Imperial College of Science, Technology and Medicine
(University of London)
Department of Computing

A Software Architecture for Selective Fault-tolerance in Distributed Systems

Ricardo de Oliveira Anido

A thesis submitted in partial fulfillment of the requirements for the degree Doctor of Philosophy of the University of London, and the Diploma of Imperial College of Science, Technology and Medicine.

May 1989

Abstract

Distributed systems present an attractive platform to implement fault-tolerant systems. However, while distribution introduces the benefit of independent failure mode of the system components, it also introduces problems due to the fact that the knowledge of the system state is distributed among the components. Thus, inconsistencies may arise in the view that each component has of the system state.

This work presents the design and implementation of a programming environment, called Cronic, that offers the user a selection of different techniques for constructing fault-tolerant distributed systems. The different mechanisms available in Cronic are designed so that they can be used in an integrated manner. With this approach, a system implementor can chose the best techniques to suit the requirements of a particular application. The software methodology adopted distinguishes between programming in the small and programming in the large, with a programming language to specify the *task modules* in the system, and a configuration language to specify the composition of those modules into a system.

The techniques available in Cronic are backward recovery, forward recovery, and replication. *Backward recovery* consists basically in saving the state of the modules at certain stages of the computation so that they can be *rolled back* in case of failure. *Forward recovery*, on the other hand, uses the knowledge about the system state at the time of the failure to compensate for the failure without rolling back. *Replication* uses redundant resources in order to mask failures. At the programming level a language construct based on the atomic action concept serves as a framework for the specification of backward and forward recovery. At the configuration level, replication is used as a way to select the level of fault-tolerance intended for different modules. The mechanisms were designed to allow an easy integration with each other. With Cronic an application can use different fault tolerant techniques to cater for the different requirements of the application components.

This work presents the design of the language constructs at the programming and configuration levels in Cronic and discusses the support needed and their implementation. The implementation is modular, so that the set of mechanisms in the Cronic environment can be easily extended. Therefore, Cronic is also a tool for experimenting with different mechanisms for fault tolerance.

Acknowledgements

I would like to express my gratitude to my supervisor, Jeff Kramer, for his useful advice, encouragement and guidance throughout the development of this work. He has been more than a supervisor – he has been a very good friend.

I would also like to thank everyone at the Distributed Systems Programming group for the discussions and friendship. In special, my gratitude to Naranker Dulay, Jeff Magee and Kevin Twidle, for their patience in explaining the details of the Conic system inwards.

This work would't have been possible without the support received from CNPq-Brazil and Unicamp – Universidade Estatual de Campinas. Also important, and most cherished, was the support received from the dear friends Arlete, Carlos, Andréia and Adriana, happy inhabitants of the Hotel Brazil.

Contents

1. Introduction

1.1 Motivation and Problem 8

1.2 System Model 11

1.3 Our Approach 12

1.4 Thesis Overview 14

1.5 Faults and Fault-tolerance 15

1.6 Failure assumptions 17

1.7 Conic System: overview 18

1.7.1 Conic Programming Language 19

1.7.2 Conic Configuration Language 21

1.8 Summary 24

2. Basic Techniques for Fault-tolerance

2.1 Introduction 26

2.2 Backward recovery 27

2.2.1 Language Constructs for backward recovery 30

2.2.2 Commit Protocol 31

2.3 Forward recovery 32

2.4 Replication 36

2.4.1 Types of replication 37

2.4.2. Overview of problems to solve 38

2.5 Integration of techniques in Cronic 39

2.6 Summary 41

3. Fault-tolerance at the programming language level

3.1 Introduction 42

3.1.1 FT-Actions 44

3.1.2 Problems with FT-Actions 45

3.2 Actions 46

3.2.1 Nesting 47

3.2.2 Participating tasks 48

3.2.3 Inherited dependencies 49

3.2.4 Inherited dependencies and nesting 51

3.2.5 Commit protocol 53

3.3 Reflections on the design of actions 53

3.4 Actions vs. transactions 54

3.5 Summary 55

4. Forward recovery in actions

4.1 Introduction 57

4.2 Exceptions in Cronic 57

4.3 Exceptions and actions 59

4.3.1 Internal propagation in actions 60

4.3.2 The abort exception 61

4.3.3 External propagation of exceptions 63

4.5 Summary 65

5. Fault-tolerance at the configuration language level: clusters

5.1 Introduction 67

5.2 Clusters 68

5.2.1 Cluster semantics 68

5.3 Passive clusters 71

5.3.1 Choosing the new primary 71

5.3.2 History-dependent passive clusters 72

5.3.3 History-independent passive clusters 74

5.4 Active clusters 76

5.4.1 Link as filters 79
5.4.2 Application specific filters 80
5.5 Reintegration 81
5.6 Summary 82
6. A configuration language supporting replication
6.1 Introduction 84
6.2 Requirements 86
6.3 Instantiation 87
6.4 Linking 90
6.5 Specifying filters 90
6.6 Summary 91
7. Implementation Issues
7.1 Introduction 92
7.2 Basic services 93
7.2.1 Global ordering service 94
7.2.2 Global status service 95
7.2.3 Atomic broadcast service 96
7.3 Basic services implementation 97
7.3.1 Chang's reliable broadcast algorithm 98
7.3.2 Failure detection 104
7.3.3 Protocol performance 105
7.3.4 Reliable multicast in Cronic 106
7.3.5 Speeding up the failure recovery 107
7.4 Implementation of the higher level mechanisms 108
7.4.1 Backward recovery implementation 108
7.4.1.1 Commit protocol 109
7.4.1.2 Commit algorithm 112
7.4.1.3 Purging old information 114
7.4.2 Forward recovery implementation 115
7.4.2.1 Exception propagation protocol 116

7.4.3 Implementation of replication.....117
7.4.3.1 History-independent passive clusters.....117
7.4.3.2 History-independent passive clusters.....118
7.4.3.3 Implementation of active clusters.....118
7.4.3.4 Reintegration.....121
7.4.4 Configuration Language implementation.....122
8. Conclusions
8.1 Summary of research effort.....123
8.2 A word about the system model used.....125
8.3 Suggestions for future research.....126
8.4 Experience and conclusion.....128

References.....131

Chapter One

Introduction

1.1 Motivation and Problem

In the early stages of computer technology, when a computer was a mass of tens of thousands of vacuum tubes, the failure of a component always meant that the computer would be inoperative for hours, and that the computation it was performing when the failure happened would have to be re-started from the beginning [Carter87]. Compared to that, the level of reliability offered by computer systems today is rather good. However, as our dependence on computer systems grew over the years, so did our reliability expectations. While in the early days computers were used only to perform more or less straightforward computations, nowadays they affect most facets of our life, from controlling our power supply distribution to performing virtually all our bank transactions.

More importantly, computer systems are now used in situations were a failure could

endanger human life. Even if today we have at our disposal better specification and verification tools to help in the design process, better hardware, and better programming environments in which to implement the intended system, we are still very much concerned about failures.

The term *fault-tolerance* involves a multitude of techniques, used in hardware and software, with the general goal of making the systems more dependable. The choice of which technique to use depends on the failure assumptions about the system and on the level of fault-tolerance intended. One may, for example, want to guarantee only the *progress* of the computation performed by the system, not taking into account the time it takes to complete that computation (that is, one is prepared to stop the computation while there is a failure in the system, as long as, when the failure disappears, the computation can be resumed from where it stopped). An example of a more stringent requirement for a system is to guarantee that the computation will never be stopped, regardless if there is or not a failure in the system. If we wish to fulfill this requirement independently of the number of failures present in the system at the same time, we would clearly need an infinite amount of resources. Thus, *absolute fault-tolerance* is impossible: what we can do is to design the fault-tolerance in the system so that the probability of the system failing is small enough for the particular application of that system.

Distributed systems seem to offer a good framework for solving to the fault-tolerance problem. Since the different components are physically separated, they present independent failure modes; that is, the failure of one component will not necessarily cause the failure of the whole system. However, this very physical separation of

components is at the same time the greatest obstacle to implementing fault-tolerant distributed systems, because the knowledge of the global system state is also distributed. Thus, the effect that a failed component will have in the system is in general difficult to predict and to withstand.

This research addresses the problem of designing and implementing distributed fault-tolerant systems. We want to investigate the requirements and nature of a set of appropriate tools for building fault-tolerant distributed systems from the systems programmer point of view. The main objective is to investigate how high level programming concepts and tools can be used to simplify the incorporation of fault-tolerance in distributed systems in a *selective* way, so that the systems programmer can choose the best technique to meet the requirements of a particular system. Specifically, this work provides a novel integration of services for selection at the interface presented to the system implementer by the programming environment. Although the results are general enough to be applied to other environments, Conic [Kramer87] has been used as a basis for this new programming environment. The ideas presented have also been tested by integrating the proposed set of tools into Conic, and by using the new programming environment (called Cronic, for reliable Conic) to implement some prototype fault-tolerant distributed systems.

Each tool in Cronic is based on a different mechanism, presenting a different effect on the fault-tolerance of the system, and also incurring a different penalty on the system's performance and size. In this way, the system implementer can choose the mechanisms that best meet the system requirements.

1.2 System Model

We view the physical system as being composed of a set of *processors*, each of which has an associated memory . The processors are interconnected by a *communication system*, through which they can send and receive messages.

Each processor executes one or more processes, or *tasks*. Tasks are the only active agents in the system, and they do not share any memory, communicating exclusively by exchanging messages. The tasks in a system will normally run in several different physical processors, and a set of tasks may run on a single, multi-programmed processor which will be called a *node*.

Each task defines a communication interface which consists of a set of *ports*, and messages are sent and received via the ports. In this way, tasks do not name directly their communicating partners, as in Ada [Ada80] and CSP [Hoare78]. Rather, communication is performed naming only the local port. A system is created, or *configured*, by instantiating task modules and *linking* the communicating ports in the desired pattern. This approach increases the system modularity, enables modules to be re-used, facilitates changes in the system configuration (only modules affected by the change need to be replaced), and also facilitates development and testing (modules can be tested and developed independently). More importantly, it leads to a software methodology that distinguishes between the *programming* of individual task modules from *configuring* those modules into a system.

1.3 Our Approach

What is proposed in this work is to extend the software methodology induced by the system model presented in the last section to incorporate the system's fault-tolerance capabilities. With this approach the user develops each functional module independently. Then, depending on the specific system reliability requirements, the user adds — where needed — the appropriate fault-tolerant mechanism. A system is therefore composed of modules with different degrees of fault-tolerance interacting with each other.

Our main interest is to construct a programming environment to help in the development of autonomous systems, which are systems where there is little opportunity for direct human intervention in detecting errors, locating faults, assessing the damage, and reconfiguring the system. In such systems intervention may be precluded throughout the system life (as in a satellite computer) or periodically for shorter periods (as in a factory controller during fully automated periods).

In contrast with more conventional systems, where the reliability concerns are time-to-failure and availability, autonomous systems must have the ability to perform without interruption and without human assistance over a period of use. In a conventional system such as a bank system, as long as the audit information is intact it is not crucially important if the system remains inoperative for a short period of time (say, some minutes), while recovery is attempted (possibly with the assistance of human intervention for reconfiguration, etc). By contrast, the control system of an aircraft must perform, even if at a degraded rate, with no interruptions while the aircraft is

airborne. This choice of application has affected the choice of mechanisms to be supported in Cronic.

We investigate the use of three common techniques to achieve fault-tolerance: backward recovery, forward recovery, and replication. *Backward recovery* consists basically in saving the state of the modules at certain stages of the computation so that they can be *rolled back* in case of failure. *Forward recovery*, on the other hand, uses the knowledge about the system state at the time of the failure to compensate for the failure without rolling back. *Replication* uses redundant resources in order to mask failures.

These techniques are made available to the system programmer by introducing a set of language constructs at the programming and configuration levels. At the configuration level, the basic technique used is replication, and the language constructs allow the specification of the service provided by replicas. Two types of replication are supported: active replication, where all replicas are executing concurrently, and passive replication, where only one of the replicas is executing while the others are kept in a standby state.

At the programming level, the basic techniques used are backward and forward recovery, and the language construct is similar to the FT-Action of [Jalote86], but this is extended to allow more modularity and flexibility. The main difference is that the set of tasks participating in an action in Cronic is determined dynamically instead of statically. An interesting result of the integration of services in Cronic is that passive replication (specified at the configuration level) can be used in conjunction with backward and forward recovery (specified at the programming level).

1.4 Thesis Overview

The remainder of this Chapter introduces some basic terminology (which is adopted from [Anderson83]) and concepts relevant to the work. It also presents an overview of the Conic system, which serves as a base for the prototype implementation of Cronic. Chapter 2 presents an overview of the basic techniques used in this work: backward recovery, forward recovery, and replication. It sets the background and discusses the problems that must be solved in supporting these techniques.

Chapter 3 introduces the language constructs designed for the programming language to support a distributed backward recovery. It presents the basic philosophy behind the design, and discusses the support needed for its implementation. Chapter 4 extends the programming language constructs to allow forward recovery to be used in conjunction with backward recovery. Chapter 5 presents our model of active and passive replication, and Chapter 6 presents the design of a configuration language supporting that model. The description of the mechanisms in Chapters 4, 5 and 6 concentrates on the philosophy of the design and on the operation of the mechanisms. Chapter 7 then describes the implementation issues. First, it presents the implementation of the basic services necessary at the distributed operating system level to support the language mechanisms presented in the previous chapters. Then it presents the implementation of the mechanisms using those basic services. Finally, Chapter 8 reports on the experience gained designing and experimenting with the system, presents our conclusions and ideas for future research.

1.5 Faults and fault-tolerance

A component *failure* occurs when, to some service request, it responds in a manner inconsistent with its specification. A component specification prescribes the state transition that a system component, such as a task or a communication link, should undergo in response to a service request. The problem of producing a correct and complete specification is not a trivial one, but it is out of the scope of this work; we will simply assume that such specification is available.

An *error* is the part of the system state that can lead to a failure. A *fault* is the algorithmic cause of an error. Notice that a failure is defined as an event that happens at the moment a deviation in the expected behaviour is noticed. The failure event occurs when an erroneous state of a component is made visible. That is, the component may have been in an erroneous state for some time, but this can only be recognized as a failure when it is eventually noticed. There is a causal relationship between fault, error, and failure: a fault (e.g., a mistake by the programmer) causes an error (i.e., a component in an erroneous state) which may cause a failure (erroneous output delivered).

The word *reliability* has been used by the computing community to mean two different concepts: a quantitative measure of the quality of the service provided by a system, and a general term to describe the approaches and techniques used to ensure that a system will operate according to its specifications. In the former sense, it is normally characterized by a function that describes the probability that a failure will have occurred during a certain period of the system operation, and it is used to estimate the

system's mean time between failures (MTBF). This reliability measure does not capture all interesting failure-related properties. For example, considering a failure has occurred, another measure, the maintainability (or repairability) is be used to express the time the service is interrupted. Also, another measure is usually defined to express the ratio (service provision / service interruption). Although these measures are not independent, each of them quantifies a unique aspect of the system behaviour, and none of them can be taken alone as a measure of a system's usefulness.

In this work the word reliability will be used in the more general sense of expressing the guarantee that the system will behave as specified. In order to achieve reliability two complementary approaches are normally used: fault-prevention and fault-tolerance.

Fault-prevention, used in the development phase of the system, attempts to ensure that no faults are present in the system. Fault-prevention can be further divided into two phases: *fault-avoidance*, in which techniques to minimize the introduction of faults are used during the system design, and *fault-removal*, in which techniques to eliminate any remaining faults are used. Examples of fault-avoidance techniques are tools and methodologies for system specification and design, tools for analysis of system specifications, etc. Quality control, and validation and testing procedures are examples of fault-removal techniques.

Fault-tolerance, used in the operational phase of the system, uses a diversity of techniques that attempt to prevent that any faults still present in the system cause a system failure. All techniques used to achieve fault-tolerance rely on some form of redundancy. Therefore, a fault-tolerant system will always use some extra resources

that would not be required if the system components were free from faults. The extra resources in a fault-tolerant system may be placed either in the hardware (e.g., in the form of extra bits used for a error-correcting/detecting code in a memory module), or in the software (e.g., two instances of the same software module, whose results are combined, running at two different locations).

1.6 Failure assumptions

As explained in section 1.2, at the hardware level we consider systems composed by processors and links. We assume that the processors and links have extensive error-correcting and error-detecting redundancy in hardware, so that they can be considered, for our purposes, to be fail-stop. That is, processors and links fail by stopping. We further consider that the executive running at each processor is free of design and implementation faults, and can be considered part of the processor 'firmware'.

The software model is mapped on top of this hardware model. Application tasks run on processors and communicate using the physical links. Collections of tasks running on a single processor are called nodes.

The failure assumptions on the hardware level affect the software level. For example, a processor that fails by stopping will cause all tasks running at that node to fail by stopping. But at the software level we want to consider also another class of faults for individual tasks, that is generally called design or implementation faults. As the name indicates, these are faults introduced at the design phase (the task design did not meet the specification) or at the implementation phase (the implementation did not implement correctly a design). Most systems assume that individual tasks are also fail-stop.

Although it may seem too restrictive, this is in fact adequate for most of the applications running in a friendly environment (such as a shop floor, where the system is controlling some industrial process) so that no malicious behaviour occur. However, once the fail-stop restriction is lifted, tasks may exhibit *any* behaviour, and it becomes difficult to define what is malicious and what it is not malicious. The problem with allowing tasks to exhibit malicious behaviour is that all the algorithms to maintain the system consistent have a much higher overhead, affecting the system performance. In Cronic we have decided for a compromise: individual tasks are not fail-stop, but since the executive at each node is fail-stop (as we assume it is part of the firmware) and it is the executive that is responsible for performing the consistency algorithms (like the commit protocol for actions, as explained in chapter 3), the damage caused by a task is restricted. It is also implicit in this assumption that the system will run in a friendly environment; in particular, user tasks will not try to emulate the executive. If that were the case, authentication algorithms would have to be introduced at the executive level.

Summarizing the failure assumptions: the executive is considered part of the processor firmware. Processors fail by stopping, and when they do so they cause all user tasks running on that node to fail by stopping. Individual user tasks may fail and continue running, but it is assumed that the executive will eventually detect the failure so that the mechanisms available in Cronic can be used to restore the system to a consistent state.

1.7 Conic System: overview

The set of tools presented in this work were experimentally developed on top of an existing software environment, Conic, which also uses the same system model of tasks and messages. Conic provides the ideal vehicle for experimenting with our ideas,

because it adopts the software methodology described in section 1.2, with a language for programming the individual modules (programming-in-the-small) and a language for configuring those modules into a system (programming-in-the-large). In this section we present an overview of the Conic system, highlighting the implementation aspects that will be useful when discussing the implementation of the fault-tolerant mechanisms proposed in the next chapters.

1.7.1 Conic Programming Language

The Conic programming language has a Pascal-like syntax, with extensions to support modularity and message passing. A *task* is the basic programming unit. Tasks define a message passing interface, composed of (typed) entryports and exitports. There are two message passing transactions defined: notify messages, which is non-blocking (that is, the sender does not wait for the message to be received), and request-reply messages, which is blocking (the sender stops processing, waiting for a response from the receiver). A small example of a task definition in Conic, with the respective graphical representation, is shown in Fig. 1.1. The example implements a simulation of a water pump which pumps water from the port *water_in* and delivers it to the port *water_out*.

```
task module pump (capacity, rate: natural);
use waterDefinitions;
entryport
    on_off: signaltype reply signaltype;
exitport
    water_in: water reply water;
    water_out: water;
var
    on: boolean; bucket: water;
begin
    receive signal from on_off reply  signal;
    on:= true;
    loop
        select
            receive signal from on_off =>
                on:= not on;
                reply  signal;
            when on
            timeout rate =>
                send capacity to water_in wait bucket;
                send bucket to water_out;
        end;
    end;
end.
```

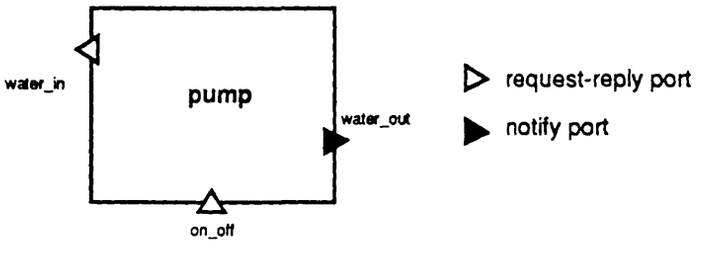


Fig. 1.1. Graphical representation of the task module pump

1.7.2 Conic Configuration Language

A system in Conic consists of a collection of modules instances, described by a configuration specification, which identifies the module types used in the system, declares the instances of those modules, and describes the interconnection between their exitports and entryports.

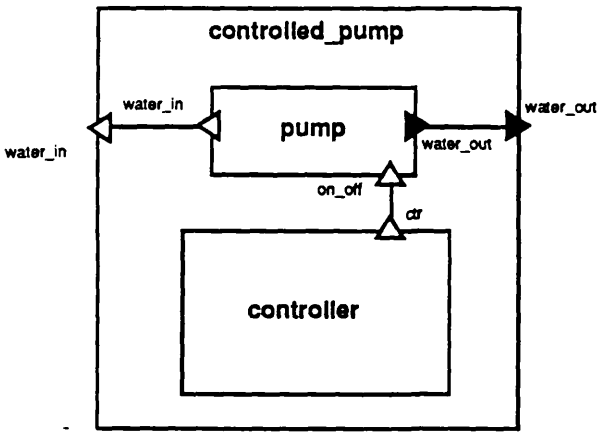


Fig. 1.2 Example of a Conic configuration specification

The module types used in a configuration specification may be either single task modules declared using the programming language or collections of modules (called group modules), declared using the configuration language, allowing a system to be defined as a hierarchical composition of modules. An example of a system specification in the Conic configuration language is given below, and a correspondent graphical description is given in Fig 1.2.

```
group module controlled_pump;
use
    waterDefinitions;
exitport
    water_out: water;
    water_in: water reply water;
use
    pump; controller;
create
    pump(5,4);   controller;
link
    pump.water_out to water_out;
    pump.water_in to water_in;
    controller.ctr to pump.on_off
end.
```

Systems in Conic may be implemented as non-distributed (on a single processor), or distributed over a network of processors, or physical nodes. Instances of modules are assigned to specific physical nodes by qualifying the **create** construct with an **at** *node_name* clause. The location of a module can only be specified at the highest level of the configuration specification. A typical Conic system specification, with logical modules being assigned to physical nodes, is given below, and its graphical representation is shown in Fig. 1.3.

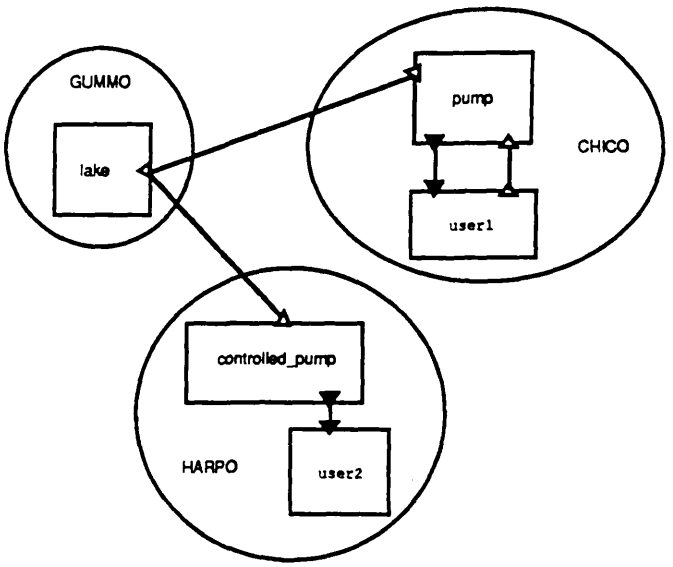


Fig. 1.3. An example of a Conic system with three nodes

```
group module water_system;
use
    pump; lake; user;
    controlled_pump;
create
    lake at gummo;
    pump(2,1) at chico;
    user1: user at chico;
    user2: user at harpo;
    controlled_pump at harpo;
link
    controlled_pump.water_in,
    pump.water_in to lake.water_out;
    pump.water_out to user1.water_in;
    user1.ctr to pump.on_off;
    controlled_pump_water_out to user2.water_in;
end.
```

1.8 Summary

The main motivation for this work was the inadequacy of existing programming environments to cater for different fault-tolerant techniques that could be used selectively according to a particular system specification. Distributed systems present an attractive platform to implement fault-tolerant systems. However, while distribution introduces the benefit of independent failure mode of the system components, it also introduces problems due to the fact that the knowledge of the system state is distributed among the components. Thus, inconsistencies may arise in the view that each component has of the system state.

In this Chapter we presented the system model adopted, which makes a distinction between "programming in the small" and "programming in the large", and our failure assumptions. The basic processing element in our system model is the task, or process, and the task functionality is specified with a programming language (this is the "programming in the small" phase of the design). Tasks do not share memory, and communicate exclusively by exchanging messages. The communication is not done naming the communicant explicitly; instead, local ports are used. In the "programming in the large" phase, tasks are instantiated and their ports are connected to other tasks' ports. This modular approach greatly increases the system flexibility, allowing components to be stored in libraries to be re-used in different parts of the system, or even in other systems.

This Chapter also presented an overview of the Conic programming environment, which uses the same system model. Conic is used as a basis for the implementation of the new programming environment incorporating fault-tolerant mechanisms, which we

call Cronic. Cronic extends the software methodology based in the programming in the small – programming in the large to incorporate the system fault-tolerant mechanisms. The next Chapter reviews the fault-tolerant techniques available in Cronic.

Chapter Two

Basic Techniques for Fault-tolerance

2.1 Introduction

The physical separation of components, which is one of the primary reasons for using distributed systems to enhance the system reliability, is at the same time the greatest obstacle in producing a reliable distributed system. To assess the damage caused by a failure, and to effect the recovery, it is necessary to know the state of the system at the time of the failure. In a centralized system, knowledge about the whole system state is available instantaneously, and this information can be used when assessing the damage and recovering from a failure. When a failure occurs, an instantaneous 'snapshot' of the system state can be produced by simply stopping all computation and looking at the state of every component. In a distributed system, however, no single processor knows the instantaneous global system state. Inconsistencies between local views of the global

state arise from the physical separation of components and are due to random communication delays and component failures. This chapter presents a review of the basic techniques for fault-tolerance available in Cronic to keep the system state consistent in the presence of failures. All techniques for fault-tolerance are based on some kind of redundancy. We can divide the redundancy in two (sometimes overlapping) classes: resource redundancy, and functional redundancy.

Resource redundancy, as the name implies, is the type where an extra amount of resources is used. In Cronic, extra software modules will be used to provide fault-tolerance, as discussed in section 2.4. *Functional* redundancy is the type of redundancy that is *included* in the functionality of the module. This type of redundancy is discussed in sections 2.2 and 2.3, which review the techniques known as backward recovery and forward recovery. Functional redundancy mechanisms map naturally into the programming level, since they are associated to *how* the function is provided. Resource redundancy maps naturally into the configuration level, since it is concerned with *where* the redundancy is used.

2.2 Backward Recovery

A well-known and widely used technique to implement functional redundancy in software is *checkpointing and rollback recovery*: at certain points in the execution the tasks save their state into a reliable storage (a memory device that survives crashes). If something goes wrong at a later point in the execution, the tasks *roll back* to their checkpoint and start executing again. Several problems arise when implementing this mechanism in distributed systems, because rolling back can make the system *global state* inconsistent. The global state of a system is defined as the set of local states, one

from each task. In our system model, in which all interaction between tasks is achieved by message passing, this notion of global consistency can be translated into "the receipt of a message cannot precede the sending of that message under any temporal frame of reference". When a task rolls back, the system could be left in an inconsistent state unless some other relevant tasks are rolled back as well.

Consider the diagram in Figure 2.1, where horizontal lines are time axes of tasks and the sloping arrows represent messages exchanged. A *cut* in this timing diagram represents a "snapshot" of the system global state, and it is *consistent* if no arrow starts at the right hand-side of the cut and ends at its left hand-side (since this would mean a task received a message not yet sent). If every task takes a checkpoint where the cut crosses its time-line, the set of checkpoints formed will be consistent if the original cut was consistent. Therefore, the first problem we have in implementing backward recovery is to produce a consistent set of checkpoints.

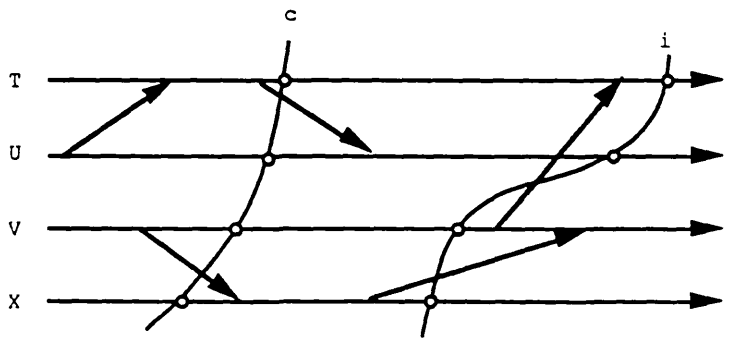


Fig. 2.1. Consistent and inconsistent cuts

There are two basic approaches to determine when the tasks should take a checkpoint. In the first, tasks take checkpoints independently and save all of them in reliable

storage. Upon a failure, the system must find a set (preferably the most recent one) of checkpoints that provide a consistent global state from all saved checkpoints. The second approach is to coordinate the checkpointing among the tasks so that each task needs to save only the most recent checkpoint. One disadvantage of this second approach is that the system must spend some time performing the coordination every time a checkpoint needs to be taken. In a sense, the first approach is more "optimistic"; however, the second approach does prevents a catastrophic situation that cannot be prevented by the first one: the so called *domino effect* [Randell75], illustrated in Fig. 2.2.

In this example, tasks T and R have taken, independently, checkpoints T_i and R_i. When T crashes, the only consistent cut in the diagram is $[T_0, R_0]$, that is, both tasks will have to roll back to their initial states, because any other cut would imply that a task received a message that hadn't been sent yet. That is, a receiver would have the receipt of the message in its checkpointed state, but the sender wouldn't have the sending of the corresponding message in its chekpointed state.

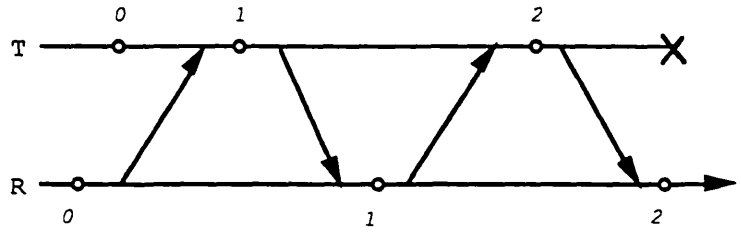


Fig. 2.2. The domino effect.

Therefore, although the approach of taking checkpoints independently is more attractive if failures are assumed to be rare, since it does not incur the overhead of synchronizing

the checkpoints, any scheme using this approach must find a way to alleviate the domino effect. A related problem that must be solved is that, since tasks take checkpoints independently, they may have to keep several consecutive checkpoints in reliable storage; at some point the system must discard the older checkpoints that are not needed any more.

2.2.1 Language constructs for backward recovery

Several proposals for language constructs which allow a programmer to program backward recovery have appeared in the literature. The first one appears to have been [Lomet77]; the better known is the "recovery block" proposed by Randell [Randell75]. The recovery block is composed of one or more *try* blocks designed to produce the same, or equivalent, computational results. At the start of a recovery block the process is checkpointed, creating a *recovery point*, and a try block is chosen for execution. Each recovery block ends with an acceptance test, which is a criterion for determining the acceptability of the execution of the try block execution. If the acceptance test fails, the process reverts to its recovery point and another try block is chosen for execution.

Randell also proposed a mechanism called *conversation*, in which a group of processes using recovery blocks communicate with each other. At the end of their communication (that is, at the end of the conversation), they each wait for the others to arrive at their acceptance tests. If they all pass the acceptance test, they commit to the information exchange that has occurred (by taking a new checkpoint, and discarding the previous one), and proceed. Should any of the processes involved fail the acceptance test, they all restore their states to their recovery points. In a conversation, a process is not allowed to *smuggle* information in or out by communicating with processes not

participating in the conversation.

The processes in a conversation take checkpoints independently, when they start the execution of their recovery blocks. But this, as explained in the last section, may lead to a domino effect in case of failure. How does the conversation overcome this problem? The answer is that the acceptance test acts as a synchronization mechanism which limits the amount of rollback. The domino effect would only be possible if a process were allowed to proceed independently after it had passed its acceptance test, without waiting for the results of the acceptance tests from the other participating processes. Moreover, the commit of a conversation is also a synchronization point for checkpoints, since, if the global result of the conversation is to be persistent, the processes must be checkpointed at the commit point.

2.2.2 Commit Protocol

The commitment of a conversation is achieved using a commit protocol. The two-phase commit protocol [Gray79], originally designed for database transactions, is probably the most used commit protocol. When used to commit conversations, it can be described briefly as following

> *Phase 1*: One of the tasks participating in the conversation is designated the coordinator. The coordinator, after succeeding in its acceptance test, takes a tentative checkpoint and sends a prepare to commit message to the other tasks in the conversation. Each task that passes the acceptance test takes a tentative checkpoint and responds with accept. If a task fails its acceptance test it responds reject.

Phase 2: If all tasks accept the conversation, the coordinator sends a commit message to all tasks; otherwise, it sends an abort. When a task receives a commit message, it makes its tentative checkpoint permanent.

The major drawback of the simple two-phase protocol is that it is blocking, in the sense that if the coordinator fails, the protocol may be blocked until the coordinator recovers, even if the coordinator had sent its decision to some of the tasks. Non-blocking commit protocols are known, but they are more expensive in time and number of messages, and are much harder to implement. Essentially, what non-blocking commit protocols implement is a failure detection mechanism and a reliable way of disseminating the coordinator decision to other tasks, so that depending on when the failure occurs, the other members may still know about the decision without waiting for the coordinator to recover.

2.3 Forward Recovery

Forward recovery is the name generally given to the technique of recovering the system without rolling back. Backward recovery aims to bring the system to a consistent state by returning it to a *previous*, consistent state. Forward recovery, on the other hand, aims to attain consistency by taking the system from the erroneous state to a *new* consistent state (new in the sense that it has not been reached before). Backward recovery can be seen as a pessimistic form of fault-tolerance. The existence of checkpoints guarantee that the system can be returned to a known, safe state. But there is a cost involved in this security, both in taking the checkpoints and in redoing the work lost since the last checkpoint. Depending on the type of the failure, and on the

moment it occurred, it may be possible to "fix" the system state without having to pay the penalty of rolling back. That is the purpose of forward recovery.

If a program (or part of a program) is thought of as function which takes a set of input states to a set of exit states, an *exception* occurs when the input state presented to the program does not produce an specified exit state. Notice that with this view there is an interval of time since the exception occurs (program invoked) until the exception is detected (program realizes the impossibility to deliver normal service). If the domain of the program (and the domain of its specification) included all the possible input states, there would be no exception occurrence when the program was invoked. However, in practice it is very rare to encounter such programs and specifications; most often, the domain will not include all possible input states, and there will be an *exceptional* domain associated with the *standard* domain.

Exceptions and exception handlers are the most common language constructs to provide forward recovery. An exception is the realization that it will be impossible to deliver the standard service specified by a program. An exception mechanism is a language construct which allows the programmer to specify that the normal sequential continuation of the program is to be replaced by an exceptional continuation (described by an *exception handler*) when an exception is detected.

Exceptions and exception handlers are also a program structuring issue. The division between exceptional and standard domains is reflected in the program structure, with code specifying normal operation being kept separate from code specifying exceptional operation. An exception mechanism is somewhat similar to an interrupt mechanism, in the sense that exceptions can be seen as asynchronous events. If it wasn't for the ability

to handle exceptions asynchronously, the programmer would have to test for the erroneous conditions in code interspersed with the code dealing with normal system service. Thus, by using exceptions and exception handlers the code becomes better structured, with code specifying normal service being kept separate from code specifying the behaviour in exceptional service.

Two issues must be addressed when designing an exception mechanism: the method of associating exceptions and handlers with segments of normal control flow, and what happens after the handler is executed.

An exception handler can be associated with an specific segment of code either statically or dynamically. This association is said to be static when the position of the handler declaration determines the segment of code it covers. This is the approach taken by Ada [Ada80] and Argus [Liskov83], for example. Alternatively, a handler can be specified dynamically by the programmer (a simple example would be to pass a handler as a parameter when calling a procedure or function), but this form of association is now generally considered undesirable [Taylor86].

There are two approaches to what should happen after an exception handler is executed. In the first, called the resuming model, when an exception is detected the handler is invoked and, when it finishes executing, the code which was interrupted by the exception is resumed. In this model the handler is responsible for removing the cause of the exception so that the normal continuation of the program can proceed.

In the terminating model, the normal continuation of the unit of code covered by the handler is totally replaced by an exceptional continuation once an exception is detected.

Once an exception is detected, the normal code for that unit is never resumed. This can also be seen as a unit having several exit points: one normal exit point, reached by normal execution, and zero or more exceptional exit points, reached by executing the associated exception handlers.

One problem with the resuming approach is that the programmer is left with the responsibility to make the exception handler bring the system to a state that is not only consistent, but also in a state suitable to resume the operation it was carrying out when the exception was detected. That is, the handler dealing with the exception becomes more dependent on where exactly within the handling context the exception happened, since it has to provide a more specific service. This, in turn, affects the trade-off that exists between the generality of the handlers and the number of exception handlers defined in a program.

The use of exceptions and exception handlers is well understood in non-distributed systems. In distributed systems several new issues must be addressed. For example, what is the handling context in a distributed computation? Most languages for distributed systems provide some kind of distributed exception, but they are in general restricted to one communication command. Conic, for example, provides a **fail** clause in request-reply transactions which is executed if the transaction cannot be completed. Ada provides for a more general distributed exception mechanism, with an exception being propagated to both the calling task and the called task if an exception is raised inside an accept statement and it is not handled there (that is, the handling context in this case includes both tasks). Even so, the Ada mechanism is restricted to the two communicating tasks. In chapter 4 we present a more general distributed exception

mechanism, which involves a set of communicating tasks. Since more than one exception may arise concurrently in a set of communicating tasks, exceptions have to be disseminated to (and voted by) the set of tasks by using a protocol similar to the commit protocol discussed in the previous section.

2.4 Replication

Replication has been proposed as a useful technique to mask failures of individual components since the first attempts to provide fault-tolerance in computers [von Neumann56], and computer hardware designers have long been familiar with techniques such as triple-modular redundancy [Lyons62]. Redundancy can also be employed at the software: an early application of redundancy in software is the SIFT aircraft control system [Wensley78]. More recently, the Circus system [Cooper85], and proposals such as N-version programming [Tso86], hot and cold standby software modules [Loques86] have also been advocating the use of replication at the software level.

The backward and forward recovery mechanisms described in the previous sections provide for a structured way to deal with failures from within the component. The philosophy behind replication is a different one: software components need not be aware of failures, because failures are masked out. An individual component does not need to take failure into account because, by replicating that component, failures will not occur (or rather, failures of an individual component instance will be masked out, and will not appear as a failure to the system). The implicit assumption in replication is that enough instances of the replicated component will survive to allow the system to

carry on normally. By replicating modules that execute important services (or even replicating *all* modules), the user is presented with a practically failure-free system. Because it is added to a system at a different implementation phase, the system reliability becomes an issue totally orthogonal to the system functionality. When designing and implementing an application, the user can ignore components failures and concentrate on its functionality. At a later stage, reliability can be incorporated into the application by replicating specific modules.

Because it allows the reliability of the system to be treated separately from its functionality, the replication technique offers an advantage from the programming methodology point of view in our system model: the reliability of a software component can be specified and introduced at the configuration stage, when the individual software components are being configured into a system.

Since our architecture model already makes use of a configuration phase, we can use that phase to specify the intended reliability of *individual* software components. Different parts of the system, with different reliability requirements, can use a different degree of replication. In Chapter 5 we present a configuration language that allows for a varied choice of services related to replication.

2.4.1 Types of replication

There are mainly two types of software replication, that are generally called active and passive replication. In *active* replication, all replicas are actively executing, and any output from the replicas is taken and combined into one single output (by voting, for example) before being used. In the *passive* form only one replica is actually executing,

and the other replicas are waiting to be activated in case the active one fails. Active replication provides a non-stop service (while there is at least one replica working) at the expense of computing resources. Passive replication uses potentially less resources, but the service is temporarily disabled while a standby replica takes over after a failure of the primary. The time it takes from the moment of the failure being detected to the moment the new active replica is ready to execute and the system resumes its normal operation is called the *switch-over* time.

2.4.2 Overview of problems to solve in replicated systems

The problems that must be solved in a replicated distributed system arise from the very fact that the replicas are distributed physically. For active replication, one problem is to synchronize correctly the operation of the replicas. This synchronization depends basically on the correct delivery of messages to and from replicas. For example, a message sent to a component that is replicated must be received by all replicas. Furthermore, all replicas must receive all messages in the same order. Another problem that must be addressed in active replication is the re-integration of replicas into a working system.

The most important issue in passive replication is the strategy to maintain the standby replicas informed on the work done by the primary, so that if the primary fails a standby can resume the operation with a minimum of disruption. To maintain the standby replicas informed on how far the primary has gone in its work, the primary sends its state to the standby replicas from time to time. In other words, the primary uses the standby replicas as a safe repository for some previous state. In [Loques86], a set of rules is derived to determine the points in the computation where a primary must

save its state to guarantee that the system will be able to recover appropriately. Depending on the communication pattern of the computation, this can be quite expensive, but in the paper they show that the set of rules presented defines the minimum number of saves without taking into account the semantics of the computation. However, if we do have information about the computation being performed, we can use much fewer save points. That is the approach taken in Cronic: in order to provide an efficient passive replication mechanism, it uses some semantic information. This is done by combining passive replication, which is basically a resource redundancy mechanism, with the functional redundancy mechanism in the form of the backward recovery.

2.5 Integration of fault-tolerance mechanisms in Cronic

In this section we present an overview of the integration of these techniques (passive replication, backward recovery and forward recovery) in a programming environment. We first re-state the basic characteristics of each technique, and the support needed to implement them:

• *Backward recovery* rolls back a set of communicating tasks to points where the system state was known to be consistent. It needs a failure detection mechanism and a mechanism that implements a reliable storage, where tasks can save their state. Tasks commit to a set of checkpoints explicitly, using a commit protocol.

• *Forward recovery* uses the information about the system state at the time it discovers any inconsistency to bring the system to a consistent state, without rolling back. In a distributed environment it needs a protocol, similar to a commit protocol, to inform all

concerned tasks of the occurrence of the exception.

• *Active replication* masks failures by providing redundant results from a replicated module. The most important support is a reliable communication system, with an appropriate naming capability, that ensures that messages reach all replicas in the same order.

• *Passive replication* implements a form of backward recovery. The support needed is an efficient scheme to determine when the primary replica must save its state to the standbys. The standbys serve as a safe repository to the primary state.

It is interesting to notice that the backward recovery mechanism at the programming level needs an implementation of a reliable storage, which can be provided by passive replication (or at least a virtual reliable storage can be provided). On the other hand, passive replication can be implemented more efficiently if some semantic information about the activities performed by the replicated module is available, which is possible if the modules are implemented using the backward recovery language construct at the programming level. Therefore, we see that backward recovery at the task level complements naturally passive replication, and this presents one aspect of the integration of fault-tolerance mechanisms in Cronic. Passive replication is used in Cronic to implement a *virtual* reliable storage, with the added benefit of increased availability over the disk based implementation of reliable storage.

Another aspect is based on the realization that forward recovery can be used in conjunction with backward recovery, so that backward recovery is used as a last recovery option. Forward recovery is used on a first instance, as a way to try to avoid

rolling back. Only if forward recovery fails will the backward recovery mechanism be used.

2.6 Summary

This Chapter introduces the basic concepts needed for this work. It reviews three techniques used for achieving fault-tolerance in distributed systems: backward recovery, forward recovery, and replication. The main objective of these techniques is to keep the system in a consistent state after the failure of some components. Backward recovery keeps consistency by checkpointing the component states into reliable storage and rolling back the affected components to those saved states in the case of failure. Forward recovery uses the information about the system state at the time of the failure to take the system to a consistent state without rolling back. Forward and backward recovery are mechanisms that can be classified as *functional* redundancy, because the component redundancy is tightly coupled with the component function.

Replication of components is an example of another type of redundancy, which we call resource redundancy. Passive replication, in the assumption that enough replicas of each component will survive to guarantee the functionality of the system in case of failures.

The next two Chapters present the design of a programming language construct to implement functional redundancy, allowing an integrated use of backward and forward recovery. Chapter 5 will present the design of the replication mechanisms and the configuration language constructs available in Cronic to provide resource redundancy.

Chapter Three

Fault-Tolerance at the Programming Language Level: Actions

3.1 Introduction

In the next two chapters we present a mechanism that allows the integrated use of backward and forward recovery. This chapter introduces the mechanism as a backward recovery tool; in the next chapter the mechanism is extended to allow forward recovery. The mechanism is based on the conversation scheme discussed in section 2.2 with extensions to include forward recovery, and is similar to the FT-Action proposed in [Jalote86]. The mechanism is available in Cronic as a programming language construct.

The main design principles observed when designing the mechanism (other than the standard ones of being simple, practical, and incurring as low overhead as possible)

were:

1) Independent recovery design. The recovery capabilities of each task should be structured and developed independently of other tasks. Allowing independent design of the recovery capabilities encourages modularity, so that systems may be designed with as little coupling among different modules as possible. This enables the creation of libraries of modules which can be re-used in different systems without any changes.

2) Transfer of uncommitted messages. Unlike [Strom85], in which individual outgoing messages are not sent until they are deemed "safe", tasks are allowed to send to other tasks information that is revocable. Not sending the messages until they are not revocable any more can be seen as a "pessimistic" approach, incurring an overhead independent of failures happening or not. We adopt a more optimistic view, in the assumption that failures in the system are the exception, not the rule.

3) Prohibition of suspicion. Each task is responsible for detecting and correcting errors that it originated. Therefore, a task cannot be suspicious of imported information (although it can be told by the exporting task that the information is erroneous). If a task were permitted to suspect that another task exported faulty information it is not possible to prevent the domino effect [Kim86].

4) Provide for backward and forward recovery. It should be possible to use forward recovery in conjunction with backward recovery. Backward recovery provides a safe recovery option (since there is a consistent state where the system can re-start from), but it is in general more costly than forward recovery (which does not guarantee that the system will eventually achieve a new consistent state). Since both have advantages and

disadvantages, the best option is to use then in an integrated manner.

3.1.1 FT-Actions

In [Jalote86], a scheme for using backward and forward recovery in a complementary way was described. The main mechanism, called *FT-action* , was based on Randell's conversation, and was designed for CSP [Hoare78].

The processes participating in an FT-Action name explicitly the other participating processes, and the system checks that all participating processes declare the same set of participating processes. An example of the proposed syntax for backward recovery in FT_Actions is given below (from [Jalote86]).

```
A: FT_Action with (P1, P2, ...., Pn)
...
P1 :: [    ...
        FT_Action A
            ensure <acceptance test>
            by <primary>
            else by <alternate>
            ...
            else by <alternate>
            else signal error
        end
    ... ]
```

As in a conversation, there are primary and alternate try blocks, and an acceptance test that must be passed before committing the FT_Action.

3.1.2 Problems with FT_Actions

When a process enters an FT-Action it stays there, maybe through different try blocks, until the computation is completed successfully (or the alternates are exhausted), and the same group of processes must be used in each of the try blocks. But, ideally, if a process fails in an attempt to achieve a common goal with one group of processes, it should be allowed to attempt the same goal, or even a different goal (such as some form of degraded service), with another set of processes.

Another problem is that a process may want to use a different number of alternative try-blocks. The fact that processes in an FT_Action must have the same number of alternates, together with explicit naming of participating tasks, increases the module *coupling* of the system: the knowledge of other processes behaviour when implementing one particular process is increased. Also, if a modification of the system structure is necessary, so that the set of participating tasks is changed, all processes in the FT-Action must be recompiled.

In the FT-Action, independence of algorithms among the different alternates is hampered by the fact that the same acceptance test must be passed, regardless of which algorithm is being executed. After being individually rolled back, the same set of processes attempts to communicate again, and eventually reach the same acceptance test. A single test must be general enough to pass results produced by the most degraded algorithm, but this might be too general to detect errors produced by other, stricter algorithms. These considerations indicate that, instead of being fixed for all alternatives, the acceptance test should be left to be specified by the alternative

algorithm itself.

3.2 Actions

The recovery capabilities of each task in Cronic are structured with the aid of actions. An *action* is a logical encapsulation of work which defines a recoverable region of a task. A task creates a new action every time it enters an *action statement*:

```
action statement ::= action by <primary block>
                     [ else by <alternate block> ]
```

Upon starting a new action, the task is checkpointed and begins executing the action block. A task tries to *commit*, or validate, the results of the action automatically when it reaches the end of the primary block; if it succeeds committing, it takes another checkpoint to make the results permanent. If the action cannot complete execution successfully, the action *aborts*: the task is rolled back and executes the (optional) alternate block. Actions can be aborted by the system (due for example to a run time error while executing the action, or failure of the processor where the task is executing) or explicitly by the user, by raising the exception *abort* (*exceptions* are discussed in the next chapter). If the action runs out of alternates, the task is rolled back to the checkpoint taken at the beginning of the action, and the standard exception *action_aborted* is raised. It is important to stress that actions are *logical* activities: the real active entities are still the tasks that are executing them.

When an action A imports information from another action B (that is, when the task executing action A receives a message from the task executing action B), action A becomes *dependent* on action B. If after their communication B aborts, A must be

aborted as well, since it used information that may be no longer valid. However, the abortion of action A does not cause the abortion of action B. If two actions are reciprocally dependent they can be considered as a single action, since they will either commit or abort together. In this case we can think of the two actions as being *merged* into a single action. Notice that a set of reciprocally dependent actions is similar to an FT_Action, the difference being that in Cronic the participating tasks are determined dynamically.

3.2.1 Nesting

Actions may be nested, but since actions are only logical activities, as explained above, an enclosing action is "suspended" until its nested action terminates. In other words, a task can have only one "active" action at a time - the innermost; all the others are suspended. As is usual in nested atomic actions ([Moss81],[Alchin83]), the commitment of a nested action is relative to the commitment of the outermost action. However, the abortion of a nested action does **not** cause the automatic abortion of the enclosing action. Rather, it causes the standard exception *action_aborted* to be raised in the nesting action:

```
action by begin
    ...
    action by begin
        ...                -- try this alternate
    end
    else by begin
        ...                -- else try this other
    end;                       -- if the action fails, action_aborted is
                               -- raised in the enclosing action
    ...
end;
```

3.2.2 Participating tasks

As explained before, the participating tasks in an action are determined dynamically, according to the dependencies formed as a result of the communication pattern. To keep track of this asymmetrical dependency information, each action maintains a set of actions which it depends upon, called the action's *contingency set*. Actions are identified by an unique *action-id*, assigned when the action is created. The action-id of an action includes the identifier of the task (the task-id) running the action. When an action is created, its contingency set is initialized to contain its own action-id, indicating the obvious fact that the action commitment depends on the action itself. When an action imports information from another action by receiving a message, it "inherits" all the dependencies of the exporter action. That is, the importer action becomes dependent on all the actions in the exporter's contingency set (which includes the exporter action itself). The contingency sets are used to determine the participating tasks in the action, as it will be explained ahead.

Each action also keeps information about the actions which became dependent on it, by

maintaining a set called the *dependency* set. Dependency sets are used mainly to propagate exceptions, as explained in chapter 4, and to purge old system information (section 3.2.7).

Maintaining the dependency set is not as straightforward as maintaining the contingency set. When an action receives a message from another action, the sender action-id is included in the message, and the receiver simply includes it in its contingency set. When an action sends a message, however, it may not know the action-id of the receiver action (in fact, the sender may not even know if the receiver is running an action), and therefore cannot include the receiver action identity into its dependency set. Instead, it includes a partial entry, containing only the receiver task identifier. From then on, when it receives a message from another action, it checks whether the message is from some task in that partial entry (which it can do because, as explained before, the action identifier includes the task identifier); if it is, it completes the entry. Notice that a partial entry in the dependency set may still be unresolved when the action reaches its commit point, if the action does not receive any message from the task included in the dependency set as a partial entry (in other words, if the actions don't become mutually dependent).

3.2.3 Inherited dependencies

If the contingency set of the exporter action grows after the import of information took place, the importer action must become dependent on the additional actions in the exporter contingency set even if it does not import any further information. In other words, a dependent action *inherits* even the dependencies the exporting action acquired

after the communication took place. Fig.3.1 illustrates this situation (dotted lines indicate the span of each action).

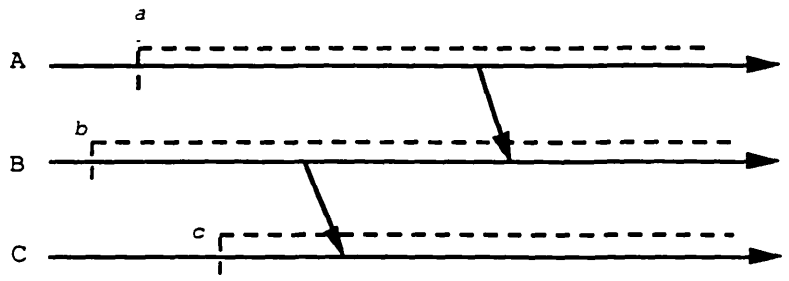


Fig. 3.1 Inherited dependencies

Action *c* becomes dependent on *b* when it receives a message from *b*. When *b* becomes dependent on *a*, *c* inherits this dependency even if it does not communicate with either a or b anymore.

A special case of inherited dependencies occurs when a cycle is formed in the dependencies, making all the tasks in the cycle reciprocally dependent on each other. Consider the situation depicted in Fig. 3.2. At first we have three actions, *a*, *b* and *c*, with contingency sets cg_set_a, cg_set_b, cg_set_c, respectively. When *b* imports information from *a*, *b* becomes dependent on *a*, causing $cg_set_b=\{b,a\}$. Action *c* then receives a message from *b*, which makes *c* dependent on both *b* and *a*, that is, $cg_set_c=\{b,a,c\}$. When finally action *a* receives a message from *c*, a cycle in the dependencies is created, with $cg_set_a = cg_set_b = cg_set_c = \{a,b,c\}$.

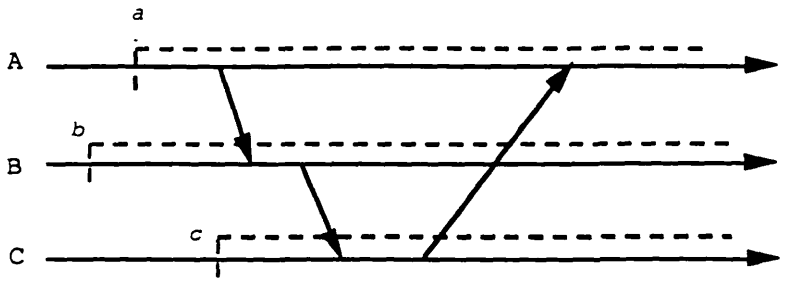


Fig. 3.2. Cyclic dependencies

In this case, the three actions may be thought of as having been merged into one single action. They will commit or abort together.

3.2.4 Inherited dependencies and nesting

Nesting complicates the management of dependencies among actions; consider, for example, the scenario of Fig. 3.3.

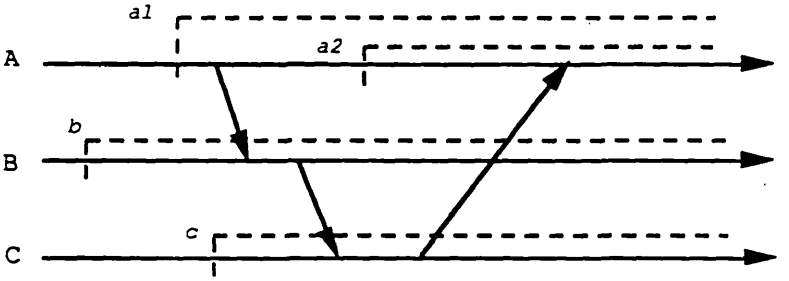


Fig. 3.3. Cyclic dependencies and nesting

The difference from the previous example is the appearance of a nested action, *a2*. When *a1* communicates with b, $cg_set_b = \{a1,b\}$; when b communicates with c, = $cg_set_c = \{a1,b,c\}$. When *a2* is created, *a1* is "suspended". Action *a2* then receives a message from *c*, and this causes $cg_set_{a2} = \{a1,b,c,a2\}$. Because of the communication pattern, action *a2* is now reciprocally dependent on a parent action,

a1; that is, *a2* can only commit if a1 commits. But *a1* is actually suspended, and cannot participate in a commit protocol, because it may have more work to do and cannot yet decide if it agrees to commit. Actions *a1* and *a2* have seemingly reached a deadlock situation. The only way to break the deadlock is to *discard* the nested action. Action *a2* is left to run to completion; if it reaches its commit point, the action is automatically committed, without running the commit protocol, and action *a1* resumes execution normally. If, however, action *a2* aborts before reaching its commit point, it is as if action *a1* had aborted, and $task_a$ reverts to the checkpoint taken at the creation of *a1*. This example also shows the way nested actions limit the domino effect: $task_a$ will never rollback further than the checkpoint taken at the creation of the outermost action, *a1*.

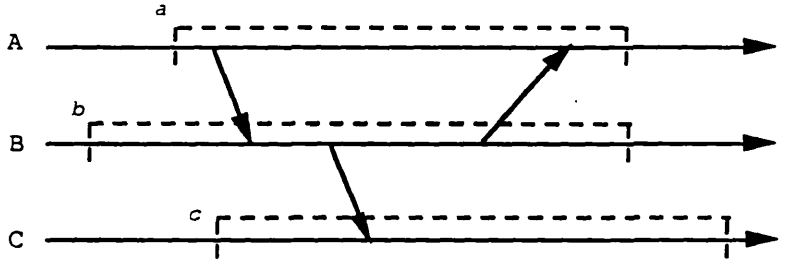


Fig. 3.4. Actions *a* and *b* may commit before action *c*

When an action reaches its commit point, some of the actions it became dependent upon may have committed already, as the example of Fig. 3.4 shows. When action *c* reaches its commit point, actions *a* and *b* have already committed (since they don't depend on *c*, they don't wait for *c* to reach its commit point). The system must maintain information on actions that have committed already, so that its dependents can also commit.

5.2.5 Commit Protocol

The commit protocol is responsible for maintaining the consistency in the system. It must guarantee that mutually dependent actions reach the same conclusion about the action outcome, and that the outcome is used to commit actions that became dependent on the finished one.

In section 7.4.1.1 we present the commit algorithm used in Cronic. It is based on the two-phase commit protocol described in section 2.2.2. In the first phase, the commit protocol chases all the dependencies and gathers the votes from the tasks involved. In the second phase it informs the tasks involved of the outcome of the protocol. If the action commits, the task is checkpointed to a reliable storage to make the result permanent. The outcome of the action is also kept in reliable storage.

3.3 Reflections on the design of actions

In this section we discuss other approaches that could have been taken for the design of actions in Cronic, and analyse their relative advantages and disadvantages.

In Jalote, a task executing an FT-Action names explicitly the other participating tasks, as a FT-Action header. This is possible because the FT-action was designed for CSP, a language in which communication statements specify the name of the target task. Conic, using locally named ports in the communication statements, is much more flexible, allowing more modularity and reusability than CSP. Consequently, the backward recovery mechanism can also be more flexible than Jalote's FT-Action.

The first decision was not to synchronize the tasks entering an action. In the FT-Action scheme, the participating tasks first agree that they are ready to perform the FT-Action. This can be seen as a pessimistic approach, a more optimistic one being not to synchronize the start of the action, assuming that they will be available normally. If the user does want to synchronize the start of an action he can always do so by writing the appropriate code at the beginning of the action; the user can never specify that he does not intend or care for the synchronized start of FT-Actions.

Another decision was to allow non-reciprocal dependencies to be formed when actions communicated using notify transactions. A first approach, [Anido86], considered that only request-reply transactions should "bind" actions together. Notify transactions could be used, but they did not incur any dependency on the communicating actions. Although this simplified greatly the tracking of dependencies, it was felt that this asymmetry between the two communication transaction types was confusing semantically. Furthermore, it also led to an implicit rule in the use of notify transactions: an action should, before sending a notify to another task, communicate using a request-reply with that task, so that if the destination task is running an action both actions are properly bound together.

3.4 Action vs. Transactions

The difference between Cronic actions and the standard transactions stems from the differences in the system model used. Transactions used in programming languages such as Argus are more closely related to database transactions. The system model in those systems can be said to be more object oriented, in the sense that the user

manipulates data (the object data) using the transactions (the allowed operations on the data). The invocation of a transaction generates a new process, or thread of control, which is totally independent from the invoking process. The transactions within an object share the object data, and since they run simultaneously, they in fact compete with each other for the data. Therefore, normal transactions need a concurrency control mechanism to guarantee the atomicity of each transaction. The concurrency control mechanism normally guarantees atomicity by enforcing serializability: two concurrent transactions are allowed to complete if the total effect is indistinguishable from running one transaction after the other. If serializability cannot be enforced, one of the transactions must be aborted.

The serializability constraint in transaction systems means that circular dependencies such as the one in Fig. 3.2 are not allowed. Cronic actions don't have this constraint, because they don't share any data. The system model is based on the fact that processes communicate only through message passing. Transaction based systems are useful in systems where *contention* for resources is likely to occur between processes. Cronic actions are intended to be used in systems where *cooperation* between processes is needed.

3.5 Summary

This chapter presented the backward mechanism designed for Cronic. It is similar to the FT-Action proposed by [Jalote86], but the participating tasks in the Cronic *action* are determined dynamically instead of being declared statically. This diminishes the coupling of the tasks involved, allowing tasks to be independently developed and

tested, and increases the reusability of tasks.

Actions may be nested, so that the user can control the amount of rollback (the creation of a nested action has the effect of causing a temporary checkpoint to be taken inside an action). Tasks become dependent on others by importing information. This dependency is tracked by the system when the action tries to commit. The commit protocol used is a two-phase protocol. In the first phase, it sends prepare to commit messages and effectively chases all dependent tasks. In the second phase it reaches a decision to either commit (if all participating tasks agree to commit) or to abort the action (if any participating task refuses to commit).

As presented, actions offer only a backward recovery mechanism. In the next chapter the backward recovery mechanism of actions is complemented by a forward recovery mechanism.

Chapter Four

Forward Recovery in Actions

4.1 Introduction

The preceding chapter described the action mechanism as a means to provide backward recovery. To provide forward recovery, the action mechanism must be integrated with an exception mechanism, and exceptions should be propagated to all tasks that can be affected by the exception. In this chapter we present an exception mechanism for Cronic, to be used complementally to the backward recovery in actions.

4.2 Exceptions in Cronic

Conic does not provide a general exception mechanism, although it does provide a restricted one, in the form of the request-reply with a fail clause (*send ... wait ... fail* => ...). For Cronic, we have chosen a simple, but effective model for the exception mechanism, similar to the one used in ADA [Ada]. It is based on the termination model, and the handling contexts are bound statically to a block or body of a procedure, function or task.

Exceptions are declared, with the same scope rules as variables, by an exception declaration. The exceptions thus declared appear in raise statements and in exception handlers within the scope of the declaration.

```
exception_declaration ::= identifier_list : [external] exception;
```

External exceptions will be explained in section 4.3.3. Examples of user defined exception declarations are:

```
too_much : exception;
no_more, clean_up: exception;
```

Certain exceptions are predefined and can be raised by the run-time system (divide_by_zero, for example). An exception may also be raised explicitly by the user with a *raise* statement.

```
raise_statement := raise [exception_name];
```

For example,

```
raise no_more;
raise;
```

The last example, of a raise statement with no exception identifier, can only appear within an exception handler, and it means that the exception which caused transfer to the handler is to be raised again (causing the interruption of this handler execution).

An exception handler may appear at the end of a *unit*, which must be a either a block, or the body of a procedure, function or task.

```
exception handler ::= when exception_choice => sequence_of_statements
exception_choice ::= identifier_list | others
```

The user may define a handler for all exceptions not named in the exception block by declaring a handler for *others*. If present, the handler for *others* must appear last.

If an exception is raised in a unit (block or body) which does not contain a handler for that exception (or an others clause), the execution of the block is abandoned and the exception is *propagated* to the enclosing unit (block or body), that is, the exception is raised again there (if the task body does not contain a handler for a raised exception the task is terminated). This kind of exception propagation, which entails propagating exceptions within a task, will be called *internal* propagation. There will be also a mechanism to propagate certain exceptions to other tasks, to be used together with the action mechanism, which will be called *external* propagation.

4.3 Exceptions and actions

The exception mechanism may be used to provide an alternative forward recovery to the backward recovery of the action mechanism. An example of using forward and backward recovery complementally in the action mechanism is to declare an exception handler for one of the action alternates:

```
action by begin
    <alternate 1>
    except
        when too_many, too_few => <handler block>;
        ...
        when mishit => <handler block>;
end
else by begin
   <alternate 2>
end
```

In this case, if an exception occurs while executing *<alternate 1>*, the normal execution of *<alternate 1>* is abandoned and the corresponding handler body is executed; the end of the handler body then becomes the end of the action, and as would happen if the normal end had been reached the task tries to commit the action. If no handler for that exception is present, the action is aborted, backward recovery takes place, and as before the next alternate is executed.

4.3.1 Internal propagation in actions

Should exceptions be allowed to propagate (internally) back through actions? Consider the situation of a exception being raised inside an action which does not define a handler for that exception. To follow the rules of exception handling, the action block must be abandoned. But should the same exception be automatically raised again in the enclosing unit, as would be the normal case? The problem is that, since the action block is to be abandoned, backward recovery will take place (to follow the rules of the action mechanism). There is no guarantee that the exception will be valid in the enclosing unit, since the cause for the exception may have been removed by restoring the previous task state. Therefore, actions introduce a variation in the internal propagation rules: an exception which caused an action to fail is not propagated automatically in the action's enclosing unit. Rather, the exception action_aborted is propagated. Another way of viewing this situation is considering that the action_aborted exception has a higher priority than any other exception that could have caused the enclosed action to fail. When the enclosed action fails, an exception resolution procedure takes place and the winning exception is the action_aborted exception, which is propagated internally.

Although it is not done automatically, propagating exceptions through actions may be useful. Suppose a user knows that a given exception is fatal to the running action, and

that he also knows that this exception was caused by some activity performed before the action was created, and the other alternates, using another algorithm, will also fail. Then, if the exception happens, the user would like the action to be aborted straight away, without trying to execute another alternate, and an exception be raised on the action's enclosing unit. That is, the user wants to specify that raising an exception is a valid result for the action, because backward recovery will not have eliminated the cause for the exception. The user specifies this behaviour by explicitly raising an exception within an action exception handler (that is, a handler defined in an action block). Therefore exceptions are not propagated through actions unless that is explicitly specified by the user, as in the following example

```
action by begin
    <alternate 1>
    except
        when too_many, too_few => raise;
        when invalid => raise mishit;
end
else by begin
   <alternate 2>
end;
```

If any of the exceptions *too_many*, *too_few* or *invalid* is raised while executing the <alternate 1>, the action is aborted, backward recovery takes place, and the appropriate exception is raised at the point where the action was created. On the other hand, if any other exception is raised within <alternate 1>, the action is aborted, backward recovery takes place, and the <alternate 2> part is executed.

4.3.2 The abort exception

Although the internal state of a task can be easily restored from a checkpoint if the

action aborts, activities which interfere with the external world, such as undoing a printing or switching a valve off, cannot be "rolled back" automatically. This is a problem intrinsic to the atomic action model, and the only way to alleviate it is to postpone all activities which affect the external world until the end of the action [Liskov83]. Notice that this approach does not solve the problem, it only diminishes the size (in time) of the vulnerability window. Besides, it may not be practical for some applications to delay all external interactions until the end of the action. What is needed is a better way to "clean up" the system when an action aborts.

Forward recovery can also be used for this purpose. The exception handler can specify the sequence of instructions needed to clean up the state before aborting the alternate. A special pre-defined exception, *abort*, is used for aborting an action. Raising the abort exception causes the alternate to abort, initiating the backward recovery. The abort exception can be used in conjunction with the *others* exception handler as an aid to restore a task local state, when this state cannot be restored exclusively by a restoring a checkpoint. It can be used, for example, in the case of tasks that are in control of switches or valves, which are accessed using a particular sequence of instructions:

```
action by begin
    <alternate 1>
    except
        when others: begin
                        <restore external state>
                        raise abort;
        end
end
else by begin
<alternate 2>
end;
```

Notice that the use of the handler for *others* inside an action must be used with care, since it could defeat the intended backward recovery scheme. If an empty handler is declared for *others*, or more precisely, if no raise abort statement appears in the *others* handle, the alternates that follow will never get executed, for backward recovery will never take place.

4.3.3 External propagation of exception in actions

In a distributed system, apart from internal propagation of exceptions, there must also be a provision for exceptions to be propagated to other tasks. In Ada, an exception can be propagated to a task when it is attempting to communicate with another task. In Conic, the fail clause of a request reply also provides a kind of exception propagation, but as in Ada, it is limited to one single communication.

The action mechanism provides a convenient vehicle for providing a distributed exception mechanism which is similar to the internal exception mechanism, in the sense that a handling context can be defined for a segment of code, which can include several communication statements, instead of being restricted to only one statement, as in Conic and Ada. Several decisions, however, must be made on the semantics of "distributed" exceptions. For example, when to propagate an exception, what exceptions must be propagated, and how to propagate them.

An action propagates exceptions to all tasks in its dependent set; it does not propagate exceptions to actions in its contingency set. The reasoning for this approach is as follows. The fact that an exception was raised indicates that the action discovered an error. Because of the prohibition of suspicion assumption, only actions which imported this erroneous information need to act on the exception, and therefore only they need to receive the notification of the exception occurrence.

Should all exceptions raised inside an action be externally propagated? This may be somewhat restrictive, since certain exceptions may have a meaning only locally, and may not affect any information that was exported. The best way is to propagate only exceptions that are explicitly declared as external exceptions. In Cronic, an exception is declared to be external by qualifying the exception declaration, as in

```
too_much, no_more : external exception;
```

The standard exception *abort* is also declared as external. External exceptions must be declared at the task level, and therefore the scope of external exceptions is the whole task.

Because of circular dependencies, when an external exception is raised an action cannot invoke the exception handler immediately after it propagates it to tasks in its dependency set. Instead, it must wait and follow an exception propagation protocol, similar to the commit protocol, because different exceptions may be raised by the actions in the dependency cycle, and the winning exception may be one that another task raised at the same time. Going back to the idea that actions in a dependency cycle are logically one action, all the tasks participating in this merged action must agree on the exception that is to be handled, should more than one external exception be raised during the action execution. This also indicates the necessity of ranking the exceptions, so that the tasks can agree on the "winning" exception. Exceptions are ranked according to the order they are declared.

One way to deal with an external exception occurrence would be simply to stop the action execution and wait for the dependent actions to reach their respective commit

points, and then to use the commit protocol to vote and propagate the exception. The result of the commit, instead of being either commit or abort, would be either commit (no exception) or an exception to be handled. However, this "passive" approach, besides being inefficient, may lead to a deadlock: if every action in a dependency chain detects an exception and waits, no action will reach the commit point to start the commit protocol, and the actions in the chain wait forever.

What is needed is an exception propagation protocol that is activated as soon as an exception occurs. The external exception propagation algorithm is discussed in section 7.4.2.1.

4.5 Summary

This chapter presented the exception mechanism designed for Cronic. The exception mechanism can be used in conjunction with the action construct to provide a complementary forward and backward recovery for tasks. The exception mechanism is based on the termination model (the unit that was executing when the exception occurred is not resumed after the exception handler is executed), and the handlers are bound statically to regions of text in the task. In particular, they can be bound to alternates in actions, in which case forward recovery will take place as a first step in attempting to recover from a failure. Only if forward recovery fails will backward recovery take place. Forward recovery can also be used as a means of cleaning up the external system states that cannot be rolled back automatically (such as valves and switches controlled by the tasks).

Certain exceptions, declared as external exceptions, are propagated to tasks that have imported information from the task were the exception originated. Since the tasks may have a circular dependency, and since more than one task may raise an external

exception, the tasks must run an exception resolution protocol to agree on the exception handler to be executed.

Chapter Five

Fault-tolerance at the Configuration Language Level: Clusters

5.1 Introduction

In Cronic, the fault-tolerance of a system is specified in two levels. At the programming level, tasks may be structured with the aid of actions. At the configuration level, those tasks are then configured into a system with the specification of the intended replication for each module. The replication varies in level (the number of replicas) and in type (passive or active redundancy). In this chapter we describe the replication mechanisms available in Cronic, and their integration with the action mechanism described in the previous chapter. We begin by defining a model of replication for Cronic. Using this model, we introduce the mechanisms for providing passive and active replication.

5.2 Clusters

A *cluster* is a set of instances of modules, distributed over the system, grouped together on a functional basis for fault-tolerance purposes. A particular instance in a cluster is be called a *member*. Although it is actually constituted by a set of instances distributed over the system, a cluster is seen by the other modules in the system as a single logical module. The redundancy in a cluster is used to provide a non-stop service for the software module. A cluster may be active or passive. In an *active* cluster, all members are executing concurrently; in a *passive* cluster only one member is actively executing, with the other members on a standby state.

A cluster is seen by the other modules in the system as a single logical module. This is achieved in Cronic by using a multicast address for the cluster. Messages sent to and from a cluster use the cluster address. In that way, any change in the cluster configuration remains unobserved by the communicants with that cluster.

A cluster is said to be *homogeneous* when all its members are of the same type, and *heterogeneous* otherwise. Members of an heterogeneous cluster must be of *compatible* types. The notion of compatibility depends on whether the cluster is active or passive and on the member semantics. Compatibility is based on two levels of equivalence: interface equivalence and execution equivalence. We describe these two levels by analysing the cluster semantics.

5.2.1 Cluster semantics

Even if a cluster is homogeneous, its behaviour depends on whether there is any non-

determinism in the behaviour of members. Thus, to describe the semantics of a cluster, we need to describe first the semantics of the cluster members. In this section we present a model of the computation performed by the individual modules in the system. Using this model, we can define precisely the semantics of clusters and the notion of compatibility between cluster members.

A module is seen as consisting of an internal state and a port interface for exchanging messages. Two modules are *interface equivalent* if they use the same external interface. This definition of interface equivalence is very conservative, but it is easy to enforce, because it can be checked at compile time. A less conservative definition (such as, for example, taking into account only ports actually used, or linked), would be more difficult to enforce, and does not seem to present much advantages in practice. The approach chosen, although conservative, it is not restrictive. A module may still use only a subset of the interface declared.

The notion of execution equivalence takes into account the module semantics. At any moment, the state information in a module represents the history of messages that passed through the module interface. For simplicity, we assume that module M interface contain a single entryport *in* and a single exitport *out*. In a *step* of the module execution, denoted by $M[S_i, m_{in}, S_o, m_{out}]$, the module M, initially at state S_i, receives a sequence of messages m_{in} from its entryport, changes its internal state to S_o, and sends the sequence of messages m_{out} to its exitport. Because of non-determinism, there may be a set of possible steps for a fixed initial internal state and a fixed sequence of input messages.

If there is only one possible step execution given S_i and m_{in}, the module is said to be

deterministic. When an active cluster consists of modules that are deterministic, its members respond identically to incoming messages and produce the *same* output messages. Previous proposals for fault-tolerance impose determinism as a requirement ([Mancini86], [Gunninberg83]).

The requirement of determinism can be relaxed by introducing an application-dependent definition of acceptable non-determinism. In this way, the application specifies the degree of non-determinism it considers acceptable. We associate with module M a *state-equivalence* relation, and with each port of M a *value-equivalence* relation. These two module-specific equivalence relations induce an equivalence relation on the step executions in the module. Two step executions $M[S_i, m_{in}, S_o, m_{out}]$ and $M[S'_i, m'_{in}, S'_o, m'_{out}]$, where S_i and S'_i are state-equivalent and m'_{in} and m'_{in} are value-equivalent, are *step execution equivalent* if

1) The new states S_o and S'_o are state equivalent, and

2) The output messages m_{out} and m'_{out} are value equivalent.

The step execution equivalence relation induces an equivalence relation on the module execution: a module is defined to be *execution-equivalent* if all step executions $M[S_i, m_{in}, S_o, m_{out}]$ are equivalent for any possible S_i and m_{in}. Clearly, a deterministic module is also execution equivalent.

An active cluster should appear to the other modules in the system as if it were a single module. Therefore, a message sent to a cluster should produce the same result as if it were sent to a single module. When a cluster consists of modules that are execution-

equivalent, an incoming message may produce a different (albeit equivalent) behaviour at each member, but with the guarantee that all cluster members will produce equivalent output messages. In a system composed of several clusters, an incoming message will in general produce a chain of messages through the other clusters. But if each cluster is composed by execution equivalent members, the system as a whole will exhibit the correct behaviour (that is, the behaviour of a system where active clusters were replaced by single modules).

5.3 Passive clusters

Passive clusters are the instrument for providing passive replication in Cronic. In a passive cluster only one member (the *primary*) is actively executing at one time. The other members (called *standby members*) are on a standby state. If the primary fails, a *switch-over* occurs and a standby member assumes the role of the primary.

5.3.1 Choosing the new primary

A general mechanism for choosing the new primary is to promote an election among the standby members. An election is justified in two situations. One is the case where the system does not provide information about failures, and the election is used for finding one surviving standby member. The other case is to achieve a better load balancing in the system, and the election is held to find the standby member that is located at the least loaded node. In Cronic, where the failure detection occurs at the system level, the first case does not apply, and we felt that the increase in the switch-time incurred by an election did not justify the flexibility offered. Furthermore, dynamic load balancing is an issue outside the scope of this work. We assume the load balance is done statically

by the application developer when configuring the system.

Therefore we took the approach of fixing (at system creation) the order in which the standby members should take over. With this order, and the information of which members are currently functioning provided by the system, the standby that should become the primary member (which we will refer as the *next primary*) is always defined. We will refer to this order as the *NextPrimary* list. The NextPrimary list is defined to be circular and static. A failed member rejoins the list in the same place assigned originally.

5.3.2 History-independent passive clusters

A module is said to be *history-independent* if its operation does not depend on past information. This is the case of cyclic tasks which are sent periodically a set of values to work on. For example, a task collecting values sent by sensors in a process control system, where the activity performed by the task depends only on current sensor values.

The only requirement for compatibility in history-independent passive clusters is that the members must be interface equivalent. Members may provide a different service: for example, a standby may provide a degraded form of the service offered by the primary. Members may be even composed of different modules. Note that a module composed hierarchically using only history-independent tasks and modules is also history-independent, and only the external interface counts in the compatibility requirements.

Assuming all modules shown in Fig 5.1 are history-independent, modules *A* and *B*

are compatible to be used in a cluster, because they use the same external interface (i.e., they are interface equivalent).

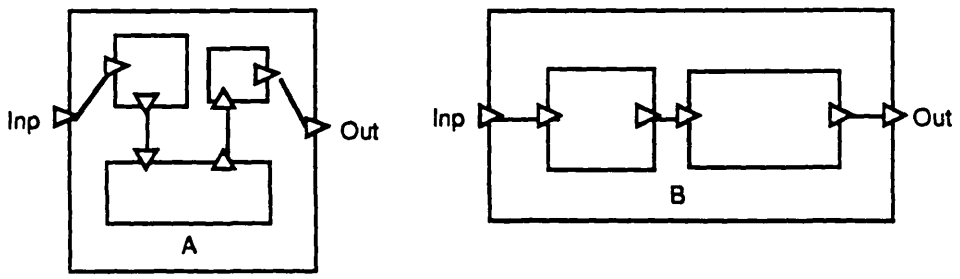


Fig. 5.1 History independent modules are compatible if they are interface equivalent

In a history-independent passive cluster, when the primary fails, the next primary is simply activated. The implementation of history-independent passive clusters is discussed further in section 7.4.3.

Although the messages addressed to a passive cluster reach all nodes where members are running, the executive only delivers them to the primary member. Therefore, when the switch-over takes place, the new primary does not have in its buffers any previous message sent to the cluster. The assumption is that a new set of messages will be sent in the next cycle. Notice however that the result for one the the cycles may be lost when the primary fails. For this reason, history-independent passive clusters are suitable for cyclic tasks that are continuously sent some values to operate on, and one occasional failure to deliver the results can be tolerated. Notice that the failure of the primary does not necessarily mean the results for one cycle will be lost. That depends on when the failure happened. If the failure happens after the results for one cycle have been sent and before the primary has received any message for the next cycle, the cluster will be

able to produce the results without interruption.

5.3.3 History-dependent passive clusters

In history-dependent passive clusters, the primary must save its state to the standby members from time to time, so that the stanbies can resume the primary work in case of failure. As discussed in section 2.5, in Cronic passive replication provides a *virtual* reliable storage for the task state when used in conjunction with actions. As opposed to a disk based reliable storage, which even in the event of total crashes retains the checkpoints recorded, the virtual reliable storage provided by replication is successful only as long as (at least) one standby remains functioning.

As discussed before, disk based reliable storage is suitable for applications where accountability is the foremost concern. Recovery may take longer, as long as the state of the computation at the time of the failure can be recovered, as in the case of an application such as distributed numerical programming. Other applications, however, have a more stringent requirement. Not only must the system state remain consistent, but it must exhibit a fast recovery. That can be provided by passive replication.

In backward recovery using disk-based reliable storage, the checkpointed states of the tasks participating in the action must be recovered before the tasks can resume execution. Backward recovery with virtual reliable storage provided by passive replication provides a faster switch-over with the advantage that there is no need for task states to be copied back. The next primary can resume execution straight away, because it is already at the last checkpointed state.

Every time a task is checkpointed in an action its state is sent to its standby members, instead of being sent to a reliable storage on disk. If the action aborts, a standby takes over as a new primary and resumes execution from the last checkpointed state. Notice that when an action aborts, there is a change of roles between primary and standby members over the system.

Fig. 5.2 illustrates the change of roles when an action rolls back. Shaded rectangles indicate the primary members in the clusters. Suppose clusters *A* and *B* are running mutually dependent actions. When the current primary in *A* fails, the action aborts and the two clusters roll back. This causes the standby members at both clusters to take over the role of primaries.

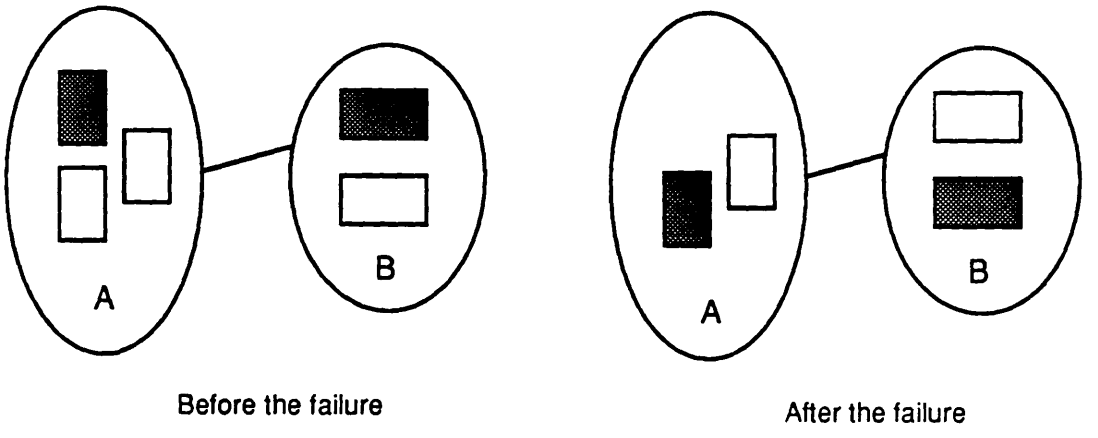


Fig. 5.2 Change of roles when a cluster rolls back

Notice that the number of members in each cluster does not need to be the same. Also, the NextPrimary list is circular, and fixed at configuration time.

There are two ways of viewing this integration of actions and passive replication. From the point of view of actions, replication provides an implementation of the reliable

storage needed to checkpoint the task state. From the point of view of replication, actions provide a synchronization point for saving the state of the primary to the standby .

History-dependent passive clusters in Cronic must be homogeneous. The reason for this restriction is the model adopted in Cronic of using passive replication with actions. Two problems arise if non-homogeneous history-dependent clusters were allowed. The first problem is that the checkpointed state must be meaningful for the standby. That is, the standby must be able to interpret correctly the received state. A more difficult problem is that of nested actions. All the cluster members must have the same structure with respect to nested actions.

The Cronic configuration language, presented in chapter 6, will provide language constructs for defining a passive cluster, specifying the location where cluster members will run, and assigning the priority order for becoming the primary member in case of failures.

5.4 Active clusters

Active replication can be used to mask two types of faults: omission in sending messages, and transmission of messages with erroneous content. The first type of fault occurs for example when members are running on fail-stop nodes. The failure of a node in the system will only cause members running at that node to cease sending messages. Another example of the first type of fault is intermittent failure of the node transmitter, which causes it not to send some messages. The second type of fault may

be caused by a defective hardware or software (in the case of heterogeneous clusters, with members running different software versions). We restrict our attention to non-malicious errors, since we assume the systems will be running in a "friendly" environment, such as a process control inside a plant. If malicious behaviour is possible, an authentication protocol could be added on top of the protocols presented here.

The basic problem that must be solved in order to support active clusters is the correct synchronization of the members operation. In our system model, in which all interaction is through message passing, this problem translates into delivering the messages in the correct sequence to all cluster members, and into combining messages sent by the cluster members into one.

Let us examine the problem of a simple module m communicating with an active cluster c. A message sent by m must be received by all cluster members. Moreover, two different messages sent to a cluster must be received in the same order by all cluster members. Otherwise, the cluster members may lose synchronization, as Fig. 5.2 illustrates. A cluster c is sent messages from modules m and n. Cluster member c_1 receives the message from m first, while cluster c_2 receives the message from n first. Member c_1 may then process first the message from m, while c_2 may process first the message from n, which could lead to different behaviours.

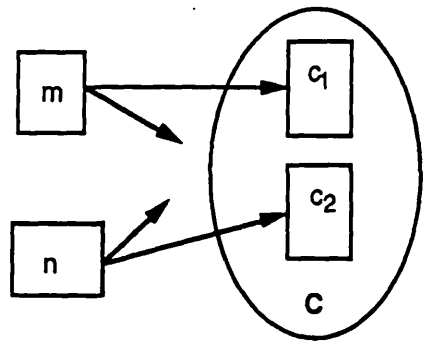


Fig. 5.2 Clusters members could loose synchronization

This example illustrates the requirement of a communication mechanism to support active clusters. The cluster addressing mechanism must be integrated with a multicast protocol that ensures that messages addressed to a cluster are delivered to all cluster members, with the guarantee that it will be ordered in the same way, relatively to other messages, by the cluster members. This reliable multicast is further discussed in section 7.3.1.

In the case where an active cluster *c* sends a message to a module *m*, several messages, originating from the several members, will be sent to *m*. However, only one copy of the message should actually reach *m*. Therefore, the messages from cluster members must be *collated* before being delivered to their destination. The next section describes our approach to this problem.

The general case of a cluster communicating with another cluster is a composition of the two previous cases. Only *one* copy of a message sent by a cluster a must be received by *every* member of the destination cluster.

The compatibility requirements for active clusters is that members must be execution equivalent. The reason is obvious. The correct operation of an active cluster is dependent on it being able to appear to other modules as a single module, and that is only possible if the members are execution equivalent.

5.4.1 Links as filters

A link is normally viewed as a simple, passive *channel* between modules in the system: if exitport *ex* is linked to entryport *ep*, messages sent to *ex* are expected to be duly delivered at the *ep*. When using active clusters, it is useful to think of a link as a *filter* between modules in the system, because some active work must be performed by the link to collate the messages originated by the cluster members into one.

Since all the messages generated by the cluster members may be made available at the link, different strategies for collating, or filtering, the messages can be used. For example, the user may want the link to deliver the first message generated by any member of a cluster, or he may want the link to perform a voting among all generated messages. Some filtering strategies are discussed in this section.

Filters are in fact an instrument for achieving different levels of fault-tolerance with an active cluster. The choice of which filter to use will depend on the failure assumptions about the cluster members.

If cluster members may fail by sending wrong messages, we can use a filter which masks those failures out, by either voting or averaging the messages. It is important to notice that this voting or averaging must be implemented in a decentralized way; if it is

centralized, it introduces a single point failure, invalidating the whole purpose of replication.

If it is assumed that cluster members are prone to omission failure only, we can use a filter that delivers to its connected exitport *one* corresponding message generated by any cluster member.

5.4.2 Application specific filters

Automatic filtering assumes the contents of the messages sent by cluster members are the same (or at least similar enough so that the receivers cannot distinguish them). If it is assumed that the members can generate messages with different contents, the user needs to provide an application specific way to collate the copies of the message into one. There is very little that can be done to collate different messages without application specific information. For example, the only voting possible without any semantic information about the message contents is the one based on bit by bit comparison.

Allowing application specific filters to be defined introduces more flexibility, since the user can use all semantic information to perform the collation of message copies. Several strategies can then be implemented by the user: majority voting, averaging, special voting schemes, etc.

Example of averaging filters that can be defined by the user are, for a set of messages $(m_1,m_2,...,m_i)$:

- the median of m_i after the lowest and highest values have been discarded.
- the average of m_i after the lowest and highest values have been discarded.

- the average of all m_i that are within Δ of i-j other arguments, etc.

Majority voting can also take advantage of semantic information, allowing the user to define admissible error margins between different copies of the message.

In the next chapter we present the facilities in the Cronic configuration language to specify different filters.

5.5 Reintegration

A cluster is resilient to failures of all but one of its members. After the failure of one of its members, a cluster has its reliability diminished. Thus, it is desirable to replace, whenever possible, the failed instance in order to restore the cluster to its full fault-tolerance capabilities. In this section we discuss the problem of reintegrating a failed member to an existing cluster.

We examine the problems of supporting reintegration for the various types of clusters. The first support needed is the ability to re-instantiate automatically a failed instance. In Cronic we provide only for automatic re-instantiation after software crashes. The re-instantiation is possible only when the executive where the instance is running is still functioning and the code for the failed task is loaded. Cronic does not provide for automatic re-instantiation in the case of hardware crashes, when node fails due to a hardware problem. There is no provision, at the moment, for automatically downloading.

When a task fails, it is automatically restarted by the local system executive running.

The reintegration of this new instance into a working cluster depends on the type of the cluster.

History-independent clusters present few problems for reintegration. Since it does not depend on past values to operate, a recovered instance can resume execution as soon as it has recovered. The reintegration of history-dependent clusters is more complex. In order to re-integrate a history-dependent cluster, the system must bring the recovered member to a state that is consistent with the working members. For homogeneous clusters, a technique similar to process migration is used.

5.6 Summary

In this chapter we presented the mechanisms available in Conic to introduce fault tolerance at the Configuration level. A *cluster* is collection of replicated modules grouped together on a functional basis. The view other modules in the system have of a cluster is that of a single logical module. Cronic provides for two types of replication within a cluster: passive replication, where only one of the replicas is executing and the others are on standby, and active replication, where all replicas are executing concurrently. Clusters may be homogeneous (members are of the same module type) or heterogeneous. We examine the use of clusters according with the component semantics, dividing our discussion in history-dependent and history-independent clusters (active and passive).

Passive clusters members may be structured with the aid of actions. In that way, the standby members serve as a virtual reliable storage for the primary state. Replication in this case provides a faster roll-back for actions than the one provided by disk-based

checkpoint.

Active clusters can be used to mask software failures without the necessity to roll back. Links in an active cluster can be viewed as a filter for the members messages. Different filtering strategies can be used to achieve different degrees of fault tolerance.

We also examined the compatibility requirements for members in a heterogeneous cluster. The compatibility is based on two levels of equivalence (interface equivalence and execution equivalence), and depends on the replication type and on the component semantics.

In the next chapter we present a configuration language that supports our model of replication.

Chapter Six

A Configuration Language Supporting Replication

6.1 Introduction

In this chapter we present a configuration language that supports our replication model and allows the specification of the replication mechanisms described in the previous chapter. Several programming environments allow some sort of configuration management. But they typically do so by allowing processes to create another processes, or at least pass the name of processes and ports to other processes, as in Unix, or Argus [Liskov83]. Because the activities that control the configuration are intermixed with the functionality of the individual processes, they are not readily available for examination by a person (or process) attempting to reason about the system.

Some environments, however, adopt a software methodology similar to ours and enforce the separation between the system specification and process description, as for example the languages Pronet and Durra. Pronet [LeBlanc87] is similar to Cronic, in

that it is constituted by two complementary languages, one to describe the processes (called Alsten), and one to specify the system (called Netsla). The system specification initiate process executions and oversees the operation of the processes it has initiated. The Durra language [Barbacci88] is concerned only with the system specification level. Users write individual tasks in any language supported by the host processors. The language is then used to specify the creation of the processes at the appropriate processors, and the communication patterns between the processes.

None of these languages, however, includes the possibility of specifying replicated processes. In this section we present a configuration language that supports our model of replication. The language is based on the Conic Configuration Language described in [Dulay86].

It is important to point out that the Cronic configuration language provides a somewhat static configuration management. The emphasis is on system consistency with a fast recovery rather on system extensibility. It does not provide for a fully dynamic reconfiguration of the system. Although it can be used as a fault-tolerant technique, dynamic reconfiguration has also the aim of providing extensibility to a running system. Extensibility is certainly a desirable characteristic of any system, but we are more concerned with the system's fault-tolerant characteristics. The system specification in our system can be seen as a *system maintenance* type of reconfiguration. That is, the system is reconfigured in case of failures, but to a limited degree, and in standard ways.

The Cronic configuration language has two primary purposes: to describe the

specification of a system, and to direct its construction and maintenance. A system in Cronic is hierarchical composition of modules, which are instantiated and then *linked* together (the establishment of a connection between two module ports is called a link) The module types used in a configuration specification may be task modules (that is, modules containing a single sequential task) or collections of modules called group modules. The highest level module group (which will be called *system* group) specifies the location and replication of each component module, that is, it allocates group modules to nodes.

6.2 Requirements

One question that must be addressed before discussing the configuration language requirements is the necessity of a configuration language at all. The question is, "is it necessary to have two distinct languages for the two different levels of programming"? One reason for providing two different languages is to enforce a separation between the two phases of the design process. Specifying the configuration policy together with the system functionality makes modules less modular, because it constrains the situations the modules could be used in. Also, the configuration management needed in Cronic, with fixed policies for reconfiguration, does not need an imperative language, a declarative language could be used instead, providing a clearer configuration specification. A configuration language for Cronic therefore should be

- conceptually simple – yet should be able to specify all mechanisms presented in the previous chapters
- modular – to provide information hiding and re-usability

- declarative – to provide a specification that can be easily understood
- hierarchical – to allow top-down design
- executable – to provide system creation and maintenance

In the next sections we introduce the Cronic configuration language. The language is an extension to the Conic Configuration Language. We concentrate on the description of the extensions. A full description of the Conic Configuration Language is in [Dulay86].

As usual in the Conic version, the **use** clause specifies the types of the modules that will be used.

6.3 Instantiation

The *create* statement is used to declare a module instantiation. A *create* statement specifies the module type, declares the actual parameters if any are needed, and assigns a name to the instance. The *create* statement is also used to assign modules to specific nodes. This location specification can only be done at the highest level of the configuration specification. An example of a module istantiation is (from the example in section 1.7.2)

```
use
    pump; user;
create
    pump(2,1) at chico
    user2: user at harpo;
```

This declares that an instance of the module type *pump*, with actual parameters (2,1),

is to be created at node *chico*. The instance name is *pump* (the absence of a name indicates that the instance will be named after the module type). An instance of the module type *user*, named *user2*, is declared to be created at node *harpo*.

The *create* statement is also used to instantiate clusters. Clusters must be named explicitly in *create* statements. The create statement is also used to specify the type of replication, active or passive, intended for the cluster. Active replication is the default replication type. An example of an active cluster declaration is

```
use
    pump; bucket;
create
    rep_pump: pump(2,2) at chico, at harpo, at gummo;
```

This example declares that the cluster called *rep_pump* is constituted by replicated instances of the module type *pump* running at the nodes named chico, harpo and gummo. This is an homogeneous cluster, that is, all members are instances of the same module type *pump*. As discussed in section 5.2, active clusters can also be heterogeneous, provided the members are execution-equivalent. An example of an active heterogeneous cluster declaration is

```
use
    pump; bucket;
create
    repli_pump:  pump(2,2) at chico, at harpo,
                 bucket(20) at gummo;
```

which declares an active cluster *repli_pump* constituted of three instances: two instances of type *pump*, and one instance of type *bucket*. Notice that not only the module types are distinct, the number of arguments is also different for module types

pump and *bucket*. To see why this is allowed, recall the definition of module compatibility for active clusters: the constituent types of a cluster must be port and execution equivalent. Execution equivalence is defined in terms of the perceived behaviour of the module types, and, unlike port equivalence, cannot be checked automatically by the configuration language compiler. In fact, it is very difficult to devise a general method for checking execution behavior. Therefore, instead of placing restrictions that could be checked by the compiler but ultimately would have no effect on the behaviour of the modules (for example, requiring that they should have the same number and types of arguments), it is left to the user to verify that the module instances are in fact execution equivalent.

Passive replication in a cluster are specified using the qualifier *backup* before the *at* location specification. The order the passive members are declared establishes the priority for a member to become the new primary in case the primary fails (i.e., it defines the NextPrimary circular list). An example of an homogeneous passive cluster creation is

```
use
    pump;
create
    rep_pump:   pump(2,2) at chico,
                    backup at harpo,
                    backup at gummo;
```

A cluster can also combine active replication with passive replication (history-independent clusters). In this case, passive instances become active and are integrated into the active cluster when any one of the active members fails:

```
create
    rep_pump:   pump(2,2) at chico, at harpo,
                backup at gummo;
```

6.4 Linking

The *link* construct remains almost unchanged from the Conic version, since after instantiation a cluster is seen as a logical single module. When specifying a link from or to a cluster, it is not necessary to link explicitly the cluster members; instead, the cluster logical name, defined in the corresponding *create*, is used:

```
link
    pipe.water_out to pump.water_in;
```

links the exitport *water* of all members of the cluster *example* to the entryport *water* of *user* (or to all of its members if *user* was also defined as a cluster). There remains one question, however, namely how to specify the type of filter service provided by the link in the case of active clusters.

6.5 Specifying filters

The specification of the filtering strategy is done at the link construct. The default strategy is bit by bit voting. If another filtering strategy is needed, the user must supply the name of a procedure that implements the required strategy, by following the name port with the *with* construct:

```
use
    pumpdef: myVoting;          -- imports the procedure
link
    control.Ctr to valve.Ctr;   -- by default bit by bit majority voting is used
```

```
control. flow to sensor.req with myvoting; --voting performed using the procedure myvoting
```

There is the possibility that a different type of filtering is necessary for the two directions of communication in a link, when the link is between request-reply ports. For example, the user may want the request message to be voted, but the reply message to be delivered in a first-come basis.

```
use
    systemdefs: Majority, RoundRobin;
link
    pipe.water_out to pump.water_in with Majority reply RoundRobin;
```

The implementation of the user specified filtering procedures is given in section 7.4.3.3.

6.6 Summary

This chapter described the Cronic Configuration Language, which supports the replication mechanisms of chapter 5. The language is an extension of the Conic Configuration Language [Dulay86]. The modifications affect only the highest level of description, where the location specification is done. The modifications allow the specification of active and passive clusters, and the specification of filtering strategies for active clusters.

Chapter Seven

Implementation Issues

7.1 Introduction

In this chapter we discuss the implementation of the mechanisms described in the previous chapters. These 'high-level' fault-tolerant mechanisms are built on top of some basic services. The purpose of this division is two-fold. First, it allows for the implementation and testing of the higher level fault-tolerant mechanisms to be done independently. And second, it allows the set of higher-level mechanisms in Cronic to be expanded in the future without affecting the existing ones.

We begin by recalling the basic support needed for the different mechanisms, so that a common set of basic services can be defined in order to support all mechanisms. We then examine the implementation of these basic services. Finally, the implementation of the higher level mechanisms using those basic services is presented.

One of the most basic support for any fault-tolerant mechanism is the provision for

failure detection. It must be able to handle both hardware and software failures.

For the implementation of clusters, there is a need for a communication service that provides group addressing. To support passive clusters, group addressing and a global status service provide all the support needed. Active clusters need one more characteristic from the group communication. So that the cluster consistency is maintained, members in an active cluster must all receive the same messages in the same order. Therefore, the replication mechanisms in Cronic need a communication service that provides group addressing and delivers the same messages in the same order to the group members.

To implement the backward recovery in actions, we need a reliable storage to record the checkpoints and outcome of the actions. We also need a commit protocol to 'chase' the dynamic dependencies and commit the actions. The forward recovery in actions needs a exception resolution protocol that is similar to the commit protocol. The commit and exception resolution protocols are run within a group, and must deliver their outcome to all concerned tasks. Therefore, they can also benefit from a communication service with group addressing and guaranteed delivery.

7.2 Basic Services

Based on the requirements for the support of the mechanisms, we decided on the three basic services to offered by the Cronic kernel as a basis for constructing tools for building fault-tolerant systems are: *global event ordering*, *node status management* and *reliable broadcast*. These services are not independent of each other. Global event ordering is concerned with providing all nodes in the system with a consistent view of the order in which they perceive the occurrence of events. Events include sending and receiving messages, failures and recoveries. The status management service, which

provides information about failures and recoveries in the system, depends on the global event ordering service, since this information must be ordered in relation to other events in the system. Atomic broadcast is the ability to send a message to a group of nodes with the guarantee that either all (correctly functioning) nodes receive the same message or none receive it, and that all messages received are delivered in the same order to all correct nodes. Therefore, it also depends on the global ordering service.

7.2.1 Global Event Ordering Service

The global event ordering service provides a global order for events in the system, that is, it makes all the nodes in the system perceive events in the same order. Typical events are sending and receiving messages, and the detection of crashes and recoveries in the system. There are two approaches to order an event globally. One is to maintain a global real-time clock in the system and to timestamp every event with the value of this global clock at the moment the event occurred. Another approach is to order only partially the events in the system, based on their causality relation. This involves deriving a global logical timestamp based on the causal order of events. That is, a global *logical* clock is maintained instead of a real-time clock.

Ordering events partially

In our system model, in which events are causally related exclusively through explicit communication, a partial order is sufficient. That is, a relative order needs to be established between two given events only if it is possible for one event to causally affect the other. In [Lamport78], Lamport defines the relation "A happened before B", where A and B are events, which induces a partial ordering and is denoted by $A \rightarrow B$, as the smallest relation satisfying:

i) if A and B are in the same node and A comes before B;

ii) if A is the sender of a message and B the receiver of such message;

iii) if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

Using this definition we can derive the global timestamp condition that must be observed to correctly order the events in the system: for any events A and B, if $A \rightarrow B$ then $T_A < T_B$, where T_X is the value if the timestamp value associated with the event X.

If we consider only messages, partial ordering can be easily implemented as follows. Each node maintains a local timestamp, which is incremented before every send. Each message is timestamped with the value of the node's timestamp at the time of the send. When a node receives a message whose timestamp is greater than the node's timestamp, it cannot accept that message straight away, because this would violate the causal relation (a message would have been received before it had been sent). Instead, it first "bumps" its own timestamp value to be greater than the message's (correcting the causal relation) and then accepts the message. It is the inherent causal relation of exchanging messages that makes this implementation simple.

7.2.2 Global Status Service

The most basic facility in a fault tolerant system is the ability to detect failures. This is the role of the global node status service, which also provides information about recoveries in the system. To do this, it must solve the *node membership* problem: it must determine which nodes are working correctly at a given time, and must ensure that all correctly functioning nodes observe the same order of failures and recoveries. It

is also important that all nodes observe status events in the same order in relation to the communication events. In other words, communication and status events must be ordered using the global order service.

Failed nodes do not normally advise the other nodes in the system that it has just failed. Even if some self-checking (by hardware or software) is performed, there is no guarantee that the checking mechanism itself will not fail. Therefore the only way to detect failures or recoveries in a distributed system must be based in the observation of the other nodes behaviour. Omission and timing faults are detected by the absence of some expected response from another node, in the form of either an acknowledgement or some other specified communication.

7.2.3 Atomic Broadcast Service

The atomic broadcast service provides the nodes with broadcast with failure atomicity: if a message is received by one node in the broadcast group, it will be received by all operational nodes. In a point-to-point network, sending a message to a node is atomic: either the message arrives or it does not. But sending a message to more than one destination node is not atomic, since the sending node could fail after before sending to all destinations. Broadcast networks such as Ethernet [Metcalf76] and ring networks would appear to provide already an atomic broadcast, but they do not. For example, a node's message buffer space may be full, the node may not be listening at the time the message is transmitted, or, in a contention network, an undetected collision may affect the reception by a few nodes.

To effect an atomic broadcast, protocols are required so that in case of message loss the message is retransmitted, and in case of failure of a node another node assumes its

duties. Therefore, the processor membership problem must be solved in order to implement an atomic broadcast.

A reliable broadcast is a broadcast that besides being atomic is also ordered. That is, in a reliable broadcast it is guaranteed that all correct processor will receive the message (atomicity), and all correct processors will receive all messages in the same order (ordering). Note that the order the nodes receive the message is not necessarily the order they were actually sent, it suffices that all nodes perceive the same order.

7.3 Basic services implementation

These three basic services are not independent. Depending on the algorithms used, the services may appear layered in different ways. For example, some process membership algorithms assume that they are built on top of a global ordering service [Cristian88], while some global ordering mechanisms assume to be built on top of the processor membership service. There are even some algorithms which provide the three services at one indivisible layer, that is, one only protocol provides event ordering, reliable broadcast and node status management, as in [Chang84]. Chang's algorithm is intended to be used in broadcast based networks, and implements a reliable broadcast (not only an atomic broadcast) by partially ordering events.

We decided to implement the basic services in Cronic using Chang's algorithm for two reasons. First, Cronic is targeted to be used in local area networks (LANs) and broadcast based media, for which Chang's algorithm is very efficient. In data processing environments, such as office networks, contention based broadcast networks such as Ethernet are already widely used. In process control environments, such as a plant floor networks, non-contention broadcast and common buses (which

also provides broadcast capability) such as MAP and fieldbuses are being standardized and are becoming to be widely used. The second reason for using Chang's algorithm is its simplicity and integration of the three services into one single protocol (in fact, as explained below, there are two protocols, one used to solve the process membership problem, the other to implement the reliable broadcast). In section 8.4 we comment on the experience of using Chang's algorithm in practice.

7.3.1 Chang's reliable broadcast algorithm

The algorithm uses a neat combination of negative acknowledgement, positive acknowledgement and token passing. Messages are numbered sequentially, but a node does not acknowledge the receipt of messages which arrive in the correct sequence. Only if a message arrives out of order does a node asks for the missing message (this is called negative acknowledgement). The responsibility to attach numbers to messages is rotated between all nodes in the system, so that it does not constitute a single point of failure. This responsibility is viewed as a token that must be passed among the nodes, and the token is passed in a pre-determined order along a *ring* of nodes. The ring is formed in a first phase of the algorithm, which in fact solves the processor membership problem and then establishes the order the token must be passed (called the *token list*). A node must pass the token after issuing an specified number of sequence numbers, or timestamps, and the mechanism of token passing uses positive acknowledgement, that is, the node accepting the token acknowledges explicitly this fact in a message. The algorithm also uses positive acknowledgement between the message source node and the current token. The source node re-broadcasts the message until the token node attaches a timestamp to it.

Normal phase: broadcast algorithm

The normal phase of the algorithm uses the token list created in the reformation phase. Token lists are also sequentially numbered. Let us denote by tl_i the version number of the token list currently used by node i. Each node is committed to only one token list in the normal phase, and only nodes using the same token list communicate.

A source node broadcasts a message and until it receives an acknowledgement from the current token node. Each broadcast message contains an identifier $B_{s,n}$ which identifies its source node s and its number n, which is incremented for each message sent. The source does not broadcast further messages until an acknowledgement is received.

The current token node acknowledges messages. In order to avoid messages being acknowledged more than once, each node i maintains an array $M_i[s]$, that records the sequence number of the next message from source node s that needs to be acknowledged. When token node i receives a broadcast message $B_{s,n}$ such that $M_i[s]=n$, it *timestamps* the message, by transmitting an acknowledgement ACK(nts_i,$B_{s,n}$), where nts_i is the next expected token node. If there are no messages to acknowledged for a certain period of time, the token node issues a message ACK(nts_i,*Null*), which acknowledges the Null message, and is used only to make the token rotate, for reliability purposes.

If node receives an acknowledgement for a message it has not yet received, it sends a request for the missing message. If a node j receives an acknowledgement ACK(ts,x) such that $ts>nts_j$, it sends a message requesting the missing ACK message. A node requesting a missing message keeps re-broadcasting its request until it is answered. The current token node is responsible for answering these requests. A node does not accept the token until it has recovered all missing messages. This

guarantees that the token node has always all the information necessary to respond to requests from other nodes.

After the message is timestamped and the token is transferred λ times, it is guaranteed that at least $\lambda+1$ nodes have received the message and the corresponding timestamp. It is at this time that the message is *committed* and delivered to the user. This is called an λ-resilient system, for messages can be retrieved as long λ or fewer nodes fail.

Reformation phase: the processor membership algorithm

The reformation phase of the algorithm is responsible for forming the list of correctly functioning nodes in the broadcast group. The broadcast group is arranged as a circular ring, which is called the token list.. The reformation algorithm is a 3-phase algorithm for the originator node, and a 2-phase algorithm for the participating nodes. The originator node is the node that coordinated the process of forming the broadcast group. Any node may become an originator, but only one will eventually succeed including all functioning nodes in the token list.

The reformation phase is responsible for validating a new token list, electing a new token node, and re-starting the normal phase of the protocol. This phase ensures that, independently of of failures, only one token list is created, and that the old token list is not used by any node.

An originator is any node that detects a failure or recovery. It invites the other nodes to form a new broadcast list. The nodes join and commit this new list. The originator then elects a new token node, passes the token, and the normal phase is resumed. Since more than one node may have detected the failure and become an originator, more than one list may be formed, but the protocol guarantees that only one list will be considered

valid and be committed.

A new list must pass three tests to be valid: the majority test, the sequence test, and the resilience test. The majority test requires that a list have a majority of nodes. The lists formed are assigned a sequence number, and the sequence test requires that a node only join a list with a higher sequence number than the one it had previously joined. The resilience test verifies that none of the messages committed by the old list is lost. The resilience test also ensures that the old token list is not used by any node not aware of the reformation.

Majority test

To perform the majority test, the number of nodes in the system, n, must be known. This is not the number of correctly functioning nodes in the system, which is one of the things the protocol must determine, but an agreed, fixed maximum number of nodes at the time of the system creation. The majority test requires that any valid list must include at least $\lfloor n/2 \rfloor + 1$ nodes. This ensures that only one valid list will be formed.

Sequence test

Each node has a unique node number. A token list *version* number is formed by concatenating a *sequence* number to the node number. That is, *version_number* = *(sequence_number, node_number)*. The originator uses the sequence number of the last list it belonged to plus one as the new list sequence number when setting the new list version number, so that every list has a unique version number.

The sequence test requires that a node only join a list with a higher version number than the one it had joined before. If a node fails the sequence test, it informs its last list

version number to the originator, which increments it to form the new list version number the next time it becomes an originator.

The majority test ensures that any two lists have at least one node in common. Combined with the sequence test, this means that the lists formed have increasing version numbers, so that version numbers indicate the sequence in which the token lists were formed.

Resilience test

In an λ-resilient system, at least $\lambda+1$ nodes have received the message before it was committed (the node that acknowledges the message plus the following λ nodes). Consider the last message committed with the old list. If any of the $\lambda+1$ nodes that follows the node which acknowledged it is included in the new list, none committed messages will be lost. Furthermore, the old list will not have the nodes needed to commit any additional message. Therefore the resilience test requires that the new list include the node which is expected to issue the next timestamp using the old list, or any one of the nodes which follow it in the old list.

To allow the originator to determine the version number of the last list, a joining node informs the originator its old list version number and the next expected timestamp. The list with the highest version number is considered the old list, and is submitted to the resilience test. The node with the highest expected timestamp in the old list is the node with the most complete information, and is designated by the originator as the provisional token node. Each node then retrieves from the provisional token node any missing messages that were committed by the old list. When all nodes have retrieved all missing information, the originator issues a real token to the token node and the normal

phase resumes.

The reformation protocol is a three phase protocol for the originator and the new token node in the new list, and a two phase protocol for the other nodes. In the first phase, the originator sends an invitation to the other nodes to form a new list. In the second phase, the originator applies the majority and resilience tests to the new list, and if they are successful, it nominates a new token node and sends the new list to the other nodes. The nodes retrieve any missing information in this phase. In the third phase, the originator issues a token to the new token node.

Originator protocol

```
phase I:   broadcast invitation;
phase II:  wait for all responses or timeout;
               newTL := [nodes which responded];
               if (all responses = yes) and (pass majority and resilience tests) then
                   broadcast newTL to nodes in newTL;
               else
                   broadcast abort to nodes in newTL;
                   Modify TL version number, wait and restart;
phase III: wait for all responses from nodes in newTL or timeout;
               if (all responses = yes) then
                   Generate a new token and pass it to new token node;
                   Commit newTL;
                   Resume normal phase;
               else
                   Rend abort to new token node;
                   Wait and restart;
```

Participant protocol

```
phase I:   Wait for invitation;
               If (invitation passes the sequence test) and (the node does not belong
```

```
                to any other list) then vote yes;
                Else vote no;
phase II:   Wait for either newTL, or abort is received, or timeout;
                If (newTL received) and (still belongs to list) then
                    Recover missing messages and vote yes;
                    Commit newTL;
                    Resume normal phase (except new token node);
                Else
                    Leave the list, wait and restart;
phase III:  (new token node only)
                Wait for either new token, or abort is received, or timeout;
                If (newTL received) and (still belongs to list) then
                    Accepts token and resume normal phase;
                Else
                    Wait and restart;
```

7.3.2 Failure detection

If the token node fails, the other nodes in the system will detect its failure when they cannot retrieve a missing message or getting an acknowledgement for a new message. If any other node in the system fails, the failure will be detected when the token node tries to transfer the token to the failed node. In the worst case, the time to detect a failed node is the time it takes for the token to be transferred a complete cycle, which depends on the amount of traffic in the system.

In the absence of broadcast messages to be acknowledged, the token node waits for T_{idle} before passing the token by acknowledging a Null message. Temporary failures may cause the process of transferring the token to take $r \times T_{retry}$, where r is the number a token node tries to pass the token to the new token node and r is the number of times it tries to pass it. Therefore, in the worst case the algorithm detects a node failure in $n \times (T_{idle} + r \times T_{retry})$. This shows the trade off that exists between the

number of control messages in the protocol and the longest it will take for the protocol to detect a failed node. If T_{idle} and T_{retry} are made smaller, there will be more control messages in the network, but failures will be detected sooner.

7.3.3 Protocol performance

The protocol uses three types of control messages: acknowledgements, token passing, and confirmation messages. In the absence of failures, the protocol transmits one acknowledgement per broadcast message. The number of token passing messages depends on the token passing rate. When the token node transfers the token after each acknowledgement, the number of confirmation messages (sent to acknowledge receiving the token when there are no broadcast messages to timestamp) is on average $1+ P_{Q_empty}$ where P_{Q_empty} is the probability that the queue of broadcast messages waiting to be acknowledged is empty. Thus, at most two control messages are needed per broadcast.

The number of confirmation messages transmitted can be reduced by changing the token transfer rate, that is, making the token node pass the token only after acknowledging A messages. This, however, increases the number of messages that need to be kept for possible retransmission by a factor of A. In a token list with n nodes, the number of messages that must be kept by every node in the network (since every node eventually accepts the token and must answer requests for old messages) is $A \times (n-1)$. This is derived from the fact that a node only accepts the token if it can answer all possible requests for missing messages, so that a node does not need to keep the messages it had kept the last time it had the token. This analysis shows the trade off that exists between the storage requirements and the number of control messages sent per broadcast message.

7.3.4 Reliable Multicast in Cronic

A Conic message contains basically the following identifiers:

srcPort srcTask srcNode msgSeq typeSig destPort destTask destNode data

An easy way to implement a multicast is to use the broadcast protocol directly. A Cronic message presented to the broadcast layer is delivered to all nodes in the system. Thus, by sending a message with a multiple destination address (a list of addresses instead of the single address), and making the receiver end of the application protocol scan the list of receivers instead of looking only at one receiver address, a reliable multicast service is achieved.

Notice that this strategy does not increase the message traffic in the network. It is only the processing done at the receiving nodes that is increased. Also, more memory is necessary so for the receivers to store the messages. However, with the decreasing cost of implementing special VLSI devices, this load can be taken off the main processor by introducing reliable broadcast processors at every node in the network. This approach is taken in [Ashtana88].

It is also interesting to notice that the failure detection capabilities of the broadcast protocol is dependent on the network load. The more broadcast messages in the network, the faster the token will circulate through the token list, and the faster a failure will be detected. Thus, a constant use of broadcasts (as long as the number of broadcast messages does not exceed the network capacity) is advantageous.

The only change needed in the addressing of messages in order to support a reliable multicast is therefore to change the destination address from single to a list of

addresses:

srcPort srcTask srcNode msgSeq typeSig destList data

with *dest_list* being of the form

count destPort destTask destNode...destPort destTask destNode

where *count* specifies the number of destination addresses present in the message.

7.3.5 Speeding up the failure detection

The protocol already detects failures, but, as explained, the time delay between the failure occurring and the detection depends on the network traffic. This can be fairly long depending on the number of nodes and the number a node retries passing the token or getting an acknowledgement before declaring a failure. To shorten the time of detection, we decided to modify the algorithm so that a user task that becomes suspicious of some failure can ask the system to do a *chase* on the nodes. This is implemented as a round of token exchanging, with the token node not waiting for the corresponding period of timeout before passing the token. The nodes pass the token as soon as they receive it (acknowledging a message if necessary) until the node which started the chase receives the token back, at which point the token will have been passed along all nodes. If the suspicion that one of the nodes was down is true, this process will cause a reformation to occur, since the node preceding the failed one will not be able to pass the token. This capability is used by the Recovery Manager tasks when committing an action (section 7.5.1.1).

7.5 Implementation of the higher level mechanisms

All modules in Cronic have a request-reply *system* port, called RecoveryPort. A standard statement is inserted in all tasks in Cronic so that the first operation of a task is the sending of a RegisterMessage to its RecoveryPort. The RecoveryPort of each task is linked to a local Recovery Manager which is part of the node executive. The Recovery Managers communicate locally with all the application tasks, and remotely to all other recovery managers. The different mechanisms in Cronic are implemented by different Recovery Managers.

In the next sections we describe the implementation of the different Recovery Managers.

7.5.1 Backward recovery implementation

An action statement in a task is translated into communication operations through the standard RecoveryPort. For example, the statement

```
action by
    <primary>
else by
    <alternate 1>
end;
```

is translated to

```
send ActionCreate to RecoveryPort wait ActionResult;
case ActionResult of
    0: <primary>;
    1: <alternate 1>
end;
send ActionCommit to RecoveryPort wait ActionResult;
```

Upon receiving an *ActionCreate* message form a task, the Recovery Manager checkpoints the task to the associated reliable storage and replies *0* to the task. The task then proceeds to execute *<primary>*. If the action aborts, the Recovery Manager restores the task state from the checkpoint (at which point the task is again "waiting for reply") but this time the Recovery Manager replies *1* to the task, which proceeds to execute *<alternate 1>*. Therefore, the task itself is oblivious as to how backward recovery is implemented.

The receipt of an ActionCommit message indicates to the Recovery Manager that the task completed the action successfully. The Recovery Manager then initiates the process of committing the action. The commit protocol used is described in the next section.

This design is simple and elegant. The tasks are unaware of the way the recovery mechanisms are implemented, so that different recovery strategies (implemented by attaching different Recovery Managers to the tasks) can be used without recompiling the tasks. The rest of the executive is also unaffected.

7.4.1.1 Commit Protocol

In this section we present the commit protocol used by the Recovery Managers to commit a task. In the following description, we assume that a single Recovery Manager, the *coordinator*, initiates the commit algorithm, and that no Recovery Manager fails during the execution of the protocol; later we extend the algorithm to handle those cases. Although the commit protocol is run by the Recovery Managers on behalf of the tasks, for simplicity of presentation we describe it as if the tasks were executing the protocol themselves. For example, we use the phrase 'the coordinator sends a message to all tasks in its contingency set' as a short form of 'the Recovery

Manager of the committing task sends a message to all Recovery Managers associated to the tasks in the committing task contingency set'.

The commit protocol is a two phase commit protocol. In the first phase, the coordinator *c* prepares to commit (by taking a tentative checkpoint), and invites all tasks in its contingency set to prepare to commit. If after receiving the responses *c* learns that all tasks have successfully prepared to commit, *c* decides that the action should commit, and all tentative checkpoints are made permanent. Otherwise the decision is to abort the action. The second phase is then started by *c*, which records its decision in reliable storage and propagates it to all other tasks.

The message prepare to commit, sent by the coordinator, contains the action identifier and the coordinator contingency set. An action *a*, when receiving a prepare to commit message, checks to see if there are any actions in its own contingency set which are not included in the message contingency set. If there are, *a* acts as a sub-coordinator, sending prepare to commit messages, with an updated contingency set, to those tasks. Action *a* then waits the responses to its invitations, and informs the result to its coordinator.

Action *a* includes the updated contingency set in its response to the coordinator, and the coordinator uses these sets to update its own contingency set. Therefore, at the end of the first phase of the commit protocol the contingency set of the coordinator is the set of all participants in the action. The coordinator can then send its decision directly to all participants using the multicast facility.

time. To see this, suppose two actions, *b* and *c*, send prepare_to_commit messages to action *a*. Suppose *a* receives *b*'s prepare_to_commit message first; then *a* sends prepare_to_commit messages to all actions in $cg_set_a - cg_set_b$. At this point, all actions in $cg_set_a \cup cg_set_b$ have been sent a prepare_to_commit, that is, all actions in cg_set_a have been sent a prepare_to_commit, either by *a* or by *b* (or previously by some action dependent on *b*). Therefore, if *a* receives another prepare_to_commit, $cg_set_a - cg_set_x = \emptyset$, so there is no need for *a* to become a sub-coordinator any more. But notice that the action must still answer the invitation, otherwise the sending action will block waiting for the reply.

7.4.1.2 Commit Algorithm

For simplicity of presentation, a fictitious task called daemon is introduced, to perform the initiation and decision process of the coordinator Recovery Manager. In practice, the activities of the daemon task are performed by the coordinator Recovery Manager itself.

Daemon task:

```
send (prepare_to_commit, ∅) to "coordinator";
receive (result, cg_set) from "coordinator";
if result="agree_to_commit" then
        send "commit" to all s ∈ cg_set
else
        send "abort" to all s ∈ cg_set;
```

All actions t:

```
status_t := {"agreed_to_commit" if t agrees to commit, "abort" otherwise};

receive (prepare_to_commit, cg_set_r) from "r";
if status_t="agreed_to_commit" then begin
        take a tentative checkpoint;
        send (prepare_to_commit, cg_set_r U cg_set_t) to all s ∈ cg_set_t — cg_set_r ;
        for all s ∈ cg_set_t — cg_set_r do
            receive status_s from "s";
        if $ s ∈ cg_set_t — cg_set_r | status_s=" abort" then
            status_t:="abort";
send (status_t, cg_set_r U cg_set_t) to "r";

receive result from "daemon";
if result="commit" then
        make tentative checkpoint permanent;
        status_t:=committed;
else
        rollback to last permanent checkpoint;
```

Every action terminates its execution of the commit algorithm. To see this, note that any action that did not import any information from any other action clearly terminates. Let x be an action with $cg_set_x \neq \emptyset$. Suppose x receives a (prepare_to_commit, cg_set_y) from y. If cg_set_y includes properly cg_set_x, or $status_x \neq$agreed_to_commit, x sends $status_x$ immediately after receiving y's invitation. Therefore, there can be no deadlock involving y waiting for x's reply and x waiting for the coordinator decision.

Action x cannot be in deadlock waiting for replies of actions in its contingency set either. To show this, note that if x receives a prepare_to commit message from y, y received a prepare_to_commit message before x does, and by the algorithm x does

not send a prepare_to_commit to *y*. Therefore, *x* will receive replies form all actions it sent prepare_to_commit messages.

7.4.1.3 Purging old information

As indicated before, the system must maintain information about committed actions so that its dependents may commit. However, at some stage this information becomes obsolete, with all dependent actions already committed or aborted. To purge obsolete information about actions, the system uses the actions' dependency sets.

In the second phase of the commit protocol the system records in stable storage the action id and the dependency set of the committing action. Suppose *a* is committed action with a non empty dependency set. When a dependent action *b* reaches its commit point, it sends a prepare_to_commit message to *a*, and this message contains *b*'s contingency set.

Remember that the dependency set is a set of *tasks*, while the contingency set is a set of actions; also, recall that the action identifier contains the identifier of the task running it. Therefore action *a* can remove from its dependency set all the tasks whose actions are included in the contingency set received, for those tasks will be informed of *a*'s outcome by the commit protocol. That is, those tasks will not need the information about *a* anymore. When all tasks have been removed from *a*'s contingency set, the information about *a*'s outcome can be discarded, for it is guaranteed not to be used any more.

7.4.2 Forward recovery implementation

The forward recovery mechanism in Cronic uses some support already provided in Conic. Although Conic does not provide exception handlers, an exceptional occurrence in Conic causes a message being sent (by the executive, on behalf of the task) to the task standard *error* port. This message contains the identification of the problem that occurred (heap violation, division by zero, etc.). The implementation of exceptions in Cronic also uses this mechanism.

The design of the forward recovery is similar to the backward recovery. For example, the exception handler declaration

```
begin
    <A>
except when
    too_many: <handler 1>
    others: <handler 2>
end;
```

is translated to

```
send ExceptCreate to RecoveryPort wait ExceptResult;
case ExceptResult of
    0: <A>;
    1: <handler 1>;
    otherwise: <handler 2>
end;
```

When a task starts executing a block that has an exception handler, a message ExceptCreate is sent to the task RecoveryPort. This causes the Recovery Manager to mark the task stack so that it can be collapsed in the case an exception happens. The

Recovery Manager then replies 0 to the task. If an exception happens, a message containing the exception identification is sent by the executive through the task *error* port (which is linked to the Recovery Manager). The Recovery Manager then collapses the stack and replies the exception identification to the task.

7.4.2.1 Exception propagation protocol

The exception protocol uses the fact that exceptions have different priorities, and in a first phase determines which of the raised exceptions has the greater priority. If all tasks in the action have declared an exception handler (possibly an empty one, or a default one) for that exception in their action bodies, the tasks execute their respective handlers and the action terminates normally (that is, all tasks try to commit the action); if not, the action is aborted.

As the commit protocol, the exception propagation protocol has two phases and is based on diffusing information among tasks. However, while in the commit the coordinator send messages to actions in its contingency set, in the exception propagation it sends messages to tasks in its dependency set.

In the first phase, the coordinator sends a exception_notice to all actions in its dependency set. The exception_notice message contains the exception raised in the coordinator, the action identifier and the coordinator dependency set. After receiving all the responses, the second phase is started by the coordinator. It decides on the exception that should be propagated, based on the responses received, and propagates it to the other actions.

To decide which exception should be propagated, the coordinator must be able to detect

cycles in the dependency. If there are no cycles, it propagates its own exception (that is, the same exception sent in the exception_notice message). If the coordinator detects a cycle, it propagates the highest priority exception received in the dependents responses.

In order to allow the coordinator to detect cycles, a dependent action *a* replies to the exception_notice message with a message containing a set with all the actions which are dependent on it, that is, a set that is the union of all dependency sets in *a*'s own dependency chain. Then the coordinator needs only to check if it itself is included in the set received, in which case there is a cyclic dependency.

As in the commit protocol, an action *a* becomes a sub-coordinator if its dependency set includes actions not present in the set received with the exception_notice message. It sends exception _notice messages with an updated dependency set, and then waits for the responses. It then sends to the coordinator its response, which contains the winning exception (or its own exception if no cycle is detected) and the set of its dependency chain.

7.4.3 Implementation of replication

7.4.3.1 History independent passive clusters

Passive clusters composed of history-independent modules are the simplest to support. Upon system creation, only the recovery manager of the primary replies to the register message. In that way, only the primary starts executing, and the other cluster members are kept waiting for the reply to their register messages.

A recovery manager keeps a list of cluster members for all members that it manages.

With that list and the status information provided by the system, the recovery manager can determine which member is the next primary. Upon failure of the primary, the recovery manager associated to the next primary only needs reply to the register message sent when the standby was instantiated. The standby receives the message and starts executing.

7.4.3.2 History-dependent passive clusters

The process is similar when the task is part of a cluster. Upon receiving an *ActionCreate* message the Recovery Manager associated to the primary member checkpoints the task state to the other cluster members. It then replies *0* to the primary member. If the action aborts, the Recovery Manager associated to the next standby replies *1* to the standby task when it receives the information that the primary aborted.

For example, the scheduler does not need an extra state to define that the task is a standby and should not be scheduled. A standby is simply another task waiting for a message (that may never come, if the primary does not abort, or fail).

7.4.3.3 Implementation of active clusters

To guarantee that the destination module receives only one copy of the message, the messages are numbered sequentially, with every new cluster corresponding message being given a new number. Two approaches are then possible: to envisage the filter as being placed at the sender side, or at the receiver side. A filter at the receiver side can be implemented by the receivers simply discarding copies of the corresponding message sent by different members. Although very easy to implement, this approach is wasteful of network resources, since all cluster members will send their copy of a corresponding

message to the network. Since it is assumed that members fail only by omission, all messages are value-equivalent, and differ only at the sender identity.

A more efficient way to mask omission failures is to place the filter at the sender end. This can be done in two ways, which will be called the first message transmission, and round-robin transmission. In the first message transmission, as the name implies, only the first message that appears in the network is transmitted. The other corresponding messages don't need to be transmitted, because they are 'copies' of the first one. In this approach, it is the responsibility of the members themselves to monitor the traffic in the network to identify first messages and not to send copies. In Cronic, this could be implemented as a modification of the multicast protocol, which would, upon receiving a new message from the application, first check if it had not received a corresponding message from other cluster member (at another node). Only if it had not received such a message would it go on and transmit the message. Notice that it could happen that more than one node actually transmits the message, but in any case the token node would acknowledge only one of them (the other, although originating from another node, is for all purposes a copy).

Round-robin transmission, in which the responsibility for sending the message is circulated among cluster members, is a better approach to implement the filter at the sender end. In this approach only one message is sent , with the consequent advantage that the other senders do not waste processing time and network bandwidth, and the receiver does not need to discard message copies. That is the approach we have chosen as the default communication between active clusters in Cronic. The order the members transmit is based on information acquired at the reformation phase, which determines

to the same set of messages), and there is no need to broadcast the result.

The way the user specifies the filter is by writing a procedure. The user is passed an array containing the messages received, the number of messages present, and it must produce the result message. The filter is specified for each link among involving active clusters, and it is called when all messages have been received (since the system has the information about the failures and recoveries, it knows how many messages to expect from a cluster – the number of active members at the time). The filter procedure has the form

```
procedure filterName    (M:array of userMsgType; n:natural;
                         resultMsg:userMsgType);
```

where userMsgType is the type for the port where the filter will be used.

7.4.3.4 Reintegration

For an active homogeneous cluster, a technique similar to process migration can be used. Process migration (usually used for load balancing purposes) enables an existing process to be moved to another machine without otherwise changing its state. To move the process, a copy is made at the new location, and the old process is then destroyed. This mechanism can be easily modified so that the old process is not destroyed.

For active heterogeneous clusters, direct process copying is impossible. What is needed is a mechanism to translate a member state into a standard external format, so that it can be transmitted to other machine types. A mechanism similar to the one presented in [Herlihy82], for example, can be used. Without help from the module programming

language, this approach is only feasible for machine-heterogeneity (members are produced from the same programming language specification, for different machines). For heterogeneity of the type introduced by N-version programming, where different versions may be based in totally different internal structures, it may be impossible to transfer automatically the state of a member to another. For example, in truly independent N-version programming, not only the control structures are different, but also the internal representation of the state is different – in other words, different versions will use different data structures and different code to implement the same functionality (in fact the effectiveness of N-version programming is based on the assumption that the versions will be totally different).

7.4.4 Configuration Language implementation

The group module specifications are compiled into scripts that perform the instantiation and linking specified in the script. For group modules other than the system module, the script makes calls to the local kernel in order to create a task, and link its ports. The create operation for a group module (in the case of a group module being created inside another group module) is performed by executing the corresponding script of the internal module.

Chapter Eight

Conclusion

8.1 Summary of research effort

This work was prompted by the inadequacy of existing programming environments to provide a flexible support for the implementation of fault-tolerant systems. The main aim was to develop an environment, and associated tools, which offered a choice of techniques that could be used to suit different application requirements. This section provides a review of the major contributions of this research.

With a topic as broad as fault-tolerance, whose literature offers a myriad of different techniques, we decided to concentrate on a few mechanisms that would provide a basis for experimenting with the idea of selective fault-tolerance. It was also important that the support needed to implement the set of chosen mechanisms should be general enough to support new mechanisms in the future. We believe we achieved that goal.

The environment presented, Cronic, proved to be a good starting point for experimenting with selective fault-tolerance.

This work is also a synthesis. It has pulled together ideas and mechanisms presented separately, shown to be useful by themselves. The techniques chosen to be available in Cronic (backward recovery, forward recovery and passive replication) have been researched extensively, but one contribution of this work is to bring them together in an integrated form, as a programming environment that can be expanded to include other techniques for fault-tolerance. This integration presented several interesting problems.

The action mechanism presented is based on the FT-Action of [Jalote86], but the action in Cronic is more flexible than the FT-Action, because the participants in an action are determined dynamically. Another novel aspect introduced is the use of semantic information (in the form of actions) to implement passive replication more efficiently than previous approaches ([Loques86],[Kim86]). In this case, the role of cluster members (primary or standby) changes when an action is aborted. Therefore, the change of roles occurs not only in the case of crashes, as it is usual in systems using passive replication, but also in the case of inconsistencies detected by a user within an action. The change of roles enables a faster backward recovery, because the checkpointed task states don't need to be retrieved from stable storage.

Another contribution of this work is the Cronic environment itself, which is an ideal vehicle for experimenting with fault tolerance mechanisms for distributed systems. Several other mechanisms and techniques can be investigated to be included in Cronic.

8.2 Comments on the system model used

The system model chosen for Cronic employs processes and messages, and this choice was influenced by two facts. First, we had access to an existing programming environment (Conic) that could serve as a basis for the new environment, and second we envisaged that a particular class of applications would be developed in the new system (embedded process control systems). Although the duality of this system model and the object oriented model has been demonstrated [Lauer78] (even the duality of the techniques for fault-tolerance in both models has been argued [Shriva87]), it should be noted that the duality refers to expressive power alone. In other words, any model may be used in the implementation of a particular application (in the sense that it *can* be done), but one model may be better suited to a particular application.

This point is relevant because we were interested mainly in supporting the development of autonomous systems, which must function for long periods of time without direct human intervention. Such systems, which are usually process control systems, require efficiency both in time and space. Also, most autonomous systems present few opportunities for concurrent access to one component – therefore, support for concurrency control is not of great concern. Instead of concurrency control, which restricts the interactions between system components, control systems are often characterized by components needing to cooperate between themselves. They are also characterized by different fault tolerant requirements for different components. The fault tolerant requirements of a component that is running in the field (at a shop floor, for example, near the process under control) is often more stringent than that of a component running at a less hazardous environment (such as the control room). Cronic

provides the possibility of meeting these diverse requirements.

8.3 Suggestions for future research

There are several interesting research questions related to the programming environment we have developed. The most obvious extension is the integration of other mechanisms and techniques.

An example of a technique that could be included in Cronic is dynamic reconfiguration [Kramer89]. Although in general intended to provide for expansion and occasional changes in the system configuration, dynamic reconfiguration can also be used for fault-tolerance. For example, it could be used in Cronic for changing the replication level of a cluster without stopping the system. This is not possible at the moment because the cluster address information is fixed at configuration time, and all tasks (including standby members) are pre-instantiated, that is, they are instatiated at the time of system creation (as discussed, this approach was determined by the decision to support mainly autonomous systems).

An interesting research problem is the development of a software methodology and associated tools for building systems using the architecture presented. Tools for analysing a given system configuration and extracting some information to help improving the design would be very convenient. For example, tools for detecting circular dependencies, and for detecting nested actions which because of the communication pattern would always be discarded, would be very useful.

Another research possibility made possible by this work is the investigation of active

redundancy together with actions. Actions could be used in conjunction with active replication to provide a synchronization point for the replicas in the event of re-integration. The fact that the cluster members are at a known state at the end of an action could be used to relax the requirement that history-dependent active clusters in Cronic must be homogeneous.

Another interesting possibility is to investigate the use of active and passive clusters in the *same* application system. More specifically, to devise a suitable mechanism to interface a passive cluster with an active one. Suppose a passive cluster p is linked to an active cluster a. The problem is that if p fails inside an action, a and p must rollback. However, a will never roll back (there should be no need for an active cluster to rollback, active replication would otherwise be a waste of resources). This situation would introduce inconsistencies.

Support for concurrency is another research topic that should be explored. More specifically, we need to determine which are the best primitives and tools that the system should provide in order to support the specification of concurrency. Some form of concurrency control would be useful if it could be used selectively in parts of the system. Most process control applications do not need concurrency control (specially the control part of the application), but the management or supervisory part of the application would benefit from it. In particular, some applications need to maintain a database. Some form of support for concurrency control would be very useful in such cases.

The concurrency issue is closely tied to the system model we have chosen, in which tasks are autonomous entities that communicate exclusively by message passing. In this

model, all concurrency must be explicitly specified at the task level. In object oriented systems the concurrency can be made transparent to the user. Semantic information about the behaviour of the object can also be used to allow a greater degree of concurrency. Some research has been done already in this direction ([Herlihy84], [Allchin83]).

8.4 Experience and conclusion

We have used a prototype implementation of Cronic to build some simple examples of systems which use selective fault tolerance. Although more experience with the system is needed, the concept of integrating different mechanisms has proved very useful.

One interesting result of the experience of using Cronic was the realization that the flexibility of the action mechanism could, paradoxically, be a problem. The fact that actions are bound dynamically may complicate the verification of the correctness of the design of a large system. That was not a problem we actually experienced when developing the prototype systems, but we felt it could be a problem in larger systems. As mentioned in the last section, a tool for helping in the design of dynamic actions would be probably essential for large systems.

Another experience worth mentioning is related to the system implementation. More specifically, to the choice of Chang's algorithm as a basis for the reliable broadcast in Cronic. Chang's algorithm is simple and elegant, and this is reflected in its actual implementation and in the implementation of the higher levels mechanisms in Cronic. For example, the fact that it provides failure detection as an integral part of the reliable

broadcast simplifies the commit protocol for actions. However, two problems intrinsic to the algorithm were detected. The first problem was the unboundedness of time in the reformation phase of the algorithm. The problem is similar to the contention resolution in contention based networks, in which the contending nodes "give-up" transmitting for a random period of time on the assumption that this will permit one of them gaining exclusive access to the medium. However, this medium contention involves only *one* transmission, and the *contention window time* tends to be relatively small. In the reformation phase of Chang's reliable broadcast, the contention is for the attention of several nodes, and therefore it comprises a *set* of transmissions and some internal computation, which makes the contention window time perceptible. In applications where fast response time is important this unboundedness presents a problem. Furthermore, the failure of one node affects even the response time of tasks that need not be rolled back, since the reformation involves all nodes in the system.

The second problem is the non-extensibility of the algorithm. The majority test used in the reformation phase relies on the knowledge of the total number of nodes present in the system. To only way to change the total number of nodes in the system is by re-initializing the whole system (since in the general case the reliable broadcast algorithm cannot make use of itself to distribute the new total number of nodes).

These two problems are in fact related, because they both refer to the node membership problem discussed in section 7.3.1. An interesting research effort would be to use another approach to implement the three basic services in Cronic (global event ordering, failure detection and atomic broadcast, see section 7.2). For example, instead of ordering events based on causal relations, ordering could be based on real clocks, with

some mechanism for clock synchronization (which could be implemented in hardware [Kopetz85]). With a global (real) clock, both the unboundedness and the extensibility problem would be solved. Diffusion algorithms based on synchronized clocks [Cristian85] can be used to provide atomic broadcast with bounded time, and this in turn can be used to solve the processor membership in bounded time [Cristian88]. However, it is still necessary to investigate an appropriate failure detection algorithm, and also to examine the changes necessary in the action commit and exception resolution algorithms.

References

[Ada] The programming language Ada reference manual, *in Lecture Notes in Computer Science*, Goos, G. and Hartmanis, J., Eds., vol. 106, Springer-Verlag, Berlin, 1980.

[Allchin83] Allchin, J. E.,"An architecture for reliable decentralized systems", *PhD Thesis*, Technical Report GIT–ICS–83/23, Georgia Institute of Technology, Atlanta, Georgia, 1983.

[Anderson83] Anderson, T. and Lee, P. A., *Fault-tolerance: principles and practice*, Prentice-Hall, Englewood Cliffs, N.J., 1983.

[Ashtana88] Ashtana, A., "Design of a vlsi module for supporting distributed fault-tolerant architectures", Personal Communication, AT&T Bell Laboratories, Murray Hill, 1988.

[Anido86] Anido, R. O. and Kramer, J., "Synchronized forward and backward recovery for communicating processes". *Proceedings of the 7th DCCS*, October, 1986, Germany.

[Barbacci88] Barbacci, M. R., Weinstock, C. B. and Wing, J. M., "Programming at the processor-memory-switch level", *Proceedings of the 10th Int. Conf. on Software Engineering*, Singapore, April 1988.

[Babaoglu86] Babaoglu, O. and Drummond, R., "Streets of Bizantium: network architectures for fast reliable broadcasts", *IEEE Trans. Soft. Eng.*, vol SE-11, No. 6, June 1986.

[Carter87] Carter, W. C., "Experiences in fault tolerant computing, 1947–1971", *in Dependable Computing and Fault Tolerant Systems*, Avizienis, A., Kopetz, H.

and Laprie, J. C., Eds., pp. 1–36, Springer Verlag, NY, 1987.

[Chang84] Chang, J. and Maxemchuk, N. F., "Reliable broadcast protocols", *ACM Transactions on Computer Systems*, vol 2, No. 3, pp. 251–273, August 1984.

[Cooper85] Cooper, E. C., "Replicated distributed programs", *Proceedings of the 10th ACM Symp. on Operating Systems Principles, Operating Systems Review*, vol 19, No. 5.

[Cristian85] Cristian, F., Aghili, H. and Strong, R., "Atomic broadcast: from simple message diffusion to byzantine agreement", *IEEE*, pp. 200–206, 1985.

[Cristian88] Cristian, F., "Agreeing who is present and who is absent in a synchronous distributed siystem", *Digest of Papers FTCS-18*, pp. 206-211, Tokio, Japan, 1988.

[Davies73] Davies, C. T., "Recovery semantics for a DB/DC system", *Proceedings of the 1973 ACM National Conference*, pp. 136-141, 1973.

[Dulay86] Dulay, N. et all, "The Conic configuration language", Research Report DOC 84/20 (updated 86), Imperial College, London, 1986.

[Drummond86] Drummond, R.,"Impact of communication networks on Fault-tolerant distributed systems", *PhD Thesis*, TR86-748, Cornell University, Ithaca, NY, 1986.

[Gray79] Gray, J., "Notes on database operating systems", *in Operating Systems: an Advanced Course*, Bayer, R., Graham, R. M. and Seegmuller, G., Eds, Springer Verlag, NY, 1979.

[Gunningberg83] Gunningberg, P., "Voting and redundancy management implemented by protocols in distributed systems", *Proceedings of the 13th FTCS*, pp. 182–185, 1983.

[Herlihy82] Herlihy, M. and Liskov, B., "A value transmission method for abstract data types", *ACM Transactions on Computer Languages and Systems*, vol 4,

No. 4, pp. 527-551, October 1982.

[Herlihy84] Herlihy, M.,"Replication methods for abstract data types", *PhD Dissertation*, Technical Report MIT/LCS/TR-319, Dept. of Electrical Eng. and Computer Science, MIT, May 1984.

[Hoare78] Hoare, C. A. R., "Communicating sequential processes", *Comm. ACM*, vol 21, No. 8, pp. 666-677, August 1978.

[Jalote86] Jalote, P. and Campbell, R. H., "Atomic actions for fault-tolerance using CSP", *IEEE Trans. Soft. Eng.*, vol SE-12, No. 1, pp. 59-68, January 1986.

[Kim86] Kim, K. H. and Abouelnaga, A., "A scheme for coordinated execution of independently designed recoverable distributed processes", *Proceedings of the 16th FTCS*, pp. 130-133, 1986.

[Koo87] Koo, R. and Toueg, S., "Checkpointing and rollback recovery for distributed systems", *IEEE Trans. Soft. Eng.*, vol SE-13, No. 1, January 1987.

[Kopetz85] Kopetz, H. and Ochsenreiter, W., "Internal clock synchronization with a VLSI synchronization unit", *Technical Report TR1985/7*, Technical University of Vienna, 1985.

[Kramer87] Kramer, J. Magee, J. and Sloman, M., "The Conic toolkit for building distributed systems", *IEE Proceedings*, vol 134, Pt. D, No.2, pp. 73-82,March 1987.

[Kramer89] Kramer, J. Magee, J. and Sloman, M., "Constructing distributed system in Conic", *IEEE Trans. Soft. Engineering*, vol 15, No.6, pp. 663-675, June 1989.

[Lamport78] Lamport, L., "Time, clocks, and the ordering of events in a distributed system", *Comm. ACM*, vol 21, No. 7, pp. 558-565, July 1978.

[Lauer78] Lauer, H. C., Needham, R. M., "On the duality of operating systems structures", *in Proceed. Second Intern, Symposium on Operating Systems,*

IRIA, October 1978.

[LeBlanc87] , "Pronet", *IBM Journal of Research and Development*, vol 6, no 2, pp. 200-209, 1987.

[Liskov83] Liskov, B. H. and Scheifler, R. W., "Guardians and actions: linguistic support for distributed systems", *ACM Transactions on Programming Languages and Systems*, July 1983.

[Lomet77] Lomet, D. B.,"Process synchronization, communication and recovery using atomic actions", *SIGPLAN Notices*, vol. 12, No.3, pp.128-137, March 1977.

[Loques86] Loques, O. G. and Kramer, J. ,"Flexible fault tolerance for distributed systems", *IEE Proceedings*, vol. 133, no. 6, pp. 319-337, November 1986.

[Lyons62] Lyons, R. E. and Vanderkulk, W., "The use of triple modular redundancy to improve computer reliability", *IBM Journal of Research and Development*, vol 6, no 2, pp. 200-209, 1962.

[Mancini86] Mancini, L., "Modular redundancy in a message passing system", *IEEE Trans. Soft. Eng.*, vol SE-12, No. 1, pp. 79–86, January 1986.

[Metcalf76] Metcalf, R. M. and Boggs, D. R., "Ethernet: distributed packet switching for local computer networks", *Comm. ACM*, vol. 19, No. 7, pp. 395-404, July 1976.

[Moss81] Moss, J. E. B. "Nested transactions: an approach to reliable distributed computing", *PhD Thesis*, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computing Science, June 1981.

[Powel87] Powell, D. R., "Fault-tolerance in Delta-4", research report No. 87064, LAAS - Laboratoire d'Automatique et d'Analyse des Systemes, Toulouse, France, February 1987.

[Randell75] Randell, B., "System structure for software fault tolerance", *IEEE Trans. Soft. Eng.*, vol SE-1, No. 3, pp. 220-232, June 1975.

[Shriva79] Shrivastava, S., "Concurrent Pascal with backward error recovery: implementation", *Software Practice and Experince*, vol 9, pp. 1021-1033, 1979.

[Shriva87] Shrivastava, S., Mancini, L., and Randell, B., "On the duality of fault-tolerant systems structure", *Workshop on experiences on distributed systems*, Kaiserslaunterm, Germany, September 1987.

[Strom85] Strom, R. E. and Yemini, S., "Optimistic recovery in distributed systems", *ACM Transactions on Computer Systems*, vol. 3, No. 3, pp. 204–226, August 1985.

[Tso86] Tso, K. S., Avizienis, A., and Kelly, J. P. J., "Error recovery in multi-version software", *IFAC SAFECOMP 86*, Sarlat, France, 1986.

[Taylor86] Taylor, D. T., "Concurrency and forward recovery in atomic actions", *IEEE Trans. Soft. Eng.*, vol SE-12, No. 1, pp. 69-78, January 1986.

[Tyrrel86] Tyrrel, A. M. and Holding, D. J., "Design of reliable software in distributed systems using the conversation scheme", *IEEE Trans. Soft. Eng.*, vol SE-12, No. 9, pp. 921-928, September 1986.

[von Neumann56] von Neumann, J. "Probabilistic logic and the synthesis of reliable organisms from unreliable components", in *Automata Studies*, Shannon, C. E. and McCarthy, J., Eds., pp. 43-98, 1956.

[Wensley78] Wensley, J. H. *et all*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control", *Proceedings of the IEEE*, vol. 66, No. 10, pp. 1240-1255, 1978.

[Wood81] Wood, W. G., "A decentralized recovery protocol", *Dig. Papers FTCS-11: 11th Annual Int. Symp. Fault-Tolerant Comp.*, Portland, OR., pp. 159-164, 1981.