed in [Spector 82]. A reliable request-reply transaction can be performed by two broadcast messages, one for the request and another for the reply. This also solves the message addressing issue, since the same physical instance of a message (request or reply) can reach both target instances (the active and the passive); however the modules instances associated to the same Rmodule have to have unique addresses within the network. Considering that message and module instance failures are rare, this would work in most of the cases. In order to cater for both possible failures the request message can be retransmited. The retry interval is set up to be not much bigger than the average time interval needed to complete the transaction in normal conditions. The scheme can cause some unnecessary delay in case of pure message failures. However, considering that message failures are rare, and that delays of the same order can occur because of module failures, this should not be considered a flaw. The use of broadcasts requires the communication interface to act on every message transmitted, and catch only those directed to module instances residing in its attached station. This should be considered in the interface design in order not to cause wastage of processing power available in the station, e.g., a separate communication processor can be used. The use of broadcast addresses also simplifies reconfiguration, as indicated in section 6.1.1.2.

Example 4: A reliable transaction can also be implemented by unicast message transmissions (notify transactions). In this case two unicast transmissions are used by a source to direct messages to each of the potential targets: the active and the passive instances; this also solves the addressing issue. Hence for each reliable transaction the transmission of at least two request and two reply messages is needed. In practice this depends on the particular implementation of the service: if a simple datagram service is used the above is true; if a more elaborate transport service is used, e.g., each message is acknowledged, more messages are needed. This scheme is used in our prototype implementation of reliable transactions. Its main advantage is that it uses services generally provided by available message transport mechanisms. The main disadvantage is the potentially high delays, which can cause "hiccups" at the application level.

It is recognised that fault tolerant systems require special purpose designs. In the reliable transaction case, this can be essential for meeting the real-time demands of some control applications in a cost effective way.

6.4 Summary of the Chapter

In this chapter, the design and implementation details of the fault tolerance support mechanisms were presented. The chapter was organized in two main sections.

The first was related to the configuration management service, which is implemented by two system modules: the configuration manager and the status collector. The reliability service required for these modules is application dependent. However, in order to support the requirements of some applications they need themselves to be fault tolerant. This is obtained by using the hot standby service for their implementation. In addition to fault tolerance, they have to provide consistent configuration control capability which requires special care in its design to be achieved in an environment where failures occur (which includes failures of instances implementing the configuration management service). The configuration manager uses a strategy that ensures that configuration operations are always completed once started. The status collector assures that every station status change is consistently communicated to the configuration manager. Thus, if failures do not occur very quickly in sequence, a stable configuration for the system is obtained. The design of the configuration magement service provides the minimum capabilities required, and has a very simple implementation. However, extensions can be made when necessary; this was considered in sections 6.1.2.1 and 6.1.3.

The second was concerned with the mechanisms which are associated with the hot standby service. They support the programming of strong failure dependency systems. Issues related to the save primitive, management of instances implementing hot standby Rmodules, and reliable request-reply transaction were presented. The save primitive semantics was specified and implementation options which allow the reduction of state information tranferred in save operations were proposed. The save operation is performed with the intervention of a standard management task that also performs the management functions of its own module instance. This includes controls of the role (passive or active) being performed by the instance, instantiation of state of new instances, and detection of failures of its counterpart. The management task can be added to application Rmodules in the development system without any change in their programming.

The requirements for the reliable request-reply transaction were identified and a simple design which meets them is proposed. It can be implemented by slight changes in the modules which provide the interprocess communication mechanisms in the present Conic implementation. The reliability and efficiency of the support mechanisms depend on the particular communication system; this was discussed in section 6.2.3.4. The mechanisms which support the transaction are orthogonal to the other mechanisms associated with the hot standby service.

CHAPTER VII

FAULT TOLERANT APPLICATION SYSTEMS

In this chapter important aspects of the construction of fault tolerant application system are presented. A methodology to develop fault tolerant application systems is described, and relevant design issues concerning strong and weak failure dependency systems are discussed. Finally, based on two examples, observations about the use of the reliability services are made.

7.1 Methodology

In this section, a methodology for developing fault tolerant application systems is briefly presented. We assume that the system is already defined, has its elements identified, and has its interaction and performance requirements specified. This can be obtained by using techniques such as those presented in [Weitzman 80]. For simplicity, consider that each element corresponds to a module. The methodology consists of the following steps:

Step 1. Each module is separately designed, modules from a library can also be used. In this step, fault-avoidance techniques for validation and verification should be used at the module level.

Step 2. The composition and interconnection structure of the system is specified using the Conic configuration language. At this stage validation and verification techniques should be used at the system level, e.g., to check the consistency of connections and to assure the absence of deadlocks caused by the interconnection structure of modules. Note that the system can be operationally tested before fault tolerance is added.

Step 3. The reliability service used by each module and the type of each port are specified in order to meet the requirements of the system. The corresponding configuration specification file is produced. The possible mappings from modules to stations are also defined. Thus the consistency of the possible mappings can be checked. The configuration specification is processed by the translator program and the configuration description file is produced. This configuration description file contains all the information needed by the configuration management entities to control the configuration. Also the support for hot standby Rmodules is provided.

Step 4. The fault tolerant system is tested in the most realistic operational conditions. The same techniques and tools used to test standard Conic systems can be used, e.g., debugger module.

Hot standby Rmodules are treated in step 3. In principle, it is possible to perform automatically all translations to support these Rmodules from their specification, i.e., provision of the management task, insertion of saves in the program of tasks, and the provision of support for reliable ports. In practice, this depends on the available development tools, and on the particular Rmodule, e.g., application defined saves can be used.

The exact order of activities performed in steps 2 and 3 depends on the development tools. Also, some feedback may be necessary in order to eliminate the problems found in any of the steps. By using this methodology the configuration description information is produced.

The methodology presented in this section deals with the mechanistic aspects of developing an application system. It can be supported by extending the standard Conic development tools. Other relevant issues for the design of fault tolerant application systems are discussed in the next sections.

7.2 Management Interface

The configuration manager can answer queries about the status of Rmodule instances, and also report changes of status of Rmodule instances. This capability is useful in the implementation of application dependent recovery activities in weak failure dependency systems. In this way, specific Rmodules would collect or be provided with status information and can, for example, perform activities such as re-initilisation of repaired Rmodule instances, or enforce a system shut-down procedure after failures.

The concentration of recovery activities in specific Rmodules seems to be a good strategy to be used for weak failure dependency systems. In a system composed of various modules, it would be difficult to equip each individual module with the necessary mechanisms. In addition, each module would have to be specially designed for a given configuration. The concentration requires only the special design of the specific Rmodules, which provide the recovery activities for the other Rmodules. Also, the Rmodules in charge of the recovery activities can be made fault tolerant by the use of the hot standby service, thus providing reliability for their functions. Even so, some assumption should be made about the frequency of faults, i.e., if faults occur concurrently with a recovery activity special care in the programming is required to allow the operation to continue and complete.

7.3 Response Time

An important consideration for the design of real-time control applications is the response time of the system. The formal calculation and validation of the response time of a system of interacting modules is a very complex issue, which is subject of current research, e.g., [Bernstein 81, Karg 84]. In this section, we resort to informal means to show that reliable request-reply transactions performed in a system of hot standby modules will terminate within a bounded time interval. This is considered below.

7.3.1 Normal Case

Here we are interested in estimating the time taken to perform a reliable a request-reply transaction in the case where no instance failure occurs during the transaction execution. This time interval is measured from the time the request message is sent to the time the reply message is received.

(1) $T_{normal} <= T_{request} + T_{acceptance} + T_{action} + T_{save} + T_{reply}$

$T_{request}$, T_{reply} are the maximum times taken to transport the request and reply messages, respectively, between both sides performing a transaction. T_{save} is the maximum time taken to perform a save operation. A save is also performed through messages. Thus these times can be specified, although they are dependent on the particular communication system.

$T_{acceptance}$ is the maximum time the request message takes to be accepted by the application task. T_{action} is the maximum time taken to perform the action. These times depend on the processing load in the station where the instance is running, which is known beforehand. Available techniques can be used to determine a maximum value for this parameter, e.g., [Leinbaugh 80].

7.3.2 Failure Case

When a failure occurs, the time taken by the active instance to take over the active role, i.e. $T_{recovery}$, should be considered.

(2) $T_{failure} <= T_{request} + T_{acceptance} + T_{action} + T_{save} + T_{recovery} + T_{reply}$

Thus,

(3) $T_{failure} <= T_{normal} + T_{recovery}$

where,

(4) $T_{recovery} <= T_{detection} + T_{activation} + T_{action}$

$T_{detection}$ is the time taken for the failure to be detected, and $T_{activation}$ is the time taken to switch the passive instance to the active role. Equation (4) includes T_{action} in order to consider the worst case where the target instance fails after executing the action and before the reply reaches the source instance. In this case the action is executed again on recovery. We assume that a new passive instance will take some time to be created by the system. Thus, on recovery, T_{save} is not counted again.

7.3.3 Comments

Equation (1) shows that the time to execute a reliable transaction, in the normal case, is equal to that for execution of standard Conic request-reply transaction plus the time to perform a save operation.

In equation (4) $T_{activation}$ is small since it is related only to the activation of the application tasks. Remember that a passive instance is fully prepared to perform application processing. Thus, equation (4) shows that the delay introduced by a module failure is dominated by $T_{detection}$ and T_{action}. In process control applications, most actions are typically short. However, if an action takes a long time to execute, an intermediate save can be used to minimize $T_{recovery}$, although this delays the action execution in the normal case. $T_{detection}$

depends on the implementation of the detection mechanism. As discussed in section 6.2, techniques to minimize this parameter can be used. These techniques can also be used for speeding up the save operation. The equations are valid for remote and local transactions; only the message transport times, -- $T_{request}$ and T_{reply} have different values.

So far, we have considered that actions are executed locally within a task, which is the most common case in control applications. However there are cases were transactions needs to be performed within an action, i.e., reliable transactions are nested. In this case equation (2) can be employed sucessively in order to calculate the response time of the top transaction. However, if the tasks participating in the transaction are in different stations it is very unlikely that all of them fail together. Under this assumption it is sufficient to consider how the failure of each one separately affects the total response time. The transaction response time is then determined by the worst response time incurred by a single task failure.

The equations presented above can be used in order to calculate the response time of reliable request-reply transactions, which is required to determine the mimimum timeout value to be associated with reliable transactions. Also the cost in time of fault tolerance can be estimated. This is important in selecting a communication system, or in evaluating the suitability of the approach for a given application.

7.4 Input/Output

Input/Output may well be the weakst link of fault tolerant systems. Many authors and designs simply assume that everything outside the computer system is reliable, and ignore the problem. However, input/output activities are important in control applications because they provide the interface with the environment. In the two following sections we first consider input and output issues separately, related comments are made in section 7.4.3.

7.4.1 Input

In some cases instead of having a single input, it is required to have multiple replicated inputs, each one connected to a different station, so that no single fault can cause the loss of incoming information. Sensors used in control applications are generally less reliable than the computer stations to which they are attached. In this case, sets of replicated sensors can be connected to the same station. A

module in the station can collect their readings and perform some filtering algorithm in order to obtain a reliable reading. Some sensors can be embedded in stations. In this case, connections are made easier since they can be specified at a logical level by the configuration language. Also, repaired sensors can be automatically reintegrated in the system by the configuration management service.

For recovery it may be necessary to remember what information has been fed into the system. This can be readily implemented by using a hot standby Rmodule, and reliable transactions to communicate the information to the rest of the system. In other class of applications, information can be redundantly fed into the system, e.g., radar applications.

7.4.2 Output

An output can be modelled as the sending of a message out of the system. In some applications, outputs cannot be repeated without harmful effects, e.g., chemical mix control [Schoeffler 79]. In this case a solution is to extend the boundaries of the system in order to include the output interface. Hot standby Rmodules can perform outputs through reliable transactions. The interface would filter out repeated outputs. In other applications, outputs can be repeated without any harmful effect, e.g., any switch activation, and writing of information to a magnetic disk. In this case, no filtering of repeated outputs is needed; the guarantee of repeatable outputs that is provided by hot standby Rmodules is enough. It should be pointed out that a system of hot standby modules can produce consistent outputs in presence of single instance failures.

It is sometimes difficult to have the interface replicated all the way to the final output. Thus the output interface should be contructed in such a way that its reliability does not compromise the rest of the system. However, an exception is provided in [Sklaroff 76], where multiple actuators are used to drive flight control surfaces. In this case, if an actuator fails, the remainder have enough power to drive the surfaces to the desired position.

7.4.3 Comments

The ideal case of considering the input/output interface reliable and accessible by any station of the system is not usually feasible. In some applications, the equipment being controlled is physically distributed, and so are the stations. The diversity of input and output devices and the specific way they are employed should also be considered. Thus it is difficult to draw general conclusions in this area; some examples are given below:

Example 1. Some devices have characteristics and applications that make their replication simple, e.g., (1) a pump is loosely coupled to its environment. Thus a number of pumps can be employed and be operated in parallel or in standby mode in order to provide fault tolerance. Devices like this can be controlled by an embedded station in which case they do not require a common physical interface with two or more stations of the system; (2) a hot standby Rmodule can be interfaced to a terminal driver module. In this case, the physical terminal can be replicated. Each replicated terminal needs to be interfaced only to each of the stations where the instances of the associated Rmodule can run. The interaction with the system can be performed through the terminal associated with the currently active instance.

Example 2. Other devices have to be connected to at least two stations for availability, e.g., a magnetic disk. In this case each disk unit is also replicated, and information is redundantly kept in the replicated disks. A reliable disk server can be implemented by a hot standby Rmodule, which can hide the replicated disks from the rest of the system.

Example 3. A weak failure dependency system can be implemented by cold standby Rmodules. In this case, system state will be lost due to failures. Thus on recovery, any previous input can be ignored and some initial state is enforced on the interface. Information provided by input sensors can be used to ensure that this initial state is compatible with the current state of the application. This requires special application programming.

Example 4. Interrupt capability is not usually found in fault tolerant system proposals. Inputs are sampled under program control. This is justified as a condition for verification and validation. In strong failure dependency systems, the treatment of interrupts must be synchronized for consistent recovery. In our proposal, this can be obtainned by using standard service modules to treat the interrupts, and using reliable transactions to feed them into the system. However the response time to interrupts cannot be as low as that of a non-replicated system.

7.5 Application Programming

In this section, two examples of application programs are considered. Firstly we discuss the approach used in order to design the support mechanisms themselves for the fault tolerant system. Secondly we show how to make a classical program fault tolerant: the dining philosophers. Some observations based on the use of the techniques are made for each example.

7.5.1 Support Mechanisms

The configuration management service modules make use of the reliability services (see discussion in section 6.1.3). The configuration manager uses the hot standby service, and keeps state information related to the allocation of instances under its control. This provides the capabilities of fast and automatic (re)configuration operations. For similar reasons the status collector Rmodule also uses the hot standby service. A somewhat special example of techniques that can be used for the programming of cold standby Rmodules is provided by the management task, used in the implementation of hot standby Rmodules (see section 6.2): This task does not keep state, thus being similar to a task of a cold standby Rmodule. The implementation of the support mechanisms allow us to make the following observations.

(1) It is not always possible to devise a partition of an application system functions in modules in such a way that a weak failure dependency system is obtained. For example, the service used by the configuration manager is dependent on the specification required: If automatic failure recovery is required, the hot standby service has to be used.

(2) According to our assumptions the failure of a module instance can be unambiguously determined by the other module instances. We took advantage of this fact in the designs of the status collector module and the management task of hot standby Rmodules. If the structure of the application under consideration is more complex, the configuration management service capabilities of reporting events and answering queries may be required.

(3) Failure recovery for cold standby Rmodule instances is mostly an application concern; the system can only create a new instance. The procedure for recovery is application dependent. In some cases it can be just the normal procedure of the Rmodule, e.g., as in the case of a sensor. In other cases, e.g., the management task which supports instances of hot standby Rmodules, it can require the recovery of information from other instances, and also synchronization with these instances.

7.6.2 Dining Philosophers

The dining philosophers is a well know problem normally used to investigate synchronizing mechanisms in co-operating sequential processes [Dijkstra 75]. However it has also been used to illustrate the application of backward error recovery techniques for fault tolerance [Shrivastava 78] (see section 3.2.1. for related discussion). The modules and their interconection structure are represented in fig. 7.1, for a group of four philosophers. The code of the Rmodules: table, fork, and philosopher are presented in figures 7.2, 7.3, and 7.4, respectively; a brief explanation of the functions of each module is also presented in the figures. The action being performed by each philosopher (thinking, sitting, eating) and the current allocation of each fork (none, left, right) are displayed in a terminal (not represented). It is assumed that the Rmodules use the hot standby service, and that all communications are performed through reliable ports. The example can have a practical interpretation: The table represents a database that is constantly read and updated by application modules. The philosophers represent the application modules; their action is in some way conditioned by the database module, e.g., a deadlock avoidance policy is enforced. The forks represent resources which have to be used in an orderly fashion by the application modules; they also perform input/output activities. The code of these Rmodules are exactly identical to the original (unreliable) Conic program, with the exception of the saves. The dining philosopher example allows us to make the following observations.

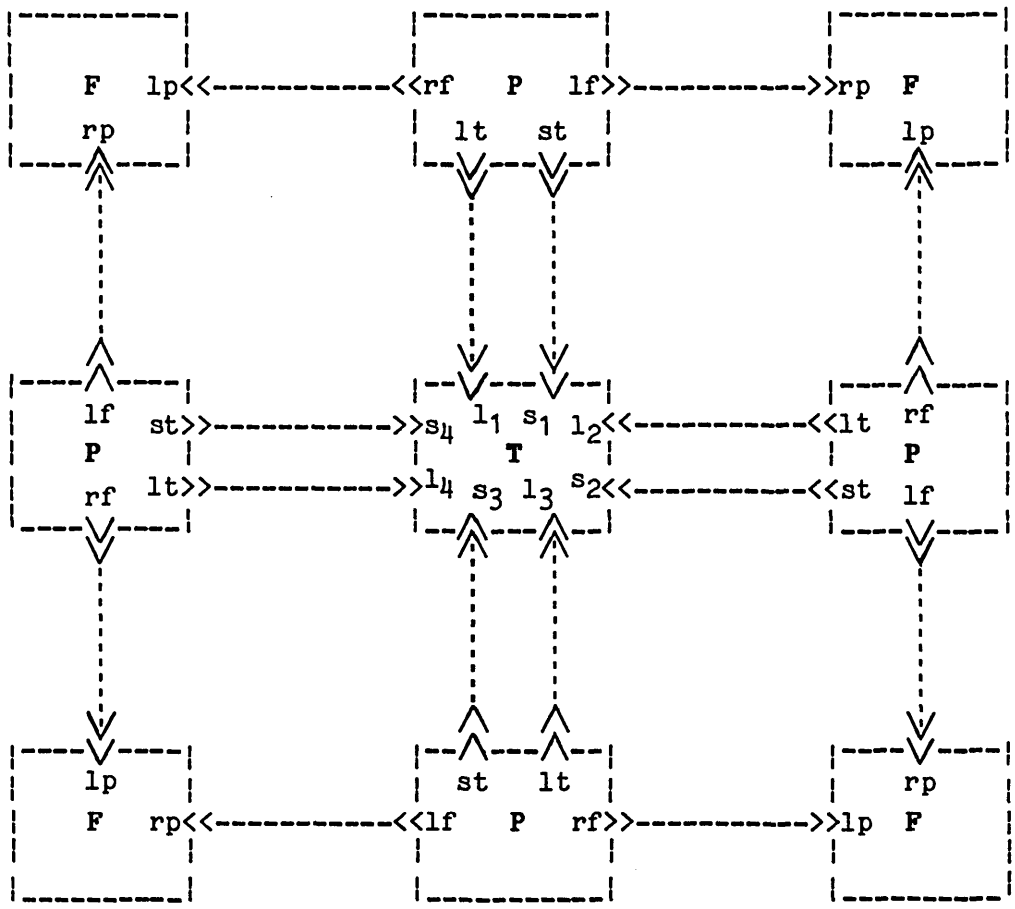
(1) It shows how immediate the application of the technique is. Two changes in relation to the standard Conic program are required. The first, is required in order to insert the transaction control code and saves at proper points in the program of the tasks. In the example only the saves were introduced in the task program (the code associated to the end-to-end control sequence numbers is not shown). This was performed manually, however the saves and control code could be automatically introduced by a preprocessor in the development system; the information required is defined by the type of the ports of the module. The second change, is required in order to transform n-to-one connections to an equivalent set of n one-to-one connections. Thus the entryports of the table module were replicated. However it should be pointed out that the transformation is very simple, and either can be performed automatically, or not be explicitly required if a more elaborate implementation is available.

(2) It shows the application of the save rules. The rule enforced by each of the saves is indicated in the figures. For the table module a save is introduced between the RECEIVE and REPLY statments. These saves are necessary to enforce rule I, and assure the repeatability of the reply messages. The fork module displays its current allocation by calling the procedure displayallocation, which performs the actual outputs through the exit port output. Thus, a save placed before the displayallocation call statement surely enforces rule II: a save is always performed before a second transaction through the output port; thus the recover of the matched reply is always possible. The other two saves enforce rule I, as in the table module. It is interesting to note that the save enforcing rule II does not need to be performed: because the structure of the fork program rule II is already enforced by the remaining saves. In an elaborate implementation, the execution of each save can be automatically decided by extra code associate to each save; the decision depends on the path taken by the program during its execution. In some cases, it is also possible to remove redundant saves in an optimisation stage. For example, the saves in the philosopher program are enough to ensure rule II for all its ports; this assures the repeatability for the transactions performed by the philosopher instances. Thus, it is not necessary to place a save before each SEND statement.

(3) It can be seen that in any single failure case the system continues working without any loss of state and that no inconsistent transactions are attempted by a module. For example, consider that a failure happens at point x, in fig. 7.4. On recovery all statements following the last save (marked by eating) will be performed again; thus in consequence of the failure the transactions will be repeated. The underlying mechanism ensures that the transactions on the leavetable, leftfork, and rightfork ports complete, and that the philosopher program makes progress. The recovery does not interfere with the activities of the other modules of the system. There is no need of special programming of any of the modules in order to guarantee consistency of their state, e.g., value of the sitting variable. However failures can lead to a degradation of the response time. For the philosopher module, this can be remedied by inserting saves within the delays, i.e., by using intermediate saves. Finally, it is interesting to note that both output ports (fork and philosopher modules) need not be reliable: The characters are written in the same positions on the screen of the terminal. Thus, in case of repeated outputs the dysplayed words are simply overwritten, the outputs are idempotent (this illustrates the discussion in section 7.4.2).

(4) It also illustrates a problem with idempotent actions, i.e., to program the modules in a way that no inconsistency arises as a consequence of repeated execution of the same action. This is an efficient way of solving communication problems in some applications [Herbert 81]. In the example, this technique can lead to strange situations. For instance, consider a fork module and suppose that a philosopher puts down a fork and fails before a confirmation of this action is received. In the meantime, a second philosopher picks up the same fork. On recovery the first philosopher has to wait until the second philosopher put down the fork. Only then, can the first philosopher continue its activities. A similar situation occurs for the table module; if a philosopher fails and the variable sitting is equal to n - 1. In this case, the table module would also have to keep more history state and have more elaborate algorithms in order to take consistent decisions, e.g., for updating the value of sitting in case of repetition of requests. Thus the use of idempotent actions makes a precise evaluation of the response time of the system more

difficult, and requires some effort to be programmmed. In our approach, reliable request-reply transactions and the recovery technique provide the required reliability for the system. However, our approach allows idempotent actions to be used when worthwhile, e.g., in the design of the configuration manager module (see section 6.1.1.3 for discussion), and in the output operations of the fork and philosopher modules (observation 3).



lf : leftfork, rf : rightfork

lp : leftphilosopher, rp : rightphilosopher

st : sittable, lt : leavetable

l_1, l_2, l_3, l_4 : leave

s_1, s_2, s_3, s_4 : sit

F : fork module, P : philosopher module, T : table module

Fig. 7.1 The Dining Philosopher System

```
MODULE dtable(n:integer);

{    -- dtable permits at most (n - 1) philosophers to sit at the tasble,
        where there are n philosophers -- n is four in this example.
}

     ENTRYPORT* leave_i : signaltype REPLY signaltype;
                sit_i   : signaltype REPLY signaltype;

     TASK table;

          ENTRYPORT  leave_i : signaltype REPLY signaltype;
                     sit_i   : signaltype REPLY signaltype;

          VAR sitting     : integer;
              request, ok : signaltype;

          BEGIN
               sitting:= 0;
               ok:= signal;
               LOOP
                    SELECT
                         RECEIVE request FROM leave_i;

                         save;  { rule I }
                         REPLY ok TO leave_i;
                         sitting:= sitting - 1;
                    OR
                         WHEN sitting < (n - 1)
                         RECEIVE request FROM sit_i;

                         save;  { rule I }
                         REPLY ok TO sit_i;
                         sitting:= sitting - 1;
                    END;
               END;
          END;
BEGIN
END.
```

Fig 7.2 Table Module

---

* Only one pair of ports is represented, see discussion in observation (1).

```
MODULE dfork;
{    -- fork receives pickup and putdown requests from philosophers, but
        can only be allocated to at most one philosopher at a time
}
     USE philos.msg, file.def;

     ENTRYPORT  rightphil : forktype  REPLY  signaltype;
                leftphil  : forktype  REPLY  signaltype;
     EXITPORT   output    : string    REPLY  signaltype;

     TASK  sharedfork;

          ENTRYPORT  rightphil : forktype  REPLY  signaltype;
                     leftphil  : forktype  REPLY  signaltype;
          EXITPORT   output    : string    REPLY  signaltype;

          TYPE allocationtype = (none, left, right);

          VAR  allocated : allocationtype;
               request   : forktype;

          PROCEDURE displayallocation;
          BEGIN
{         -- displays in the terminal the allocation of the fork, uses
             exitport output.
}         END;
          BEGIN
               allocated:= none;
               LOOP
                    save;  { rule II }
                    displayallocation(allocated);
                    SELECT
                         WHEN ((allocated=none)) or (allocated=left))
                         RECEIVE request FROM leftphil;
                         save;  { rule I }
                         REPLY signal TO leftphil;
                         CASE request OF
                              pickup  : allocated:= left;
                              putdown : allocated:= none;
                         END;
                    OR
                         WHEN ((allocated=none)) or (allocated=right))
                         RECEIVE request FROM rightphil;
                         save; { rule I }
                         REPLY signal TO rightphil;
                         CASE request OF
                              pickup  : allocated:= right;
                              putdown : allocated:= none;
                         END;
                    END;
               END;
          END;
BEGIN
END;
```

Fig. 7.3 Fork Module

```
MODULE dphilosopher(thinktime, eattime: integer);
{    -- Philosopher repeatedly thinks, sits down at the table, picks up his
        left and right fork and eats. He requests permission before each of
        his actions.
}

     USE philos.msg, file.def;

     EXITPORT rightfork  :  forktype   REPLY signaltype;
              leftfork   :  forktype   REPLY signaltype;
              leavetable :  signaltype REPLY signaltype;
              sittable   :  signaltype REPLY signaltype;
              output     :  string     REPLY signaltype;

     PROCEDURE  displayactivity(activity : activitytype);
     BEGIN
{    -- displays  in  the terminal a figure corresponding to  the  activity
        being performed, uses the exitport output.
}
     END;

     TASK philosopher;
          EXITPORT rightfork  :  forktype   REPLY signaltype;
                   leftfork   :  forktype   REPLY signaltype;
                   leavetable :  signaltype REPLY signaltype;
                   sittable   :  signaltype REPLY signaltype;
                   output     :  string     REPLY signaltype;

          BEGIN
               request:= signal;
               pickuprequest:= pickup;  putdownrequest:= putdown;
               LOOP

     { thinking }   save;  { rule II }
                    displayactivity(thinking);
                    DELAY thinktime;
                    SEND request  TO sittable  WAIT  ok;

     { sitting  }   save  { rule II }
                    displayactivity(sitting);
                    SEND pickuprequest TO leftfork  WAIT ok;
                    SEND pickuprequest TO rightfork WAIT ok;

     { eating   }   save { rule II }
                    displayactivity(eating);
                    DELAY eattime;
                    SEND request TO leavetable WAIT ok;
                    SEND putdownrequest TO leftfork  WAIT ok;
                    SEND putdownrequest TO rightfork WAIT ok;
     {     x    }
               END;
          END;
     BEGIN
     END.
```

Fig. 7.4 Philosopher Module

7.6 Summary of the Chapter

In this chapter, the construction of fault tolerant application systems was discussed. A development methodology for obtaining the static configuration information needed by the configuration management service was presented. It can be implemented in a development system by extending the standard Conic development tools. Other relevant design issues were discussed. Specifically, in section 7.2 we discussed the use of the configuration management interface to implement recovery activities for weak failure dependency systems; in section 7.3 we presented an informal method to calculate the response time for a reliable request-reply transaction; this is needed in order to calculate the timeout value associated to reliable ports, and to check if real-time application requirements are met; in section 7.4 we discussed the issue of input/output interfacing in a fault tolerant system. Finally, in section 7.5, based on the evaluation of the design of the configuration management service modules and on an example, we make some observations. They are useful as a guidance in the development of fault tolerant application systems.

CHAPTER VIII

CONCLUSIONS and SUGGESTIONS for FURTHER WORK

This thesis has presented a proposal for a fault-tolerant distributed computer control system. This chapter reviews the important features of the design and suggests further work which would improve the applicability of the proposed system.

8.1 Review of the goals

In the following we relate the relevant aspects of the design with the goals that were set up in chapter I.

Simplicity:

The system supports two reliability services: hot and cold standby. By specifying the desired service, and providing the required redundancy, module fault tolerance is automatically obtained. The required support mechanisms are very simple.

On the one hand this is achieved by the centralized configuration management service design, and the original configuration capability of Conic. The centrallized design permits the efficient implementation of different reconfiguration algorithms. Only the stations supporting this service need to have the necessary resources, the rest of the stations require only the standard Conic support mechanisms. The built-in configuration capability of Conic simplifies the implementation of module instance replacement operations. The required configuration description information is easily obtained off-line, in a development system. At run-time, configuration operations are readily implemented by the configuration management service, which uses a simple algorithm to control the allocation of module instances. The configuration operations are performed through messages sent to the standard operating system of each station.

On the other hand the achievement of simplicity is aided by the error confinement assumption. This assumption simplifies the detection of failures in the system since their only visible effect is the stopping of module instances. A malfunctioning module cannot exibit arbitrary and malicious behaviour thereby corrupting critical information and/or leading other modules to fail. Since state is not corrupted through error propagation the design of weak and strong failure dependency systems is simplified, e.g., a simple checkpointing technique can be employed for the support of strong failure dependency systems. This also benefits the design of the configuration management service: A straightforward strategy can ensure the consistency of configuration operations in presence of failures during their execution. Also station and module failures can be unambiguously detected by checking timing constraints taken locally in a module; this simplifies the obtention of failure detection capability.

Transparency:

This goal should be discussed separately for weak and strong failure dependency systems:

Weak failure dependency systems are implemented by modules using the cold standby service. Provided that resources are available replacement of failed cold standby instances is automatically performed. In some cases this is enough to recover the module services, but in general recovery activities should be explicitly programmed by the application designer, e.g., recovery of state for the module instance. For some applications failures can be detected by the application modules themselves and resolved within their context. However, application independent capability of detecting application failures is provided by the configuration management service, and these failures can be reported to specific modules of the system. This can be used to implement application dependent failure recovery strategies in this class of systems.

Strong failure dependency systems can be transparently supported. In principle the system can be programmed without any concern for fault tolerance and afterwards automatically transformed in order to obtain that capability. However, enough flexibility is available to deal with application dependent situations, e.g., reliable and unreliable transactions can be used when desirable. In these cases explicit enforcement of the recovery technique is required. This can be obtained by using explicitly programmed saves, which requires program edition. Standard Conic development tools can be used for both types of system.

Modular Design:

The mechanisms which provide fault tolerance were implemented by standard modules, and require for their support the standard Conic run-time mechanisms. Configuration management capability is implemented by two modules: The configuration manager and the status collector modules. They can be independently tailored to meet the needs and selectively used in each application. The configuration management service is enough to provide the requirements of weak failure dependency systems; extra mechanisms are required for the support of strong failure dependency systems. The mechanisms for the programming of strong failure dependency systems are orthogonal to the configuration management mechanisms: Support for reliable transactions is obtained by simple changes in the standard communication system modules, in order to perform end-to-end controls which do not introduce any unreasonable overhead and are essential in any fault tolerant system. Hot standby modules require a standard management task which provides for transference of state to stable storage and the management of the role performed by a module instance. This task can be automatically included in hot standby modules and operates independently of the other support mechanisms.

Hardware Independence:

The mechanisms are directly transportable to any hardware supporting the basic Conic machine. Their reliability depends on the validity of the error confinement assumption. This can be ensured within a given probability according to the reliability requirements of each application. In fact programmers have long assumed aproximations of the error confinement environment. Software techniques can be used to ensure the assumption, although more efficiency and coverage can be given by using available semiconductor components which provide built in error detection mechanisms in the hardware. The current integrated circuit technology prices and the advantages that result of assuming error confinement make the option very attractive.

We have designed the modules which support fault tolerant application systems. Their relevant details were implemented in a small scale and tested in a prototype. They do not rely on any additional feature or service which may be as difficult to provide, e.g., a global-time reference available to all stations. Thus we believe that the critical issues were solved. The current design supports closed application systems. However, due to its modularity the modules can be tailored to fit the needs of different applications.

8.2 Suggestions for Further Work

Further work can be performed in different areas. It can improve the applicability of the approach in a variety of ways.

(1) The configuration language extensions we have suggested for specifying and building fault tolerant application systems can be fully integrated in the Conic development system. This would allow the reliability support required by each module to be automatically obtained from the system specification, e.g., the programming translations for the support of hot standby modules: management task and language primitives. Configuration language mechanisms for specifying more elaborated reconfiguration control strategies can also be studied and implemented.

(2) The refinement and extension of the fault tolerance support mechanisms: The configuration management service can be extended in the directions mentioned in chapter VI, e.g., it can support more dynamic configurations, and be distributed in different stations of the network. The mechanisms to provide an interface between application entities and the configuration management service can be fully defined and made available. This depends on their specification at the configuration language level. Support for hot standby modules could also be extended. Rmodules of this type can be allowed to use more than one passive instance. This would provide more reliability at some cost in efficiency and resources. It should be noted that the concept of reliable transactions and the associated recovery technique are independent of the number of passive instances, thus this option requires changes only in the underlying implementation. For optimised implementation of some applications, tasks of a same module can use shared variables; capability for dynamically linking reliable ports can also be desirable in some applications. The support of these options for hot standby modules could be provided.

(3) Reliable communication system. In the thesis we have specified some minimum properties for the communication system, which are enough to ensure the reliability of our design. However, the properties of a specific communication system can have influence in some points of our design, e.g., a communication system using a token based access control protocol inherently provides station failure detection capability. In addition this capability can be distributed in all stations of the system, which allows direct implementation of the status collection and the local detection of failures of hot standby instances. The influence of the communication system in the implementation of reliable transactions were also discussed in section 6.2.3.4. The design of a communication system taking in consideration these interactions can benefit the whole system.

(4) The examples we presented and the simplicity of our approach have shown its potential applicability. However, the development of further application systems is essential for its evaluation and the possible identification of points for refinement and extensions of the present support mechanisms. Also, the experience acquired will allow application independent functions (and probably transactions) to be identified and thus standardized for general application.

(5) The construction of reliable systems requires the use of several techniques. Tools for functions such as: the quantitative evaluation of reliability, the validation of response time, and verification of properties such as absence of deadlocks in application configurations, can be provided in the development system. The structural information needed for these functions can be obtained from the configuration specification of each application system. The identification of the other information needed, the selection of the suitable techniques, and their integration in useful tools would help the development of reliable application systems.

(6) Finally some system design questions are worthy of further investigation. For example, -- how does the partitioning of the functions of a system influence its reliability requirements ?

8.3 Conclusions

We have proposed a basic approach to the provision of fault tolerant systems. This has included software techniques for programming fault tolerant applications and the design of the mechanisms to support the approach. The basic system is able to support a class of embedded real-time applications; however its range of applicability can be extended as discussed in the previous section. The design was based on an existing system, Conic, to allow the identification and discussion of the critical and general issues. This work provides a useful classification of system types: weak and strong dependency systems, an analysis of the problems related to the error assumptions which can be adopted in the design of fault tolerant systems, and proposes simple solutions which can be used to provide fault tolerance to other systems. We believe that this work contributes to the understanding and construction of fault tolerant systems.

APPENDIX A

THE CONIC APPROACH

This appendix provides a concise outline of the Conic approach [Kramer 83]. Its purposes are to present the concepts and support mechanisms we use in the thesis.

A.1 Programming Language

Conic is based in Pascal [Wirth 76], which was extended with capability of concurrency and message passing. The primitive software component is a module, which can be seem as a logical abstraction of a component in a control application. Modules are are separately designed and compiled and different instances of a same module type can be used in a system. Each module contains at least one task, which are the units of program concurrency in Conic. The task's interface is defined by a number of exit and entry ports. These are of two kinds: exitports, through which messages are sent to other tasks, and entryports through which messages are received. A module interface is defined by the exit and entry ports exported by its tasks, only these ports can be used to perform inter module communication. A separate configuration language is used to specify the connections between entry and exit ports (see section A.2), and thus define the communication structure of a system of modules. Intra module port connections are specified by a link statement within the module.

A.1.1 Module & Task Structure

The general structure of a module and of a task are presented below.

```
MODULE <moduleidentifier> (parameters)
     USE <filename>|<filename>
     <entry and exit port declarations>
     <type definitions>
     <variable declarations>
     <procedure declarations>
     <taskname>|<taskname>
     <local link declarations>
BEGIN
     <initialisation code>
END

TASK <taskname> <priority>
     <entry and exit port declarations>
     <type definitions>
     <variable declarations>
     <procedure declarations>
BEGIN
     <taskcode>
END
```

Examples of module type definitions are shown below in outline. They form part of a patient monitoring system, formely presented in [Magee 83b], that is used as an example throughout this appendix.

```
MODULE nurseunit;
     USE patientypes;
     ENTRYPORT alarms[1..maxbed]:alarmstype;
     EXITPORT  query[1..maxbed]:signaltype REPLY patientstatustype;
{    -- The module displays alarms received on 'alarms' on a terminal
     and in response to input at the terminal displays the status of a
     particular patient by requesting it on 'query'[i];     }
END.
```

```
MODULE bedmonitor (scanrate:integer);
     USE patienttypes;
     EXITPORT  alarms:alarmstype;
     ENTRYPORT status:signaltype REPLY patientstatustype;
{    -- The module scans sensors attached to a patient every
     'scanrate' seconds. When the sensor readings are outside ranges
     set at a bed-side terminal display it displays an alarm at the
     bedside terminal and sends alarm messages to 'alarms'. Data on
     sensor readings and ranges are sent to 'status' in response to a
     'signal' request.   }
END.
```

The USE construct is used to specify **definition files** which contain sets of module related data types, e.g.

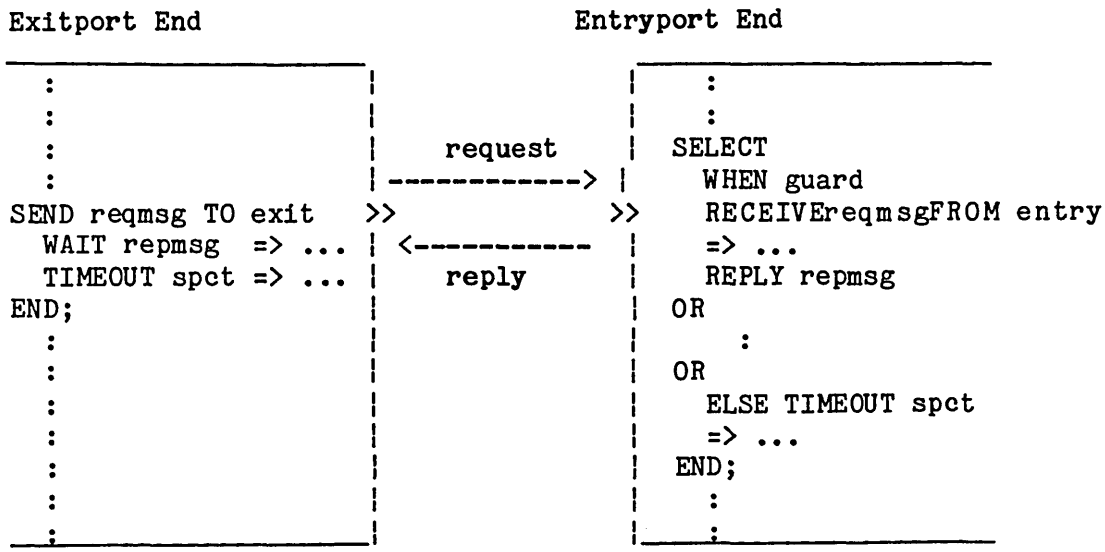
```
CONST maxbed       =  16;
TYPE  sensortype   = (bloodpressure, skinresistance,
                      temperature,pulse);
      readingtype  = record
           .          status: ok,notok;
                      value : integer;
                    end;
      alarmtype    = (outofrange, sensorfault), noalarm);
      readingstype = array[sensortype] of readingtype;
      alarmstype   = array[sensortype] of alarmtype;
      patientstatustype = record
                            name   : array[1..20] of char;
                            readings: readingstype;
                            alarms: alarmstype;
                          end;
```

A module specifying a particular file has access to the types in that file. Thus definition files can be used in defining the interface of each module, which allows type consistency checks of module interfaces to be peformed at the compilation time.

A.1.2 Communication Primitives

Two communication primitives are available, they are used to specify different communication styles required for the programming of control applications [Kramer 81].

Request-Reply: This primitive specifies a bidirectional, synchronous, request-reply transaction (figure A.1). After sending the request the sender task is suspended until the corresponding reply message is received back from the receiver side. At the language level these transactions are specified through SEND-WAIT and RECEIVE-REPLY statments. At the receiver task a SELECT statment may be used in order to allow a nondeterministic choice of one of the messages specified by enclosed RECEIVEs. Optionally a WHEN statment can be used to specify a logical condition (guard) to receive messages through an entryport. At the sender task a TIMEOUT statment may be used in order to exceptionally complete the transaction after an user specified time period(spct) expires. A LINKFAIL statment (not shown) may be used to signal that the exitport is not connected when a transaction is attempted. The port declarations associated to this primitive are also presented in figure A.1.

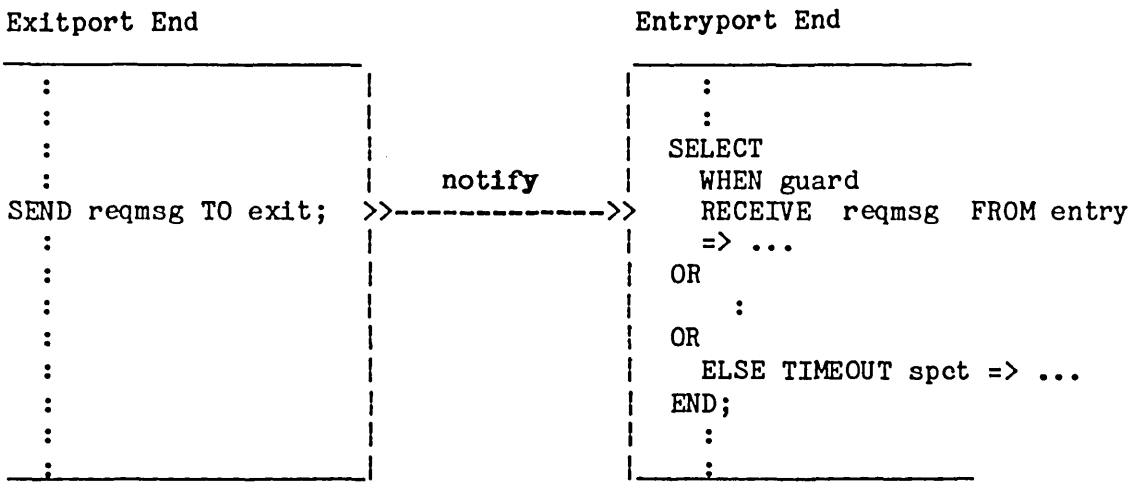


Port declarations for the request-reply transaction:

```
EXITPORT exit:request_message_type   REPLY reply_message_type;
ENTRYPORT entry:request_message_type REPLY reply_message_type;
```

Fig. A.1 Request-Reply Primitive

Notify: This primitive specifies a unidiretional assynchronous transaction (fig. A.2). The sender task continues execution after sending the notify message. At the language level this transaction is specified by SEND and RECEIVE statments. As for the request-reply SELECT, TIMEOUT, and guards can be used at the entryport side. The NOTIFY entryport declaration has an optional QUEUE part, which is used to specify the maximum size of a circular queue of message buffers associated with the entryport. If this queue gets full the oldest message is overwritten when a new buffer is required.



Port Declarations for the notify transaction:

EXITPORT exit:notify_message_type;

ENTRYPORT entry:notify_message_type QUEUE(integer constant);

Fig. A.2 Notify Primitive

A.1.3 Other Details

Link declarations describe the message port interconnections between tasks within a module. Only ports of the same transaction type may be linked, e.g., notify exitports to notify entryports.

LINK exitportname TO entryportname;

Two statements not standard in Pascal are provided:

(1) <loop> ::= LOOP <statment sequence> end

The sequence of statements is repeatedly executed.

(2) <delay statement> ::= DELAY <integer expression>

The delay statement delays the execution of the next statement by the time corresponding to the value of the integer expression.

A full description of the Conic programming language is available in [Magee 83a].

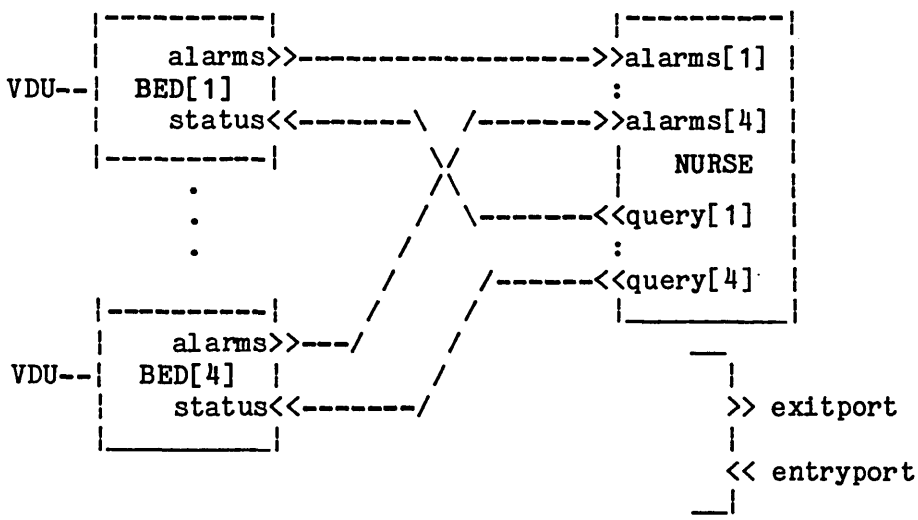
A.2 Configuration Language

Systems, in Conic, consist of interconnected sets of module instances. These are the smallest units in the configuration of a system and must reside in a single station. It is possible however to have more than one module instance in a station. A configuration language is used to specify a system, an example is presented in figure A.3. The specification identifies the module types from which the system will be constructed, declares the instances of these types which will exist in the system and describes the inter-connection of instances. These three functions are achieved by the USE, CREATE, and LINK constructs respectively:

USE - This construct provides a context of the module types from which a system is to be constructed. It allows instances of these types to be used in the system. In the example the module types are bedmonitor and nurseunit.

CREATE - This construct specifies named instances of module types which will exist in the system. The name of module instances must be unique within a specification. Instances can be parameterised, e.g., bedmonitor, and 'families' of module instances may be also specified, e.g., bed.

LINK - Module instances are connected together using this construct. It specifies the binding of entryports to exitports. The constraint of this binding is that an exitport must have the same type (transaction and message) as the entryport to which it is linked. All the links of this example are from one exitport to one entryport; i.e., the connections are one-to-one. However, entryports may have more than one exitport linked to them; i.e., n-to-one connections are allowed. The connections are specified by instancename.portname pairs.




```
SYSTEM  ward;
USE  bedmonitor, nurseunit;
CONST  nbed = 4;
CREATE  bed[1..nbed]: bedmonitor(100); <1,2,3,4>
        nurse       : nurseunit; <5>
LINK   bed[1..nbed].alarms  TO nurse.alarms;
       nurse.query[1..nbed] TO bed[1..nbed];
END;
```

Fig. A.3 System Specification (Ward Monitoring)

The mapping of modules to physical stations is currently specified by annotating the configuration specification whith the address to which each module instance should be allocated (numbers between < and >). Currently these specifications are used to create in a development system a load image for each station. In this case the configuration of instances and the binding information are static. Extensions of the configuration language, e.g., for allowing structuring of system specifications, and provision of the capability of changing at run-time the specification of a system (and consequently the configuration), are currently being studied [Magee 83b].

A.3 Run-Time Support

At run-time module instances are supported by the Conic machine (or operating system), this is illustrated in figure A.4. This base machine is replicated at every station of the system. The kernel provides for, tasking, local message passing, interrupt handling, time functions, and local link capability. Tasks are schedulled to run according the priorities associated to their definition. The communication system [Sloman 83], provides for remote message passing (communication among modules allocated in different stations). However local and remote message passing have identical semantics. The local management layer is outlined in figure A.5. The module manager supports the dynamic instantiation of module instances; the code of a module type is loaded in the station storage by using the store access module, which also provides storage reading capability. The link manager provides for the binding of entryports to exitports. The error manager provides run-time error reporting capability. This and the storage reading capability are used mainly for debugging purposes. It is interesting to note that with the exception of the kernel the base machine is programmed in Conic.

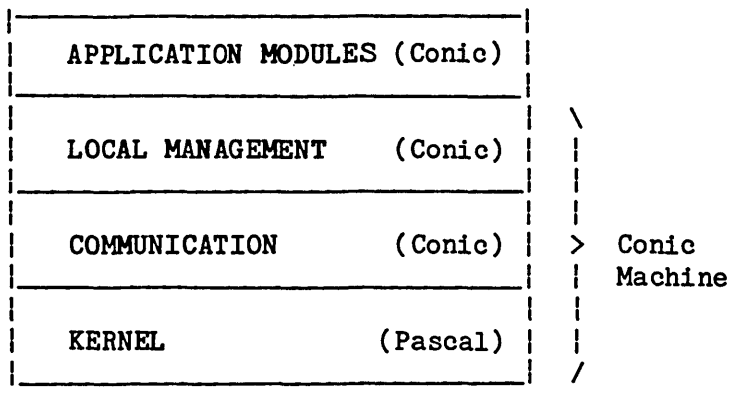


Figure A.4 Layer Model of a Conic Station

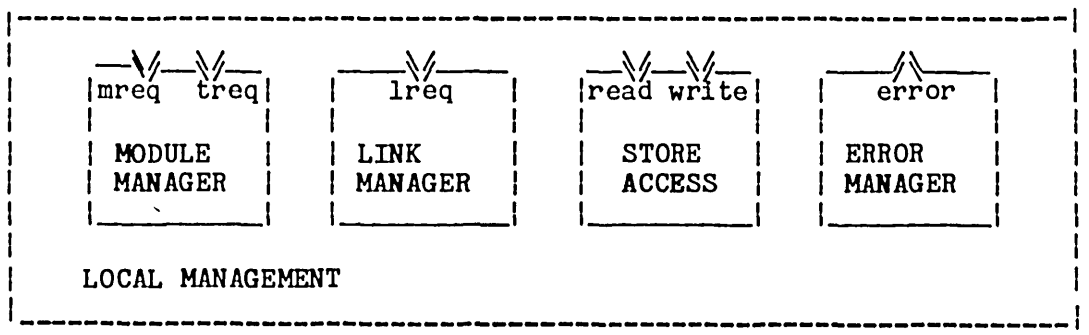


Fig. A.5 Local Management Layer

The services provided by the management layer are required for supporting dynamic configuration (the ability to modify a system while it is running). They provide support for configuration operations -- load module type, and create, link, and start module instances, at the station level. These operations are used to enforce the system specification. Configuration operations are requested through standard messages sent to the appropriated management modules. These messages contain system identifiers of objects in the target system (moduletype, moduleinstance, port). In general it may be necessary to undo configuration operations, thus the complementary operations are also provided. These services help the achievement of fault-tolerance capability, other uses of these services are discussed in [Magee 83b].

Other utility services can be provided by standard modules. Examples: a file server, a module type loader, a terminal driver. They can be introduced in the specification of a system at the development stage.

REFERENCES

[Adrion 82] Adrion, W., et al, Validation, Verification, and Testing of Computer Software, ACM Computing Surveys, Vol.14, N.2, June 82.

[Anderson 81] Anderson, T., Lee, P., Fault Tolerance Principles and Practice, Prentice/Hall International, 81.

[Anderson 82] Anderson, T., Lee, P., Fault Tolerance Terminology Proposals, 12th Symposium on Fault-Tolerant Computing, June 82.

[Andrews 79] Andrews, D., Using Executable Assertions for Testing and Fault Tolerance, 9th Symposium on Fault Tolerant Computing, June 79.

[Avizienis 71] Avizienis, A., et al, The STAR (Self-Testing and Repairing Computer, An Investigation of the Theory and Practice of Fault-Tolerant Computer Design, IEEE Tran. on Computers, Vol. C-20, November 71.

[Avizienis 76] Avizienis, A., Fault-Tolerant Systems, IEEE Transactions on Computers, Vol. C-25, 1976.

[Avizienis 82] Avizienis, A., The Four-Universe Information System Model for the Study of Fault-Tolerance, 12th Symposium on Fault-Tolerant Computing, June 82.

[Barigazzi 82] Barigazzi, G., et al, Reconfiguration Procedure in a Distributed Multiprocessor System, 12th Symposium on Fault-Tolerant Computing, June 82.

[Bartlet 81] Bartlet, J., A Non-Stop Kernel, Proc. of the Eight Symposium on Operating Systems Principles, December 81.

[Bernstein 79] Bernstein A., Scheneider, F., On Restrictions to Ensure Reproducible Behaviour in Concurrent Programs, TR79-374, Dept. of Computer Science, Cornell University, New York, November 80.

[Bernstein 81] Bernstein, A., Harter, P., Proving Real-time Properties of Programs with Temporal Logic, Proceedings of the Eighth Symposium on Operating Systems Principles, December 81.

[Borr 81] Borr, A., Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing, Proc. 7th Int. Conf. on Very Large Data Bases (VLDB), Cannes, France, September 81.

[Boyd 81] Boyd, R., Hughes J., High Availability Rings for Control of Telecommunication Switching. Cambridge Ring Modelling and Simulation Special Interest Group, March 81.

[Brown 83] Brown, I., Bosch E., The Synergism of Microcomputers and PLCs in a Network, 5th IFAC Workshop on Distributed Computer Control Systems, Transwal, South Africa, May 83.

[Chavade 82] Chavade, J., Crouzet, Y., The P.A.D: A Self-Checking Circuit for Fault-Detection in Microcomputers, 12th Symposium on Fault Tolerant Computing, June 82.

[Dijkstra 68] Dijkstra, E., Cooperating Sequential Processes, in Programming Languages, F. Genuys (Ed.), Academic Press, New York 68.

[Dijkstra 75] Dijkstra, E., Hierarchical Ordering of Sequential Processes, Acta Informatica, 1, N. 2, 75.

[Disparte 81] Disparte, C., A Self-Checking VLSI Microprocessor for Electronic Engine Control, 11th Symposium on Fault Tolerant Computing, June 81.

[Fischer 82a] Fischer, M., Lynch, N., A Lower Bound for the Time to Assure Interactive Consistency, Information Processing Letters, Vol.1, N. 4, June 82.

[Fischer 82b] Fischer, M., Lynch N., Impossibility of Distributed Consensus with one Faulty Process, MIT, MA. 02139, September 82.

[Frison 82] Frison, S., Wensley J., Interactive Consistency and its Impact on the Design of TMR Systems, 12th Symposium on Fault-Tolerant Computing, June 82.

[Garg 84] Garg, K., Design and Performance Validation Techniques for Distributed Systems Using Timed Petri Nets, Ph.D. Thesis (in preparation), Imperial College, London, England.

[Garcia 82] Garcia-Molina, H., Elections in a Distributed Computing System, IEEE Transactions on Computer, Vol. C-31, N. 1, January 82.

[Gaude 80] Gaude, G., et al, Design and Appraisal of Operating Systems Matched in Selective Active Redundancy, 10th Symposium on Fault-Tolerant Computing, October 80.

[Geitz 81] Geitz, G., Schmitter, E., BFS--Realization of a Fault-Tolerant Architecture, The 8th Annual Symp. on Computer Architecture, Sigarch Newsletter, Vol.9, N.3, May 81.

[Goetz 78] Goetz, F., Plisch, D., Hardware vs. Software Design Tradeoffs for Maintenance Functions in High-Reliability Real Time Systems, Proc. of the IEEE, Vol. 66, N. 10, October 78.

[Goldeberg 80] Goldberg, J. et al, Development and Evaluation of a Software-Implemented Fault-Tolerance(SIFT) Computer: SIFT Operating System, Interim Technical Report 2, SRI International, April 80.

[Gray 78] Gray, J., Notes on Data Base Operating Systems, in Operating Systems: An Advanced Course, Spring Verlag 78.

[Hamming 50] Hamming, R., Error Detecting and Error Correcting Codes, Bell System Technical Journal, Vol. 26, N.2, April 50.

[Herbert 81] Herbert, A., Needhan, R., Sequencing Computation Steps in a Network, Proceedings of the Eighth Symposium on Operating Systems Principles, December 81.

[Hetch 76] Hecht, H., Fault Tolerant Software for Real-Time Applications, Computing Surveys, 8, December 76.

[Hopkins 80] Hopkins, A. L., Fault-Tolerant System Design: Broad Brush and Fine Print, Computer, IEEE, March 80.

[Hopkins 78] Hopkins, et al, FTMP--A High Reliable Fault-Tolerant Multiprocessor for Aircraft, Proc. of the IEEE, Vol. 66, N. 10, October 78.

[Ihara 78] Ihara, H., et al, Fault-Tolerant System With Three Symmetric Computers, Proc. of the IEEE, Vol. 66, N. 10, October 78.

[Kain 80] Kain, R., Franta, W., Interprocess Communication Schemes Supporting System Reconfiguration, Compsac - Proc. Computer Software and Applications Conference, October 80.

[Kaiser 78] Kaiser, C., et al, Design of a Continuously Available Distributed Real-Time System, 8th Symposium on Fault-Tolerant Computing, June 78.

[Kant 83] Kant, K., Efficient Local Checkpointing for Software Fault Tolerance, Operating Systems Review, ACM, Vol. 17, N.2, April 83.

[Katsuki 78] Katzuki, D. et al, Pluribus-- An Operational Fault-Tolerant Multiprocessor, Proc. of the IEEE, Vol. 66, N. 10, October 78.

[Kim 79] Kim, K., Error Detection, Reconfiguration and Recovery in Distributed Computing Systems, The 1st Int. Conf. on Distributed Computing Systems, Huntsville, Alabama, October 79.

[Kleinrock 80] Kleinrock, K., et al, A Highly Reliable Distributed Loop Architecture, 10th Symposium on Fault-Tolerant Computing, October 80.

[Kopetz 82a] Kopetz, H., et al, An Architecture for a Maintainable Real Time System (MARS), Report MA 82/2, Technische Universitat Berlin, April 82.

[Kopetz 82b] Kopetz, H., The Failure Fault Model, 12th Symposium on Fault-Tolerant Computing, June 82.

[Kopetz 83] Kopetz, H., Real Time in Distributed Systems, 5th IFAC Workshop on Distributed Computer Control Systems, Transwal, South Africa, May 83.

[Kramer 81] Kramer, J., et al, Intertask Communication Primitives for Distributed Computer Control Systems, 2nd Conf. Distributed Computer Control Systems, Paris, April 81.

[Kramer 83] Kramer, J., et al, Conic: An Integrated Approach to Distributed Control Systems, IEE Proc., Vol. 130, Pt. E, N. 1, January 83.

[Lamport 78] Lamport, L., The Implementation of Reliable Distributed Multiprocess Systems, Computer Networks 2, 78.

[Lampson 81] Lampson, B., Atomic Transactions, in Distributed Systems Architecture and Implementation, an Advanced Course, Spring Verlag, 81.

[Lantz 80] Lantz, K., Uniform Interfaces for Distributed Systems, PhD thesis, Computer Science Dept., Univ. of Rochester, May 80.

[Laprie 82] Laprie, J., Costes A., Dependability: A Unifying Comcept for Reliable Computing, 10th Symposium on Fault-Tolerant Computing, June 82.

[Lee 82] Lee, P. A., Morgan D. E., Fundamental Concepts Of Fault Tolerant Computing - Progress Report, 12th Symposium on Fault-Tolerant Computing, June 82.

[Leinbaugh 80] Leinbaugh, D., Guaranteed Response Times in a Hard Real-Time Environment, IEEE Trans. on Software Eng., March 80.

[Leung 80] Leung C., Fault Tolerance in Packet Communication Computer Architectures, PhD Thesis, MIT, September 80.

[Levy 78] Levy J., A Multiple Computer System for Reliable Transaction Processing, Sigsmall Newsletter, October 78.

[Liskov 79] Liskov, B., et al, CLU Reference Manual, in Lecture notes in Computer Science, Vol. 114, Springer-Verlag, New York 81.

[Liskov 81] Liskov, B., On Linguistic Support for Distributed Programs, Proceedings IEEE Symp. on Reliability in Distributed Software and Data Base Systems, Pittisburgh, July 81.

[Liskov 83] Liskov, B., Scheifler, R., Guardians and Actions: Linguistic Support for Robust Distributed Programs, ACM Trans. on Programming Languages and Systems, Vol. 5, N. 3, July 83.

[Lomet 77] Lomet,D., Process Structuring, Synchronization, and Recovery Using Atomic Actions, in Proc. of an ACM Conference on Language Design for Reliable Software, Raleigh North, Carolina, March 77.

[Loques 83] Loques, O., Configuration Control, Conic Group Internal Report, Imperial College, January 83.

[Magee 83a] Magee, J., et al, The Conic Programming Language, Version 1.1, Research Report DOC 82/11, Dept. of Computing, Imperial College, London, March 83.

[Magee 83b] Magee, J., Kramer, J., Dynamic System Configuration for Distributed Real-time Systems, Real-time Systems Symposium, IEEE, Arlingnton, Virginia, December 83.

[Magee 84] Magee, J., Provision of Flexibility in Distributed Systems, PhD thesis (in preparation), Imperial College, 84.

[Marchal 82] Marchal, P., Courtois, B., On detecting the Hardware Failures Disrupting Programs in Microprocessors, 12th Symposium on Fault Tolerant Computing, June 82.

[McConnel 79] McConnel, S., et al., The Measurement and Analysis of Transient Errors in Digital Computing Systems, 9th Symposium on Fault-Tolerant Computing, IEEE, 79.

[McDermid 80] McDermid J., Checkpointing and Error Recovery in Distributed Systems, Royal Signals & Radar Establishment, Memorandum N. 3271, September 80.

[Merlin 77] Merlin, P., Randell, B., Consistent State Restoration in Distributed Systems, Report N. TR113, Computer Laboratory, Univ. of New Castle upon Tyne, 77.

[Morgan 77] Morgan, D., et al, A Survey of Methods for Improving Computer Network Reliability and Availability, IEEE Computer Magazine, November 77.

[Moss 81] Moss, E., Nested Transactions: An Approach to Reliable Distributed Computing. PhD thesis, MIT-Lab. for Comp. Science, April 81.

[Nelson 81] Nelson, B., Remote Procedure Call, Xerox Research Report, CSL-81-9, also PhD thesis report, Carnegie Mellon University, CMU-CS-81-119.

[Ohm 79] Ohm, V., Reliability Considerations for Semiconductor Memories, COMPCON Spring 79, IEEE 79.

[Osaki 80] Osaki, S., Nishio, T., Evaluation of Some Fault Tolerant Computer Architectures, in Lecture Notes in Computer Science 97, Springer Verlag, Berlin, 80.

[Pease 80] Pease, M. et al, Reaching Agreement in Presence of Faults, Journal of the ACM, April 80.

[Peterson 72] Peterson W., Weldon E., Error-correcting Codes, Cambridge, MA, MIT Press, 72.

[Powell 82] Powell, R., Dependability Evaluation of Communication Support Systems for Local Area Distributed Computing, 12th Symposium on Fault Tolerant Computing, June 82.

[Prince 81] Prince, S., Sloman, M., Communication Requirements of a Distributed Computer Control System, IEE proc., Vol. 128, Pt. E, N. 1, January 81.

[Randell 75] Randell, B., System Structure for Software Fault Tolerance, IEEE Trans. on Software Enginnering, Vol. SE-1, N. 2, June 75.

[Randell 78] Randell, B., et al, Reliability Issues in Computing System Design, ACM Computing Surveys, June 78.

[Rennels 78] Rennels, D., et al, A Study of Standard Building Blocks for the Design of Fault Tolerant Distributed Computing Control Systems, 8th Symposium on Fault Tolerant Computing, June 79.

[Robinson 82] Robinson, A. S., A User Oriented Perpective of Fault-Tolerant System Models and Terminologies, 12th Symposium on Fault-Tolerant Computing, June 82.

[Russel 80] Russel, D., State Restoration in Systems of Communicating Processes, IEEE Trans. on Software Engineering, March 80.

[Saltzer 81] Saltzer, J., et al, End-To-End Arguments in System Design, Proc. of the 2nd Int'l Conf. on Distributed Computing (IEEE), Paris, France, April 81.

[Scheneider 81] Scheneider, F. B., Schlichting, R., Towards Fault-Tolerant Process Control Software, 11th Symposium on Fault-Tolerant Computing, June 81.

[Schlichting 80] Schlichting 80, R., Scheneider F., Verification of Fault-Tolerant Software, TR. 80-446, Dept. of Comp. Science, Cornell University, New York, November 80.

[Schlichting 82] Schlichting 81, R., Scheneider F., Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems, TR. 81-479, Dept. of Comp. Science, Cornell University, New York, November 81.

[Schmid 82] Schmid, M., et al, Upset Exposure by Means of Abstraction Verification, 12th Symposium on Fault Tolerant Computing, June 82.

[Schoeffler 79] Schoeffler, J., Software Data Architecture for Distributed Data Acquisition and Control Systems, IFAC 79.

[Sedmak 80] Sedmak, R., Liebergot, H., Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration, IEEE Trans. on Computers, Vol c-29, N. 6, June 80.

[Sheridan 78] Sheridan, C., Space Shuttle Software, Datamation 24, July 78.

[Shoch 80] Shoch, J., Hupp, J., Measured Performance of an Ethernet Local Area Network, Comm. of ACM, Dec. 80.

[Shooman 68] Shooman, M., Probablistic Reliability: An Engineering Approach, McGraw-Hill, New York, 68.

[Shooman 83] Shooman, M., Software Engineering, Mcgraw-Hill, New York, 83.

[Shrivastava 79] Shrivastava, S., Concurrent Pascal with Backward Error Recovery: Language Features and Examples, Software-Practice and Experience, Vol. 9, 79.

[Sklaroff 76] Sklaroff, J., Redundancy Management Techniques for Space Shuttle Computers, IBM J. of Research and Development, January 76.

[Sloman 83] Sloman, M., et al, A Flexible Communication System for Distributed Computer Control, 5th IFAC Workshop on Distributed Computer Control Systems, Transwal, South Africa, May 83.

[Smith 75] Smith, B., A Damage and Fault Tolerant Input/Output Network, IEEE Trans. on Computers, Vol. C-24, n. 5, May 75.

[Souza 80] Souza, J., A Unified Method for the Benefit Analysis of Fault-Tolerance, 10th Symposium on Fault-Tolerant Computing, October 80.

[Spector 82] Spector, A., Performing Remote Operations Efficiently on a Local Computer Network, Comm. of ACM, April 82.

[Tillman 82] Tilman P., ADDAM - The ASWE Distributed Data Base Management, Distributed Data Bases, North Holland Publishing Company, 82.

[Toy 78] Toy, W., Fault-Tolerant Design of Local ESS Processors, Proc. of the IEEE, Vol. 66, N.10, October 78.

[USA-DOD 80] USA Dept. of Defence. Reference Manual for the ADA Programming Language, July 80.

[Wakerly 76] Wakerly J., Microcomputer Reliability Improvement Using Triple Modular Redundancy, Proc. of the IEEE,Vol. 64, N. 6, June 76.

[Watson 81] Watson, R., Timer-Based Mechanisms in Reliable Transport Protocol Connection Management, Computer Networks, Vol. 5, North-Holland Publishing Company, 1981.

[Wensley 76] Wensley, H., et al, The Design, Analysis, and Verification of The SIFT Fault Tolerant System, Proc. of the 2sd Int'l Conf. on Software Enginnering, October 82.

[Wensley 78] Wensley, H., et al, SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control. Proceedings of the IEEE, October 78.

[Wirth 76] Wirth, N., Jensen, K., Pascal User Manual and Report, Second Edition, Springer Verlag, 1976.

[Wolf 79] Wolf, J., Design of a Distributed Fault-Tolerant Loop Network, 9th Symposium on Fault Tolerant Computing, june 79.

[Wood 80] Wood, G. G., Review of Common Practice and Accepted Hardware Standards in Process Control, Real-time Data Handling and Process Control: Common Practices, Status and Future Trends, Proceedings of the First European Symposium, Berlin, October 79.

[Wulf 75] Wulf, W., Reliable Hardware/Software Architecture, IEEE Transactions on Software Engineering, June 75.

[Zave 76] Zave, P., On the Formal Definition of Processes, Proc. of Int. Conf. on Parallel Processing, IEEE, August 76.

[Zielinski 83] Zielinsky, K., Dynamic Module Allocation Algorithm for Fault-Tolerant Distributed Computer Control Systems, Working paper, Cambridge Computer Laboratory, June 83.